Jim's Git Journey

Week 1: Learning Git in 6 Levels

Meet Jim

> *Jim is a software engineer who just started at a tech company. Today is his first day using Git to manage his code. Let's follow Jim's journey as he learns Git commands through real-world scenarios.*

What Jim will learn today:

- Level 1: Starting a new project
- Level 2: Checking out previous code
- Level 3: Fixing mistakes
- Level 4: Organizing with tags and branches
- Level 5: Merging Branches
- Level 6: Advanced workflows

Level 1: Jim's First Day

Monday Morning

> *Jim arrives at his desk. His manager asks him to start working on a new feature for the company's web application. Jim opens his terminal and creates a new project folder called "user-profile".*

Jim's first task: Set up Git to track his work

Jim Initializes His Gallery

> *"I need to create my drawing room first," Jim thinks, remembering the gallery metaphor from training.*

```
mkdir user-profile
cd user-profile
git init
```

What happened:

- Git created a hidden `.git` folder (the **Gallery**)
- Jim is now in the **main** branch (his first **Room**)
- His **Canvas** (Working Directory) is ready!

Jim Creates His First File

> *Jim creates a new file called "profile.js" with basic user profile code.*

```
$ git status
On branch main
Untracked files:
  profile.js
```

Jim's observation:

- The file exists in his **Canvas** (Working Directory)
- But it's like a **yellow Post-it note** — not yet pinned!

Jim Pins His First Note

> *"This code looks good! I should pin it to my board," Jim decides.*

```
$ git add profile.js
$ git status
On branch main
Changes to be committed:
  new file: profile.js
```

What Jim did:

- Used `git add` to **pin the note to his Pin Board**
- The file is now in the **Index (Staging Area)**

Jim's First Snapshot

> *Jim is satisfied with his work. Time to save this moment in history (repository)!*

```
$ git commit -m "Add basic user profile structure"
[main a1b2c3d] Add basic user profile structure
 1 file changed, 25 insertions(+)
```

What happened:

- Git took a **snapshot** of Jim's pinned notes
- Stored it in the **Gallery Collection** (Repository)
- Gave it a unique ID: `a1b2c3d`

Jim Continues Working

> *Jim adds more features: a profile photo upload function and a bio section. He creates two new files: "upload.js" and "bio.js".*

```
$ git add upload.js bio.js
$ git commit -m "Add photo upload and bio features"
[main e4f5g6h] Add photo upload and bio features
 2 files changed, 48 insertions(+)
```

Jim's progress:

- **Two snapshots** now exist in history
- His project is growing safely!

Jim Wants to See History

> *"How many snapshots have I created?" Jim wonders.*

```
$ git log --oneline
e4f5g6h (HEAD -> main) Add photo upload and bio features
a1b2c3d Add basic user profile structure
```

Jim learns:

- `git log` shows the **Gallery's timeline**
- Each commit has a unique **hash** (ID)
- **HEAD** points to where he is now in the "main" branch

Level 2: Checking out previous code

> *Jim suspects his latest changes introduced a bug. He recalls the code worked fine **two commits ago** and wonders: "How can I see what my code looked like before?"*

His colleague Sarah shows him the magic of `git checkout` — a command to **travel back** and **view any past version** of the project.

The Time Machine: git checkout

> *"Think of `git checkout` as a time machine for your code," Sarah explains. "It lets you travel to any point in your project's history!"*

```
# Check out a specific commit
git checkout <commit-hash>

# Check out a specific file from a commit
git checkout <commit-hash> -- <filename>

# Return to the latest version
git checkout main
```

Jim's First Time Travel

> *Jim wants to see his code from two commits ago. First, he checks the commit history:*

```
> git log --oneline
a3b4c5d (HEAD -> main) Fix button color issue
f2e1d0c Add user authentication
b9a8c7d Implement login form
e5f4g3h Initial project setup
```

> *"I want to go back to the 'Implement login form' commit," Jim says.*

```
> git checkout b9a8c7d

Note: switching to 'b9a8c7d'.

You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard
any commits you make in this state without impacting
any branches by switching back to a branch.
```

Understanding Detached HEAD

"What's a detached HEAD?" Jim asks, worried.

