**BASIC GIT COMMAND**

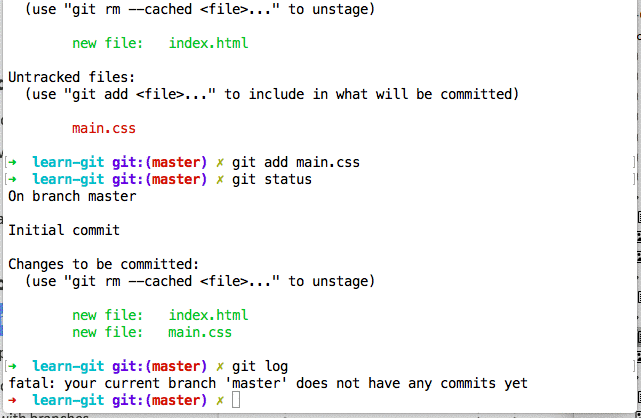
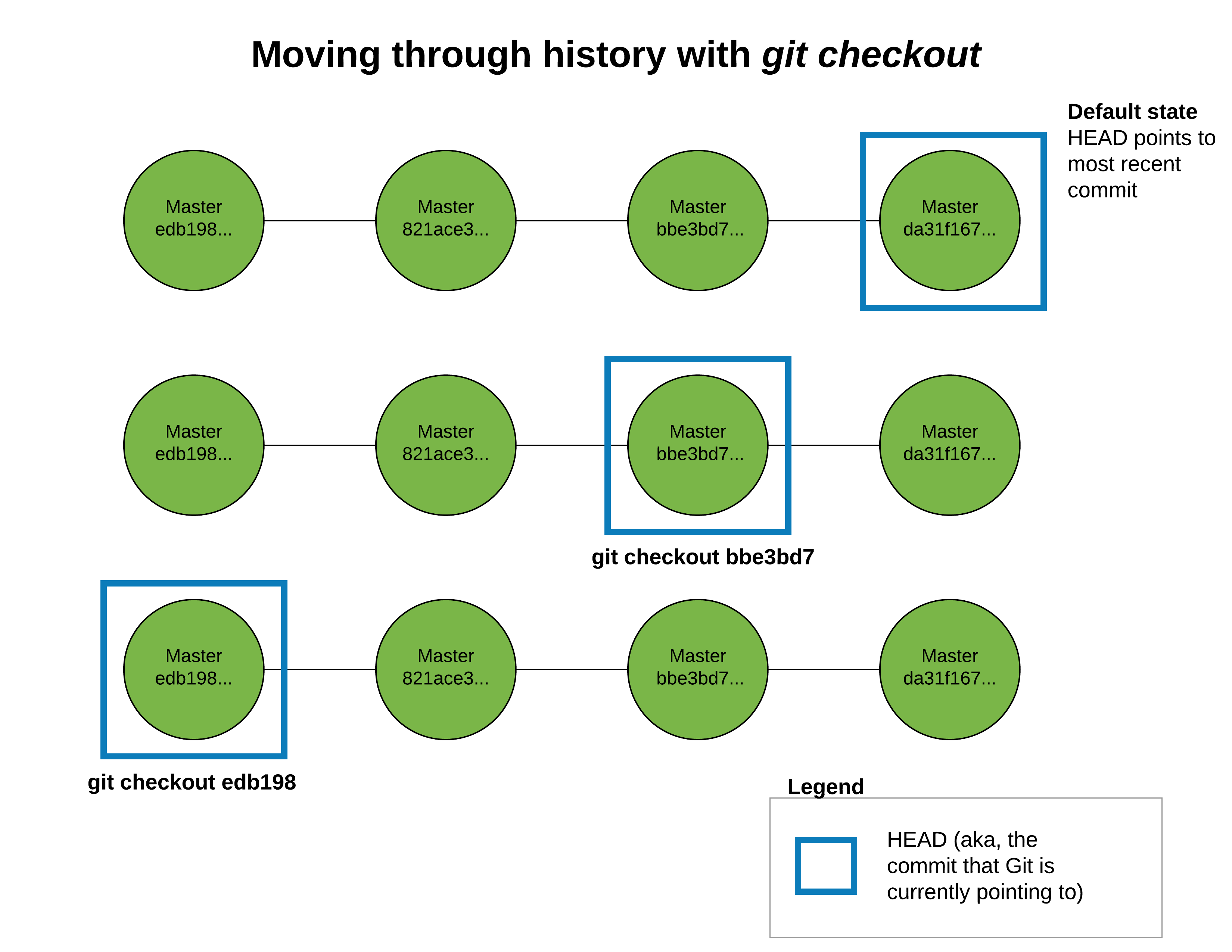
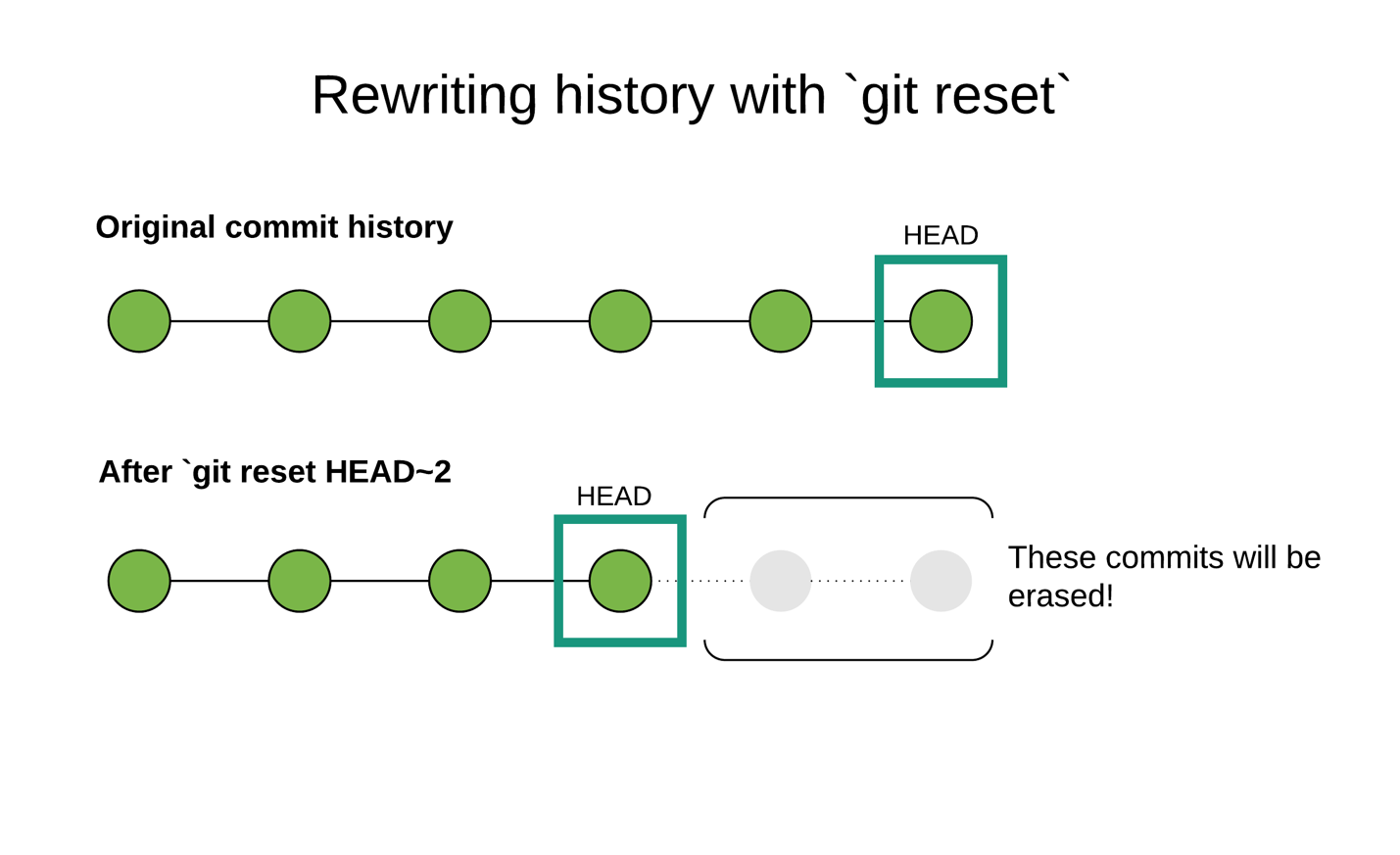
We'll cover:

* git init - used to initialize a new Git repository
* git status - used to find out the current state of the repository (aka, repo, for short)
* git add - used to *stage* new or changed files. We'll discuss the idea of staging changes in a moment.
