05_GIT

08 June 2025 21:29

1. Initialize Repository

git init
git status

2. Stage & Commit

git add .
git commit -m "YOUR MESSAGE"

3. Remotes

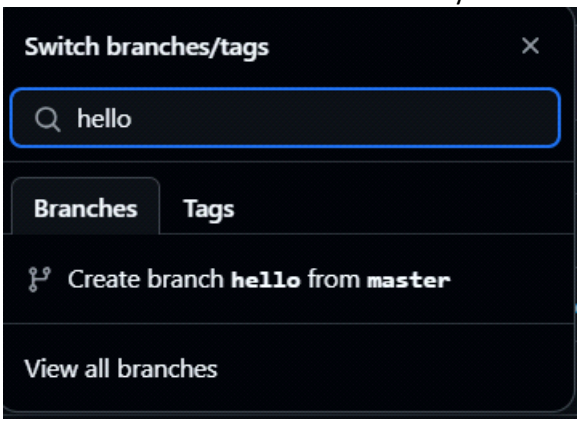
git remote add origin <REMOTE-URL>
git remote -v
git push origin master

4. History & Config

git log
git log --oneline
cat .git/config

5. Branching

 # Create
git branch -c sprint1
 # List
git branch -a
 # Switch
git checkout sprint1
(or) git switch sprint1
You can also create branches directly on **GitHub** via the web UI—



any branch you create there will contain the same data as your current branch once you pull it locally.

6. File Operations

 # Remove
git rm saturn6.py saturn7.py saturn8.py saturn9.py
 # Rename
git mv saturn1.py saturn11.py
 # Then:
git add .

git commit -m "describe your changes"

7. Push & Pull Branches

git push origin sprint1
git pull

8. Create & Work on New Branch

git pull
git checkout -b sprint2
add files, e.g.:
touch sun earth venus mercury
git add .
git commit -m "planets and start"
git push origin sprint2
git branch -a

9. Switching & Merging into Master

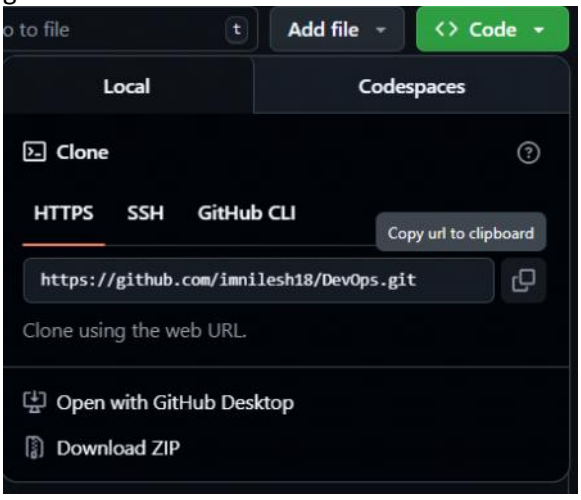
git switch master
(or) git checkout master
git merge sprint1
resolve conflicts, save merge message
git push origin master
To push all branches:
git push --all origin

10. .gitignore

vim .gitignore
add patterns, e.g.:
*.log
then:
git add .gitignore
git commit -m "Add .gitignore"
git push origin <branch>

11. Cleanup & Clone

rm -rf foldername
git clone <REMOTE-URL>



Rollback in Git: Essential Commands & Explanations

1. Overview of Rollback Techniques

Git provides multiple ways to undo changes or revert to previous states. Each method has different effects on your commit history and working directory:

Command	Effect	History Impact
git reset HEAD*	Unstages files (moves from "Staged" back to "Modified").	No new commits; history unchanged.
git reset --hard HEAD^	Resets working directory and index to one commit before HEAD (destructive).	Moves branch pointer back; destroys data.
git revert <commit-hash>	Creates a new commit that undoes changes introduced by <commit-hash>.	Preserves complete history.
git checkout .	Discards all local modifications in the working directory (for tracked files only).	No commits created; working tree cleared.

