Merge Conflicts

What a merge conflict is, how to read conflict markers (<<<<<<<, =======, >>>>>>>), how to resolve one with real Java code examples, the same mechanics during both merge and rebase, and safely canceling with git merge --abort/git rebase --abort.

Intermediate 30 min

Last updated: August 27, 2026
Read online: https://www.learnforgex.com/en/topics/merge-conflicts

---

You've seen merge conflicts mentioned twice already — once in "Branches & Merging," once in "Rebase & Squash" — and deliberately left them for this dedicated lesson. Here we go through the full mechanics: exactly what Git shows you, exactly what it's asking you to do, and a repeatable workflow for resolving conflicts calmly instead of panicking.

What Is a Merge Conflict?

A **merge conflict** happens when Git tries to combine two sets of changes and can't automatically decide which version is correct, because both sides changed the same lines of the same file in different ways. Git stops the merge (or rebase) partway through and asks you — a human — to decide.

Why Conflicts Happen

Git merges automatically whenever it can — most changes across a team don't conflict at all, because they touch different files or different, non-overlapping lines. A conflict specifically requires **both branches to have modified the same lines**. Two developers adding unrelated methods to the same class usually merges cleanly; two developers changing the *same line* of the *same method* is what triggers a conflict.

Understanding Conflict Markers

When a conflict happens, Git writes both versions directly into the file, wrapped in markers:

```
public int calculateDiscount(Order order) {
<<<<<<< HEAD
    return order.getTotal() > 100 ? 10 : 0;
=======
    return order.getTotal() > 100 ? 15 : 5;
>>>>>>> add-loyalty-discount
}
```

Read this precisely:

- `<<<<<<< HEAD` marks the start of **your current branch's version**.
- `=======` separates the two versions.
- `>>>>>>> add-loyalty-discount` marks the end of **the incoming branch's version**, and names that branch.

Git isn't confused about *what* changed — it's showing you both valid options and asking you to pick (or combine) the correct outcome, since only you know the actual intent behind each side.

Resolving a Merge Conflict

Resolving means editing the file so it contains exactly the code you want — removing the markers and choosing (or merging) the content between them:

```
public int calculateDiscount(Order order) {
    return order.getTotal() > 100 ? 15 : 5;
}
```

There's no special "conflict resolution mode" — you're just editing a normal text file. The markers are the only unusual part, and once they're gone and the code is correct, Git treats the file as resolved.

Completing a Merge

Once every conflicted file is edited and the markers are gone, stage the resolved files and commit:

```
#!/bin/sh
# Merging a branch, hitting a conflict, resolving it, and completing the merge.

git merge add-loyalty-discount
# Output:
# Auto-merging DiscountCalculator.java
# CONFLICT (content): Merge conflict in DiscountCalculator.java
# Automatic merge failed; fix conflicts and then commit the result.

git status
# Output:
# On branch main
# You have unmerged paths.
#   (fix conflicts and run "git commit")
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#         both modified:   DiscountCalculator.java

# DiscountCalculator.java now contains conflict markers:
#
# public int calculateDiscount(Order order) {
# <<<<<<< HEAD
#     return order.getTotal() > 100 ? 10 : 0;
# =======
#     return order.getTotal() > 100 ? 15 : 5;
# >>>>>>> add-loyalty-discount
# }
#
# ... edit the file, keep the correct logic, remove all three marker lines ...

git add DiscountCalculator.java
git commit
# Output:
# [main 3f4a5b6] Merge branch 'add-loyalty-discount'

git status
# Output:
# nothing to commit, working tree clean
```

`git add` here means something slightly different than usual — it's not staging a new change, it's telling Git "this file's conflict is resolved." The commit that follows is the merge commit itself.

Conflicts During Rebase

Conflicts can also happen during a rebase — since rebase replays commits one at a time, a conflict can occur on any individual commit in the sequence, and Git pauses exactly at that commit.

Resolving Rebase Conflicts

The resolution mechanics are identical to a merge conflict — edit the file, remove the markers, `git add` the resolved file — but the command to move forward is different, because a rebase might still have more commits to replay after this one:

```
git add ResolvedFile.java
git rebase --continue
```

If more commits in the sequence also conflict, Git will stop again at each one — repeat the same resolve-and-continue steps until the rebase finishes.

Aborting a Merge

If a conflict turns out to be more complicated than expected, or you realize the merge shouldn't happen at all right now, you can cancel it entirely and return to exactly how things were before you started:

```
git merge --abort
```

Aborting a Rebase

The equivalent for a rebase in progress:

```
git rebase --abort
```

Both `--abort` commands are completely safe — there's no partial or broken state to worry about. If a conflict is confusing enough that you're not confident in your resolution, aborting and re-approaching (perhaps after talking to whoever wrote the conflicting change) is always a reasonable choice.

A Practical Conflict Resolution Workflow

A calm, repeatable process for any conflict:

1. Run `git status` to see exactly which files are conflicted.
2. Open each conflicted file and read *both* sides of every marker block before touching anything.
3. Understand what each side was trying to accomplish — not just the syntax, the intent.
4. Edit the file to the correct final result, removing all markers.

5. Run the project's tests if you can — a resolved conflict that compiles isn't necessarily a resolved conflict that's *correct*.
6. `git add` each resolved file.
7. Complete with `git commit` (merge) or `git rebase --continue` (rebase).

Common Mistakes

- **Accidentally leaving conflict markers in the committed code** — this genuinely happens, and it's a syntax error (or worse, silently wrong logic) that ships to `main`. Always re-read the file after resolving, and let your build/tests catch what a rushed read might miss.
- **Picking a side without understanding why the other side made its change** — sometimes the "correct" resolution is neither version alone, but a combination of both.
- **Panicking mid-conflict** — `--abort` always gets you back to safety; there's no need to push through a confusing conflict under pressure.

Best Practices

- Read both sides of a conflict fully before editing — don't just delete whichever one looks unfamiliar.
- Run tests after resolving, before completing the merge/rebase — a conflict resolution that compiles can still be logically wrong.
- When a conflict is genuinely unclear, talk to the person whose change you're conflicting with — they usually know the intent behind their side immediately.

Summary, Cheat Sheet, and Glossary

Summary

- A merge conflict happens when both branches changed the same lines of the same file — Git can't pick automatically, so it asks you to.
- Conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) show both versions directly in the file; resolving means editing them into the correct final content.
- The same resolve-and-continue mechanics apply during both merges (`git commit`) and rebases (`git rebase --continue`).
- `git merge --abort` and `git rebase --abort` are always safe escape hatches back to the pre-conflict state.

Cheat Sheet

```
git status                    # see which files are conflicted
# ... edit files, remove <<<<<<< ======= >>>>>>> markers ...
git add <resolved-file>       # mark a file's conflict as resolved
git commit                    # complete a merge
git rebase --continue         # complete one step of a rebase
git merge --abort             # cancel a conflicted merge entirely
git rebase --abort            # cancel a conflicted rebase entirely
```

Glossary

- **Merge conflict** — when Git can't automatically combine two branches' changes to the same lines.
- **Conflict marker** — the `<<<<<<<` / `=======` / `>>>>>>>` lines Git inserts to show both conflicting versions.
- **Resolve** — editing a conflicted file down to the single, correct final version.