GIT/GITHUB - PART 2

# Chapter 1: Git Workflow Enhancements & `HEAD` Concept

## 1. Basic Git Workflow Reminder:

• Standard process: Make changes → Stage them → Commit them

• Purpose of staging: Allows combining multiple changes into a single commit.

## 2. Skipping the Staging Step:

Use: git commit -a -m "message"  
• -a flag:  
 • Automatically stages all tracked and modified files.  
 • Skips git add for already tracked files.  
 • Does not work on new (untracked) files – must run git add first.

## 3. Use Cases for `git commit -a`

• Best for small, quick changes where staging isn’t needed.  
• Not ideal for:  
 • New files (since they’re untracked).  
 • Complex changes requiring detailed commit messages.

## 4. Limitations of `-m` Flag

• -m flag is used to add a commit message inline.  
• Not suitable for:  
 • Long or complex commit descriptions.  
 • Messages that require following best practices (e.g., multiline messages).

## 5. Caution with `-a` Shortcut

• You skip the staging area, so:  
 • You can’t selectively include files.  
 • All modified, tracked files are committed at once.  
 • Be sure you’ve finalized everything you want in the commit.

## 6. Checking the Commit

• Use git log to confirm:  
 • The latest commit appears at the top.  
 • The HEAD now points to this new commit.

## 7. Understanding `HEAD`

• HEAD = pointer to the currently checked-out snapshot.  
• Tracks where you are in the Git history.  
• Usually points to the latest commit in the current branch.

HEAD Behaviors:  
• Used in commands like git diff, git branch, git status.  
• Can move to older commits (e.g., during rollbacks or reverts).  
• Think of it as a bookmark in your project timeline.

## 8. HEAD & Branches

• In branch workflows, HEAD points to a commit on a specific branch.  
• Acts as a reference point for most Git operations.  
• You’ll encounter HEAD often when working with:  
 • Undoing changes  
 • Rolling back commits  
 • Merging branches

# Chapter 2: Advanced Git Usage: Viewing and Managing Changes

## 1. Viewing Commit History with More Detail

🔹 git log  
• Shows the commit message, author, and date.  
• Useful for basic tracking of changes.

🔹 git log -p  
• The -p flag stands for patch.  
• Displays the exact lines changed in each commit.  
 o Plusses (+) indicate added lines.  
 o Dashes (-) indicate removed lines.  
• Format matches diff -u output.  
• Automatically uses a pager to help navigate large output (Page Up, Page Down, arrow keys).  
• Helpful for tracking down bugs or outages by examining line-level changes.

## 2. Using git show for Specific Commit Details

• Syntax: git show <commit\_id>  
• Displays:  
 o The commit metadata.  
 o The changes (patch) associated with that commit.  
• Commit ID can be found using git log.

## 3. Using git log --stat

• Shows statistics per commit:  
 o Files changed.  
 o Lines added and removed.  
• Gives a summarized view of the impact of each commit.

## 4. Git Diff for Uncommitted Changes

🔹 git diff  
• Shows differences between:  
 o Working directory and staging area.  
• Useful before committing when changes have accumulated.  
• Supports narrowing down to specific files:  
 git diff <filename>

## 5. Interactive Staging with git add -p

• The -p (patch) flag with git add lets you:  
 o Review each change before staging it.  
 o Choose what to stage (great for partial commits).

## 6. Checking Staged vs. Unstaged Changes

🔹 git diff --staged  
• Shows the difference between:  
 o Staging area and the last commit.  
• Useful for final verification before committing.

## 7. Best Practices & Workflow Tips

• Git gives full visibility into what was changed and when.  
• Tools like:  
 o git diff  
 o git log -p  
 o git show  
 o git add -p  
Help ensure accurate and thoughtful commits.  
• Don’t worry about memorizing everything immediately—practice and repetition will build familiarity.  
• Git stores all historical information in the repository—you can always refer back.