2. Explanation of Each Command

1. **git reset HEAD***
   - Unstages files that were added with git add.
   - Does **not** modify the working directory; only the index.
2. **git reset --hard HEAD^**
   - Resets the current branch to one commit before HEAD (HEAD^).
   - **Warning:** Discards all uncommitted changes in both the index and working directory.
3. **git revert <commit-hash>**
   - Generates a new "revert" commit that inverses the changes introduced by the specified <commit-hash>.
   - Safe for public branches, since history remains intact.
4. **git checkout .**
   - Replaces all modified files in the working directory with their last committed versions.
   - Only affects tracked files; untracked files remain untouched.

3. Rollback Hands-On Workflow

3.1 Initial File Editing

1. Open and edit the file:

   ```
   vim jupiter1.rb
   ```

2. View file contents:

   ```
   cat jupiter1.rb
   ```

3.2 Discarding Changes Before Staging

- Restore to last committed state:

```
git checkout jupiter.rb
```

- Verify contents:

```
cat jupiter.rb
```

3.3 Checking Status & Differences

1. Check repository status:

```
git status
```

2. See unstaged changes:

```
git diff
```

   - Shows line-by-line modifications in working directory.

3.4 Staging & Reviewing Staged Changes

1. Stage all changes:

```
git add .
```

2. Confirm staging:

```
git status
```

3. View staged (cached) differences:

```
git diff --cached
```

3.5 Unstaging a File

- To unstage jupiter.rb without discarding its changes:

```
git restore --staged jupiter.rb
```

- Then verify:

```
git status
git diff        # Shows changes remain in working dir
```

4. Rolling Back After a Commit

1. **Commit your changes**:

```
git add .
git commit -m "playbook"
```

2. **Inspect history and diffs**:

```
git status
git diff            # Working vs. staged (should be none immediately after commit)
git diff --cached   # Staged vs. HEAD (none)
git log --oneline   # List recent commits
```

```
git diff 358d7f8..a886cb6   # Compare two specific commits
```

4.1 Safe Revert (Keeps History)

- Revert the latest commit:

```
git revert HEAD
```

- Or revert a specific commit:

```
git revert <commit-id>
```

- **Behavior:** Opens an editor to enter a "revert" commit message. History shows both original and revert commits.

4.2 Hard Reset (No History)

- Reset to a specific commit, discarding all subsequent commits and uncommitted changes:

```
git reset --hard 358d7f8
```

- **Effect:** Branch pointer moves to 358d7f8; all later commits are no longer referenced.

5. Key Differences: reset --hard vs. revert

Aspect	git revert	git reset --hard
History Preservation	Yes – adds a new commit for the revert.	No – rewrites history, discards commits.
Collaboration Suitability	Safe for shared/public branches.	Dangerous on shared branches.
Data Recovery	Reversible (you can revert the revert).	Irrecoverable without backups.

6. Chapter Wise Summary

- **Rollback Commands**: Four primary commands (reset HEAD*, reset --hard, revert, checkout .) to undo changes at different stages.
- **Unstaging & Discarding Changes**: Use git restore --staged to remove from staging, git checkout . or git diff & git status to inspect and discard.
- **After Commit**: git revert for safe undo with history, git reset --hard for destructive rollback without history.
- **Best Practices**:
    - Use revert on public branches to avoid rewriting history.
    - Use reset --hard only on local or private branches when history rewrite is acceptable.

These notes cover every command and scenario from the lecture, providing clear purpose, usage examples, and best-practice guidelines.

Git SSH Login with a Remote

Repository: Essential Commands & Explanations

1. Overview

SSH (Secure Shell) allows you to authenticate with GitHub (or any Git remote) without repeatedly typing your username and password. Instead, you use an SSH key pair (private + public) to establish a secure connection.

2. Inspect & Clean Existing SSH Configuration

1. **View current Git config**

   ```
   cat .git/config
   ```
   - Shows your repository's remote URLs (HTTP or SSH) and other settings.

2. **Remove all existing SSH keys**

   ```
   rm -rf ~/.ssh/*
   ```
   - Deletes all files in your ~/.ssh/ directory so you can start fresh.

