03 Git and GitHub Lecture Notes

Comprehensive Git and GitHub Lecture Notes

---

1. Introduction to Version Control

What is Version Control?

Version control is a system that records changes to files over time, allowing you to:

- Track modifications to your code
- Revert to previous versions
- Compare changes over time
- Collaborate with multiple developers
- Maintain different versions simultaneously

Types of Version Control Systems

1. **Local Version Control**: Files stored locally with version history
2. **Centralized Version Control**: Single server contains all versions (SVN, CVS)
3. **Distributed Version Control**: Every clone is a full backup (Git, Mercurial)

---

2. Git Fundamentals

What is Git?

Git is a distributed version control system that allows multiple developers to work on a project simultaneously while tracking changes efficiently. Created by Linus Torvalds in 2005 for Linux kernel development.

Why Git is Popular

1. **Distributed Architecture**: Complete repository copy on every machine
2. **Speed and Efficiency**: Optimized for performance with large repositories
3. **Powerful Branching**: Lightweight branches and advanced merging
4. **Collaboration Support**: Efficient multi-developer workflows
5. **Large Community**: Extensive ecosystem and tool support
6. **Flexibility**: Adaptable to various development workflows
7. **Stability**: Proven reliability in large-scale projects

Git Architecture

- **Working Directory**: Your local files
- **Staging Area (Index)**: Files prepared for commit
- **Local Repository**: Your local Git database
- **Remote Repository**: Shared repository on server

File States in Git

1. **Modified**: File changed but not staged
2. **Staged**: File marked for next commit
3. **Committed**: File safely stored in database

---

3. Essential Git Commands

Repository Initialization and Cloning

Initialize a New Repository

```
git init
```

Creates a new Git repository in current directory.

Clone an Existing Repository

```
git clone <repository-url>
git clone <repository-url> <directory-name>
```

Basic File Operations

Check Repository Status

```
git status
git status --short  # Abbreviated output
```

Add Files to Staging Area

```
git add <filename>           # Add specific file
git add .                    # Add all files in current directory
git add -A                   # Add all files in repository
git add *.js                 # Add all JavaScript files
git add --interactive        # Interactive adding
```

Commit Changes

```
git commit -m "Commit message"
git commit -am "Message"     # Add and commit tracked files
git commit --amend           # Modify last commit
git commit --amend -m "New message"  # Change last commit message
```

Viewing History and Changes

View Commit History

```
git log
git log --oneline            # Condensed view
git log --graph              # Visual representation
git log --stat               # Show file statistics
git log -p                   # Show patches
git log -n 5                 # Show last 5 commits
git log --since="2024-01-01" # Commits since date
git log --author="John Doe"  # Commits by author
git log --grep="bug"         # Search commit messages
```

View Changes

```
git diff                     # Unstaged changes
git diff --staged            # Staged changes
git diff HEAD~1              # Compare with previous commit
git diff branch1 branch2     # Compare branches
git diff --name-only         # Show only file names
```

Show Specific Commit

```
git show <commit-hash>
git show HEAD~2              # Show 2 commits back
```

4. Branching and Merging

Branch Management

List Branches

```
git branch                   # Local branches
git branch -r                # Remote branches
git branch -a                # All branches
git branch -v                # Verbose output
```

Create and Switch Branches

```
git branch <branch-name>     # Create branch
git checkout <branch-name>   # Switch to branch
git checkout -b <branch-name> # Create and switch
git switch <branch-name>     # Modern way to switch
git switch -c <branch-name>  # Create and switch (modern)
```

Rename Branches

```
git branch -m <old-name> <new-name>  # Rename branch
git branch -m <new-name>             # Rename current branch
```

Delete Branches

```
git branch -d <branch-name>     # Delete merged branch
git branch -D <branch-name>     # Force delete branch
git push origin --delete <branch-name>  # Delete remote branch
```

Merging

Basic Merge

```
git checkout main
git merge <feature-branch>
```

Merge Strategies

```
git merge --no-ff <branch>      # No fast-forward merge
git merge --squash <branch>     # Squash commits
git merge -X theirs <branch>    # Prefer their changes in conflicts
git merge -X ours <branch>      # Prefer our changes in conflicts
```

Rebasing

Basic Rebase

```
git checkout feature-branch
git rebase main
```

Interactive Rebase

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

Interactive rebase actions:

- `pick` : Keep commit as-is
- `reword` : Change commit message
- `edit` : Modify commit
- `squash` : Combine with previous commit
- `drop` : Remove commit

Difference Between Merge and Rebase

Merge:

- Preserves commit history
- Creates a merge commit
- Shows true development timeline
- Safer for shared branches

Rebase:

- Creates linear history
- No merge commits
- Cleaner history
- Rewrites commit history (dangerous for shared branches)

