# Git Fundamentals

Dr. Prashant Aparajeya

# Course Objectives

- Understand what version control is and why Git is used
- Learn the core Git commands and workflows
- Master branching, merging, and conflict resolution
- Work confidently with remote repositories and GitHub
- Develop effective collaboration strategies for team-based projects

# Module 1: Introduction to Version Control with Git

# Local Copy Methods

Before formal version control systems, developers often relied on simple local copy methods:

- Manual backups: Copying files to separate folders with date stamps
- Commenting out code: Preserving old versions within the same file
- Zip archives: Compressing project folders at different stages

These methods, while simple, have significant drawbacks:

- Limited history tracking
- Difficulty in collaboration
- Increased risk of data loss
- Inefficient storage use

# Formal Version Control Systems

Modern version control systems offer sophisticated features:

- Comprehensive history: Detailed logs of all changes
- Branching and merging: Support for parallel development
- Collaboration tools: Mechanisms for multiple developers to work together
- Backup and restore: Easy recovery of previous versions

# Git: A Distributed Version Control System

Git is a distributed version control system created by Linus Torvalds in 2005. It has become the de facto standard for version control, with nearly 95% of developers reporting it as their primary system as of 2022.

Key features of Git include:

- **Distributed nature**: Every developer has a full copy of the repository
- **Speed and efficiency**: Designed for performance, even with large projects
- **Branching and merging**: Powerful tools for non-linear development
- **Data integrity**: Uses cryptographic hashing to ensure data consistency

# Basic Git Terminology

To understand Git, it's crucial to familiarize yourself with these key terms:

**Repository**: A container for a project, including all files and their version history.

**Commit**: A snapshot of changes at a specific point in time. Each commit has a unique identifier and contains:

- Changes made to files
- Author and timestamp
- Message describing the changes

**Working Directory**: The folder on your local machine where you're actively working on files.

**Staging Area**: An intermediate area where changes are prepared before committing. Also known as the "index," it allows you to selectively choose which changes to include in the next commit.

**HEAD**: A pointer that refers to the latest commit in the current branch. It represents your current working state and the base for new changes.

# Git Workflow

The basic Git workflow involves three main states:

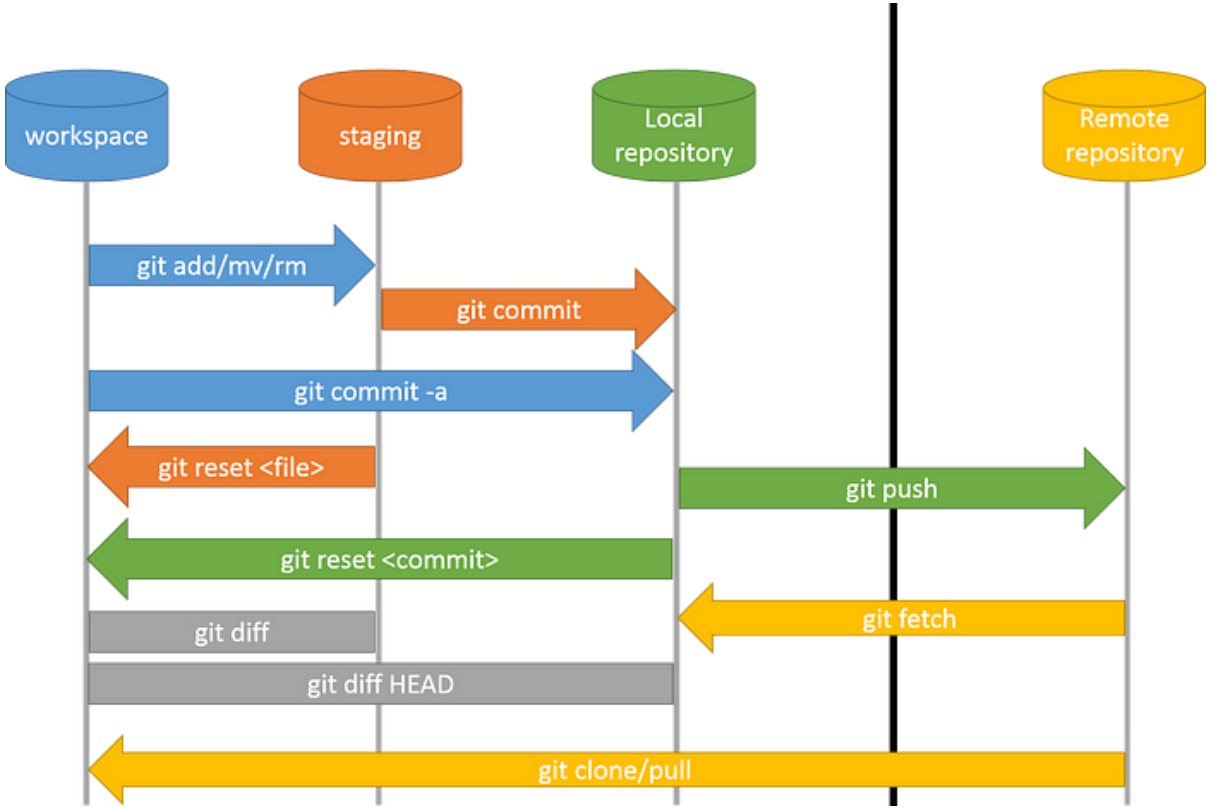
**Modified**: Changes have been made to files in the working directory but not yet staged or committed.