3. Generate a New SSH Key Pair

1. **Invoke key generation utility**

   ```
   ssh-keygen.exe
   ```
   - Follow prompts to:
     - Choose file location (~/.ssh/id_rsa by default)
     - Enter passphrase (optional, but recommended)

2. **Verify generated key files**

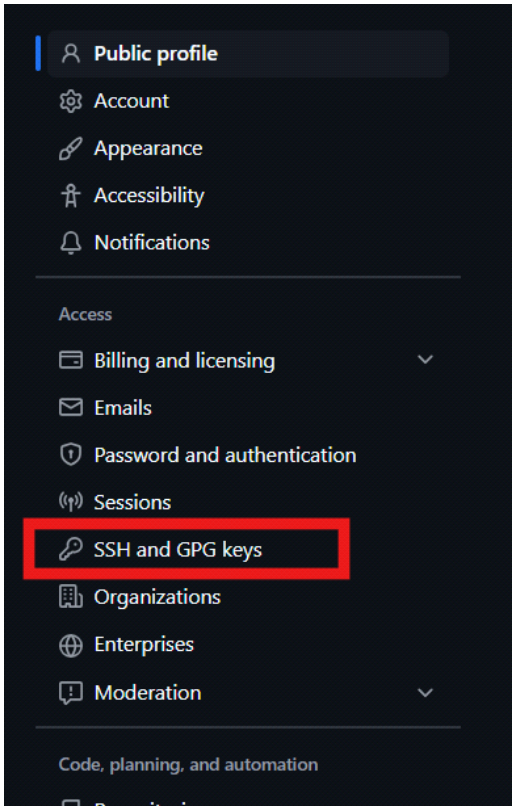
   ```
   ls ~/.ssh

   # Expected output:
   id_rsa     id_rsa.pub
   ```
   - id_rsa is your **private** key (keep it secure)
   - id_rsa.pub is your **public** key (share this)

4. Add Your Public Key to GitHub

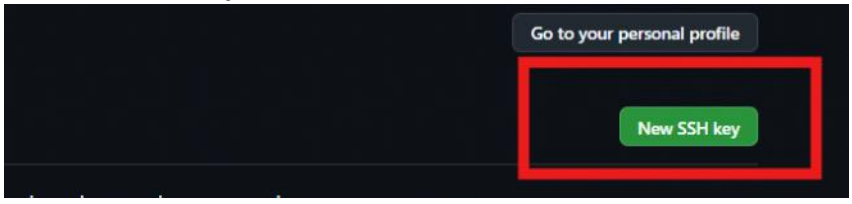
1. **Display the public key**

   ```
   cat ~/.ssh/id_rsa.pub
   ```

2. **Copy** the entire contents (ensure it starts with ssh-rsa and ends with your email).
3. **On GitHub** (in your account settings):
   - Navigate to **"SSH and GPG keys"**



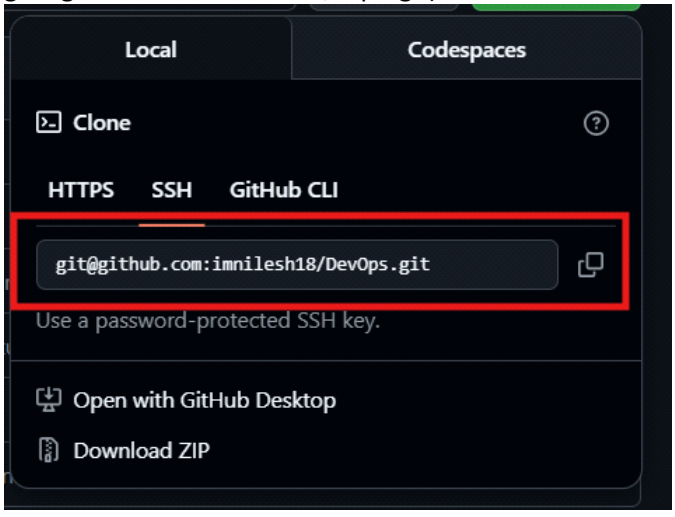
- Click **“New SSH key”**



- **Paste** the public key
- **Save** (double-check you’ve pasted the public, not the private, key)

5. Clone a Private Repository Over SSH

1. **Obtain SSH clone URL**
   - On your GitHub repo page → click **“Code”** → select **SSH** → **Copy** URL (e.g., git@github.com:username/repo.git)



1. **Clone using SSH**

   git clone git@github.com:username/repo.git
   - The client uses your **private** key to prove identity against the **public** key on GitHub.