---

5. Working with Remote Repositories

Remote Management

View Remotes

```
git remote                    # List remote names
git remote -v                 # Show URLs
git remote show origin        # Detailed remote info
```

Add/Remove Remotes

```
git remote add <name> <url>
git remote remove <name>
git remote rename <old> <new>
```

Synchronizing with Remotes

Fetch Changes

```
git fetch                     # Fetch from all remotes
git fetch origin              # Fetch from specific remote
git fetch origin main         # Fetch specific branch
```

Pull Changes

```
git pull                    # Fetch and merge
git pull origin main        # Pull specific branch
git pull --rebase           # Fetch and rebase
git pull --no-ff            # No fast-forward merge
```

Push Changes

```
git push                    # Push current branch
git push origin main        # Push to specific branch
git push -u origin main     # Set upstream and push
git push --force            # Force push (dangerous)
git push --force-with-lease # Safer force push
git push --tags             # Push tags
```

Tracking Branches

Set Upstream Branch

```
git branch --set-upstream-to=origin/main main
git push -u origin main     # Set upstream while pushing
```

---

6. Git Stash

What is Git Stash?

Git stash temporarily saves your uncommitted changes, allowing you to switch branches or pull updates without committing incomplete work.

Basic Stash Operations

Create a Stash

```
git stash                   # Stash tracked files
git stash save "Work in progress"  # Stash with message
git stash -u                # Include untracked files
git stash -a                # Include all files (even ignored)
git stash push -m "Message" # Modern syntax with message
```

List Stashes

```
git stash list
# Output: stash@{0}: WIP on main: 1234567 Last commit
```

Apply Stashes

```
git stash apply             # Apply most recent stash
git stash apply stash@{1}   # Apply specific stash
git stash pop               # Apply and remove stash
git stash pop stash@{1}     # Apply specific stash and remove
```

View Stash Contents

```
git stash show              # Show files in stash
git stash show -p           # Show full diff
git stash show stash@{1}    # Show specific stash
```

Remove Stashes

```
git stash drop              # Remove most recent stash
git stash drop stash@{1}    # Remove specific stash
git stash clear             # Remove all stashes
```

Advanced Stash Operations

```
git stash branch <branch-name>  # Create branch from stash
git stash push -- <file>        # Stash specific files
git stash push -p               # Interactive stashing
```

Stash Use Cases

1. **Quick Branch Switching:** Save work before switching branches
2. **Emergency Fixes:** Stash feature work to fix urgent bugs
3. **Pulling Updates:** Clean working directory before pull
4. **Experimenting:** Save current work before trying different approaches

---

7. Advanced Git Operations

Undoing Changes

Unstage Files

```
git reset HEAD <file>       # Unstage specific file
git reset                   # Unstage all files
```

Undo Commits

```
git reset --soft HEAD~1     # Undo commit, keep changes staged
git reset --mixed HEAD~1    # Undo commit, unstage changes
git reset --hard HEAD~1     # Undo commit, discard changes
```

Revert Commits

```
git revert <commit-hash>    # Create commit that undoes changes
git revert HEAD~2           # Revert commit 2 steps back
git revert -n <commit>      # Revert without committing
```

Restore Files

```
git restore <file>          # Restore working directory file
git restore --staged <file>  # Unstage file
git restore --source=HEAD~1 <file>  # Restore from specific commit
```

Tagging

Create Tags

```
git tag <tag-name>          # Lightweight tag
git tag -a <tag-name> -m "Message"  # Annotated tag
```

```
git tag -a <tag-name> <commit-hash>  # Tag specific commit
```

List and Show Tags

```
git tag                      # List all tags
git tag -l "v1.*"            # List tags matching pattern
git show <tag-name>          # Show tag details
```

Push Tags

```
git push origin <tag-name>   # Push specific tag
git push --tags              # Push all tags
git push origin --tags       # Push tags to specific remote
```

Delete Tags

```
git tag -d <tag-name>        # Delete local tag
git push origin --delete <tag-name>  # Delete remote tag
```

Cherry Picking

```
git cherry-pick <commit-hash>  # Apply specific commit
git cherry-pick -n <commit>    # Apply without committing
git cherry-pick <commit1>..<commit2>  # Apply range of commits
```

Bisect (Finding Bugs)

```
git bisect start
git bisect bad               # Mark current commit as bad
git bisect good <commit>     # Mark known good commit
# Test and mark commits as good/bad
git bisect reset             # End bisect session
```

Reflog

```
git reflog                   # Show reference log
git reflog show HEAD         # Show HEAD movements
git reset --hard HEAD@{2}    # Reset to specific reflog entry
```

8. GitHub Overview

What is GitHub?

GitHub is a web-based hosting service for Git repositories that adds collaboration features, project management tools, and social coding capabilities.

