Git Lesson 2: Git Deep Dive

Objectives / Goals	1. Git Branch 2. Git Merge 3. Merge Conflicts 4. Git Stash 5. Reverting changes 6. Rebasing - **Optional (will not be covered in class)**

★ Objective 1: Git Branch

"Branch" is not an unfamiliar word. It is too common to hear the words such as the branch of a tree, branch of a bank, a branch of science, etc. We use them quite often in our lives.

What is a Branch in Git?

Branch in Git is similar to the branch of a tree. Analogica :ral part of the tree called the trunk. While branches can generate and fall off, the trunk remains compact and is t ee is alive and standing. Similarly, a branch in Git is a way to keep developing and coding a new feature or mo fecting the main part of the project. We can also say that branches create another line of development in the n Git is the master branch (similar to a trunk of the tree). As soon as the repository creates, so does the mai

Why do we need a Branch in Git and Why Branche

Git branches come to the rescue at many different places during the development of a project. As mentioned above, branches create another line of development that is entirely different or isolated from the main stable master branch.

Using branches help you organize the workflow more efficiently and rather effortlessly.

Let's say, you are building a software for a company in a team of 5 people. It would be a good idea to have a branch for each developer because usually they work on different things. And because of the way Git works, you can keep working on your branch regardless of the work that is happening in other branches.

A common practice in a lot of companies is to have one branch where the code is ready to shipped and another branch for testing. So, In that case, you work on the testing branch and if everything is working as expected, you add it in to the main production branch.

These are just a couple of ways in which you can use branches. There can be a lot more cases based on your use case.

Refer to this link GIT branching commands for git branch commands

---

What is HEAD?

The term HEAD refers to the current commit you are viewing.

By default, you'll view the tip of the master branch on a repository, unless the main branch of your repository has a different name. The tip of the master branch is the most recent commit on the main branch of your codebase.

How does Git know what branch you're currently on? It keeps a special pointer called `HEAD`. In Git, this is a pointer to the local branch you're currently on. In this case, you're still on `master`. The `git branch` command only *created* a new branch — it didn't switch to that branch.

HEAD pointing to a branch.

You can easily see this by running a simple `git log` command that shows you where the branch pointers are pointing. This option is called `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

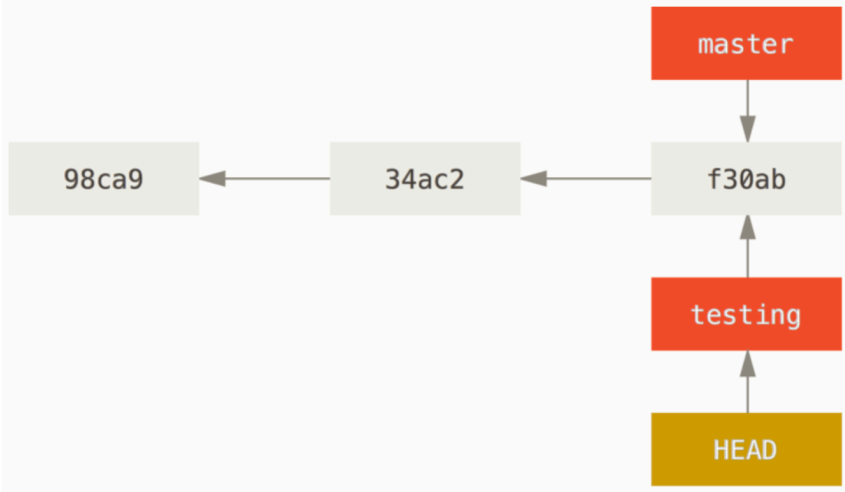
You can see the `master` and `testing` branches that are right there next to the `f30ab` commit.

Switching Branches

To switch to an existing branch, you run the `git checkout` command. Let's switch to the new `testing` branch:

```
$ git checkout testing
```

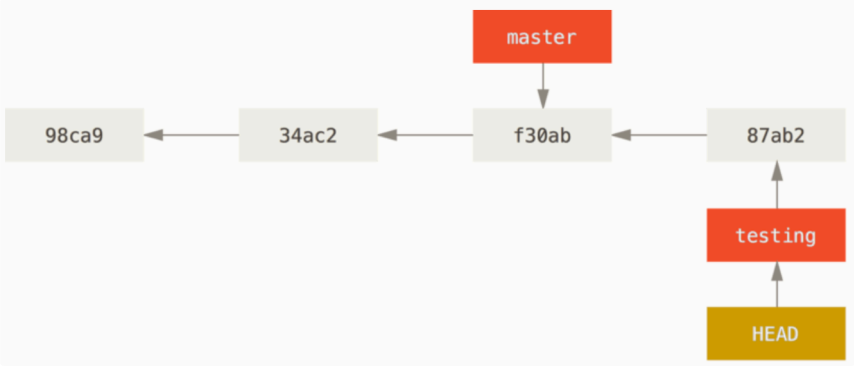
This moves `HEAD` to point to the `testing` branch



HEAD points to the current branch

What is the significance of that? Well, let's do another commit:

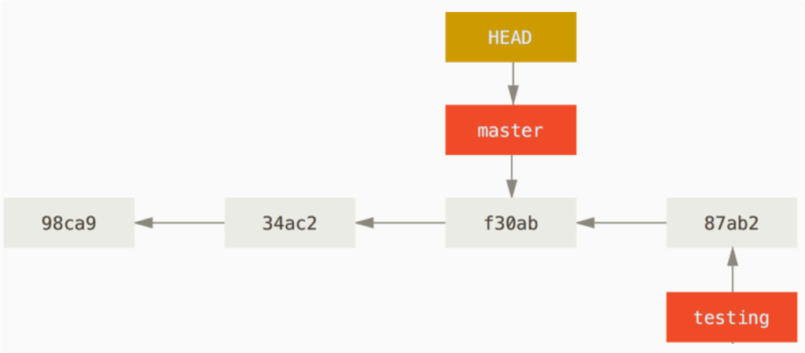
```
$ vim test.rb
$ git commit -a -m 'made a change'
```



The HEAD branch moves forward when a commit is made

This is interesting, because now your `testing` branch has moved forward, but your `master` branch still points to the commit you were on when you ran `git checkout` to switch branches. Let's switch back to the `master` branch:

```
$ git checkout master
```