2. **Confirm host authenticity**
   - On first connect, you'll see a fingerprint check:

     The authenticity of host 'github.com (IP address)' can't be established.
     RSA key fingerprint is SHA256:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.
     Are you sure you want to continue connecting (yes/no/[fingerprint])?
   - Type **yes** to accept and add GitHub to your known_hosts.

3. **Result**
   - Clone proceeds without prompting for username/password.
   - Future operations over SSH are seamless and secure.

6. Benefits of SSH over HTTP

Aspect	SSH	HTTPS
Authentication	Key-based; no password prompts	Username/password or PAT every time
Security	Strong crypto handshake	Encrypted, but relies on credentials
Convenience	Once set up, no further prompts	Must enter credentials or use credential helper
Suitable For	Private & public repos on shared machines	Public repos or simple setups

7. Chapter Wise Summary

- **Inspect & Clean**: Checked .git/config, removed old keys (rm -rf ~/.ssh/*).
- **Key Generation**: Ran ssh-keygen.exe, producing id_rsa & id_rsa.pub.
- **Add to GitHub**: Copied id_rsa.pub contents, pasted in GitHub's **SSH and GPG keys** settings.
- **Clone via SSH**: Used git clone git@github.com:..., accepted fingerprint, no credentials needed.
- **Benefits**: SSH offers secure, password-less authentication, ideal for private repositories and automation.

These notes capture every step and command from your lecture, organized for clarity and future reference.

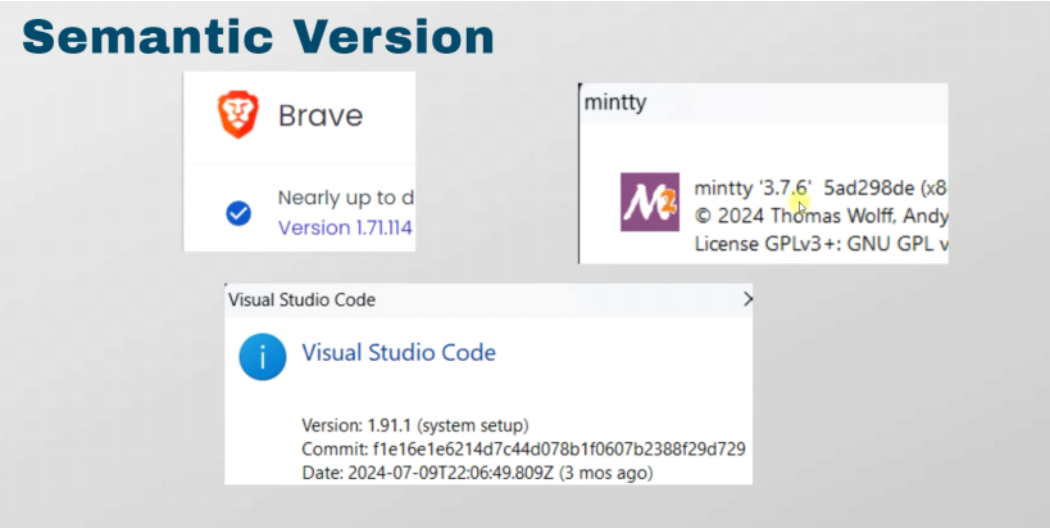
Git Tags & Semantic Versioning

1. Introduction to Git Tags

Git Tags & Semantic Versioning

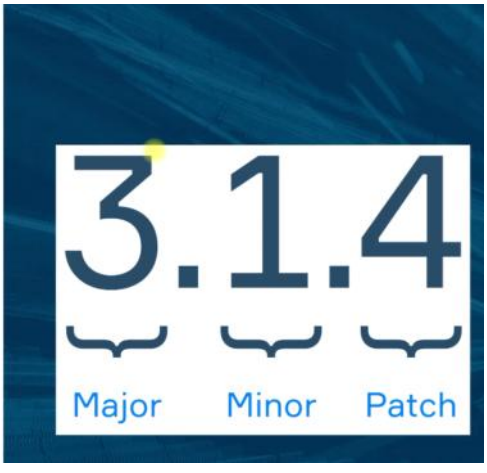
1. Introduction to Git Tags

- **Purpose:** Attach meaningful version labels (“tags”) to commits, typically used for releases.
- **Use Case:** Semantic versioning (SemVer) in modern software projects.