Key Differences: Git vs GitHub

Git:

- Distributed version control system
- Command-line tool
- Local repository management
- Core version control functionality

GitHub:

- Web-based hosting platform

- Built on top of Git
- Remote repository hosting
- Additional collaboration features:
  - Pull requests
  - Issues tracking
  - Project boards
  - Wiki pages
  - Actions (CI/CD)
  - Security scanning
  - Code review tools

GitHub Features

Repository Management

- Public and private repositories
- Repository templates
- Forking capabilities
- Repository insights and analytics

Collaboration Tools

- Pull requests for code review
- Issues for bug tracking and feature requests
- Project boards for task management
- Wiki for documentation
- Discussions for community interaction

Development Tools

- GitHub Actions for CI/CD
- Codespaces for cloud development
- Security advisories and scanning
- Dependency management
- Code search and navigation

---

9. Collaboration Workflows

Fork and Pull Request Workflow

1. **Fork Repository:** Create personal copy
2. **Clone Fork:** Download to local machine
3. **Create Feature Branch:** Work on specific feature
4. **Make Changes:** Implement functionality
5. **Push Changes:** Upload to your fork
6. **Create Pull Request:** Request merge into original repository
7. **Code Review:** Collaborate on improvements
8. **Merge:** Integrate changes

Commands for Fork Workflow

```
# Clone your fork
git clone <your-fork-url>

# Add upstream remote
git remote add upstream <original-repo-url>
```

```
# Create feature branch
git checkout -b feature/new-feature

# Keep fork updated
git fetch upstream
git checkout main
git merge upstream/main
git push origin main
```

Git Flow Workflow

A branching model with specific branch purposes:

- **main/master**: Production-ready code
- **develop**: Integration branch for features
- **feature/***: New features
- **release/***: Release preparation
- **hotfix/***: Critical fixes

```
# Start new feature
git checkout develop
git checkout -b feature/user-authentication

# Finish feature
git checkout develop
git merge feature/user-authentication
git branch -d feature/user-authentication
```

GitHub Flow (Simplified)

1. Create branch from main
2. Make changes
3. Create pull request
4. Review and discuss
5. Merge to main
6. Deploy

---

10. Best Practices

Commit Message Guidelines

Structure

```
<type>(<scope>): <subject>

<body>

<footer>
```

Types

- `feat` : New feature
- `fix` : Bug fix
- `docs` : Documentation changes
- `style` : Formatting changes
- `refactor` : Code refactoring

- `test` : Adding tests
- `chore` : Maintenance tasks

Examples

```
feat(auth): add user login functionality

Implement OAuth2 authentication with Google and GitHub providers.
Includes login form, session management, and user profile display.

Closes #123
```

.gitignore Best Practices

Common Patterns

```
# Dependencies
node_modules/
vendor/

# Build outputs
dist/
build/
*.exe
*.dll

# Environment files
.env
.env.local
config/secrets.yml

# IDE files
.vscode/
.idea/
*.swp

# OS files
.DS_Store
Thumbs.db

# Logs
*.log
logs/

# Temporary files
tmp/
temp/
```

Language-Specific Examples

JavaScript/Node.js:

```
node_modules/
npm-debug.log*
.npm
.env
dist/
coverage/
```

Python:

```
__pycache__/
*.pyc
*.pyo
venv/
.env
*.egg-info/
```

Java:

```
*.class
*.jar
target/
.gradle/
build/
```

Pull Request Guidelines

What to Consider When Reviewing

1. **Code Quality**
   - Clean, readable code
   - Proper indentation and formatting
   - Meaningful variable and function names
   - No redundant or dead code
2. **Functionality**
   - Code works as intended
   - Handles edge cases
   - Doesn't break existing features
   - Includes appropriate error handling
3. **Testing**
   - Adequate test coverage
   - Tests pass
   - Tests are meaningful
4. **Documentation**
   - Code is self-documenting
   - Complex logic is commented
   - API changes are documented
5. **Performance**
   - No obvious performance issues
   - Efficient algorithms used
   - Resource usage considered

---

11. Troubleshooting Common Issues

Merge Conflicts

Identifying Conflicts

```
git status
# Shows "Unmerged paths"
```

Conflict Markers

```
<<<<<<< HEAD
Your changes
=======
Incoming changes
>>>>>>> branch-name
```

Resolving Conflicts

1. Open conflicted files
2. Choose which changes to keep
3. Remove conflict markers
4. Stage resolved files
5. Commit the merge

```
git add <resolved-file>
git commit -m "Resolve merge conflict"
```

Common Problems and Solutions

Undo Last Commit (Not Pushed)

```
git reset --soft HEAD~1  # Keep changes
git reset --hard HEAD~1  # Discard changes
```

