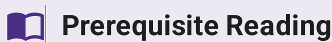
a1c # tooling

Git Branching and Pull Requests

TCSS 305 Programming Practicum

This guide introduces Git branching and the pull request workflow. You'll learn why branches isolate work safely, how to create and work on feature branches, and how to merge your changes back to main via pull requests. Assignment 1c requires this workflow: you'll develop on a separate branch and merge to main through a pull request.



Prerequisite Reading

This guide builds on Git Version Control: Getting Started, which covers repositories, commits, push, and the basic workflow. Review that guide first if these terms are unfamiliar.

Why Use Branches?

Imagine you're working on a feature and halfway through, your code doesn't compile. If you've been working directly on `main`, your main branch is now broken. You can't easily:

- Share working code with teammates
- Switch to fixing an urgent bug
- Compare your broken state against what was working

Branches solve this problem by letting you **isolate your work** from the stable codebase.

```
┌─────────────────────────────────────────────────────────────┐
│  main branch (stable, always works)                          │
│  ═══════════════════════════════════════════════════════════│
│         ↓                                                      │
│         ├── feature branch (your experimental work)          │
│         │   ────────────────────────►                        │
│         │   (can be broken, incomplete, messy)               │
│         │                                                      │
│        ◄── merge back when complete and tested               │
│                                                                │
│  ═══════════════════════════════════════════════════════════│
│  main branch (still stable, now includes your feature)       │
└─────────────────────────────────────────────────────────────┘
```

Benefits of Branching

Benefit	Description
Isolation	Your work-in-progress doesn't affect the stable codebase
Safe experimentation	Try ideas without fear of breaking anything permanently
Parallel work	Multiple features can be developed simultaneously
Clean history	Main branch shows completed features, not half-finished work
Code review	Changes can be reviewed before merging

> **Important**
>
> In professional development, the `main` branch is considered sacred. It should always compile, always pass tests, and always be deployable. Feature branches are where the messy work of development happens.

CI/CD: When Main Means Production

In many projects, pushing to `main` doesn't just update files—it **automatically deploys to production**. This is called **Continuous Integration/Continuous Deployment (CI/CD)**.

CI/CD in brief:

- **Continuous Integration (CI):** Every push triggers automated tests. If tests fail, the team is alerted immediately.
- **Continuous Deployment (CD):** When tests pass on `main`, the code is automatically deployed to users.

This means a broken commit to `main` can break the live product for real users within minutes.

Example 1: This Course Website

The site you're reading right now uses CI/CD:

1. Course materials are written in Markdown on a feature branch
2. Changes are merged to `main` via Pull Request
3. When changes hit `main`, GitHub Actions automatically builds the site

4. The built site is deployed to GitHub Pages within minutes
5. Students see the updated content at cfb3.github.io/TCSS305-GUIDES

If broken Markdown or incomplete content were pushed directly to `main`, the site could fail to build—or students would see unfinished materials not ready for publication.

Example 2: A Web Application

Imagine a team building an e-commerce site:

1. Developer creates `feature/new-checkout` branch
2. Developer writes and tests the new checkout flow
3. Developer creates a Pull Request to merge into `main`
4. Team reviews the code, automated tests run
5. After approval, the PR is merged to `main`
6. CI/CD pipeline automatically deploys to production
7. Real customers can now use the new checkout flow

Without branches, a half-finished checkout feature could go live and lose sales.

Tip

Even for solo projects, using branches builds habits that will serve you well in team environments where CI/CD is standard.

Branches: Conceptual Model

Before understanding branches, let's clarify how Git stores your project.

How Git Stores Your Project

When you **commit**, Git takes a snapshot of all your files at that moment and stores it permanently. Each commit has:

- A unique ID (a long hexadecimal string like `a1b2c3d`)
- A pointer to the previous commit (its "parent")
- The actual file contents at that point in time

Your repository—whether local or on GitHub—stores **every commit you've ever made**. This is why you can always go back to any previous version. Git doesn't just store the latest files; it

stores the complete history.

Not Just Diffs

Unlike some version control systems, Git stores complete snapshots, not just the changes between versions. It's smart about storage (identical files aren't duplicated), but conceptually each commit is a full picture of your project.

What Is a Branch?

A **branch** is simply a pointer (a label) that points to a specific commit. That's it—just a name attached to a commit ID.

When you make new commits on a branch, the branch pointer moves forward to point to the new commit. The old commits don't go anywhere; you just have a new "current" commit.