**Staged**: Modified files have been marked to be included in the next commit.

**Committed**: Changes have been safely stored in the local repository.

This workflow allows for a flexible and controlled approach to managing changes in your project.

# Git Workflow Diagram



# Git Installation

How to install Git?

https://github.com/git-guides/install-git

# Git Setup

Configure user name and email using:

```
$ git config --global user.name "Your Name"

$ git config --global user.email "you@example.com"
```

# Git Setup

Creating a Repository:

– Open a terminal and create a new directory, then initialize Git:

```
$ mkdir <repo_name>

$ cd <repo_name>

$ git init
```

# Git Setup

Basic Commands:

– Create a file (e.g., hello.txt), add text, and view status:

```
$ echo "Hello, Git!" > hello.txt
$ git status
```

– Stage the file and commit:

```
$ git add hello.txt
$ git commit -m "Initial commit: add hello.txt with greeting"
```

– Show commit history with:

```
$ git log --oneline --graph
```

# Publishing Local Git Repo to Github

STEP 1: **Create a New Repository on GitHub**

1. Go to GitHub
2. Click on the "+" icon in the top-right corner and select New repository.
3. Give your repository a name (**same name as you did on the local**), set it to public or private, and DO NOT initialize it with a README (since you already have a local repo).
4. Click Create repository.
5. GitHub will provide you with a URL (something like https://github.com/your-username/repository-name.git). Keep this handy.

# Publishing Local Git Repo to Github

**STEP 2: Link Your Local Repository to GitHub**

1. In your terminal or command prompt, navigate to your local project folder where Git is already initialized:

   ```
   $ cd /path/to/your/project
   ```

2. Now, add GitHub as the remote origin:

   ```
   $ git remote add origin https://github.com/your-username/repository-name.git
   ```

3. Check if the remote was added successfully:

   ```
   $ git remote -v
   ```

# Publishing Local Git Repo to Github

**STEP 2: Push Your Local Code to GitHub**

1. Run the following commands:
   a. `$ git branch -M main` # Ensure your branch is named 'main' (or 'master' if you're using an older setup)
   b. `$ git add .` # Stage all files
   c. `$ git commit -m "Initial commit"` # Commit the files
   d. `$ git push -u origin main` # Push your code to GitHub

   If prompted, enter your GitHub credentials or use a Personal Access Token (PAT) if GitHub has disabled password authentication.

1. Go back to your GitHub repository page and refresh it. Your code should now be there!
2. Set up SSH authentication to avoid entering credentials every time.

