GIT & GITHUB COMPLETE TRAINING GUIDE

Hands-On Training for Interns

TABLE OF CONTENTS

---

1. INTRODUCTION TO GIT & GITHUB

What is Git?

Git is a distributed version control system that tracks changes in your code over time. It allows multiple developers to work on the same project without overwriting each other's work.

What is GitHub?

GitHub is a cloud-based hosting service for Git repositories. It provides a web interface for Git and adds collaboration features like pull requests, issue tracking, and project management tools.

The Analogy: A Photo Album

Think of Git as managing a photo album:

- **Repository:** The entire photo album
- **Commit:** Each snapshot/photo you add with a caption describing what's in it

- **Branch:** A parallel timeline (like "College Years" vs "Work Life")
- **Merge:** Combining two timelines back together
- **Remote (GitHub):** A backup copy of your album stored in the cloud

Why Use Version Control?

- Track every change made to your code
- Revert to previous versions if something breaks
- Collaborate with team members without conflicts
- Maintain multiple versions of your project simultaneously
- Understand who changed what and when

---

2. GIT BASICS

Installing Git

Windows:
Download from https://git-scm.com/download/win

Mac:

```
brew install git
```

Linux (Ubuntu/Debian):

```
sudo apt-get install git
```

Verify Installation:

```
git --version
```

Initial Configuration

Before using Git, configure your identity:

```
# Set your name
git config --global user.name "Your Name"
```

```
# Set your email
git config --global user.email "your.email@example.com"

# Set default branch name to main
git config --global init.defaultBranch main

# View your configuration
git config --list
```

Creating Your First Repository

Method 1: Create a new repository locally

```
# Create a new directory
mkdir my-project
cd my-project

# Initialize Git repository
git init

# Check status
git status
```

Method 2: Clone an existing repository

```
# Clone from GitHub
git clone https://github.com/username/repository.git

# Clone to a specific folder
git clone https://github.com/username/repository.git my-folder-name
```

The Three States of Git

Git has three main states for your files:

1. **Working Directory:** Where you modify files
2. **Staging Area (Index):** Where you prepare files for commit
3. **Repository (.git directory):** Where Git stores your snapshots

Analogy: Think of it like sending a package:

- **Working Directory:** Items scattered on your desk

- **Staging Area:** Items you've put in the box, ready to ship
- **Repository:** The package after it's been sealed and recorded

Basic Git Workflow

```
# 1. Check status of your files
git status

# 2. Add files to staging area (preparing for commit)
git add filename.txt          # Add specific file
git add .                     # Add all files in current directory
git add *.js                  # Add all JavaScript files
git add folder/               # Add entire folder

# 3. Commit changes (take a snapshot)
git commit -m "Add login feature"

# 4. View commit history
git log
git log --oneline             # Condensed view
git log --graph --all         # Visual branch representation
```

Understanding Git Add

```
# Add specific files
git add file1.txt file2.txt

# Add all modified files
git add -A    # or --all

# Add all files in current directory and subdirectories
git add .

# Interactive staging (choose what to stage)
git add -p    # or --patch
```

Understanding Git Commit

```
# Commit with message
git commit -m "Brief description of changes"
```

```
# Commit with detailed message (opens editor)
git commit

# Add and commit in one step (only for tracked files)
git commit -am "Update user profile"

# Amend the last commit (change message or add forgotten files)
git add forgotten-file.txt
git commit --amend -m "Updated commit message"
```

Viewing Changes

```
# See unstaged changes
git diff

# See staged changes
git diff --staged

# See changes in a specific file
git diff filename.txt

# See changes between commits
git diff commit1 commit2
```

Undoing Changes

```
# Discard changes in working directory (CAREFUL!)
git checkout -- filename.txt

# Unstage a file (keep changes in working directory)
git reset HEAD filename.txt

# Undo last commit, keep changes staged
git reset --soft HEAD~1

# Undo last commit, keep changes unstaged
git reset HEAD~1

# Undo last commit, discard all changes (DANGEROUS!)
git reset --hard HEAD~1
```

Ignoring Files (.gitignore)

Create a `.gitignore` file to exclude files from version control:

```
# .gitignore example
node_modules/
*.log
.env
.DS_Store
dist/
build/
*.tmp


# Create .gitignore
echo "node_modules/" > .gitignore
git add .gitignore
git commit -m "Add gitignore file"
```

---

3. BRANCHING STRATEGIES

What is a Branch?