* git commit - used to take a snapshot of the repo at a point in time.
* git diff -- used to see what has changed in a repo since the last commit
* git reset -- used to reset your repository to a prior state.
* git checkout -- used to look at a prior state of the repository

With these commands, you can take snapshots of your code and, if you need to, go back to an earlier state.

Be sure to follow along and enter the commands we discuss to begin memorizing them.

**Key Terms**

* Initialize
* Git repository
* Commit
* Stage
* Snapshot
* **From new project, to new repository, to initial commit**
* From the command line, navigate to your projects folder. Inside that directory, create a new sub-directory called learn-git and move into it:
* *// <tab> means you should practice using tab completion!*
* **cd** **pro**<**tab**> *// move into projects folder*
* **mkdir** learn-git
* **cd** lea<**tab**> *// move into learn-git folder*
* Next, we're going to create two files: index.html and main.css. From the same command line console, run the command touch index.html main.css to initially create these files. At this point, they'll be empty.
* Now that we've got some files, we're ready to *initialize* our project as a Git repository. All that really means is that we're telling Git that we want to use its version control system in this folder. Once initialized, Git will be able to tell us things like which files it's currently tracking (that is, which ones it pays attention to when it takes snapshots) and which ones have changed since a previous commit.
* From the same command line window, run the command git init. You should see a message like Initialized empty Git repository in /Users/sallystudent/projects/learn-git/.git/. Your message will vary according to your user name and the location of your project directory.
* You can confirm what this log says by running ls -a. You should see that alongside index.html and main.css, we also have an invisible folder, .git. If you're curious, you can run ls .git to see the contents of that folder. Know that Git adds this folder and these files, but that 99% of the time, you won't ever need to touch them. Indeed, unless you have a specific reason to alter these files and you know exactly what you're doing, you should avoid modifying them, as it may mess up your Git repository.
* **What's the scenario? git status knows**
* In the previous step, we created a new repository, but we didn't add or commit either of our files (index.html and main.css). In other words, we haven't yet taken a snapshot. We'll do that in a moment, but how can we tell what the status of our repository is?
* That's the purpose of the aptly named git status command. git status can be run in any repo, at any point, and you'll get a rundown of which files from previous commits have changed, which files are new and not currently tracked, and which files have been moved or deleted.
* To see for yourself, run git status from the same command line window as earlier. You should get back something like this:
* On branch master
* Initial **commit**
* Untracked files:
* (**use** "git add <file>..." **to** **include** **in** what will be committed)
* index.html
* main.css
* **nothing** added **to** **commit** but untracked files **present** (**use** "git add" **to** track)
* That message tells us a few things. First, we can see that we're on the master branch. We'll discuss branches in detail in the next assignment, but for now, recall from the previous assignment that the master branch is the default branch. When you initialize a new Git repo, it will have a single branch: master.
* Next, note that Git tells us that we have *untracked files* in our repository, and it specifically lists index.html and main.css.
* So what is an untracked file? To understand this, you need to understand *how* Git takes snapshots of your project. If you were to point a camera at a well-lit room and snap a photo, it would record an image of everything in the room.
* With Git, snapshots work a little differently. With Git, you have to explicitly tell it to start *tracking* files (or objects in the room, to follow our photographic metaphor). It's like having a camera where you can say, "When you take a picture of this room, record the sofa and the coffee-table, but not the herb garden (which is a work in progress I don't want you to record just yet)".
* In short, Git only tracks the files within a repo that you tell it to track. So how do we start tracking files? The answer to that is also in the status message from above:
* **nothing** added **to** **commit** but untracked files present (use "git add" **to** track)
* To start tracking files, we need to use the git add command.
* Back on the command line, run the following commands:
* git **add** index.html
* git status
* You should get back something that looks like this:
* On branch master
* Initial **commit**
* Changes **to** be committed:
* (**use** "git rm --cached <file>..." **to** unstage)
* **new** **file**: index.html
* Untracked files:
* (**use** "git add <file>..." **to** **include** **in** what will be committed)
* main.css
* git status is now telling us that we have a set of changes that are *staged* to be committed. We'll circle back in a moment to nail down exactly what we mean by "staged", but for now, know that if we call git commit, any and all changes indicated in the "Changes to be committed" part of the message will be reflected in that commit.