> *Sarah reassures him: "Don't panic! It just means you're looking at a past version. Think of it like watching a movie of your past code - you can see everything, but you're not really there."*

Jim explores his old code:

```
# Look at the files
ls
cat login.js
```

He finds that the bug wasn't there!

Returning to the Present

> *"How do I get back to my current work?" Jim asks. "Easy!" Sarah says. "Just checkout your branch again:"*

```
> git checkout main
Switched to branch 'main'
```

Checking Out Specific Files

Later, Jim realizes he only needs the old version of one file. Sarah shows him a more targeted approach

> *This brings the old version of login.js into your current working directory," Sarah explains. "You can then commit it if you want to keep it.*

```
# Get just the login.js file from the old commit
cp login.js login.js.backup # if you want to keep local copy
git checkout b9a8c7d -- login.js
```

Jim's New Superpower

Jim now feels confident using `git checkout`:

- ✓ He can explore any past commit
- ✓ He can retrieve specific files from history
- ✓ He can always return safely to his current work

> *"It's like having a perfect memory of every version of my code!" Jim exclaims.*

Tips from Sarah

Sarah adds that:

1. **Always note where you are**: Use `git status` to see if you're in detached HEAD state
2. **Create a branch if needed**: If you want to make changes to old code, create a new branch: `git checkout -b fix-from-old-version`
3. **Use descriptive commit messages**: They make time travel easier when you can understand what each commit did

Level 3: Fixing mistakes

> *After lunch, Jim realizes he made an error handling bug in "bio.js". "I need to fix this!" he thinks.*

Jim's problem:

- The current commit has incomplete code
- He wants to add the fix to the same commit
- He doesn't want a messy history

Jim Amends His Commit

> *Jim fixes the bug in bio.js error handling and wants to update his last commit.*

```
$ git add bio.js
$ git commit --amend -m "Fix bug in bio.js (with error handling)"
[main e4f5NEW] Fix bug in bio.js (with error handling)
```

What happened:

- Git **replaced** the last commit with a new one
- The old commit `e4f5g6h` is gone
- New commit `e4f5NEW` includes the fix
- **The history stays clean!**

Jim Makes Another Mistake

> *Jim accidentally adds a file "temp.js" to the staging area.*
>
> *"Wait! I don't want to commit this file!" he realizes.*

```
> git add temp.js
> git status
Changes to be committed: new file: temp.js

> git reset temp.js
> git status
Untracked files: temp.js
```

Jim learns: `git reset NOTE` can **un-pin** the NOTE from the board!

A Bigger Mistake

> *Jim commits a change that breaks the entire application. His teammate notices and tells him: "We need to undo that last commit!"*

Two approaches Jim can use:

1. **Reset** - Go back in time (destructive)
2. **Revert** - Create a new commit that undoes changes (safe)

Jim Reverts the Bad Commit

> *Jim's teammate suggests: "Use revert - it's safer when working with others because it does not delete the history!"*

```
$ git revert HEAD
[main x7y8z9] Revert "Broken feature"
```

Why revert is safer:

- Creates a **new commit** that undoes changes
- Safe for **shared repositories** as otther teammates won't have conflicts

The example of git revert

Let's say we have two commits, and we want to make it *as if commit b2 never happened*.

```
> git log --oneline
b2 Add wrong line
a1 Add hello.txt
> git revert b2
> git log --oneline
c3 Revert "Add wrong line"
b2 Add wrong line
a1 Add hello.txt
```

`git revert b2` creates a new commit (c3) that undoes the changes introduced by b2, while keeping the full history intact.

The Magic of `git revert`

`git revert` doesn't *delete added files* or *copy files from a previous commit*.
Instead, it creates a **new commit** whose *diff* (set of changes) is the **inverse** of the target commit.

1. Git finds the diff (patch) that `b2` introduced compared to its parent (`a1`).
2. Git creates a new commit (`c3`) that applies the *opposite* of that diff.
    - If `b2` added a line, `c3` removes it.
    - If `b2` deleted a line, `c3` adds it back.