A branch is an independent line of development. Think of it as a parallel universe where you can experiment without affecting the main timeline.

Analogy: Imagine writing a book with multiple endings. Each ending is a branch, and you can work on different endings simultaneously without them interfering with each other.

Why Use Branches?

- Develop features independently
- Fix bugs without affecting production code
- Experiment safely
- Enable parallel development by multiple team members

Basic Branch Commands

```
# List all branches
git branch
git branch -a          # Include remote branches

# Create a new branch
git branch feature-login

# Switch to a branch
git checkout feature-login

# Create and switch in one command
git checkout -b feature-payment

# Modern alternative (Git 2.23+)
git switch feature-payment
git switch -c new-feature    # Create and switch

# Rename current branch
git branch -m new-name

# Delete a branch
git branch -d feature-login        # Safe delete (only if merged)
git branch -D feature-login        # Force delete
```

Branch Workflow Example

```
# Start from main branch
git checkout main

# Create feature branch
git checkout -b feature-user-authentication

# Make changes and commit
git add .
git commit -m "Add user login form"

# Make more commits
git add .
git commit -m "Add password validation"

# Switch back to main
git checkout main
```

```
# Your feature branch still exists with your changes
git checkout feature-user-authentication
```

Viewing Branch History

```
# See branch structure visually
git log --graph --oneline --all

# See which branches contain a specific commit
git branch --contains commit-hash

# See the last commit on each branch
git branch -v
```

Common Branching Strategies

1. Git Flow (Traditional)

- `main` : Production-ready code
- `develop` : Integration branch for features
- `feature/*` : New features
- `release/*` : Preparing for production release
- `hotfix/*` : Emergency fixes for production

2. GitHub Flow (Simpler)

- `main` : Always deployable
- `feature/*` : Any new work

3. Trunk-Based Development

- `main` : Everyone commits here frequently
- Short-lived feature branches (1-2 days max)

---

4. MERGING & RESOLVING CONFLICTS

What is Merging?

Merging combines changes from different branches. It's like weaving two stories together into one timeline.

Types of Merges

1. Fast-Forward Merge
When the target branch hasn't changed since the feature branch was created.

```
# On main branch
git checkout main
git merge feature-branch

# Result: main simply moves forward to feature-branch's latest commit
```

2. Three-Way Merge
When both branches have new commits since they diverged.

```
git checkout main
git merge feature-branch

# Git creates a new merge commit combining both histories
```

3. Squash Merge
Combines all feature commits into a single commit.

```
git checkout main
git merge --squash feature-branch
git commit -m "Add entire feature"
```

Basic Merge Workflow

```
# 1. Ensure you're on the target branch
git checkout main

# 2. Update your branch
git pull origin main

# 3. Merge the feature branch
git merge feature-login
```

```
# 4. Push to remote
git push origin main
```

Understanding Merge Conflicts

A merge conflict occurs when Git can't automatically resolve differences between two commits.

Common Conflict Scenarios:

- Two people edit the same line in a file
- One person deletes a file while another edits it
- Two people add files with the same name

Conflict Markers:

```
<<<<<<< HEAD
Your current branch's code
=======
The incoming branch's code
>>>>>>> feature-branch
```

Resolving Merge Conflicts

```
# 1. Attempt to merge
git merge feature-branch

# If conflicts occur, Git will tell you which files have conflicts

# 2. Check which files have conflicts
git status

# 3. Open the conflicted files and look for conflict markers

# 4. Edit the file to resolve conflicts:
# - Keep your changes
# - Keep their changes
# - Combine both changes
# - Write something entirely new

# 5. Remove conflict markers

# 6. Stage the resolved files
git add resolved-file.txt
```

```
# 7. Complete the merge
git commit -m "Merge feature-branch and resolve conflicts"
```

Practical Conflict Resolution Example

Before (conflicted file):

```
<<<<<<< HEAD
function login(username, password) {
    return authenticateUser(username, password);
}
=======
function login(email, password) {
    return authenticateUserByEmail(email, password);
}
>>>>>>> feature-email-login
```

After (resolved):

```
function login(identifier, password) {
    // Support both username and email
    if (identifier.includes('@')) {
        return authenticateUserByEmail(identifier, password);
    }
    return authenticateUser(identifier, password);
}
```

Aborting a Merge

```
# If you want to cancel the merge
git merge --abort

# Check status to confirm
git status
```

Merge Tools

```
# Configure a merge tool
git config --global merge.tool vimdiff
```

```
# Use the merge tool for conflicts
git mergetool
```