2. Semantic Versioning (SemVer) Basics

- **Format:** Major.Minor.Patch
  1. **Patch** (e.g. 1.71.114 → 114):
     - Bug fixes or tiny tweaks; fully backward-compatible.
  2. **Minor** (e.g. 3.7.6 → 7):
     - New features or improvements; backward-compatible.
  3. **Major** (e.g. 1.91.1 → 1):
     - Breaking changes; not backward-compatible.
     - **Example:** Upgrading support from JDK **11 to JDK 17 requires bumping the Major version, since older clients cannot run the new build.**



3. Creating & Managing Git Tags

TAG A COMMIT

- git tag TagName commit
- git show tag

ANNOTATED TAGS

- git tag -a TagName -m"message" [commit]
- git tag -a v2.1.6 -m"Release for something"

PUSH TAGS

- git push origin tag TagName
- git push --tags

3.1. Listing & Showing Tags

- **List all tags:**

  git tag

- **Show tag details:**

  git show <tag-name>
  - Displays: tagged commit, author of tag, tag message, diff of that commit.
  - Example: git show v2.0.0 → press Q to quit the pager.

3.2. Annotated Tags (Recommended)

- **Create an annotated tag:**

  git tag -a <TagName> -m "Your message" [<commit-id>]
  - If <commit-id> is omitted, tags the current HEAD.
  - <TagName> follows SemVer (e.g. v2.1.6).
  - Example:
    git tag -a v2.1.6 -m "Release for feature X"

3.3. Pushing Tags to Remote

- **Push a single tag:**

  git push origin <TagName>

- **Push all local tags:**

  git push --tags

- **Alternative (combined):**
  - In VS Code Command Palette → **Git: Push and Follow Tags**

4. Hands-On Exercise: Tagging a Forked Repo

4.1. Fork & Clone the Repository

1. **Fork on GitHub:**
   - Go to github.com/hkhcoder/vprofile-project → click **Fork**.

- Rename to proton (or your chosen name).
- Uncheck copying other branches if prompted.

2. **Clone locally:**
   - Click **Code → HTTPS**, copy URL.
   - In VS Code:
     - **Source Control** → **Clone Repository** → paste URL → choose local folder (…/learninggit/).
     - **Open** project in VS **Code when prompted.**

4.2. Switch to the atom Branch

- In VS Code status bar or **Branches** menu → select branch atom.

4.3. Make & Commit a Change

1. **Edit** README.md:
   - Change text (e.g., "JDK 17" → "JDK 21"), add extra hash.
   - **Save** (Ctrl **+ S).**
2. **Commit** with VS **Code UI:**
   - **Source Control** → **Commit** → enter message "read file changes" → **Accept**.

4.4. Tagging via Terminal

1. **Open integrated terminal:**
   - Ctrl + Shift + P → "Select Default Profile" → choose **Git Bash** (or desired shell).
   - View via **View → Terminal**.
2. **List existing tags:**
   ```
   git tag
   ```
3. **Show latest tag details:**
   ```
   git show v2.0.0    # example tag
   ```
   - Press Q to exit.
4. **Create a new tag:**
   - Suppose last tag was v3.5.2.
     ```
     git tag -a v3.5.3 -m "Patch: bug fix for README version"
     ```
5. **Verify new tag:**
   ```
   git tag
   ```
6. **Experiment:**
   - Make additional commits.
   - Create several tags (bump Patch, Minor, or Major) to practice SemVer.

5. Pushing Commits & Tags from VS Code

5.1. Sign In to GitHub

1. **Install Extension:**
   - **Extensions** pane → search "GitHub Pull Request" → **Install**.
2. **Authorize:**
   - In **GitHub Pull Request** view → **Sign in** → allow in browser → **Authorize Visual Studio Code**.

5.2. Push Changes

1. **Sync commits:**
   - **Source Control** → **Sync Changes** → confirm.
2. **Push tags:**
   - Command Palette (Ctrl + Shift + P) → "Git: Push Tags"
   - Or use "Git: Push and Follow Tags" to push both commits & tags at once.