* Note that the untracked files part of the message now only has main.css in it. We just told Git to start tracking index.html but not main.css so that's why it's telling us that main.css is not tracked.
* Now run:
* git **add** main.css
* git status
* This time you should get back a log that looks like this:
* On branch master
* Initial **commit**
* Changes **to** be committed:
* (**use** "git rm --cached <file>..." **to** unstage)
* **new** **file**: index.html
* **new** **file**: main.css
* In the "Changes to be committed" section, Git is telling us that there are two new files that are staged to be committed. Notice how the "Untracked files" section has been omitted this time when we run git status. That's because there are no longer any untracked files in the repo. We've told Git about all of the files in our project folder at this point.
* Next, let's run git log. This command gives you a list of all the commits (or snapshots) of your code that have been made.
* Uh oh! When you run git log, you should get back something that looks like this:
* fatal: your **current** branch 'master' does **not** have **any** commits yet
* Git is telling us we don't have any commits yet. That's because, indeed, we haven't told Git to *commit* our changes yet. At this point, we have only *staged* them.
* We're now ready to discuss *tracking* vs. *staging* vs. *committing*. When we say that Git is *tracking* a file, we mean that Git's version control system knows about a file and is watching it. If a tracked file has been included in a commit, Git will be able to tell us how it's changed since the last commit and the present moment.
* Files are *tracked* by Git, while changes are *staged*. When you *stage* a change to a file, that's like telling Git, "Hey pay attention to the (new) state of this file at this particular moment. If I call git commit right now, I want you to take a snapshot of this file at this particular moment." You can stage changes to multiple files, and even multiple changes to the same file, all before ever calling commit. The key is that the snapshot of your code doesn't happen until you finally call git commit.
* That brings us to *committing*. With Git, you commit a set of staged changes. Let's go ahead and commit our code so far. From the command line, run the command git commit.
* This will cause an interactive commit dialog to open up in your command line terminal. By default, this interface is not in editing mode, and what we need to do is type a commit message. Type : (colon) and then i, hit enter, and you'll be put in "insert" mode, which will allow you to edit the commit message. Type "initial commit" and hit enter. Then, to save your commit message and exit out of the commit prompt, hit the esc key to exit insert mode and then type :wq, which stands for write and quit, and hit enter.
* The prompt should close, and you should see a log that looks like this:
* [master (root-**commit**) a462979] **initial** **commit**
* 2 files **changed**, 0 insertions(+), 0 deletions(-)
* **create** **mode** 100644 index.html
* **create** **mode** 100644 main.css
* This log has information about what happened in the commit. It tells us the branch the commit happened on (master), the SHA (Secure Hash Algorithm) of the commit (which is a unique identifier that Git assigns to each commit). Git also tells us that two files were changed and that it "created" index.html and main.css. The idea here is that this is the first time these files had their snapshot taken by Git.
* Now that you have made a commit, run git log again. This time, you'll get an interactive prompt that lists all the commits that have happened for this repo (in this case, just the one). Notice that in the log, you can see the SHA of the commit, its message, when the commit was made, and who made it (this last part will correspond to the user name and email you set up when we configured Git in the previous assignment).
* Finally, run git status one more time. This time you should see:
* **On** branch master
* **nothing** **to** **commit**, working tree clean
* This message is telling us that the master branch is "clean". That means that there's a 1-to-1 correspondence between Git's most recent snapshot of the repository and its current state. Git knows about all the files in the repository (i.e., they're all tracked), and its snapshot of each file is up to date.
* 
* **Subsequent snapshots**
* Once you've got Git tracking files and you've made an initial commit, as you make additional changes, you'll keep using the git status, git add and git commit commands to save your work as you go.
* Let's make some changes to our project. Open index.html in your text editor of choice, and copy/paste/save the following HTML:
* <!DOCTYPE html>
* <html lang="en">
* <head>
* <title>learn git</title>
* <link rel="stylesheet" type="text/css" href="main.css">
* </head>
* <body>
* <p>This is here so I can learn Git</p>
* </body>
* </html>
* Next, open main.css and copy/paste/save the following code in that file:
* body {
* background-color: green;
* }
* Recall that when we initially created and then committed index.html and main.css, they were empty. Now we've added some content to both files.
* What does Git have to think about all this? To find out, run git status, and you'll see something like this:
* On branch master
* 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: index.html
* modified: main.css
* Our status message is telling us that we're still on the master branch but that we have some *unstaged* changes. Git knows all about index.html and main.css because we started tracking those files in our first commit. But since that commit, those files have changed, and they look different than Git's most recent snapshot of them.