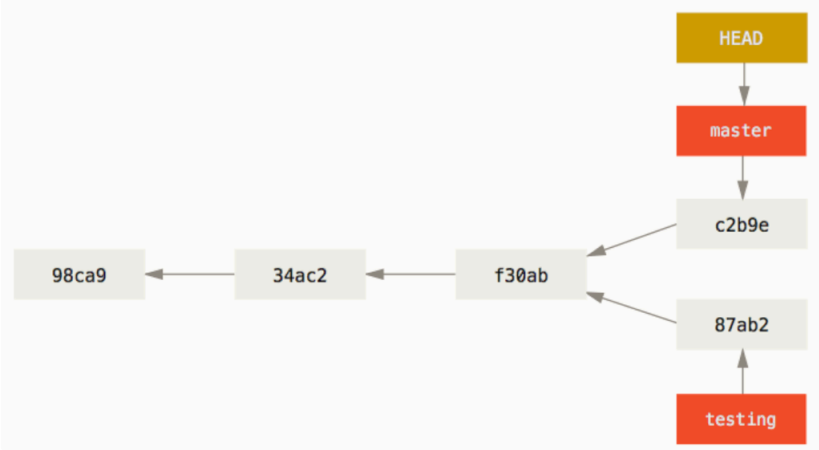
HEAD moves when you checkout

That command did two things. It moved the HEAD pointer back to point to the `master` branch, and it reverted the files in your working directory back to the snapshot that `master` points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your `testing` branch so you can go in a different direction.

Let's make a few changes and commit again:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Now your project history has diverged. You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple `branch`, `checkout`, and `commit` commands.



Creating & Switching Branches

`git branch`

List all of the branches in your repository. This is synonymous with `git branch --list.`

`git branch <branch>`

Create a new branch called `<branch>`. This does *not* check out the new branch.

`git branch -d <branch>`

Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

`git branch -D <branch>`

Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

`git branch -m <branch>`

Rename the current branch to `<branch>`.

`git branch -a`

List all remote branches.

Creating Branches

It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer, it doesn’t change the repository in any other way. If you start with a repository that looks like this:

Then, you create a branch using the following command:

```
git branch test1
```

The repository history remains unchanged. All you get is a new pointer to the current commit:

Note that this only *creates* the new branch. To start adding commits to it, you need to select it with `git checkout`, and then use the standard `git add` and `git commit` commands.

Git Checkout Vs Git Switch

Creating a new branch and switching to it at the same time

It’s typical to create a new branch and want to switch to that new branch at the same time — this can be done in one operation with `git checkout -b <newbranchname>`.

This is shorthand for:

```
$ git branch test1
$ git checkout test1
```

From Git version 2.23 onwards you can use `git switch` instead of `git checkout` to:

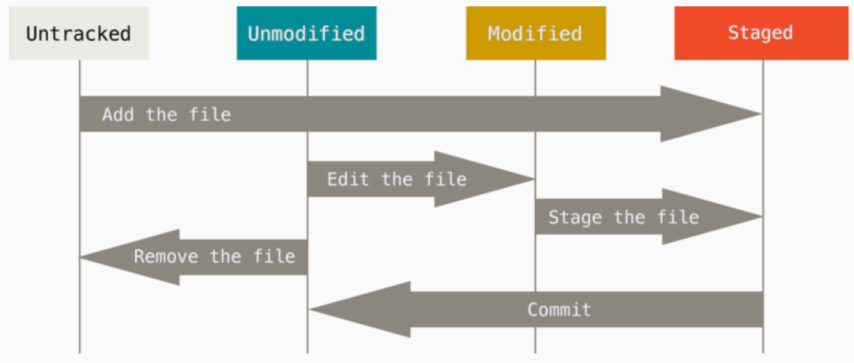
- Switch to an existing branch: `git switch test1`.
- Create a new branch and switch to it: `git switch -c test1`. The `-c` flag stands for create, you can also use the full flag: `--create`.
- Return to your previously checked-out branch: `git switch -`.

Tracked files

Each file in your working directory can be in one of two states: *tracked* or *untracked*. Tracked files are files that were in the last snapshot, as well as any newly staged files; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

As you edit files, Git sees them as modified, because you’ve changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.



Checking the Status of Your Files

The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server.

Let's say you add a new file to your project, a simple `README` file. If the file didn't exist before, and you run `git status`, you see your untracked file like so:

```
$ echo 'My Project' > README.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README.md

nothing added to commit but untracked files present (use "git add" to track)
```

You can see that your new `README.md` file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit), and which hasn't yet been staged; Git won't start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including `README.md`, so let's start tracking the file.

Tracking New Files

In order to begin tracking a new file, you use the command `git add`. To begin tracking the `README` file, you can run this:

```
$ git add README.md
```

If you run your status command again, you can see that your `README` file is now tracked and staged to be committed:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

    new file:   README.md
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the subsequent historical snapshot. You may recall that when you ran `git init` earlier, you then ran `git add <files>` — that was to begin tracking files in your directory. The `git add` command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file called `CONTRIBUTING.md` and then run your `git status` command again, you get something that looks like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

The `CONTRIBUTING.md` file appears under a section named “Changes not staged for commit” — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the `git add` command. `git add` is a multipurpose command — you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add precisely this content to the next commit” rather than “add this file to the project”. Let’s run `git add` now to stage the `CONTRIBUTING.md` file, and then run `git status` again:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.md
    modified:   CONTRIBUTING.md
```

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in `CONTRIBUTING.md` before you commit it. You open it again and make that change, and you’re ready to commit. However, let’s run `git status` one more time:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.md
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

What the heck? Now `CONTRIBUTING.md` is listed as both staged *and* unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the `git add` command. If you commit now, the version of `CONTRIBUTING.md` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`. If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Short Status

While the `git status` output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run `git status -s` or `git status --short` you get a far more simplified output from the command:

```
$ git status -s
```

```
 M README.md
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

New files that aren't tracked have a `??` next to them, new files that have been added to the staging area have an `A`, modified files have an `M` and so on. There are two columns to the output — the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the `README.md` file is modified in the working directory but not yet staged, while the `lib/simplegit.rb` file is modified and staged. The `Rakefile` was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Deleting a branch LOCALLY

Git will not let you delete the branch you are currently on so you must make sure to checkout a branch that you are NOT deleting. For example: `git checkout master`

Delete a branch with `git branch -d <branch>`.

For example: `git branch -d fix/authentication`

The **-d** option (**–delete**) will remove your local branch if you have already pushed and merged it with the remote branch.

The **-D** option (**–delete –force**) will remove the local branch regardless of whether it's been merged or not.

The branch is now deleted locally.

Deleting a branch REMOTELY

Here's the command to delete a branch remotely: `git push <remote> --delete <branch>`.

For example: `git push origin --delete fix/authentication`

The branch is now deleted remotely.