Reset Breaks History, Then When to Use Reset

> *Later, Jim's working alone on a private branch. He makes 2 bad commits that no one has seen yet.*

```
> git log --oneline
f6g5h4 Bad commit 2
i3j2k1 Bad commit 1
m0n9o8 Last good commit
> git reset --hard m0n9o8
HEAD is now at m0n9o8 Last good commit
```

When to reset:

- Working **alone** on local commits
- Commits haven't been **shared** yet

Level 4: Organizing with tags and branches

> *Jim's manager says: "Great work yesterday! We're releasing version 1.0 today. Can you mark this version in Git?"*

Jim's new challenge:

- Mark important milestones
- Make versions easy to find
- No need to remember long hash numbers

Jim Creates His First Tag

> *"Tags are like bookmarks," Jim realizes.*

```
> git tag -a v1.0 -m "First public release"
> git tag
v1.0

> git log --oneline
p7q8r9 (HEAD -> main, tag: v1.0) Final touches for release
```

What Jim learned:

- **Tags** mark important points in history
- Easy to remember: `v1.0` instead of `p7q8r9`
- Perfect for **releases** and **milestones**

Jim Gets a New Feature Request

> *"Jim, we need you to build a dark mode feature. But don't break the main code - it's in production now!" says his manager.*

Jim's challenge:

- Experiment with new features
- Keep the main branch stable
- Work without fear of breaking things

Solution: Create a new **branch** (a new room in the gallery!)

Jim Creates a Feature Branch

> *"I'll create a separate room to experiment with dark mode!"*

```
> git checkout -b feature/dark-mode
Switched to a new branch 'feature/dark-mode'

> git branch
  main
- feature/dark-mode
```

What happened:

- Created a **new branch** (new room!)
- Switched to it with **checkout**
- The `*` shows Jim is now in the dark-mode room
- Main branch is safe!

Important Distinction

- `git checkout <commit>` → *Detached HEAD*
  - You’re visiting an old snapshot.
  - It’s safe to look around, but **don’t make new changes** here — they won’t be attached to any branch.
- `git checkout -b <branch>` → *Create and move to a new branch*
  - A **new room** for experimentation.
  - You can safely **edit, commit, and merge** your work later.

Jim Works on Dark Mode

> *Jim spends the morning coding the dark mode feature. He makes several commits on his feature branch.*

```
> git add dark-theme.css
> git commit -m "Add dark mode CSS"

> git add toggle.js
> git commit -m "Add dark mode toggle button"

> git log --oneline --graph --all
- b5c6d7 (HEAD -> feature/dark-mode) Add dark mode toggle
- a4b5c6 Add dark mode CSS
- p7q8r9 (tag: v1.0, main) Final touches for release
```

Meanwhile, on Main Branch

> *While Jim works on dark mode, his teammate Sarah fixed a critical bug on the main branch!*

```
$ git checkout main
$ git log --oneline
z1x2y3 (main) Fix critical security bug
p7q8r9 (tag: v1.0) Final touches for release
```

The power of branches:

- Jim's experimental code in `feature/dark-mode`
- Sarah's bug fix in `main`
- **Both work simultaneously without conflicts!**

Level 5: Merging Branches

> *"Dark mode is ready! My manager approved it. Time to bring it into the main branch," Jim announces.*

Jim's task:

- Combine the dark-mode room with the main room
- Keep both Sarah's bug fix and his dark mode
- This is called **merging**!

Jim Performs His First Merge

> *Jim switches to main and merges his feature branch.*

```
$ git checkout main
$ git merge feature/dark-mode
Merge made by the 'recursive' strategy.
 dark-theme.css | 50 ++++++++++
 toggle.js      | 30 +++++++
 2 files changed, 80 insertions(+)
```

What happened:

- Git performed a **3-way merge**
- Combined both branches' changes
- Created a **merge commit**

Visualizing the Merge

```
> git log --oneline --graph --all
*   m9n8o7 (HEAD -> main) Merge feature/dark-mode
|\
| * b5c6d7 (feature/dark-mode) Add dark mode toggle
| * a4b5c6 Add dark mode CSS
* | z1x2y3 Fix critical security bug
|/
* p7q8r9 (tag: v1.0) Final touches for release
```

Jim sees:

- Two parallel development paths
- Successfully merged together
- Both features are now in main!

Jim Encounters a Conflict

> *Jim creates another branch to refactor some code. But Sarah also modified the same file on main. When Jim tries to merge, Git stops with a conflict!*

```
> git merge feature/refactor
Auto-merging profile.js
CONFLICT (content): Merge conflict in profile.js
Automatic merge failed; fix conflicts and commit.
```

Jim's first conflict!