---

5. COLLABORATION WORKFLOWS

Connecting to GitHub

```
# Add a remote repository
git remote add origin https://github.com/username/repository.git

# View remotes
git remote -v

# Change remote URL
git remote set-url origin https://github.com/username/new-repo.git

# Remove a remote
git remote remove origin
```

Push and Pull

```
# Push your commits to GitHub
git push origin main

# Push and set upstream branch
git push -u origin main

# Push all branches
git push --all

# Pull changes from GitHub (fetch + merge)
git pull origin main

# Pull with rebase instead of merge
git pull --rebase origin main

# Fetch without merging
git fetch origin
```

Authentication Methods

1. HTTPS with Personal Access Token:

```
git clone https://github.com/username/repo.git
# Username: your-username
# Password: your-personal-access-token
```

2. SSH (Recommended):

```
# Generate SSH key
ssh-keygen -t ed25519 -C "your.email@example.com"

# Copy public key to clipboard
cat ~/.ssh/id_ed25519.pub

# Add to GitHub: Settings > SSH and GPG keys > New SSH key

# Clone with SSH
git clone git@github.com:username/repo.git
```

Fork and Pull Request Workflow

Step 1: Fork the repository on GitHub
(Click "Fork" button on the repository page)

Step 2: Clone your fork

```
git clone https://github.com/YOUR-USERNAME/repository.git
cd repository
```

Step 3: Add upstream remote

```
git remote add upstream https://github.com/ORIGINAL-OWNER/repository.git
git remote -v
```

Step 4: Create a feature branch

```
git checkout -b feature-awesome-feature
```

Step 5: Make changes and commit

```
git add .
git commit -m "Add awesome feature"
```

Step 6: Push to your fork

```
git push origin feature-awesome-feature
```

Step 7: Create Pull Request on GitHub

(Go to your fork on GitHub and click "Compare & pull request")

Step 8: Sync your fork with upstream

```
git checkout main
git fetch upstream
git merge upstream/main
git push origin main
```

Team Collaboration Best Practices

```
# Always pull before starting work
git checkout main
git pull origin main

# Create feature branch
git checkout -b feature-name

# Regularly sync with main
git checkout main
git pull origin main
git checkout feature-name
git merge main

# Push your branch frequently
git push origin feature-name

# After PR is merged, clean up
git checkout main
git pull origin main
```

```
git branch -d feature-name
git push origin --delete feature-name
```

Working with Multiple Remotes

```
# Add multiple remotes
git remote add origin https://github.com/you/repo.git
git remote add upstream https://github.com/original/repo.git
git remote add colleague https://github.com/colleague/repo.git

# Fetch from specific remote
git fetch upstream

# Pull from specific remote
git pull upstream main

# Push to specific remote
git push origin feature-branch
```

---

6. GITHUB FEATURES

Pull Requests (PRs)

Purpose: Propose changes, review code, and discuss before merging.

Creating a Pull Request:

1. Push your branch to GitHub
2. Navigate to the repository on GitHub
3. Click "Compare & pull request"
4. Add title and description
5. Select reviewers
6. Create pull request

PR Best Practices:

- Write clear, descriptive titles
- Explain what changes were made and why
- Reference related issues (#123)
- Keep PRs small and focused

- Respond to feedback promptly

Command Line PR (using GitHub CLI):

```
# Install GitHub CLI
# https://cli.github.com/

# Create PR
gh pr create --title "Add user authentication" --body "Implements login a

# List PRs
gh pr list

# View PR
gh pr view 42

# Checkout a PR locally
gh pr checkout 42

# Merge PR
gh pr merge 42
```

Issues

Creating Issues:
Issues track bugs, feature requests, and tasks.

Using Issues Effectively:

```
# Reference issues in commits
git commit -m "Fix login bug - closes #123"
git commit -m "Work on #45 - Add password reset"

# Keywords that close issues:
# close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved
```

GitHub Actions (CI/CD)

Example: Simple Test Workflow

Create `.github/workflows/test.yml` :

```
name: Run Tests

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Node.js
      uses: actions/setup-node@v2
      with:
        node-version: '16'
    - run: npm install
    - run: npm test
```

GitHub Pages

```
# Create gh-pages branch
git checkout --orphan gh-pages

# Add your website files
git add .
git commit -m "Initial GitHub Pages commit"

# Push to GitHub
git push origin gh-pages

# Your site will be at: https://username.github.io/repository
```

GitHub Collaboration Features

1. Code Review:

- Comment on specific lines
- Suggest changes

- Request changes before approval

2. Project Boards:

- Kanban-style task management
- Organize issues and PRs

3. Wiki:

- Documentation for your project

4. Releases:

```
# Create a tag
git tag -a v1.0.0 -m "Release version 1.0.0"
git push origin v1.0.0

# Create release on GitHub from this tag
```

5. Protected Branches:
Settings > Branches > Add rule

- Require PR reviews before merging
- Require status checks to pass
- Require signed commits

---

7. ADVANCED GIT TOPICS

Rebasing

What is Rebasing?
Rebasing moves or combines commits to create a linear history.