## ✅ Summary of Commands

|  |  |
| --- | --- |
| Command | Purpose |
| git log | Show commit history |
| git log -p | Show commit history with line-level changes |
| git log --stat | Show summary of changes in each commit |
| git show <commit> | Display details of a specific commit |
| git diff | Show unstaged changes |
| git add -p | Interactively stage specific changes |
| git diff --staged | Show staged but uncommitted changes |

Here’s your content reformatted with proper **Microsoft Word formatting style** – including section headers, bullet points, inline code, and bold/italic emphasis. You can download the .docx version using the link at the end.

**Chapter 3: Removing, Renaming & Ignoring Files in Git**

**1. Removing Files with git rm**

You can remove files from your Git repository using the git rm command.

This **deletes** the file from your working directory **and** stages the removal for the next commit.

**Steps:**

1. Check contents of directory:
2. ls
3. Remove the file:
4. git rm <filename>
5. Confirm removal:
6. ls
7. Check the status:
8. git status
9. Commit the removal:
10. git commit -m "Deleted unneeded file"

**Result:**

* The file is removed.
* Deletion is tracked and committed with a message.
* Git reports the number of **deleted lines** and shows the **file was deleted**.

**2. Renaming or Moving Files with git mv**

If a file name no longer reflects its purpose, rename it using:

git mv <old\_filename> <new\_filename>

**Example:**

git mv disk\_usage.py checkFreeSpace.py

Then check status:

git status

Commit the change:

git commit -m "Renamed script to checkFreeSpace.py"

**Note:**

* git mv works like the regular mv in Linux.
* It can be used to move files across directories in your repo.

**3. The Role of git status**

git status is extremely useful to:

* Show **tracked vs. untracked** files.
* Display which files are **added, modified, deleted, or renamed**.
* Keep your repo organized and clean.

Avoid clutter: Long lists of untracked/generated files can make it hard to spot important changes.

**4. Ignoring Files with .gitignore**

To avoid tracking unnecessary files (e.g., OS-generated files), use a .gitignore.

**Steps:**

1. Create .gitignore file:
2. echo .DS\_Store > .gitignore
3. View hidden files:
4. ls -la
5. Add .gitignore to the repo:
6. git add .gitignore
7. Commit:
8. git commit -m "Add .gitignore to skip OS artifacts"

**Notes on .gitignore:**

* Files listed here will be skipped by git status, git add, etc.
* Helps reduce **noise** in Git output and improves clarity.

**✅ Summary of Commands**

| **Command** | **Purpose** |
| --- | --- |
| git rm <file> | Remove and stage file for deletion |
| git mv <old> <new> | Rename or move files |
| git status | Show tracked/untracked/modified file status |
| ls, ls -la | View files (including hidden ones) |
| .gitignore | File to list what Git should ignore |
| git add .gitignore | Stage the ignore list for commit |

Here are the **key points from the video transcript**, formatted as **MS Word-style notes** using proper headings, bullets, and indents.

**Chapter 4: Git Reverting Techniques**

**🔁 Undoing Changes in Git**

**🧾 1. Reverting Unstaged Changes**

* Use git checkout <filename> to **revert a file back** to its last committed version.
* Example:
  + Modify a file (e.g., remove a function).
  + Realize it broke the script.
  + Use git checkout to restore the file.
* Git status will show:
  + File modified.
  + Not yet staged.
* ✅ **Tip:** Think of git checkout as checking out the original file from the last snapshot.

**🔧 2. Reverting Part of a File**

* Use git checkout -p <filename> to **interactively select changes** to discard.
* Git will prompt for **each change** whether to keep or revert.

**📥 Undoing Staged Changes**

**🧾 3. Unstaging a File**

* If you accidentally stage a file (e.g., via git add \*), you can **unstage** it with:
* git reset HEAD <filename>
* HEAD refers to the **latest snapshot** you have checked out.
* git reset is the **counterpart to git add**:
  + git add ➝ adds changes to staging.
  + git reset ➝ removes changes from staging.