# Git Cheat Sheet

## Create a Repository

From scratch -- Create a new local repository
```
$ git init [project name]
```

Download from an existing repository
```
$ git clone my_url
```

## Observe your Repository

List new or modified files not yet committed
```
$ git status
```

Show the changes to files not yet staged
```
$ git diff
```

Show the changes to staged files
```
$ git diff --cached
```

Show all staged and unstaged file changes
```
$ git diff HEAD
```

Show the changes between two commit ids
```
$ git diff commit1 commit2
```

List the change dates and authors for a file
```
$ git blame [file]
```

Show the file changes for a commit id and/or file
```
$ git show [commit]:[file]
```

Show full change history
```
$ git log
```

Show change history for file/directory including diffs
```
$ git log -p [file/directory]
```

## Working with Branches

List all local branches
```
$ git branch
```

List all branches, local and remote
```
$ git branch -av
```

Switch to a branch, my_branch, and update working directory
```
$ git checkout my_branch
```

Create a new branch called new_branch
```
$ git branch new_branch
```

Delete the branch called my_branch
```
$ git branch -d my_branch
```

Merge branch_a into branch_b
```
$ git checkout branch_b
$ git merge branch_a
```

Tag the current commit
```
$ git tag my_tag
```

## Make a change

Stages the file, ready for commit
```
$ git add [file]
```

Stage all changed files, ready for commit
```
$ git add .
```

Commit all staged files to versioned history
```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history
```
$git commit -am "commit message"
```

Unstages file, keeping the file changes
```
$ git reset [file]
```

Revert everything to the last commit
```
$ git reset --hard
```

## Synchronize

Get the latest changes from origin (no merge)
```
$ git fetch
```

Fetch the latest changes from origin and merge
```
$ git pull
```

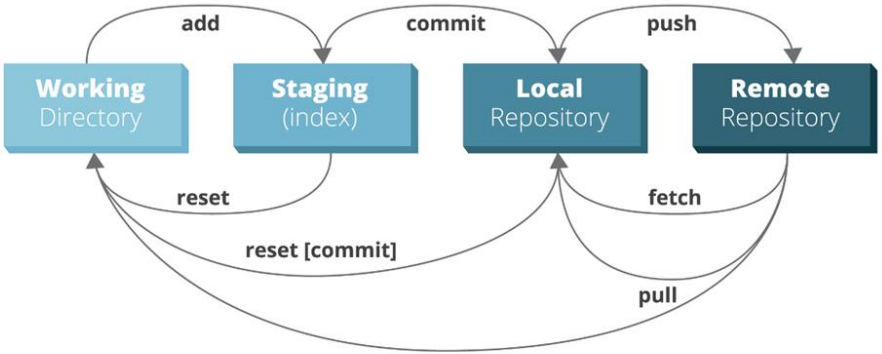
Fetch the latest changes from origin and rebase
```
$ git pull --rebase
```

Push local changes to the origin
```
$ git push
```

## Finally!

When in doubt, use git help
```
$ git command --help
```

Or visit https://training.github.com/ for official GitHub training.



# Module 2: Branching and Merging

# What Are Branches in Git?

A branch in Git is a:

- Lightweight, movable pointer to a commit.
- It allows developers to create an isolated environment to work on:
  - features,
  - bug fixes, or
  - experiments

  without affecting the main codebase.

Branches enable **parallel development** and provide a way to **merge changes** back into the **main project** when they are ready.

**Default Branch**: When a Git repository is initialized, the first branch created is the default branch. Historically, this branch was named `master`, but many platforms like GitHub and GitLab now use `main` as the default for inclusivity and standardization. The default branch typically represents the production-ready version of the codebase

# Create a Branch Locally

You can create a branch using different methods:

**Method 1:** Using git branch & git checkout (Older Way)

```
$ git branch new-branch  # Create a new branch
$ git checkout new-branch # Switch to the new branch
```

**Method 2:** Using git checkout -b (Shortcut)

```
$ git checkout -b new-branch
```

**Method 3:** Using git switch (Recommended for Newer Git Versions)

```
$ git switch -c new-branch
```

After creating the branch, verify it using:

```
$ git branch # This will list all branches, with the current one highlighted
```

# Push the New Branch to GitHub

- Once the branch is created, push it to GitHub:

```
$ git push -u origin new-branch
```

The -u flag sets this branch to track the remote branch, so future git push or git pull commands will work without specifying the branch name.

- Alternative Ways to Push the New Branch

If you've already created and switched to the new branch, you can use:

```
$ git push --set-upstream origin new-branch
```

or simply:

```
$ git push origin new-branch
```

*(but this won't set upstream tracking automatically)*

# Making Changes on a Branch

Modify a file (e.g., update hello.txt with new content) and commit.

# Merging Branches

Merging branches in Git allows you to integrate changes from one branch into another.

There are several ways to merge branches, depending on the needs. Here are the most common methods:

1. Fast-Forward
2. Three-Way Merge
3. Squash Merge
4. Rebase Merge
5. Conflict Resolution

# Fast-Forward

**Use When**: The target branch has no new commits since the branch was created.

**Effect**: Moves the target branch pointer to the new branch without creating a merge commit.

**How to Use**:

```
$ git checkout main    # Switch to the main branch
$ git merge feature-branch # Merge the feature branch into main
```

or using switch:

```
$ git switch main
$ git merge feature-branch
```

No extra commit is created.

# Three-Way Merge (Default)

**Use When**: Both branches have new commits.

**Effect**: Creates a new merge commit.

**How to Use**:

```
$ git checkout main     # Switch to the main branch
$ git merge feature-branch # Merge the feature branch into main
```

# Squash Merge

**Use When**: You want to combine multiple commits into a single commit before merging.

**Effect**: Keeps the history clean by squashing all changes into one commit.

**How to Use**:

```
$ git checkout main    # Switch to the main branch
$ git merge --squash feature-branch # Merge the feature branch into main
$ git commit -m "Merged feature-branch with squashed commits"
```

# Rebase Merge

**Use When**: You want a linear history without merge commits.

**Effect**: Moves the feature branch commits to the latest main branch, avoiding a merge commit.

**How to Use**:

```
$ git checkout feature-branch
$ git rebase main   # Replays commits on top of main
$ git checkout main
$ git merge feature-branch   # Fast-forward merge
```

Or

```
$ git rebase main feature-branch   # Rebase without switching branches
```

# Merge with Conflict Resolution

**Use When**: There are conflicting changes in the same file on both branches.

**Effect**: Git prompts you to manually resolve conflicts before merging.

**How to Use**:

```
$ git checkout main       # Switch to the main branch
$ git merge feature-branch  # Merge the feature branch into main
```

If conflicts occur:

```
$ git status # See conflicting files
$ nano filename # Edit and resolve conflicts manually
$ git add filename
$ git commit -m "Resolved merge conflict"
```

# Which Merge Strategy Should You Use?

| Merge Type | Best For |
| --- | --- |
| **Fast-Forward** | Small updates, keeping history simple |
| **Three-Way Merge** | When multiple branches have diverged |
| **Squash Merge** | Clean history for feature branches |
| **Rebase Merge** | Linear history, avoiding merge commits |
| **Conflict Resolution** | When there are conflicts |

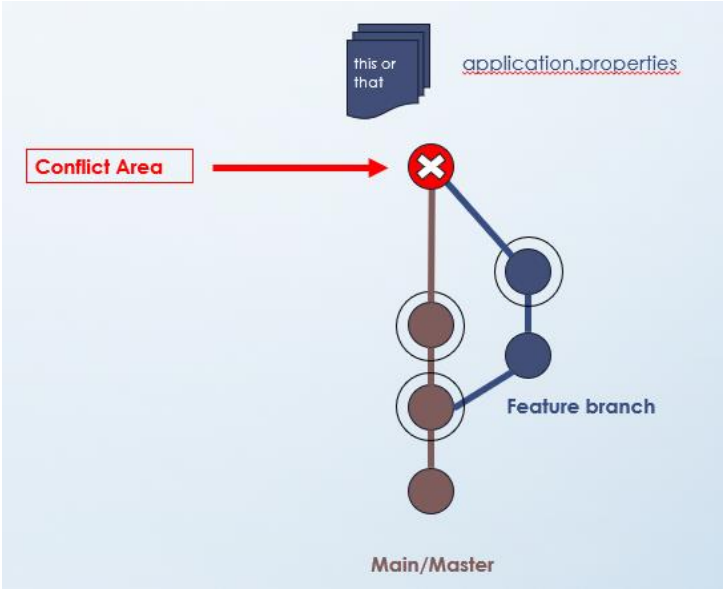
# Visualising Branches