6. Creating a GitHub Release

1. **On GitHub** → navigate to your **proton** repo → **Releases** → **Draft a new release**.
2. **Select tag:** choose one of your newly pushed tags.
3. **Title & description:** e.g. "UI bug fix in README version".
4. **Publish release** → tags become formal releases.

7. Key Takeaways

- **Git tags** are lightweight labels pointing to commits—ideal for marking releases.
- **Semantic versioning** follows Major.Minor.Patch, conveying compatibility and change magnitude.
- **Annotated tags** (-a) store metadata (author, date, message).
- **Pushing tags** is separate from pushing commits.
- **VS Code integration** simplifies committing, tagging, and releasing.

Next Steps: As you progress to CI/CD pipelines (e.g., Jenkins), these tags drive automated builds and deployments. Ensure your team follows SemVer conventions and consistently tags releases.

GitHub Copilot

1. Introduction

- **Topic:** What GitHub Copilot is, how it helps developers, and how to integrate it into VS **Code.**

2. What Is GitHub Copilot?

- **Definition:** An AI-powered coding assistant developed by GitHub.
- **Training Data:** Billions of lines of public code repositories.
- **Integration:** Works inside popular code editors (e.g., VS **Code).**

3. Core Capabilities

1. **Code Suggestions & Completion**
   - Autocompletes lines or entire functions as you type.
2. **Inline Documentation**
   - Generates comments or docstrings based on code context.
3. **Error Fixes & Refactoring**
   - Proposes corrections or improvements for existing code.
4. **Learning Tool**
   - Reveals idiomatic patterns and best practices as you review its suggestions.

4. Benefits for Day-to-Day Development

- **Speed:** Write code faster with fewer keystrokes.
- **Accuracy:** Reduce syntax errors and common bugs.
- **Learning:** Discover new approaches and patterns.
- **Productivity:** Maintain flow without context-switching to external docs.

5. Licensing & Trial

- **Free Trial:** GitHub Copilot offers an initial free trial period.
- **Subscription Required:** After trial, a paid subscription is necessary for continued use.
- **Course Usage:**

- Entire course will rely on the free-trial version only.
- Monitor usage before trial ends; subscription optional later.

6. Integrating Copilot into VS Code

6.1. Prerequisites

- **Code Editor:** Visual Studio Code (VS **Code).**
- **GitHub Account:** Logged in via your default browser.

6.2. Installation Steps

1. **Open Extensions Pane**
   - Click the Extensions icon in VS Code's Activity Bar.
2. **Search & Install**
   - Type "Copilot" → choose **GitHub Copilot** (includes Chat). → **Install**.
3. **Authenticate**
   - Click the Copilot icon in the status bar.
   - Select **Sign in to use Copilot** → browser opens → **Continue with GitHub** → return to VS Code.

6.3. Verification & Usage Indicators

- **Copilot Icon:** Appears in VS **Code's status bar.**
- **Usage Stats:**
  - E.g., "88.6% of code completions used"
  - "37% of chat messages used"
- **Limits:** Free trial quotas apply; plan suggestions accordingly.

7. Copilot Chat

- **Accessing Chat:**
  - Click the Copilot icon → **Open chat** panel appears.
- **Functionality:**
  - Ask coding questions, request code snippets, explanations.
  - Limited number of chat messages in free trial.

8. Post-Setup Workflow

- **Future Lectures:** Hands-on demonstrations of:
  - Autocomplete in real coding scenarios.
  - Writing inline comments with Copilot.
  - Fixing bugs and refactoring suggestions.
- **DevOps Integration:** Applying Copilot suggestions in build scripts, pipelines, and automation tasks.

9. Subscription Details (Optional)

- **30-Day Free Trial:** Begins upon activation.
- **Checking Plans:**
  - Visit GitHub Copilot subscription page in your account settings.
- **Upgrade Anytime:** Allows uninterrupted access to Copilot features after trial.

10. Chapter Wise Summary

- **Lecture Goal:** Understand GitHub Copilot's purpose, capabilities, and integration process.
- **Key Commands & Actions:**
  1. Install **GitHub Copilot** extension in VS **Code.**
  2. Sign in via GitHub through your browser.

3. Monitor usage stats in the status bar.
4. Access **Copilot Chat** for conversational AI assistance.

- **Next Steps:** In upcoming lectures, apply Copilot in coding exercises across various languages and DevOps workflows.