You can also use this shorter command to delete a branch remotely: `git push <remote> :<branch>`

For example: `git push origin :fix/authentication`

If you get the error below, it may mean that someone else has already deleted the branch.

```
error: unable to push to unqualified destination: remoteBranchName The destination refspec neither m
```

★ Objective 2: Git Merge

Intro to Merging

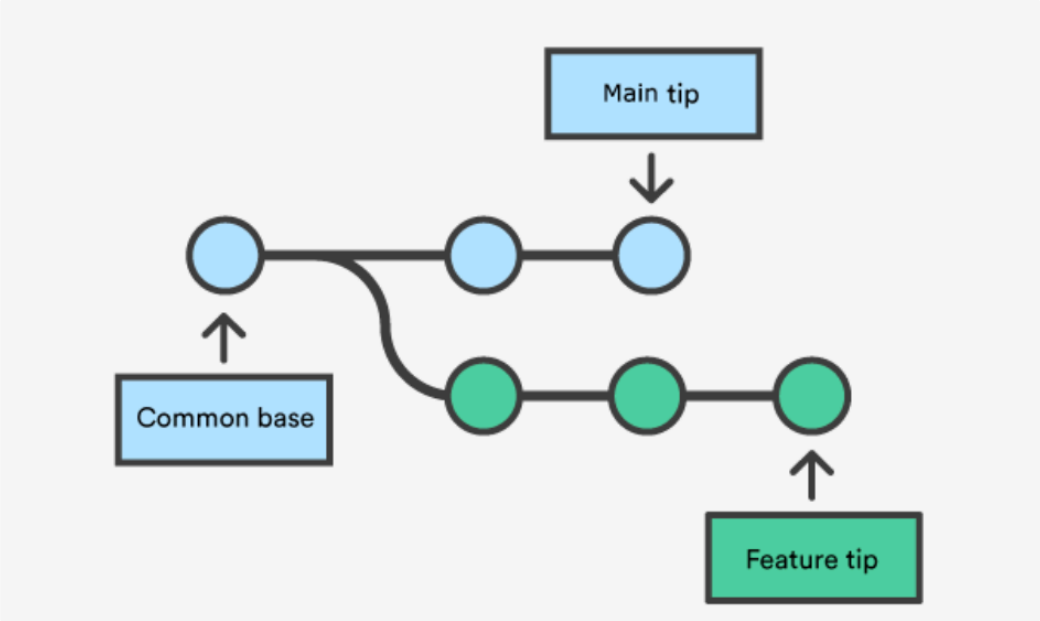
Merging is Git's way of putting a forked history back together again. The `git merge` command lets you take the independent lines of development created by `git branch` and integrate them into a single branch.

Note that all of the commands presented below merge into the current branch. The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. Again, this means that `git merge` is often used in conjunction with `git checkout` for selecting the current branch and `git branch -d` for deleting the obsolete target branch.

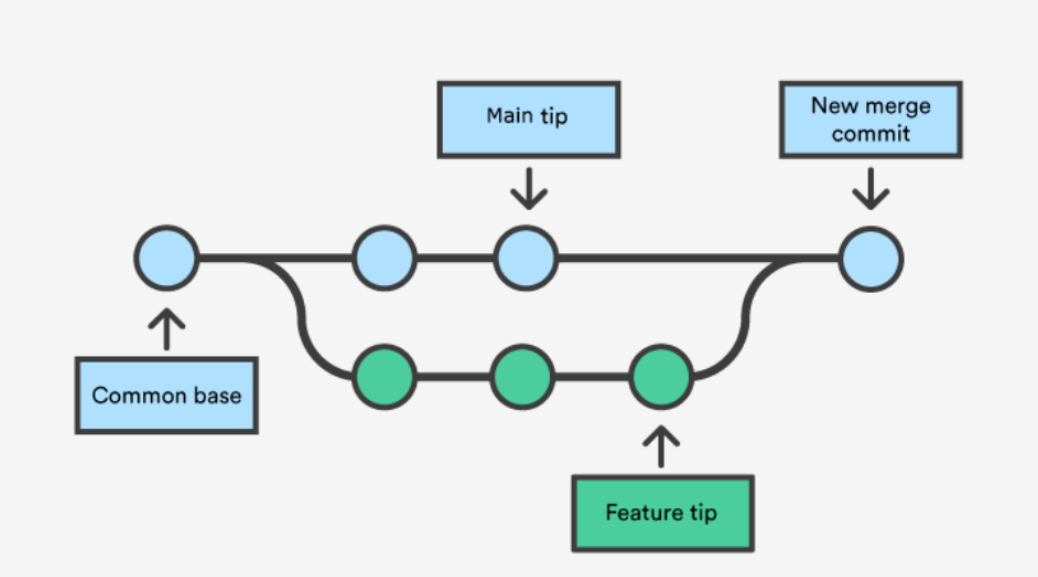
How it works

`Git merge` will combine multiple sequences of commits into one unified history. In the most frequent use cases, `git merge` is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, `git merge` takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.

Say we have a new branch feature that is based off the `main` branch. We now want to merge this feature branch into `main`.