Analogy: Instead of documenting where two timelines merged, rewriting history so your changes appear to have always come after the main branch's changes.

```
# Rebase your feature branch onto main
git checkout feature-branch
git rebase main

# Interactive rebase (edit commits)
git rebase -i HEAD~3
```

```
# Continue after resolving conflicts
git rebase --continue

# Abort rebase
git rebase --abort
```

Rebase vs Merge:

- **Merge:** Preserves complete history, creates merge commits
- **Rebase:** Creates linear history, rewrites commit history

GOLDEN RULE: Never rebase commits that have been pushed to a shared repository!

Interactive Rebase

```
# Rebase last 3 commits interactively
git rebase -i HEAD~3

# In the editor, you can:
# - pick: keep commit as-is
# - reword: change commit message
# - edit: amend commit
# - squash: combine with previous commit
# - fixup: like squash but discard commit message
# - drop: remove commit
```

Cherry-Picking

What is Cherry-Picking?
Applying specific commits from one branch to another.

```
# Apply a specific commit to current branch
git cherry-pick commit-hash

# Cherry-pick multiple commits
git cherry-pick commit1 commit2 commit3

# Cherry-pick without committing (stage changes only)
git cherry-pick -n commit-hash
```

Stashing

What is Stashing?

Temporarily saving uncommitted changes without committing them.

```
# Stash current changes
git stash

# Stash with a message
git stash save "Work in progress on login feature"

# List stashes
git stash list

# Apply most recent stash
git stash apply

# Apply and remove stash
git stash pop

# Apply specific stash
git stash apply stash@{2}

# Delete a stash
git stash drop stash@{0}

# Clear all stashes
git stash clear

# Create a branch from stash
git stash branch new-branch-name
```

Git Tags

What are Tags?

Tags mark specific points in history, typically for releases.

```
# Create lightweight tag
git tag v1.0.0

# Create annotated tag (recommended)
git tag -a v1.0.0 -m "Version 1.0.0 release"
```

```
# List tags
git tag
git tag -l "v1.*"

# Show tag information
git show v1.0.0

# Push tags to remote
git push origin v1.0.0
git push origin --tags

# Delete tag
git tag -d v1.0.0
git push origin --delete v1.0.0

# Checkout a tag
git checkout v1.0.0
```

Git Reflog

What is Reflog?
A record of all changes to HEAD, helping you recover "lost" commits.

```
# View reflog
git reflog

# Recover a deleted branch or commit
git checkout commit-hash
git checkout -b recovered-branch

# Reset to a previous state
git reset --hard HEAD@{2}
```

Git Bisect

What is Bisect?
Binary search through commits to find when a bug was introduced.

```
# Start bisect
git bisect start
```

```
# Mark current commit as bad
git bisect bad

# Mark a known good commit
git bisect good commit-hash

# Git checks out a middle commit - test it
# Then mark it as good or bad
git bisect good   # or
git bisect bad

# Continue until bug is found

# End bisect session
git bisect reset
```

Git Submodules

What are Submodules?
Include other Git repositories within your repository.

```
# Add a submodule
git submodule add https://github.com/user/library.git libs/library

# Clone repository with submodules
git clone --recurse-submodules https://github.com/user/repo.git

# Initialize and update submodules after cloning
git submodule init
git submodule update

# Update submodules to latest
git submodule update --remote

# Remove a submodule
git submodule deinit libs/library
git rm libs/library
```

Git Hooks

What are Hooks?
Scripts that run automatically at certain Git events.