```
$ git log --graph --oneline --all
```

# Module 3: Conflict Resolution

# Objectives

- Learn to identify and resolve conflicts manually.
- Practice using merge tools and understanding Git’s conflict markers.
- Develop best practices to minimize conflicts.



# How Conflicts Occur in Git

Conflicts in Git occur when two branches modify the same lines in a file or when one branch modifies a file that another branch deletes. These situations arise during a merge or rebase operation, as Git cannot automatically determine which changes to keep.

Example of Conflict Scenario:

Branch A (Current Branch) modifies line 10 of file.txt

Branch B (Incoming Branch) also modifies line 10 of file.txt differently.

When merging Branch B into Branch A, Git detects the conflicting changes and pauses the merge process, requiring manual intervention.

# Conflict Markers in Git

When a conflict occurs, Git marks the conflicting sections in the file with conflict markers. These markers highlight the differences between the current branch (HEAD) and the incoming branch.

Example of Conflict Markers

```
<<<<<<< HEAD
Current branch changes
=======
Incoming branch changes
>>>>>>> feature-update
```

**<<<<<<< HEAD** : Marks the start of the changes from your current branch.

**=======** : Separates the changes from your current branch and the incoming branch.

**>>>>>>> [branch-name]** : Marks the end of the incoming branch's changes.

# Resolving Conflicts

To resolve conflicts, you must manually edit the file to decide which changes to keep. You can:

1. Keep one version (either HEAD or incoming branch).
2. Combine both changes into a new version.
3. Remove unnecessary lines.

# Tools to Help Resolve Conflicts

**Command Line**: Use