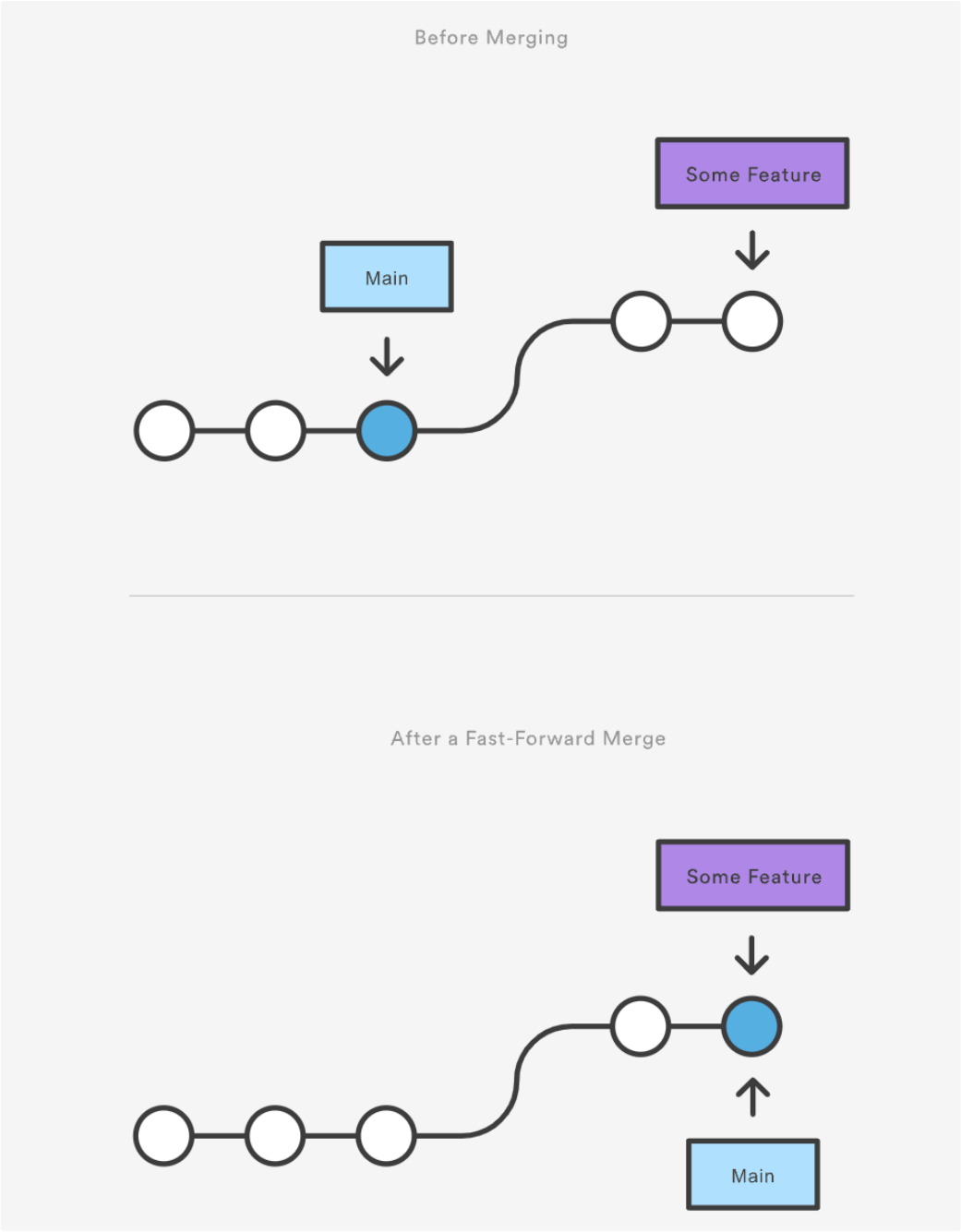
Invoking this command will merge the specified branch feature into the current branch, we'll assume `main`. Git will determine the merge algorithm automatically (discussed below).



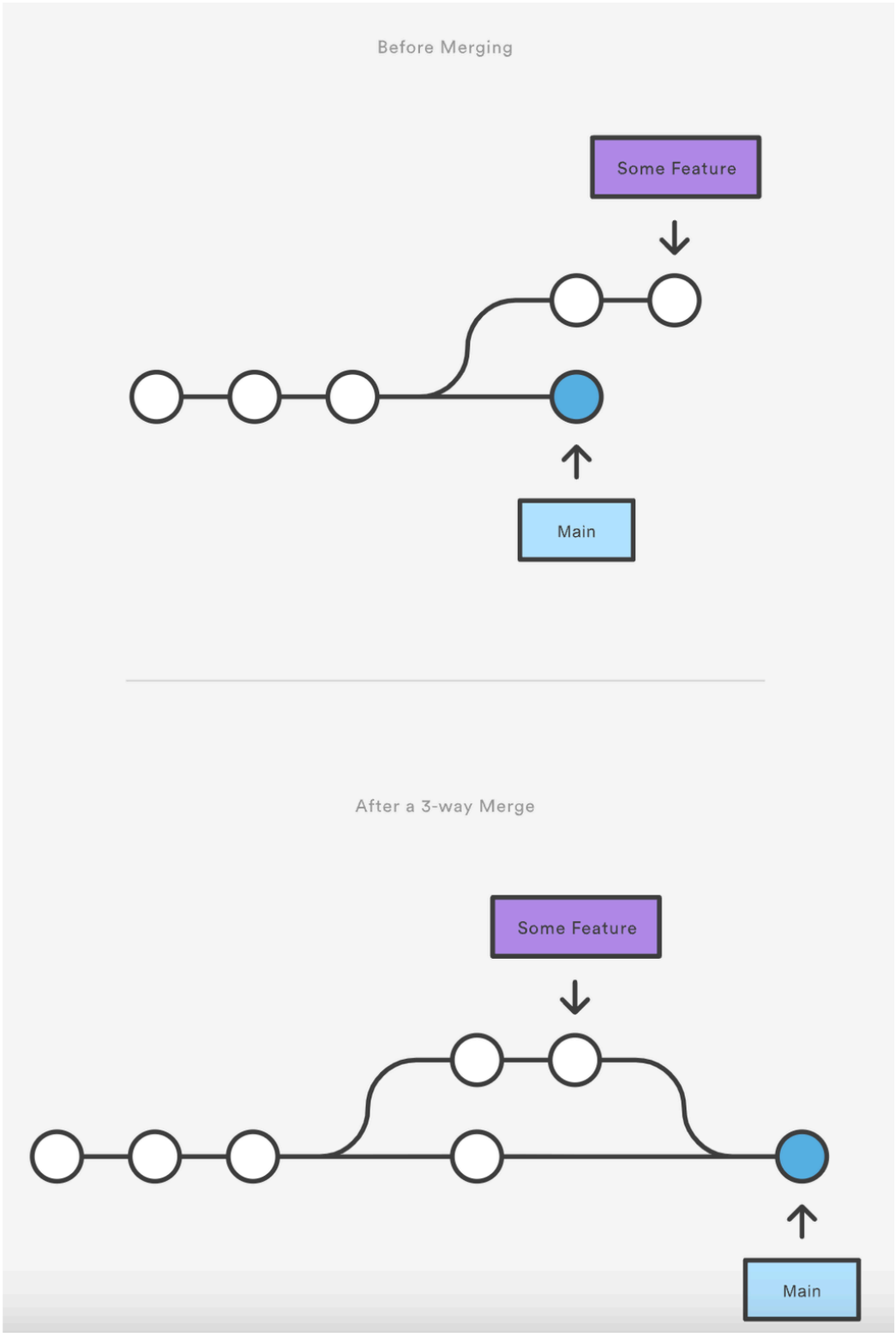
Merge commits are unique against other commits in the fact that they have two parent commits. When creating a merge commit Git will attempt to auto magically merge the separate histories for you. If Git encounters a piece of data that is changed in both histories it will be unable to automatically combine them. This scenario is a version control conflict and Git will need user intervention to continue.