* To stage these changes, we use the same git add command that we initially used to start tracking these files. This time, we'll use the -u flag, which causes Git to stage any files that it's already tracking but that have unstaged changes. Run the following commands:
* git **add** -u
* git status
* Now git status should give you back this message:
* **On** branch master
* Changes **to** be committed:
* (**use** "git reset HEAD <file>..." **to** unstage)
* modified: index.html
* modified: main.css
* We no longer have unstaged changes, and if we make a commit right now, we'll get a snapshot of the HTML and CSS we pasted into those two files.
* Before making another commit, let's add an additional change to main.css. In your text editor, add the following code to the top of the file, then save:
* \* {
* box-sizing: border-box;
* }
* Run git status again, and you should see:
* On branch master
* Changes to be committed:
* (**use** "git reset HEAD <file>..." **to** unstage)
* modified: index.html
* modified: main.css
* 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: main.css
* What this message is telling us is that our initial changes to index.html and main.css are staged. Git knows about them, and if we run git commit right now, we'll get those changes in our snapshot.
* However, it's also telling us that there are *unstaged* changes since the last time we ran git add -u. That's because of the rule we just added to main.css.
* We can stage these new changes the same way as before, by either running git add -u or git add main.css. Run one of those commands and then git status, and you should see the following message:
* **On** branch master
* Changes **to** be committed:
* (**use** "git reset HEAD <file>..." **to** unstage)
* modified: index.html
* modified: main.css
* This is telling us that if we call git commit our snapshot will be up to date with the current state of index.html and main.css.
* Let's commit our changes. This time, though, we're going to use the -m flag, which allows you to supply a commit message inline with the git commit command.
* Run git commit -m 'added content to files', and you should get back something like this:
* [master 1c2febb] added content to files
* 2 files changed, 17 insertions(+)
* Like with our previous commit, we are told the branch and SHA of the commit, and we can see the commit message. We also get data about how many files were changed. Git is able to tell when lines of code are added, changed, or removed (adds and changes correspond to "insertions" in Git speak, and removal to "deletions").
* Run git log again, and this time you should see info about both commits we've made, with the most recent one at the top. Type :q to exit out of this prompt if it does not quit automatically.
* If you run git status, you'll see the same message from before indicating that the master branch is clean.
* **What's changed? git diff knows.**
* The git diff can be used to understand what has changed about tracked files. To see how it works, let's make some changes to index.html.
* Replace the current body elements with the following two paragraphs and save:
* <p>This is here so I can learn Git. Now it's modified</p>
* <p>This is a new paragraph</p>
* Let's also make a change to main.css. Edit the style rule for body so it looks like this and then save:
* body {
* background-color: green;
* font-size: 24px;
* }
* Like before, if you run git status, Git will tell you that there are unstaged changes to index.html and main.css.
* But what if we need to know precisely what has changed? This is where git diff comes in. With git diff, we can see the changes made to a specific file, to all tracked files in the repo, or get an overview of the changes.
* To see changes to a specific file, you run git diff path-to-changed-file. Let's try that by running git diff index.html. That will bring up an interactive prompt that has something like the following information:
* diff *--git a/index.html b/index.html*
* index 418bd80..c07bcf1 100644
* *--- a/index.html*
* +++ b/index.html
* @@ -5,6 +5,7 @@
* <link rel="stylesheet" **type**="text/css" href="main.css">
* </head>
* <**body**>
* - <p>This **is** here so I can learn Git</p>
* + <p>This **is** here so I can learn Git. Now it's modified</p>
* + <p>This **is** a **new** paragraph</p>
* </**body**>
* </html>
* \ No newline **at** **end** **of** file
* (**END**)
* This message tells us precisely which lines of code have changed in index.html since our most recent commit. The original <p>This is here so I can learn Git</p> has been removed, as indicated by the - sign. Git sees that in its place, two insertions have been made:
* + <p>This **is** here so I can learn Git. Now it*'s modified</p>*
* + <p>This **is** a **new** paragraph</p>
* These are precisely the changes we made to our index.html file. Again, type :q to quit out of the prompt if it doesn't automatically return to the console.