The Main Branch

When you clone a repository, you start on the `main` branch (historically called `master`). This is the primary branch—the pointer that represents your project's official state.

```
main → [commit A] ← [commit B] ← [commit C]
                                     ↑
                                 You are here
```

Creating a Feature Branch

When you create a new branch, Git creates a second pointer starting at the same commit:

```
main ──────→ [commit C]
                  ↑
dev ─────────────┘
```

Both branches point to the same commit initially. As you make commits on your new branch, it moves forward while `main` stays put:

```
main ──────→ [commit C]
                  ↑
                  └─── [commit D] ← [commit E]
                                        ↑
                                    dev (your work)
```

Your commits on `dev` don't affect `main`. The `main` branch still points to commit C.

Merging Back: The Pull Request

When your feature is complete, you want to bring those changes back into `main`. This is called **merging**, and on GitHub, you propose a merge through a **Pull Request (PR)**.

A Pull Request says: "I have commits on my branch that I'd like to merge into main. Please review and approve."

Here's what happens when the PR is merged:

```
Before merge:
main ────────→ [commit C]
                   ↑
                   └─── [commit D] ← [commit E]
                                        ↑
                                      dev

After merge:
main ────────→ [commit C] ←─── [commit D] ← [commit E] ← [merge commit F]
                                                               ↑
                                                          main (moved!)

                   ↑
             dev (can be deleted)
```

The merge creates a new commit (F) that combines the histories. Now `main` points to this merge commit, which includes all your work from `dev`. The `dev` branch has served its purpose and can be deleted.

Note

We'll cover Pull Requests in detail later in this guide. For now, understand that a PR is the mechanism for proposing and reviewing a merge on GitHub.

Feature Branch Workflow

The feature branch workflow is the standard approach for making changes:

1. **Create a branch** for your feature or task
2. **Checkout (switch to)** that branch
3. **Make commits** on that branch
4. **Push the branch** to GitHub
5. **Create a Pull Request** to propose merging
6. **Review and merge** the changes into main

7. **Checkout main** and pull the merged changes

Checkout means switching your working directory to a different branch. When you checkout a branch:

- Your files change to match that branch's latest commit
- New commits you make will be added to that branch
- Other branches are unaffected

Think of it like switching between parallel universes—each branch is a different version of your project, and checkout teleports you between them.

This workflow is used at virtually every software company. Learning it now prepares you for professional development.

Gen AI & Learning: Git Workflow Commands

AI coding assistants can help you with Git commands, especially when you encounter unfamiliar situations or error messages. However, blindly running AI-suggested Git commands can be dangerous. Always understand what a command does before running it—especially commands involving `reset`, `force`, or `--hard`. These can permanently lose work. Ask the AI to explain commands if you're unsure.

Creating a Branch

Using IntelliJ

1. Open the Git menu: **Git > Branches...**
2. Click **+ New Branch**
3. Enter a branch name (e.g., `dev` or `feature/shopping-cart`)
4. Check "Checkout branch" to switch to it immediately
5. Click **Create**

Using the Command Line

```
# Create a new branch and switch to it
git checkout -b dev

# Or, in two steps:
git branch dev          # Create the branch
git checkout dev        # Switch to it
```

Using GitHub Desktop

1. Click the **Current Branch** dropdown in the top bar
2. Click **New Branch**
3. Enter a branch name and click **Create Branch**

> **Branch Naming Conventions**
>
> Use descriptive, lowercase names with hyphens:
>
> - `dev` — general development branch
> - `feature/shopping-cart` — a specific feature
> - `fix/null-pointer-bug` — a bug fix
> - `refactor/extract-helper-methods` — code cleanup
>
> For TCSS 305 assignments, `dev` is usually sufficient.

Checking Your Current Branch

Always know which branch you're on before making changes.

In IntelliJ

Look at the bottom-right corner of the window. The current branch name is displayed there.

Using the Command Line

```
git branch
# Shows all local branches, with * marking the current one:
#   main
# * dev

git status
# First line shows: On branch dev
```

In GitHub Desktop

The current branch is shown in the "Current Branch" dropdown at the top.

Warning

A common mistake is making commits on the wrong branch. Before you start working, verify you're on your feature branch, not `main`.

Working on Your Branch

Once you're on your feature branch, work as normal:

1. Edit files in your IDE
2. Commit changes with meaningful messages
3. Repeat until your feature is complete

All your commits go to your current branch. The `main` branch remains untouched.