Fast Forward Merge

A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into `main` would look something like the following:



However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.



While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through rebasing) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

Our first example demonstrates a fast-forward merge. The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.

```
# Start a new feature
git checkout -b new-feature main
# Edit some files
git add <file>
git commit -m "Start a feature"
# Edit some files
git add <file>
git commit -m "Finish a feature"
# Merge in the new-feature branch
git checkout main
git merge new-feature
git branch -d new-feature
```

This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

Also note that Git should not complain about the `git branch -d`, since new-feature is now accessible from the main branch.

In the event that you require a merge commit during a fast forward merge for record keeping purposes you can execute `git merge` with the `--no-ff` option.

```
git merge --no-ff <branch>
```

This command merges the specified branch into the current branch, but always generates a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

Merge Conflicts

Version control systems are all about managing contributions between multiple distributed authors (usually developers). Sometimes multiple developers may try to edit the same content. If Developer A tries to edit code that Developer B is editing a conflict may occur. To alleviate the occurrence of conflicts developers will work in separate isolated branches. The `git merge` command's primary responsibility is to combine separate branches and resolve any conflicting edits.

Understanding merge conflicts

Merging and conflicts are a common part of the Git experience. Git makes merging super easy. Most of the time, Git will figure out how to automatically integrate new changes.

Conflicts generally arise when two people have changed the same lines in a file, or if one developer deleted a file while another developer was modifying it. In these cases, Git cannot automatically determine what is correct. Conflicts only affect the developer conducting the merge, the rest of the team is unaware of the conflict. Git will mark the file as being conflicted and halt the merging process. It is then the developers' responsibility to resolve the conflict.

Types of merge conflicts

A merge can enter a conflicted state at two separate points. When starting and during a merge process. The following is a discussion of how to address each of these conflict scenarios.

Git fails to start the merge

A merge will fail to start when Git sees there are changes in either the working directory or staging area of the current project. Git fails to start the merge because these pending changes could be written over by the commits that are being merged in. When this happens, it is not because of conflicts with other developer's, but conflicts with pending local changes. The local state will need to be stabilized using `git stash`, `git checkout`, `git commit` or `git reset`. A merge failure on start will output the following error message:

```
error: Entry '<fileName>' not uptodate.
Cannot merge. (Changes in working directory)
```

Git fails during the merge

A failure DURING a merge indicates a conflict between the current local branch and the branch being merged. This indicates a conflict with another developers code. Git will do its best to merge the files but will leave things for you to resolve manually in the conflicted files. A mid-merge failure will output the following error message:

```
error: Entry '<fileName>' would be overwritten by merge.
Cannot merge. (Changes in staging area)
```

How to identify merge conflicts

Example:
Initial State: The main branch contains a file with the content "Hello, World!".
Developer A's Changes: In their own branch, Developer A changes the line to "Hello, Cat!".
Developer B's Changes: Meanwhile, Developer B changes the same line in the main branch to "Hello, Dog!".

Simulating the Merge Conflict
When Developer A attempts to merge their changes back into the main branch, a conflict arises because Git cannot automatically determine which version of the line to keep. Here's how this conflict is represented:

```
<<<<<<< HEAD
Hello, Cat!
=======
Hello, Dog!
>>>>>>> developer_b_branch
```

The markers <<<<<<<, =======, and >>>>>>> indicate the conflicting sections between Developer A's changes and Developer B's changes.

How to resolve merge conflicts using the command line

To resolve this conflict, you need to manually edit the file to choose which changes to keep or how to combine them. In this simulation, let's assume we decide to keep Developer A's changes:

```
Hello, Cat!
```

Git commands that can help resolve merge conflicts

General tools

```
git status
```

The status command is in frequent use when a working with Git and during a merge it will help identify conflicted files.

```
git log --merge
```

Passing the `--merge` argument to the `git log` command will produce a log with a list of commits that conflict between the merging branches.

```
git diff
```

`diff` helps find differences between states of a repository/files. This is useful in predicting and preventing merge conflicts.

Tools for when git fails to start a merge

```
git checkout
```

`checkout` can be used for *undoing* changes to files, or for changing branches

```
git reset --mixed
```

`reset` can be used to undo changes to the working directory and staging area.

Tools for when git conflicts arise during a merge

```
git merge --abort
```

Executing `git merge` with the `--abort` option will exit from the merge process and return the branch to the state before the merge began.

```
git reset
```

`Git reset` can be used during a merge conflict to reset conflicted files to a know good state

---

★ Objective 3: Git Stash

`git stash` temporarily shelves (or *stashes*) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

Stashing your work

The `git stash` command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. For example:

```
$ git status
On branch main
Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html

$ git stash
Saved working directory and index state WIP on main: 5002d47 our new homepage
HEAD is now at 5002d47 our new homepage

$ git status
On branch main
nothing to commit, working tree clean
```

At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

Re-applying your stashed changes

You can reapply previously stashed changes with `git stash pop`:

```
$ git status
On branch main
nothing to commit, working tree clean
$ git stash pop
On branch main
Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)
```

Popping your stash removes the changes from your stash and reapplies them to your working copy.

Alternatively, you can reapply the changes to your working copy *and* keep them in your stash with `git stash apply`:

```
$ git stash apply
On branch main
Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html
```

This is useful if you want to apply the same stashed changes to multiple branches.

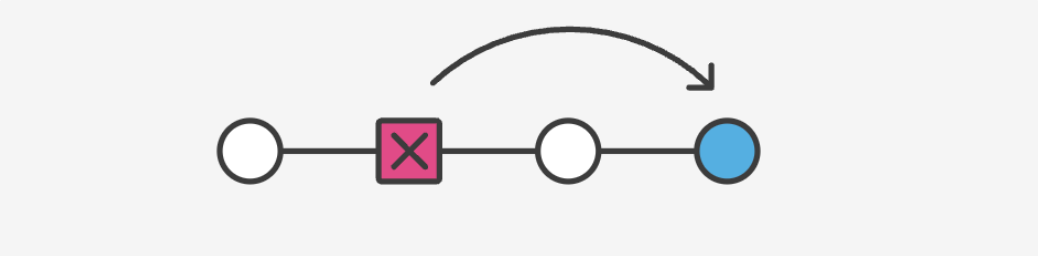
Now that you know the basics of stashing, there is one caveat with `git stash` you need to be aware of: by default Git *won't* stash changes made to untracked or ignored files.

★ Objective 4: Reverting changes

The `git revert` command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse

content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

Reverting should be used when you want to apply the inverse of a commit from your project history. This can be useful, for example, if you're tracking down a bug and find that it was introduced by a single commit. Instead of manually going in, fixing it, and committing a new snapshot, you can use `git revert` to automatically do all of this for you.