```
$ git status
```

to identify conflicted files and manually edit them.

**Conflict Editors**: Many IDEs (e.g., VSCode, IntelliJ) provide visual tools for resolving conflicts.

**Git Platforms**: Platforms like GitHub or GitLab offer inline editors for resolving simple conflicts directly on their interfaces

# Module 4: Using Remote Repositories

# Objectives

1. Cloning
2. Synchronizing
3. Configuring Remote Tracking Branches in Git

# How to Clone a Repository from a Remote Server

Cloning a repository creates a local copy of a remote repository, allowing you to work on the project locally.

**Basic Command**:

`$ git clone <repository-url>` # Replace <repository-url> with the HTTPS or SSH URL of the repository (e.g., from GitHub, GitLab, etc.).

**Cloning into a Specific Directory**:

`$ git clone <repository-url> <directory-name>` # This clones the repository into the specified directory.

**Cloning a Specific Branch**:

`$ git clone -b <branch-name> <repository-url>` # Use this to clone only the branch you need.

**Cloning with SSH (requires SSH key setup)**:

`$ git clone git@github.com:user/repository.git` # This method is secure and avoids repeated password prompts.

# Understand How to Synchronize Local and Remote Changes

To ensure your local repository is up-to-date with the remote repository, you need to synchronize changes.

1. Fetch Changes:

   **`$ git fetch`** # This downloads changes from the remote repository but does not merge them into your local branch.

2. Pull Changes:

   **`$ git pull`** # This fetches changes and merges them into your current branch. If there are conflicts, Git will pause for manual resolution.

3. Push Local Changes to Remote:

   **`$ git push`** # This uploads your committed changes from the local branch to the corresponding remote branch.

# Set Up and Configure Remote Tracking Branches

Remote tracking branches link your local branches to their remote counterparts, enabling seamless synchronization.

1. Fetch All Remote Branches:
   **`$ git fetch --all`** # Retrieves metadata for all remote branches without checking them out locally.
2. Create a Local Branch Tracking a Remote Branch:
   **`$ git checkout -b <local-branch-name> origin/<remote-branch-name>`** # This creates a new local branch and sets it to track the specified remote branch automatically.
3. Set an Existing Local Branch to Track a Remote Branch:
   **`$ git branch --set-upstream-to=origin/<remote-branch-name> <local-branch-name>`** # This links an already existing local branch to a remote branch for tracking purposes.
4. Change Tracking for an Existing Branch:
   **`$ git branch -u origin/<new-remote-branch> <local-branch-name>`** # Updates the tracking configuration if you need to point your local branch to a different remote branch.

# Module 5: Strategies for Team Collaboration Using Git

# Objectives

1. Understand different collaborative workflows and select one that fits your team’s needs.
2. Set up a team workflow using branches, pull requests, and CI.
3. Learn techniques for handling complex merge scenarios and large-team challenges.

# Popular Git Workflows for Collaboration

Git workflows define how teams collaborate on repositories, manage branches, and integrate changes. Below are three widely-used Git workflows:

1. Feature Branch Workflow
2. Git Flow Workflow
3. Forking Workflow

# Feature Branch Workflow

Overview: Each new feature or bug fix is developed in its own branch, separate from the **`main`** branch.

**Advantages**:

1. Isolates development work, ensuring the **`main`** branch remains stable.
2. Simplifies code reviews and testing before merging.

**Steps**:

1. Create a new branch for the feature:
   **`$ git checkout -b feature/feature-name main`**
2. Work on the feature and commit changes.
3. Push the branch to the remote repository:
   **`$ git push -u origin feature/feature-name`**
4. Open a pull request (PR) for review and merge into main when approved.

# Git Flow Workflow

**Overview**: A more structured approach with multiple branches:

- develop,
- release,
- Hotfix

  to manage different stages of development.

**Branch Types**:

- main : Production-ready code.
- develop : Integration branch for features.
- feature/* : Individual features branched off ***develop***.
- release/* : Prepares code for release.
- hotfix/* : Urgent fixes branched off **main**.

**Advantages**:

1. Clear separation of stable and experimental code.
2. Ideal for projects with scheduled releases.

**Commands**:

```
$ git checkout -b develop main
$ git checkout -b feature/feature-name develop
```

# Forking Workflow

**Overview**: Each developer forks the central repository into their own remote repository. Changes are made in the fork and submitted as pull requests to the main repository.

**Advantages**:

1. Useful for open-source projects with external contributors.
2. Maintains control over the central repository.

**Steps**:

1. Fork the repository on GitHub/GitLab.
2. Clone your fork locally:

   ```
   $ git clone <forked-repo-url>
   ```

3. Make changes in a new branch, push to your fork, and open a PR.

# Integrating Code Reviews, Continuous Integration (CI), and Automated Testing

Collaboration is enhanced by integrating tools that ensure code quality and streamline development processes.

a) Code Reviews

- Conducted via pull requests (PRs).
- Developers propose changes, and team members review them for quality, consistency, and functionality before merging into main.
- Use comments and suggestions to improve collaboration.

b) Continuous Integration (CI)

- CI automates testing whenever changes are pushed to a repository or merged into key branches like develop or main.
- Tools like GitHub Actions, Jenkins, or Travis CI clone the repository, build the project, and run tests automatically.
- Example CI Workflow with GitHub Actions:

```
name: CI Pipeline
on: [push, pull_request]
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run Tests
        run: npm test
```

- Benefits:
  - Detects issues early in development.
  - Ensures merged code is functional and stable.

c) Automated Testing

- Write unit tests to cover critical parts of your codebase.
- Configure CI tools to run these tests automatically on every commit or PR.
- Example Tools: Jest (JavaScript), PyTest (Python), JUnit (Java).
- Use a build matrix to test across multiple environments (e.g., OS or language versions).

# References:


- GIT-SCM.COM – Git SCM Book – Getting Started
- FREECODECAMP.ORG – freeCodeCamp “What is Git? A Beginner’s Guide”
- EDUCATION.MOLSSI.ORG – MolSSI Education: Introduction to Version Control using Git

# Practice Coding – A LOT

# Any Questions