Commit Often

Make small, focused commits. Each commit should represent a logical unit of work:

```
Good commit messages:
- "Add constructor parameter validation"
- "Implement getPrice method"
- "Fix equals comparison for null"
- "Add unit tests for discount calculation"

Not so good:
- "Fixed stuff"
- "WIP"
- "asdfasdf"
- "Final version (for real this time)"
```

Tip

Commit messages should complete the sentence: "If applied, this commit will..." Your message should describe what the commit does, not what you did.

Pushing Your Branch to GitHub

Your branch exists only on your local machine until you push it. To share your work (and back it up), push to GitHub.

First Push (New Branch)

The first time you push a new branch, you need to tell Git where to push it:

Using IntelliJ

1. **Git > Push...**
2. IntelliJ automatically detects the new branch and sets up tracking
3. Click **Push**

Using the Command Line

```
# First push: set up tracking
git push -u origin dev

# Subsequent pushes: just push
git push
```

The `-u` (or `--set-upstream`) flag tells Git to remember that your local `dev` branch corresponds to the remote `origin/dev` branch.

Using GitHub Desktop

Click **Publish branch** in the top bar. GitHub Desktop handles the upstream setup automatically.

Subsequent Pushes

After the first push, just push normally:

```
git push
```

Or use IntelliJ's Git > Push, or click "Push origin" in GitHub Desktop.

What Is a Pull Request?

A **Pull Request** (PR) is a request to merge one branch into another. It's not a Git feature—it's a GitHub feature built on top of Git.

Pull requests enable:

- **Code review**: Others can see and comment on your changes

- **Discussion**: Questions and suggestions before merging
- **CI/CD integration**: Automated tests can run on the proposed changes
- **History**: A record of what was merged and why

Important

In professional settings, code almost never goes directly into main. Everything goes through pull requests with code review. This practice catches bugs, improves code quality, and shares knowledge across the team.

The Pull Request Flow

```
┌─────────────────────────────────────────────────────────────┐
│  1. You create a PR: "Please merge 'dev' into 'main'"       │
│         ↓                                                   │
│  2. GitHub shows a diff of all changes                      │
│         ↓                                                   │
│  3. Reviewers comment, request changes, or approve          │
│         ↓                                                   │
│  4. You address feedback (push more commits if needed)      │
│         ↓                                                   │
│  5. PR is approved and merged                               │
│         ↓                                                   │
│  6. Changes now appear in main                              │
└─────────────────────────────────────────────────────────────┘
```

Creating a Pull Request

After pushing your branch, create a PR on GitHub.

Using GitHub Web Interface

1. Go to your repository on GitHub.com
2. You'll often see a banner: "dev had recent pushes" with a **Compare & pull request** button. Click it.
3. Or, click **Pull requests** tab, then **New pull request**
4. Set the base branch (usually `main`) and compare branch (your feature branch)
5. Write a descriptive title and description
6. Click **Create pull request**

Using GitHub Desktop

1. After pushing, click **Create Pull Request** in the banner
2. GitHub Desktop opens the PR creation page in your browser
3. Complete the title and description, then create the PR

Writing a Good PR Description

A good PR description helps reviewers understand your changes:

```
## Summary
Brief description of what this PR does.

## Changes
- Added validation to constructor
- Implemented equals and hashCode methods
- Fixed bulk discount calculation

## Testing
- All existing tests pass
- Added 5 new test cases for edge conditions

## Notes
Any additional context, questions, or areas to focus review on.
```

For course assignments, your PR description should summarize what you implemented and any challenges you faced.

Merging a Pull Request

Once your PR is ready (and approved, if required), you can merge it.

On GitHub

1. Open your pull request on GitHub.com
2. Scroll to the bottom of the PR page
3. Click **Merge pull request**
4. Click **Confirm merge**
5. Optionally, delete the feature branch (it's been merged, so you don't need it)

Merge Options

GitHub offers three merge strategies:

Strategy	Result	When to Use
Merge commit	Creates a merge commit combining the branches	Default, preserves full history
Squash and merge	Combines all commits into one	Clean history for small features
Rebase and merge	Replays commits on top of main	Linear history, advanced use

For TCSS 305, the default **Merge commit** is fine.

After Merging: Syncing Your Local Repository

After merging on GitHub, your local `main` branch is behind. You need to update it.

Update Local Main

```
# Switch to main
git checkout main

# Pull the merged changes
git pull
```

Or in IntelliJ: **Git > Pull** while on the main branch.