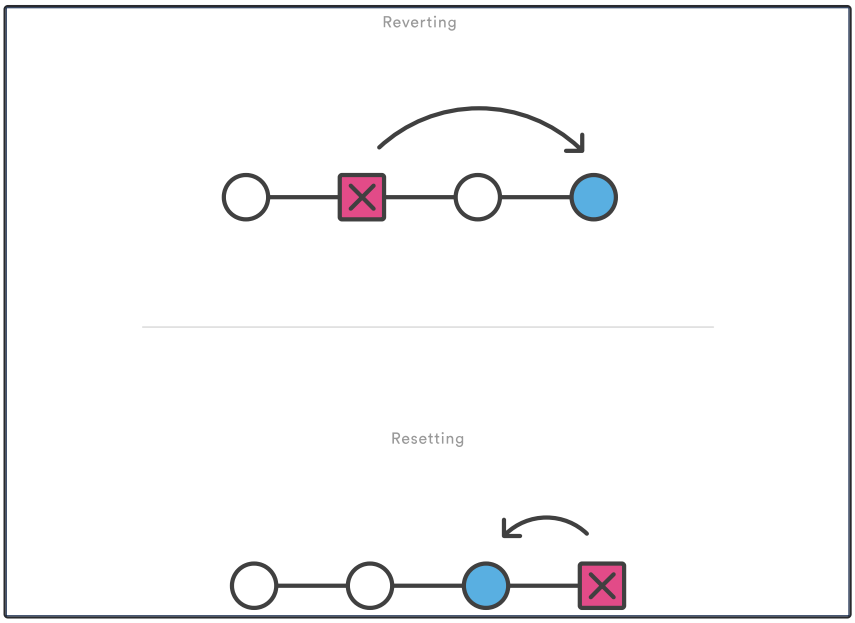


How it works

The `git revert` command is used for undoing changes to a repository's commit history. Other 'undo' commands like, `git checkout` and `git reset`, move the `HEAD` and branch ref pointers to a specified commit. `Git revert` also takes a specified commit, however, `git revert` does not move ref pointers to this commit. A revert operation will take the specified commit, inverse the changes from that commit, and create a new "revert commit". The ref pointers are then updated to point at the new revert commit making it the tip of the branch.

Resetting vs. Reverting

It's important to understand that `git revert` undoes a single commit—it does not "revert" back to the previous state of a project by removing all subsequent commits. In Git, this is actually called a reset, not a revert.



Revert

- **Purpose**: `git revert` creates a new commit that undoes the changes of a specified previous commit. This approach maintains the integrity of the commit history.
- **When to Use**:
  - Use `git revert` when you need to undo changes on a public branch or after pushing commits, as it preserves the history by adding a new commit that negates the previous one
  - It is safer for collaborative projects because it does not alter existing commits

Example

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

# Revert the last commit
git revert HEAD
```

Reset

- **Purpose**: `git reset` is used to move the branch pointer to a previous commit, effectively discarding any commits that were made after that point. This command can modify the commit history.
- **Types**:
  - **Soft**: Moves the branch pointer to a previous commit but keeps all changes staged.
  - **Hard**: Moves the branch pointer and deletes all changes from the working directory.
- **When to Use**:
  - Use `git reset` when you want to permanently remove commits from your local history, especially on private branches where you are the only contributor
  - Avoid using it on public branches or after pushing changes, as it rewrites history and can cause issues for collaborators

Example

```
# Move back to a previous commit, keeping changes staged
git reset --soft HEAD~1

# Move back and delete changes
git reset --hard HEAD~1
```

Reverting has two important advantages over resetting. First, it doesn't change the project history, which makes it a "safe" operation for commits that have already been published to a shared repository. For details about why altering shared history is dangerous, please see the git reset page.

Second, `git revert` is able to target an individual commit at an arbitrary point in the history, whereas `git reset` can only work backward from the current commit.

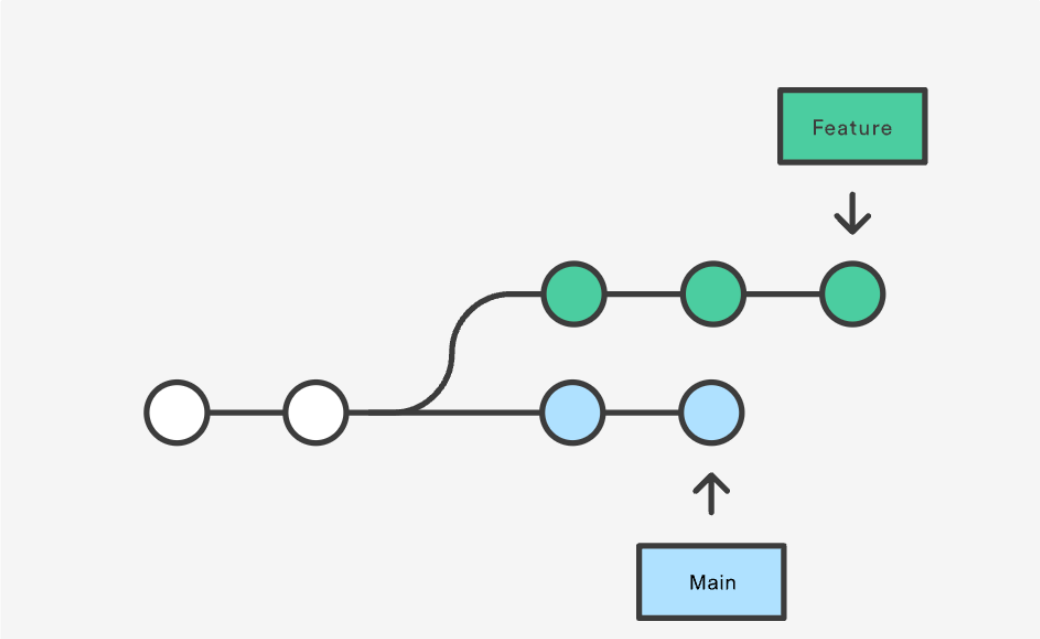
For example, if you wanted to undo an old commit with `git reset`, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits. Needless to say, this is not an elegant undo solution. For a more detailed discussion on the differences between `git revert` and other 'undo' commands see Resetting, Checking Out and Reverting.

In summary, choose `git reset` for private, local changes and `git revert` for public or shared branches to maintain a consistent project history.

★ Objective 5: Rebasing

The first thing to understand about `git rebase` is that it solves the same problem as `git merge`. Both of these commands are designed to integrate changes from one branch into another branch—they just do it in very different ways.

Consider what happens when you start working on a new feature in a dedicated branch, then another team member updates the `main` branch with new commits. This results in a forked history, which should be familiar to anyone who has used Git as a collaboration tool.



Now, let's say that the new commits in `main` are relevant to the feature that you're working on. To incorporate the new commits into your `feature` branch, you have two options: merging or rebasing.

The Merge Option

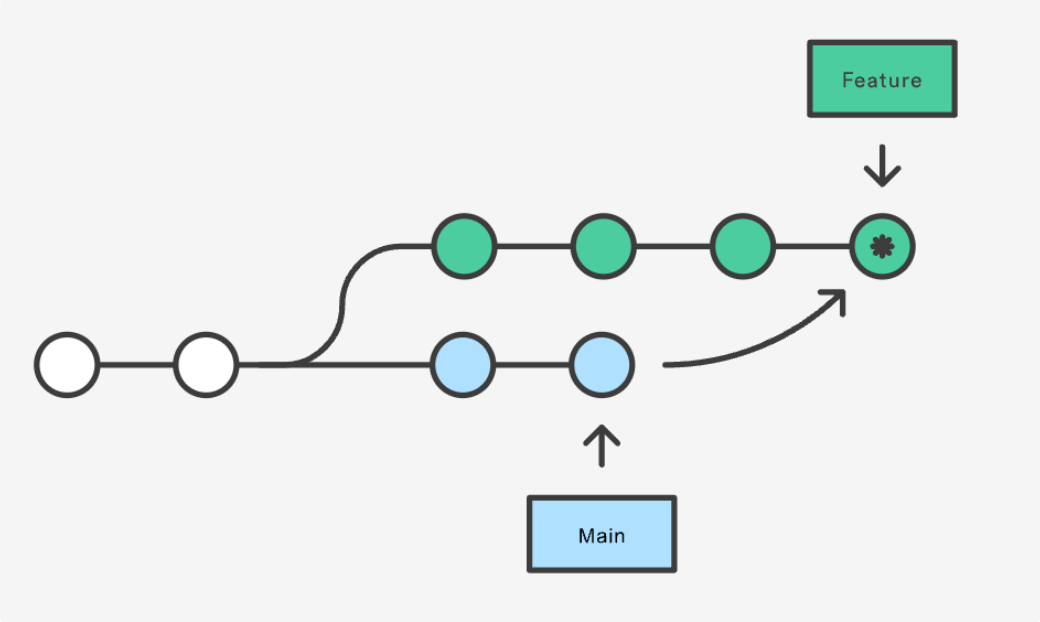
The easiest option is to merge the `main` branch into the feature branch using something like the following:

```
git checkout feature
git merge main
```

Or, you can condense this to a one-liner:

```
git merge feature main
```

This creates a new "merge commit" in the `feature` branch that ties together the histories of both branches, giving you a branch structure that looks like this:



Merging is nice because it's a *non-destructive* operation. The existing branches are not changed in any way. This avoids all of the potential pitfalls of rebasing (discussed below).

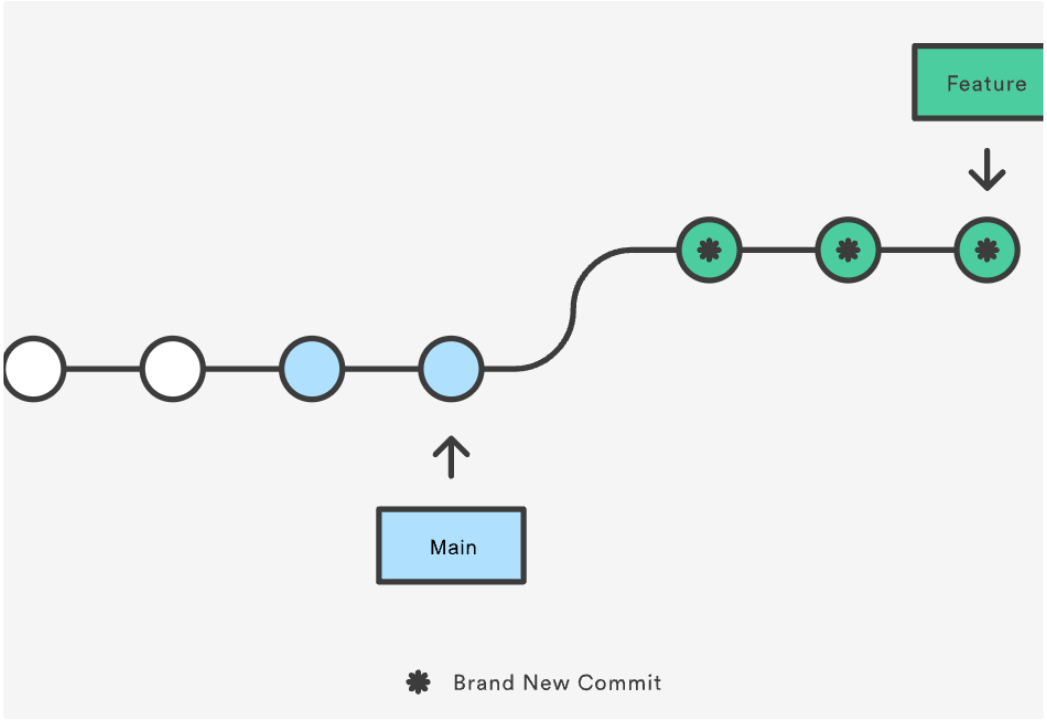
On the other hand, this also means that the `feature` branch will have an extraneous merge commit every time you need to incorporate upstream changes. If `main` is very active, this can pollute your feature branch's history quite a bit. While it's possible to mitigate this issue with advanced `git log` options, it can make it hard for other developers to understand the history of the project.

The Rebase Option

As an alternative to merging, you can rebase the `feature` branch onto `main` branch using the following commands:

```
git checkout feature
git rebase main
```

This moves the entire `feature` branch to begin on the tip of the `main` branch, effectively incorporating all of the new commits in `main`. But, instead of using a merge commit, rebasing *re-writes* the project history by creating brand new commits for each commit in the original branch.



The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by `git merge`. Second, as you can see in the above diagram, rebasing also results in a perfectly linear project history—you can follow the tip of `feature` all the way to the beginning of the project without any forks. This makes it easier to navigate your project with commands like `git log`, `git bisect`, and `gitk`.

But, there are two trade-offs for this pristine commit history: safety and traceability. If you don't follow the Golden Rule of Rebasing, re-writing project history can be potentially catastrophic for your collaboration workflow. And, less importantly, rebasing loses the context provided by a merge commit—you can't see when upstream changes were incorporated into the feature.

Interactive Rebasing

Interactive rebasing gives you the opportunity to alter commits as they are moved to the new branch. This is even more powerful than an automated rebase, since it offers complete control over the branch's commit history. Typically, this is used to clean up a messy history before merging a feature branch into `main`.

To begin an interactive rebasing session, pass the `i` option to the `git rebase` command:

```
git checkout feature
git rebase -i main
```

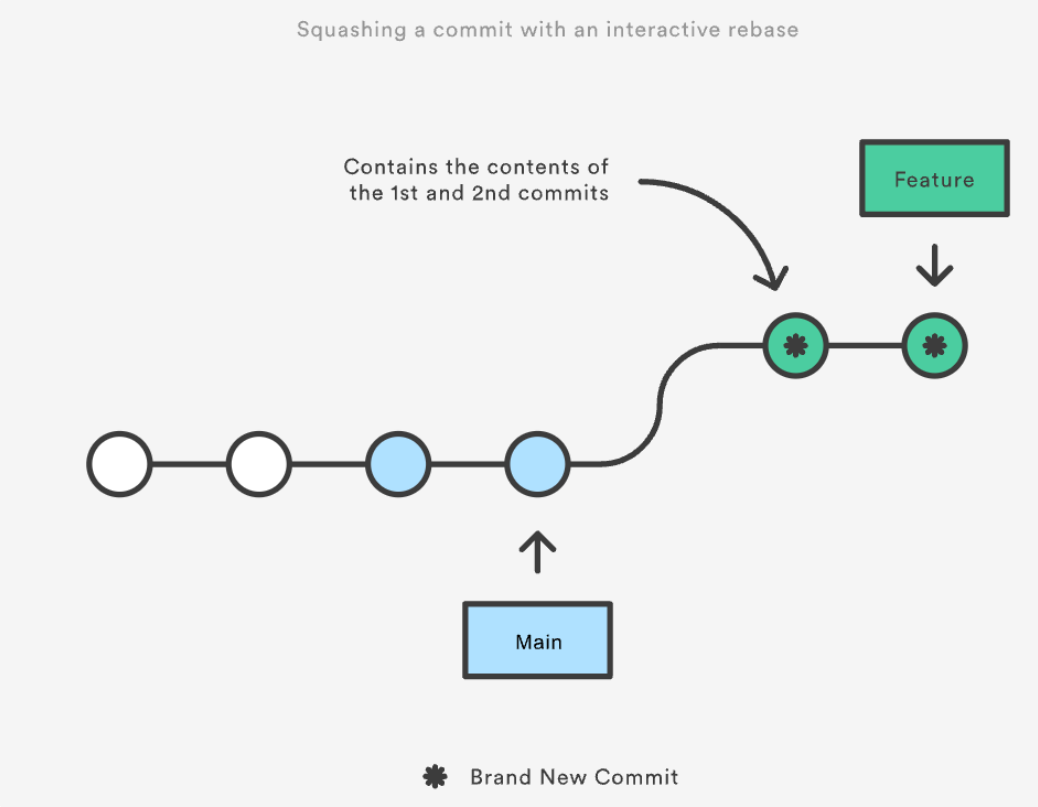
This will open a text editor listing all of the commits that are about to be moved:

```
pick 33d5b7a Message for commit #1
pick 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```

This listing defines exactly what the branch will look like after the rebase is performed. By changing the `pick` command and/or re-ordering the entries, you can make the branch's history look like whatever you want. For example, if the 2nd commit fixes a small problem in the 1st commit, you can condense them into a single commit with the `fixup` command:

```
pick 33d5b7a Message for commit #1
fixup 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```

When you save and close the file, Git will perform the rebase according to your instructions, resulting in project history that looks like the following:

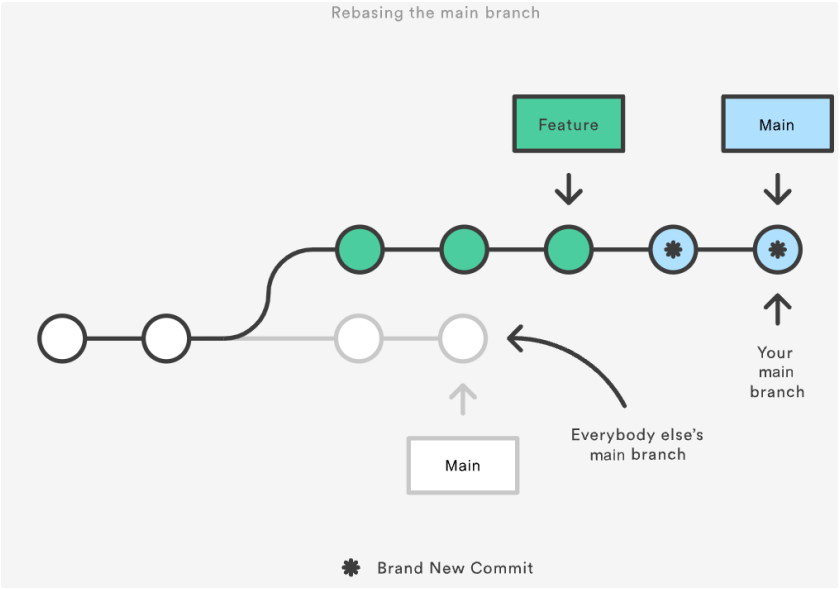


Eliminating insignificant commits like this makes your feature's history much easier to understand. This is something that `git merge` simply cannot do.

The Golden Rule of Rebasing

Once you understand what rebasing is, the most important thing to learn is when *not* to do it. The golden rule of `git rebase` is to never use it on *public* branches.

For example, think about what would happen if you rebased `main` onto your `feature` branch:



The rebase moves all of the commits in `main` onto the tip of `feature`. The problem is that this only happened in *your* repository. All of the other developers are still working with the original `main`. Since rebasing results in brand new commits, Git will think that your `main` branch's history has diverged from everybody else's.

The only way to synchronize the two `main` branches is to merge them back together, resulting in an extra merge commit *and* two sets of commits that contain the same changes (the original ones, and the ones from your rebased branch). Needless to say, this is a very confusing situation.

So, before you run `git rebase`, always ask yourself, “Is anyone else looking at this branch?” If the answer is yes, take your hands off the keyboard and start thinking about a non-destructive way to make your changes (e.g., the `git revert` command). Otherwise, you’re safe to re-write history as much as you like.