- Both branches changed the **same lines**
- Git can't decide which version to keep
- Jim must **manually resolve** it

Jim Resolves the Conflict

> *Jim opens profile.js and sees Git's conflict markers.*

```
<<<<<<< HEAD
function getProfile() {
  return user.profile;
}
=======
function getUserProfile() {
  return user.data.profile;
}
>>>>>>> feature/refactor
```

Jim's decision:

- Code above `=======` is from **main** branch
- Code below is from **feature/refactor**
- He needs to **choose or combine** them

Finishing the Conflict Resolution

> *Jim decides to keep Sarah's changes but use his better function name.*

```
function getUserProfile() {
  return user.profile;
}
```

```
> git add profile.js
> git commit -m "Merge feature/refactor with resolved conflicts"
[main c3d4e5] Merge feature/refactor
```

Conflict resolved! Jim learned that:

- Conflicts are **normal** in teamwork
- Communication helps **prevent** conflicts
- Git helps you **see** and **fix** conflicts

Level 6: Advanced Workflow

> *Jim's team lead says: "Great work, Jim! Now let me show you some advanced techniques the senior developers use."*

Advanced topics:

- **Rebase** - Keep history linear
- **Cherry-pick** - Take specific commits

Jim Learns About Rebase

> *"Merge commits create a tangled history. What if we could make it look linear?" the team lead explains.*

Current situation:

```
    A – B – C (main)
         \
          D – E (feature)
```

After rebase:

```
    A – B – C (main) – D' – E' (feature)
```

Commits D and E are **replayed** on top of C!

Jim Rebases His Branch

> *Jim has a feature branch that's behind main. Instead of merging, he'll rebase.*

```
> git checkout feature/optimization
> git rebase main
First, rewinding head to replay your work on top of it...
Applying: Optimize image loading
Applying: Reduce memory usage

> git log --oneline --graph
- f8g9h0 (HEAD -> feature/optimization) Reduce memory usage
- e7f8g9 Optimize image loading
- c3d4e5 (main) Latest main branch commit
```

Result: A **clean, linear** history!

We have two branches

```
    A – B – C (main) * <– Merge from main
          \
           D – E (feature)
```

Merge:

- We checkout to the main branch
- We (three-way) merge the feature branch

```
A – B – C —— M   (main)
      \     /
       D – E     (feature)
```

```
    A – B – C (main)
          \
           D – E (feature) <– Rebase from feature
```

Rebase:

- We checkout to the feature branch
- We compute the changes (B -> D) and apply the changes on C, and compute (D -> E) and apply again on the feature branch.

```
    A – B – C (main) – D' – E' (feature)
```

When to Rebase vs Merge

> *Jim's team lead explains the golden rule.*

Use REBASE when:

- Working on a **private** branch
- Want a **clean, linear** history

Use MERGE when:

- Working on **shared** branches
- Want to **preserve** exact history

⚠️ **Never rebase commits that others are using!**

Jim Needs Cherry-Pick

> *"Oh no! I made an important bug fix on the wrong branch. I need just that one commit on main, but not the other experimental changes."*

```
$ git log --oneline feature/experimental
x9y8z7 Experimental feature (not ready)
bbbbbb Important bug fix (need this!)
aaaaaa More experimental code
```

Jim's need: Just the bug fix commit `w6v5u4`, not the others!

Jim Cherry-Picks the Fix

```
> git checkout main
> git cherry-pick w6v5u4
[main w6v5NEW] Important bug fix
 Date: Thu Oct 26 14:30:00 2025
 1 file changed, 5 insertions(+), 2 deletions(-)
> git log --oneline
w6v5NEW (HEAD -> main) Important bug fix
c3d4e5 Latest main branch commit
```

What happened:

- Git computes the changes from aaaaaa to bbbbbb
- Applied it to the main branch
- Gave it a new hash `w6v5NEW`
- Other experimental commits stayed behind!

Jim's Workflow Wisdom

> *After a week of learning, Jim reflects on his Git journey.*

Jim's Git principles:

1. **Commit early, commit often** - Small snapshots are better
2. **Use branches** - Experiment without fear
3. **Write clear messages** - Help your future self
4. **Communicate** - Tell teammates before rebasing
5. **Don't panic** - Git can recover almost anything!