Location: `.git/hooks/`

Common Hooks:

- `pre-commit` : Run before commit (e.g., linting, tests)
- `commit-msg` : Validate commit message
- `pre-push` : Run before push
- `post-merge` : Run after merge

Example pre-commit hook:

```
#!/bin/sh
# .git/hooks/pre-commit

npm test
if [ $? -ne 0 ]; then
    echo "Tests failed! Commit aborted."
    exit 1
fi
```

Advanced Git Configuration

```
# Set up aliases
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual 'log --graph --oneline --all'

# Now you can use:
git co main
git st
git visual

# Configure default editor
git config --global core.editor "code --wait"
```

```
# Set up diff tool
git config --global diff.tool vscode
git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOT

# Enable auto-correct
git config --global help.autocorrect 1

# Color output
git config --global color.ui auto
```

---

8. BEST PRACTICES

Commit Message Guidelines

Good commit message structure:

```
Type: Brief summary (50 characters or less)

More detailed explanation if needed (wrap at 72 characters).
Explain what changed and why, not how.

- Bullet points are okay
- Use imperative mood: "Add feature" not "Added feature"

Closes #123
```

Commit Types:

- `feat` : New feature
- `fix` : Bug fix
- `docs` : Documentation changes
- `style` : Code style changes (formatting)
- `refactor` : Code refactoring
- `test` : Adding tests
- `chore` : Maintenance tasks

Examples:

```
git commit -m "feat: Add user password reset functionality"
git commit -m "fix: Resolve null pointer exception in login"
```

```
git commit -m "docs: Update API documentation for v2.0"
git commit -m "refactor: Simplify user authentication logic"
```

Branch Naming Conventions

```
feature/description        # New features
bugfix/description         # Bug fixes
hotfix/description         # Urgent production fixes
release/version           # Release preparation
docs/description          # Documentation updates
refactor/description      # Code refactoring

Examples:
feature/user-authentication
bugfix/login-error-handling
hotfix/critical-security-patch
release/v1.2.0
```

When to Commit

Do commit when:

- You complete a logical unit of work
- You fix a bug
- You add a feature
- Before switching tasks
- Before trying something risky
- At the end of your workday

Don't commit:

- Broken code (unless it's a WIP branch)
- Commented-out code
- Debugging code (console.logs, etc.)
- Generated files or dependencies

Pull Request Best Practices

1. **Keep PRs small** (under 400 lines if possible)
2. **Write descriptive titles and descriptions**
3. **Reference related issues**
4. **Ensure tests pass**

5. **Request specific reviewers**
6. **Respond to feedback constructively**
7. **Update your branch if main has moved forward**
8. **Delete your branch after merging**

Code Review Guidelines

As a Reviewer:

- Be constructive and respectful
- Explain why changes are needed
- Praise good code
- Focus on the code, not the person
- Ask questions instead of making demands

As an Author:

- Don't take criticism personally
- Respond to all comments
- Ask for clarification when needed
- Thank reviewers for their time

Security Best Practices

```
# Never commit sensitive data
# Add to .gitignore:
.env
*.pem
*.key
config/secrets.yml
credentials.json

# If you accidentally commit sensitive data:
# 1. Remove from current commit
git rm --cached sensitive-file.txt
git commit --amend

# 2. If already pushed, consider history rewriting
# (Use tools like BFG Repo-Cleaner or git-filter-branch)

# 3. Rotate any exposed credentials immediately!

# Use environment variables
```

```
# Create .env file (add to .gitignore)
API_KEY=your_api_key_here

# Use git-secrets tool to prevent accidental commits
git secrets --install
git secrets --register-aws
```

---

9. COMMON ISSUES & SOLUTIONS

Problem: Accidentally committed to wrong branch

```
# Solution 1: Move commits to new branch
git branch new-branch-name
git reset --hard HEAD~3  # Remove last 3 commits from current branch
git checkout new-branch-name

# Solution 2: Cherry-pick to correct branch
git checkout correct-branch
git cherry-pick commit-hash
git checkout wrong-branch
git reset --hard HEAD~1
```

Problem: Need to undo last commit

```
# Keep changes, uncommit
git reset --soft HEAD~1

# Discard changes entirely
git reset --hard HEAD~1

# Already pushed? Create revert commit
git revert HEAD
```

Problem: Merge conflict seems overwhelming

```
# Abort and start over
git merge --abort
```

```
# Or rebase instead
git checkout feature-branch
git rebase main

# Use a merge tool
git mergetool
```

Problem: Accidentally deleted a branch

```
# Find the commit where branch was
git reflog

# Recreate branch
git checkout -b recovered-branch commit-hash
```

Problem: Need to remove a file from Git but keep locally

```
# Remove from Git but keep in working directory
git rm --cached filename.txt
git commit -m "Remove file from tracking"

# Add to .gitignore to prevent re-adding
echo "filename.txt" >> .gitignore
```

