Branches & Merging

Creating, deleting, and renaming branches, switching with git switch, the difference between a fast-forward merge and a merge commit, an introduction to merge conflicts, and a realistic Spring Boot feature branch workflow. Why git switch is preferred over git checkout.

Intermediate 35 min

Last updated: August 27, 2026
Read online: https://www.learnforgex.com/en/topics/branches-and-merging

---

Every non-trivial project needs a way to work on multiple things at once without them interfering with each other — a new feature, a bug fix, an experiment — while keeping `main` always in a working state. That's what branches are for. This lesson covers creating, switching between, and merging branches using a realistic feature-branch workflow on a small Spring Boot project.

What Is a Branch?

A **branch** is simply a movable pointer to a commit. When you commit, the branch you're currently on moves forward to point at the new commit. `main` (sometimes still called `master` on older projects) is just a branch like any other — it has no special technical status, only a conventional one as "the primary line of development."

Because a branch is just a lightweight pointer (not a copy of the entire project), creating one is instant and cheap — there's no reason to hesitate before branching for even a small change.

Creating Branches — git branch

```
git branch add-password-reset       # create a new branch, but stay on the current one
git branch                          # list all local branches
```

`git branch <name>` alone only *creates* the branch — it does not move you onto it. You'll almost always want to create and switch in one step, covered next.

Switching Branches — git switch

`git switch` moves you onto a different branch, updating your working tree to match:

```
git switch main
git switch add-password-reset
```

Switching branches changes which files you see in your working tree — this is Git actually rewriting your project folder to match that branch's last commit, not some kind of view or filter.

> Git will refuse to switch branches if doing so would overwrite uncommitted changes it can't safely merge. Commit or stash (covered later in this course) your work before switching if you're not done with it.

Creating and Switching — git switch -c

The combined, everyday version — create a new branch and move onto it in one command:

```
git switch -c add-password-reset
```

This is what you'll actually type dozens of times; treat `git branch` alone as the rarer, "just list or manage branches" command.

Listing Branches

```
git branch          # local branches, current one marked with *
git branch -v       # local branches with their latest commit
git branch -a       # local AND remote-tracking branches
```

Renaming Branches

```
git branch -m old-name new-name     # rename a branch (from another branch)
git branch -m new-name              # rename the branch you're currently on
```

Deleting Local Branches

Once a branch's work has been merged, delete it to keep your branch list manageable:

```
git branch -d add-password-reset    # safe delete: refuses if not fully merged
git branch -D add-password-reset    # force delete: deletes even if unmerged
```

Prefer the lowercase `-d` — it's a built-in safety check that stops you from accidentally losing commits that only exist on that branch.

Merging Branches — git merge

`git merge` brings the changes from one branch into another. You run it *from* the branch you want to receive the changes:

```
git switch main
git merge add-password-reset
```

What happens next depends entirely on whether `main` has moved since the branch was created — which brings us to the two possible outcomes.

Fast-Forward Merge

If `main` hasn't changed at all since `add-password-reset` branched off, Git doesn't need to create anything new — it simply moves `main`'s pointer forward to match `add-password-reset`'s latest commit:

```
Before:  main → A
                 \
                  add-password-reset → B → C

After:   main → A → B → C
```

This is called a **fast-forward** merge because Git is just "fast-forwarding" a pointer — no new commit is created, and the history stays perfectly linear.

Merge Commit

If `main` *has* moved forward in the meantime (someone else merged something else first), a fast-forward isn't possible — Git instead creates a new **merge commit** with two parents, one from each branch:

```
Before:  main → A → D
                 \
                  add-password-reset → B → C

After:   main → A → D ------→ M   (M = merge commit, parents: D and C)
                 \           /
                  B → C ----
```

This is the normal, expected outcome once more than one person is merging into the same branch — it's not a problem, just a different (and more common in real teams) shape of history than a fast-forward.

Merge Conflicts

Sometimes both branches changed the *same lines* of the *same file* in different ways — Git can't automatically decide which version is correct, so it stops and asks you to resolve it manually. This is a **merge conflict**. We give conflicts a full dedicated lesson later in this course ("Merge Conflicts") — for now, know that it's a normal, expected part of merging, not a sign something went wrong.