Remove File from Repository but Keep Locally

```
git rm --cached <file>
git commit -m "Remove file from tracking"
```

Change Remote URL

```
git remote set-url origin <new-url>
```

Recover Deleted Branch

```
git reflog
git checkout -b <branch-name> <commit-hash>
```

Fix "Detached HEAD" State

```
git checkout main
# Or create new branch from current position
git checkout -b new-branch
```

12. Real-World Scenarios

Scenario 1: Feature Development

Situation: You need to develop a new user authentication feature.

```
# 1. Start from main branch
git checkout main
git pull origin main

# 2. Create feature branch
git checkout -b feature/user-auth

# 3. Work on feature (make commits)
git add .
git commit -m "feat: add login form"
git add .
git commit -m "feat: implement OAuth integration"

# 4. Push feature branch
git push -u origin feature/user-auth

# 5. Create pull request on GitHub

# 6. After approval, merge and cleanup
git checkout main
git pull origin main
git branch -d feature/user-auth
```

Scenario 2: Hotfix for Production Bug

Situation: Critical bug discovered in production.

```
# 1. Create hotfix branch from main
git checkout main
git checkout -b hotfix/critical-security-fix

# 2. Fix the issue quickly
git add .
git commit -m "fix: resolve SQL injection vulnerability"

# 3. Push and create urgent PR
git push -u origin hotfix/critical-security-fix

# 4. After testing and approval, merge immediately
# 5. Deploy to production
# 6. Cleanup
git checkout main
git pull origin main
git branch -d hotfix/critical-security-fix
```

Scenario 3: Collaborating with Team Member's Work

Situation: Need to continue work on colleague's unfinished feature.

```
# 1. Fetch latest changes
git fetch origin

# 2. Check out colleague's branch
git checkout -b feature/colleague-work origin/feature/colleague-work

# 3. Continue development
git add .
git commit -m "feat: complete user profile section"
```

```
# 4. Push changes
git push origin feature/colleague-work
```

Scenario 4: Accidental Commit to Wrong Branch

Situation: Made commits on main instead of feature branch.

```
# 1. Create new branch from current position
git checkout -b feature/correct-branch

# 2. Reset main to previous state
git checkout main
git reset --hard HEAD~2  # Remove last 2 commits

# 3. Continue work on correct branch
git checkout feature/correct-branch
```

Scenario 5: Synchronizing Fork with Upstream

Situation: Your fork is behind the original repository.

```
# 1. Add upstream remote (if not already added)
git remote add upstream <original-repo-url>

# 2. Fetch upstream changes
git fetch upstream

# 3. Update main branch
git checkout main
git merge upstream/main
git push origin main

# 4. Update your feature branch
git checkout feature/my-feature
git rebase main
```

Scenario 6: Code Review Feedback Implementation

Situation: Reviewer requested changes to your pull request.

```
# 1. Make requested changes
git add .
git commit -m "refactor: address code review feedback"

# 2. If reviewer wants commit history cleaned up
git rebase -i HEAD~3  # Interactive rebase to squash commits

# 3. Force push (since history changed)
git push --force-with-lease origin feature/my-feature
```

Scenario 7: Release Management

Situation: Preparing for version 2.1.0 release.

```
# 1. Create release branch
git checkout main
git checkout -b release/2.1.0

# 2. Update version numbers, changelog
git add .
git commit -m "chore: prepare for v2.1.0 release"
```

```
# 3. Final testing and bug fixes
git add .
git commit -m "fix: resolve last-minute issues"

# 4. Tag the release
git tag -a v2.1.0 -m "Version 2.1.0"

# 5. Merge to main and develop
git checkout main
git merge release/2.1.0
git checkout develop
git merge release/2.1.0

# 6. Push everything
git push origin main develop --tags

# 7. Clean up
git branch -d release/2.1.0
```

Conclusion

Git and GitHub are essential tools for modern software development. Understanding these concepts and commands will help you:

- Track changes effectively
- Collaborate seamlessly with team members
- Manage complex projects with multiple contributors
- Maintain clean and organized code history
- Implement various development workflows

Remember that mastery comes with practice. Start with basic commands and gradually incorporate more advanced features as you become comfortable with the fundamentals.

Additional Resources for Learning

- Official Git documentation
- GitHub's learning resources
- Interactive Git tutorials
- Git visualization tools
- Command line practice environments

Quick Reference Card

Essential Daily Commands:

```
git status              # Check current state
git add .               # Stage all changes
git commit -m "message" # Commit with message
git push                # Push to remote
git pull                # Pull from remote
git checkout -b branch  # Create and switch branch
git merge branch        # Merge branch
git stash               # Stash changes
git log --oneline       # View history
```

This comprehensive guide should serve as your go-to reference for Git and GitHub operations, from basic usage to advanced collaboration scenarios.