Problem: Committed large file by mistake

```
# Remove from last commit
git rm --cached large-file.zip
git commit --amend -m "Remove large file"

# If already pushed, use BFG Repo-Cleaner
# Download from: https://rtyley.github.io/bfg-repo-cleaner/

java -jar bfg.jar --strip-blobs-bigger-than 100M repo.git
cd repo.git
git reflog expire --expire=now --all
git gc --prune=now --aggressive
```

Problem: Pull request has conflicts

```
# Update your branch with latest main
git checkout feature-branch
git fetch origin
git merge origin/main

# Resolve conflicts
# ... edit files ...
git add .
git commit -m "Resolve merge conflicts"
git push origin feature-branch
```

Problem: Wrong commit message

```
# Last commit, not yet pushed
git commit --amend -m "Correct message"

# Last commit, already pushed
git commit --amend -m "Correct message"
git push --force-with-lease

# Older commit
git rebase -i HEAD~3
# Change 'pick' to 'reword' for the commit
```

Problem: Multiple small commits need to be combined

```
# Interactive rebase
git rebase -i HEAD~5

# Change 'pick' to 'squash' (or 's') for commits to combine
# Save and edit the combined commit message
```

Problem: Pushed sensitive data

```
# URGENT STEPS:
# 1. Remove from repository
```

```
git rm --cached sensitive-file
git commit -m "Remove sensitive file"
git push

# 2. Rewrite history (if needed)
git filter-branch --tree-filter 'rm -f sensitive-file' HEAD
git push --force

# 3. IMMEDIATELY rotate/change any exposed credentials!

# 4. Consider making repository private temporarily
```

HANDS-ON EXERCISES

Exercise 1: Basic Git Workflow

```
# 1. Create a new directory and initialize Git
mkdir git-practice
cd git-practice
git init

# 2. Create a file
echo "# My Project" > README.md

# 3. Check status
git status

# 4. Stage and commit
git add README.md
git commit -m "Initial commit: Add README"

# 5. Make changes
echo "This is my practice project" >> README.md

# 6. View changes
git diff

# 7. Stage and commit
git add README.md
git commit -m "Update README with description"
```

```
# 8. View history
git log
git log --oneline
```

Exercise 2: Branching and Merging

```
# 1. Create and switch to new branch
git checkout -b feature-about

# 2. Create new file
echo "# About" > about.md
echo "This project helps me learn Git" >> about.md

# 3. Commit changes
git add about.md
git commit -m "Add about page"

# 4. Switch back to main
git checkout main

# 5. Merge feature branch
git merge feature-about

# 6. View history
git log --graph --oneline --all

# 7. Delete feature branch
git branch -d feature-about
```

Exercise 3: Handling Conflicts

```
# 1. Create two branches
git checkout -b branch-a
echo "Line from Branch A" > conflict.txt
git add conflict.txt
git commit -m "Add line from Branch A"

git checkout main
git checkout -b branch-b
echo "Line from Branch B" > conflict.txt
git add conflict.txt
```

```
git commit -m "Add line from Branch B"

# 2. Try to merge
git checkout main
git merge branch-a  # This works
git merge branch-b  # This causes conflict!

# 3. Resolve conflict
# Edit conflict.txt and resolve
git add conflict.txt
git commit -m "Merge branch-b and resolve conflicts"
```

Exercise 4: GitHub Collaboration

```
# 1. Create repository on GitHub

# 2. Connect local repo to GitHub
git remote add origin https://github.com/yourusername/git-practice.git

# 3. Push to GitHub
git push -u origin main

# 4. Create feature branch
git checkout -b feature-contact
echo "# Contact" > contact.md
git add contact.md
git commit -m "Add contact page"

# 5. Push branch to GitHub
git push origin feature-contact

# 6. Create Pull Request on GitHub

# 7. After PR is merged, update local repo
git checkout main
git pull origin main
git branch -d feature-contact
```

QUICK REFERENCE CHEAT SHEET

Setup

```
git config --global user.name "Your Name"
git config --global user.email "email@example.com"
git init
git clone <url>
```

Basic Commands

```
git status
git add <file>
git add .
git commit -m "message"
git commit -am "message"
git log
git log --oneline
git diff
```

Branching

```
git branch
git branch <name>
git checkout <branch>
git checkout -b <branch>
git merge <branch>
git branch -d <branch>
```

Remote

```
git remote add origin
```