Clean Up the Merged Branch

The feature branch has served its purpose. You can delete it locally:

```
git branch -d dev
```

The `-d` flag only deletes if the branch has been merged. Use `-D` to force delete (be careful!).

Handling Merge Conflicts

Sometimes, changes on your branch conflict with changes on main. This happens when:

- Someone else modified the same lines you did
- You branched from an older version of main

Git can't automatically decide which changes to keep, so it asks you to resolve the conflict.

What a Conflict Looks Like

When you try to merge and there's a conflict, Git marks the file:

```
public String getName() {
<<<<<<< HEAD (main)
    return this.name;
=======
    return myName;
>>>>>>> dev
}
```

The markers show:

- `<<<<<<< HEAD` to `=======` — what's in the target branch (main)
- `=======` to `>>>>>>>` — what's in your branch (dev)

Resolving Conflicts

1. **Find the conflict markers** in the affected files
2. **Decide what the code should be** (pick one version, combine them, or write something new)
3. **Remove the conflict markers**
4. **Save the file**
5. **Stage and commit** the resolved file

Example resolution:

```
// After deciding to use 'name' instead of 'myName':
public String getName() {
    return this.name;
}
```

Using IntelliJ's Merge Tool

IntelliJ provides a visual merge tool:

1. When conflicts occur, IntelliJ shows a dialog
2. Click **Merge** to open the three-way merge tool

3. Left panel: main's version
4. Right panel: your version
5. Center panel: the result
6. Use the arrows to accept changes from either side
7. Manually edit the center if needed
8. Click **Apply**

Tip

The best way to avoid merge conflicts is to:

- Keep branches short-lived (merge frequently)
- Pull main regularly and merge it into your feature branch
- Communicate with teammates about who's working on what

Best Practices for Branching and PRs

Branch Hygiene

- **Keep branches focused**: One feature or fix per branch
- **Keep branches short-lived**: Merge within days, not weeks
- **Delete merged branches**: Clean up to avoid clutter
- **Name branches descriptively**: Future you will thank present you

Pull Request Etiquette

- **Write clear descriptions**: Explain what and why
- **Keep PRs reasonably sized**: Smaller PRs are easier to review
- **Respond to feedback**: Address comments before expecting merge
- **Don't merge your own PRs** (in team settings without review)

The Single Branch Assignment Workflow

For TCSS 305 assignments using branches:

1. Clone the repository (you start on `main`)
2. Create and switch to `dev` branch

3. Do all your work on `dev`, committing often
4. Push `dev` to GitHub
5. Create a PR from `dev` to `main`
6. Merge the PR on GitHub
7. Pull `main` locally to sync

Assignment 1c Requirement

Assignment 1c specifically requires this workflow. You must develop on a branch (not `main`) and merge via a pull request. Your commit history and PR will be reviewed as part of the assignment.

Summary

Concept	Key Point
Branches	Isolate work from the stable codebase
Feature branch workflow	Create branch, work, push, PR, merge
Creating branches	`git checkout -b branchname` or use IDE/Desktop
Pull Requests	Propose and discuss merging changes
Merging	Combine branches; happens on GitHub for PRs
Merge conflicts	Occur when same lines changed; must resolve manually
After merging	Update local main with `git pull`, delete old branch

Further Reading

External Resources

- GitHub Flow Guide - GitHub's official workflow guide
- Atlassian: Git Feature Branch Workflow - Detailed workflow tutorial
- GitHub: Creating a Pull Request - Official PR documentation
- Pro Git: Branching and Merging - Comprehensive coverage

References

Primary Resources:

- Chacon, S., & Straub, B. (2014). Pro Git (2nd ed.). Apress. — Free online book, Chapters 3 (Branching) and 5 (Distributed Workflows)

GitHub Documentation:

- GitHub Docs: About Branches — Official branch documentation
- GitHub Docs: About Pull Requests — Official PR documentation
- GitHub Docs: Resolving Merge Conflicts — Conflict resolution guide

Git Documentation:

- Git: git-branch Documentation — Official git-branch reference
- Git: git-merge Documentation — Official git-merge reference
- Git: git-checkout Documentation — Official git-checkout reference

Tooling:

- IntelliJ IDEA: Manage Branches — Branch management in IntelliJ
- IntelliJ IDEA: Resolve Conflicts — Conflict resolution in IntelliJ
- GitHub Desktop: Creating a Branch — Branching in GitHub Desktop

---

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.