Resolving Merge Conflicts

At a high level: Git marks the conflicting section directly inside the file, you edit the file to keep the correct content, then stage and commit to complete the merge:

```
# after a conflict, edit the file to resolve it, then:
git add ConflictedFile.java
git commit
```

The full mechanics — what the conflict markers actually look like, and a complete resolution walkthrough — are covered in depth in the "Merge Conflicts" lesson.

Deleting Merged Branches

Once `add-password-reset` has been merged into `main`, it's served its purpose:

```
git switch main
git branch -d add-password-reset
```

Keeping merged branches around indefinitely just clutters `git branch`'s output — deleting them costs nothing since their commits already live permanently in `main`'s history.

A Realistic Feature Branch Workflow

Putting it all together, for a small Spring Boot project:

```
#!/bin/sh
# A complete feature branch workflow on a small Spring Boot project: branch,
# work, merge back, clean up.

git switch -c add-password-reset
# Output:
# Switched to a new branch 'add-password-reset'

# ... edit PasswordController.java and PasswordService.java ...
git add .
git commit -m "Add password reset endpoint"
# Output:
# [add-password-reset f7e1a92] Add password reset endpoint
#  2 files changed, 31 insertions(+)

git switch main
# Output:
# Switched to branch 'main'

git merge add-password-reset
# Output (fast-forward, since main hasn't moved since branching):
# Updating a1b2c3d..f7e1a92
# Fast-forward
#  PasswordController.java | 18 ++++++++++++++++++
#  PasswordService.java    | 13 +++++++++++++
#  2 files changed, 31 insertions(+)

git branch -d add-password-reset
# Output:
# Deleted branch add-password-reset (was f7e1a92).
```

This exact loop — branch, work, merge back, delete — is the backbone of how professional teams organize day-to-day work, and every remaining lesson in this course builds on top of it (adding GitHub, Pull Requests, and code review around the same core flow).

A Note on git checkout

Before `git switch` existed (it was introduced in Git 2.23), branch switching was done with `git checkout <branch>`. You'll still see `checkout` constantly in older tutorials, Stack Overflow answers, and existing projects, so you need to recognize it — but for new code, prefer `switch` (for changing branches) and `restore` (for undoing file changes, covered in the previous lesson). The reason: `checkout` was overloaded to do *both* of those unrelated jobs plus a few more, which made it a common source of mistakes. `switch` and `restore` split those responsibilities into focused, harder-to-misuse commands.

Common Mistakes

- **Committing directly on `main`** instead of creating a feature branch first — makes it much harder to keep `main` always in a working, deployable state.

- **Forgetting which branch you're on** before making changes — `git status` always shows the current branch on its first line; get in the habit of checking.
- **Leaving merged branches undeleted indefinitely**, making `git branch` output noisy and hard to scan.

Best Practices

- One branch per feature or fix — keep branches focused and short-lived.
- Merge (or rebase, covered later) frequently to avoid a branch drifting too far from `main`.
- Delete branches immediately after merging — nothing is lost, since their history lives on in `main`.

Summary, Cheat Sheet, and Glossary

Summary

- A branch is a lightweight, movable pointer to a commit — creating one is instant.
- `git switch -c <name>` creates and moves onto a new branch in one step.
- Merging is a fast-forward (pointer moves, no new commit) if possible, otherwise a merge commit with two parents is created.
- Merge conflicts happen when both branches changed the same lines — resolved manually, then completed with `git add` + `git commit`.
- `git checkout` is the older, multi-purpose predecessor to `git switch` / `git restore` — recognize it, but prefer the newer commands.

Cheat Sheet

```
git switch -c <name>       # create and switch to a new branch
git switch <name>          # switch to an existing branch
git branch                 # list local branches
git branch -d <name>       # delete a merged branch (safe)
git merge <branch>         # merge <branch> into the current branch
git branch -m <new-name>   # rename the current branch
```

Glossary

- **Branch** — a movable pointer to a commit; the basis for working on things in parallel.
- **Fast-forward merge** — merging by simply moving a branch pointer forward, no new commit.
- **Merge commit** — a commit with two parents, created when a fast-forward isn't possible.
- **Merge conflict** — when Git can't automatically combine changes because both branches edited the same lines.