1. Setting Up

Starting a repository

git init & git clone

Start a new project or copy an existing one.

New Project (`git init`)

Turns the current directory into a Git repository.

```
mkdir my-project
cd my-project
git init
# Creates a hidden .git/ folder
```

Existing Project (`git clone`)

Downloads a repository from a remote server (like GitHub).

```
git clone https://github.com/user/repo.git
# Automatically sets up remote tracking
```

2. The Three States

Understanding how Git tracks files.

The Zones

1. **Working Directory**: Where you edit files.
2. **Staging Area (Index)**: Where you prepare the snapshot.
3. **Repository (HEAD)**: Where changes are permanently stored.

File Status

- **Untracked**: New files Git hasn't seen.
- **Tracked**:
- *Unmodified*: Clean.
- *Modified*: Changed but not staged.
- *Staged*: Ready to commit.

Staging Changes

Moving from "Modified" to "Staged".

Check Status

Always check status before adding!

```
git status
```

Add Files

```
# Stage a specific file
git add filename.txt

# Stage all changes (new, modified, deleted)
git add .

# Interactive staging (Review changes chunk by chunk)
git add -p
```

Configuration

Set your identity before you start committing.

Identity (`git config`)

```
# Set globally (all projects)
git config --global user.name "John Doe"
git config --global user.email "john@example.com"

# Set locally (current project only)
git config user.name "John Doe"
git config user.email "john@example.com"

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

Committing

Saving the snapshot to history.

A commit captures the state of the Staging Area.

```
# Commit with a message
git commit -m "Add login feature"

# Add and Commit in one step (skips untracked files)
git commit -am "Fix typo in header"
```

Best Practice: Write clear, concise commit messages. Use the imperative mood ("Fix bug" not "Fixed bug").

3. Inspecting History

Seeing what happened

git log & git diff

Viewing History (`git log`)

```
# Standard log
git log

# One line summary (cleaner)
git log --oneline --graph --all
```

Viewing Changes (`git diff`)

```
# Diff Working Directory vs Staging (What have I changed but not added?)
git diff

# Diff Staging vs Repository (What am I about to commit?)
git diff --staged
```

4. Undoing Things

"I made a mistake"

HEAD & Reset

What is HEAD?

`HEAD` is a pointer to the current commit you are viewing. Usually, it points to the tip of your current branch.

Git Reset

Move the current branch backward in history.

```
# Soft Reset: Move HEAD back, keep changes staged
git reset --soft HEAD~1

# Mixed Reset (Default): Move HEAD back, keep changes unstaged
git reset HEAD~1

# Hard Reset: Move HEAD back, DESTROY changes (Dangerous!)
git reset --hard HEAD~1
```

Unstaging & Restoring

Unstage Changes

You added a file to staging by mistake, but want to keep the changes in the file.

```
# Removes file from Staging, keeps it in Working Directory
git restore --staged <file>
# OR (older syntax)
git reset HEAD <file>
```

Discard Changes

You messed up a file and want to revert it to the last commit state (Dangerous!).

```
# Discards changes in Working Directory
git restore <file>
# OR (older syntax)
git checkout -- <file>
```

Reverting Commits

Safely undoing history that has already been shared.

`git revert`

Creates a **new commit** that is the exact opposite of an existing one.

```
# Undo the changes introduced by the last commit
git revert HEAD

# Undo a specific commit by hash
git revert <commit-hash>
```

Why? Unlike `git reset`, this doesn't rewrite history, making it safe for public branches.

.gitignore

Essential Topic: Telling Git what *not* to track.

Create a file named `.gitignore` in your root.

```
# .gitignore example

# Ignore node_modules
node_modules/

# Ignore env files with secrets
.env

# Ignore build artifacts
dist/
build/
```

5. Branching

Parallel Development

Branching Basics

Branches allow you to work on features isolated from the main code.

```
git branch                          # List branches
git branch feature-login            # Create a new branch
git switch feature-login            # Switch to a branch
git checkout feature-login          # Switch to a branch (legacy)
git checkout -b feature-login       # Create AND switch
git branch -d feature-login         # Delete a branch (safe)
git branch -D feature-login         # Delete a branch (force - carefully!)
```

6. Remote & Auth

Working with GitHub

Auth with GitHub

Modern GitHub requires **SSH Keys** or **Personal Access Tokens (PAT)**. Password auth is deprecated.

Recommended: SSH

1. Generate key: `ssh-keygen -t ed25519 -C "email@example.com"`
2. Add public key (`id_ed25519.pub`) to GitHub Settings → SSH Keys.
3. Test: `ssh -T git@github.com`

Linking Remote

```
git remote add origin https://github.com/user/repo.git         # Add a remote named 'origin'
git remote -v                                                  # Show remotes
git remote rename origin destination                           # Rename a remote
git remote remove destination                                  # Remove a remote
git remote set-url origin https://github.com/user/new-repo.git # Change remote URL
```

Push, Fetch & Pull

Push

Upload local commits to remote.

```
# First push (set upstream)
git push -u origin main

# Subsequent pushes
git push
```

Fetch vs Pull

- **Fetch**: Downloads data from remote but does **not** change your files.

```
git fetch
```

- **Pull**: Fetches **AND** Merges immediately.

```
git pull
```

7. Integration

Merge vs Rebase

layout: two-cols

Merge vs Rebase

Merge

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

✅ Non-destructive
❌ Messy history

Rebase

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

✅ Clean, linear history
❌ Rewrites history (shared branches)

Conflicts

When Git gets confused.

Occurs when the same line is modified in two different branches.

1. Git pauses the merge/rebase.
2. Open files; look for markers:

```
&lt;&lt;&lt;&lt;&lt; HEAD
Current Change
=======
Incoming Change
&gt;&gt;&gt;&gt;&gt; feature-branch
```

1. Edit file to choose the correct code.
2. **Add** the file.
3. **Commit** (or `git rebase --continue`).

8. Advanced Tools

Stash & Worktrees

Git Stash

"Save it for later."

Useful when you are not ready to commit but need to switch branches.

```
# Save changes to a temporary stack
git stash

# List stashes
git stash list

# Apply the last stash and remove it from stack
git stash pop

# Apply but keep in stack
git stash apply
```

Git Worktrees

Pro Tip: Working on multiple branches simultaneously.

Instead of switching branches (which changes your files), check out a branch into a *separate folder*.

```
# Create a new folder linked to a specific branch
git worktree add ../new-folder-name feature-branch

# List worktrees
git worktree list

# Remove worktree
git worktree remove ../new-folder-name
```

- Useful for fixing a hot-fix bug while in the middle of a massive feature refactor.

Surgical Tools

Precision Git operations.

Cherry Pick

Apply a specific commit from one branch to another without merging the whole branch.

```
git cherry-pick <commit-hash>
```

Bisect

Find the commit that introduced a bug using binary search.

```
git bisect start
git bisect bad            # Current version is broken
git bisect good <commit>  # This old version worked
# Git checks out a middle commit. You test it.
git bisect good/bad       # Tell Git the result
# Repeat until the culprit is found.
git bisect reset
```