* If we wanted to see the "diff" of all changed files in the repo, we can run git diff without specifying a path. Try that, and you should get an output that looks like this:
* diff --git a/index.html b/index.html
* index 418bd80..c07bcf1 100644
* *--- a/index.html*
* *+++ b/index.html*
* @@ -5,6 +5,7 @@
* <link rel="stylesheet" type="text/css" href="main.css">
* </head>
* <body>
* - <p>This is here so I can learn Git</p>
* + <p>This is here so I can learn Git. Now it's modified</p>
* + <p>This is a new paragraph</p>
* </body>
* </html>
* \ No newline at end of file
* diff --git a/main.css b/main.css
* index 366e4d6..81b9e81 100644
* *--- a/main.css*
* *+++ b/main.css*
* @@ -4,4 +4,5 @@
* body {
* background-color: green;
* + font-size: 24px;
* }
* Now we get the same diff we saw for index.html just a moment ago, followed by the diff for main.css. Note that depending on the size of your command line window when you run this, you may need to use the down arrows to see the entirety of the diff. As before, type :q to quit out of this prompt if necessary.
* Sometimes you just want to see a high-level view of unstaged changes. For that you can use git diff --stat. Run that command and you should see something like:
* index.html | 3 ++-
* main.css | 1 +
* 2 files changed, 3 insertions(+), 1 deletion(-)
* (END)
* This message is telling you that there were 3 total changes to index.html: two insertions (++) and one deletion (-). main.css had one insertion. We also see that "2 files changed" and that there were a total of "3 insertions(+)" and "1 deletion(-)". With the --stat flag we aren't able to see which lines in particular changed, but we can get a high-level view.
* It's important to note that git diff will tell you the difference between the most recent commit and *unstaged changes*. Once your changes are staged, they won't show up in git diff. To verify this, run git add index.html, followed by git diff --stat. This time, only main.css will show up as having a diff.
* Run git add main.css to stage these outstanding changes, and then git commit -m 'updated html and css' to commit them.
* **Unstaging changes**
* Imagine you're working on a project and you make changes to a file, stage them, then stage changes to a bunch of other files, only to realize that you don't actually want to commit the changes you made to the first file. What do you do?
* **"Time travel" with git checkout**
* Git's ability to take snapshots of your code is only valuable insofar as you can "time travel" between snapshots.
* The first "time-travely" command we'll consider is git checkout. In the next reading, you'll learn all about *branches* which are different versions of the same repository. git checkout is used to move between different branches, but it can also be used to have a look at earlier snapshots of your repository. This can be valuable when you're working on larger projects with long commit histories, and you want to know how some part of your code worked in the past (perhaps because you want to return to that state, or perhaps because you're working on some new project, and the way you initially implemented a feature in this previous project is exactly what you want to do in the next project -- you just can't remember how you did it!).
* Crucially, git checkout *doesn't rewrite history*, unlike the git reset command, which we'll get to in a moment. When you use git checkout to inspect the state of your repository at a different commit, it's like you're paging through a photo-book and looking at different snapshots. When you look at a photo of yourself from ten years ago, it has no effect on a photo taken of you yesterday. The same is true for using git checkout.
* Let's see how it works. Run git log inside your learn-git project folder. You should see something like this:
* commit 477f29291342f378a3266fcf5a7ae5bb95000f83
* Author: Benjamin White <benjamin.e.white1@gmail.com>
* Date: Tue May 9 10:06:38 2017 -0400
* updated html **and** css
* commit 1c2febba9ebdcab9d182e83a47b795a8ac05245f
* Author: Benjamin White <benjamin.e.white1@gmail.com>
* Date: Mon May 8 16:52:51 2017 -0400
* added content to files
* commit a46297929ba8d13af0955bfd310dcb5e468b6594
* Author: Benjamin White <benjamin.e.white1@gmail.com>
* Date: Mon May 8 16:23:06 2017 -0400
* initial commit
* To use git checkout to look at an earlier snapshot, you supply it the SHA of the commit you want to see. Let's have a look at the second commit we made in this repository. Copy the SHA for the "added content to files" commit (1c2feb... above, but yours will be different), and then enter :q to quit out of the git log interface if necessary.
* Now run the command git checkout <paste-the-SHA-you-just-copied>. You should see a log that looks like this:
* Note: checking out '1c2febba9ebdcab9d182e83a47b795a8ac05245f'.
* 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** performing another checkout.