**🛠 4. Interactive Reset**

* Use git reset -p to **interactively unstage specific changes**.

**✅ Summary**

* git checkout <file>: Revert unstaged changes.
* git checkout -p: Interactively revert changes.
* git reset HEAD <file>: Unstage staged changes.
* git reset -p: Interactively unstage specific changes.

Here's a neatly formatted set of **MS Word-style notes** for this segment of the video on fixing commits in Git, using proper structure, bulleting, and technical clarity:

**Chapter 5: Fixing Commits in Git – Notes**

**❌ Common Commit Mistakes**

* Forgetting to include a file in a commit.
* Writing an incomplete or unclear commit message.
* Committing with incorrect content or description.

**🛠 Fixing Recent Commits with git commit --amend**

**🧾 1. What It Does**

* Replaces the **most recent commit** with a new one.
* Reuses the commit workflow using the current **staging area**.
* Updates the commit message if desired.

**🔧 2. Use Cases**

* Add a file that was **missed in the last commit**.
* Fix or improve the **commit message** (e.g., add a bug link, better description).
* Modify both files and message in one go.

**🧪 Example Workflow**

1. Add the forgotten file to the staging area using git add.
2. Run:
3. git commit --amend
4. Git opens an editor:
   * Shows old commit message.
   * Shows the updated list of staged changes.
5. Edit the commit message (optional).
6. Save and exit the editor.
7. ✅ Commit is updated!

**✏️ Fixing Just the Commit Message**

* If no files are staged, running git commit --amend **only opens the message editor**.
* You can then **change just the description** without altering the content.

**⚠️ Important Caution**

* Do **NOT use --amend on public/shared commits**.
  + It **rewrites history** by replacing the previous commit.
  + Can **confuse collaborators** and **break shared history**.
* ✅ Safe to use for **local commits** before pushing.

**✅ Summary**

* git commit --amend:
  + Fix missed files.
  + Edit commit messages.
  + Combine staged changes into previous commit.
* Safe for local use, avoid on public commits.

**Chapter 6: Reverting Bad Commits in Git – Notes**

**🔍 Why Revert?**

* Sometimes a bug is introduced **after a commit** is made.
* You may need to **roll back** to a previous stable state.
* Git allows fixing such issues **without rewriting history**.

**❌ Scenario: A Faulty Commit**

* You made changes to a shared script and committed it.
* After some time, users report issues.
* Instead of manually debugging, you decide to **rollback the changes quickly**.

**🔄 Using git revert**

**📌 What It Does:**

* Creates a **new commit** that **inverts** the changes made in a specific previous commit.
* Does **not delete** the commit; instead, it adds a **reversal commit**.
* Keeps history **intact and clear**, making it safe for **public/shared repos**.

**🧪 Basic Workflow**

1. Revert the latest commit:
2. git revert HEAD
3. Git opens a text editor for the **new commit message**:
   * Pre-filled with a default message like:
   * Revert "Add call to disk\_full function"
   * Includes the original **commit hash**.
   * 🔍 It's good practice to **add a reason** for the revert (e.g., "function not defined").
4. Save and exit the editor.
5. Git creates a **new commit** that **undoes the last commit**.

**🧾 Check the Log**

* View the last two commits and their differences:
* git log -p -2
* You'll see:
  + The original commit showing **added lines** (with +).
  + The revert commit showing **removed lines** (with -).

**⚠️ Important Notes**

* git revert is **safe for public repositories** (unlike git reset or --amend).
* It’s ideal when:
  + A commit is already pushed.
  + You want to maintain **team collaboration and traceability**.

**🔄 Coming Up**

* Reverting older commits (not just the latest) will be discussed in the next session.

Would you like me to export this as a **.docx file** for printing or documentation use?