* If you want to create **a** new branch to retain commits you create, you may
* do so (now **or** later) **by** **using** -b **with** **the** checkout **command** **again**. **Example**:
* git checkout -b <new-branch-name>
* HEAD is now **at** 1c2febb... added content to files
* "detached HEAD state" probably sounds like a bad thing, but don't worry: nothing is broken. We'll circle back to what that means in just a moment, but first let's see what happened when we returned to this previous snapshot.
* Open your project folder in your text editor, and have a look at index.html and main.css. You'll find that the most recent changes we made in those files are no longer there. index.html should look like this:
* <!DOCTYPE html>
* <html lang="en">
* <head>
* <title>learn git</title>
* <link rel="stylesheet" type="text/css" href="main.css">
* </head>
* <body>
* <p>This is here so I can learn Git</p>
* </body>
* </html>
* and main.css should look like this:
* \* {
* box-sizing: border-box;
* }
* body {
* background-color: green;
* }
* This is *exactly* how these files looked when we made this commit. To get back to our "normal" state, run git checkout master, which will put you back "in the present" where we started. When you do that, you should see something like this:
* **Previous** HEAD position was 1c2febb... added content to **files**
* Switched to branch 'master'
* That worrisome "detached HEAD" message is gone. Phew!
* But what did it mean? And what is this "HEAD"? In Git, *HEAD* is a variable that points to a commit, by default, the most recent commit. To continue with our photo-book metaphor, *HEAD* is like the page in the photo-book you're currently looking at, and the commit SHA is like the page number in the photo-book.
* 
* The chart above depicts what happens to HEAD when you checkout earlier commits. When you checkout a specific commit, Git points the HEAD variable to that commit. This has the effect of putting Git in the *detached HEAD state*, which basically means that if you were to start making and committing changes to your repo from this state, and then you go back to your main branch (git checkout master), you'd have no way of recovering these changes. If you really do want to checkout an earlier commit of your repo and then create a new version from that point that you can return to, then you need to create a new branch (which we'll cover in the next reading) after checking out the old commit.
* We saw that you can use git checkout <SOME-SHA> to have a look at an earlier snapshot of your repo. You should also be aware of an alternative syntax for this that uses the HEAD variable. Our current project only has three commits in it, and we said that HEAD defaults to the most recent commit. To look at our first commit, we could checkout the SHA for it, or we could run this command (which you should try): git checkout HEAD~2. This command says to look at the snapshot that was two commits earlier than wherever HEAD is currently pointing to.
* Go ahead and run git checkout master to get out of the detached HEAD state.
* **Rewriting history with git reset**
* If you've ever seen the classic eighties film [**Back to the Future**](https://en.wikipedia.org/wiki/Back_to_the_Future) or heard of the [**grandfather paradox**](https://en.wikipedia.org/wiki/Grandfather_paradox), you know that time travel can be *dangerous*. Specifically, things can get messy, hard to think about, and possibly even broken when you go back in time and change something that happened.
* Git gives you the ability to "rewrite history" by using the git reset command, which is the final command we'll explore in this reading. This command should be used with caution because rewriting history is indeed the dangerous part of time travel. That said, there is one good use case for it: undoing commits that *you haven't shared with anyone else*.
* At this point you may be scratching your head about this idea of sharing commits with others since we haven't covered that yet. We'll get to that in a later checkpoint when we discuss GitHub, which is the platform that will allow you to share your repositories (and therefore your commits) with others. For now, know that a "commit you haven't shared with anyone else" basically means a commit that exists on your local computer only and that you haven't pushed up to a GitHub repo.
* So imagine that it's one in the morning and you've had a few too many coffees and are feeling inspired. You somewhat hastily decide that you're going to add a new feature to your app, and because we haven't covered branching yet, you don't know that you should do this work on a separate branch!
* You code away and before you know it, it's five in the morning, the birds are chirping, and you've made fifteen new commits to your repository. You head to bed to catch a few hours of sleep, exhausted, but eager to continue on this feature in the evening.
* Fast forward to the evening: after getting rested and slowly thinking through the feature you've added, you realize it was a horrible idea, and there's no way that feature should be shipped to your end users. At the same time, you've gotten an urgent bug report that you need to fix *right now*. The problem is that if you make new commits to your code and then publish it in production, you'd send out the feature you coded last night, that you now realize was a horrible idea.
* What's a programmer to do? This is where git reset comes in. As with git checkout , you can point git reset to the SHA of a commit or to a number of commits back relative to HEAD. In the scenario we're exploring, you'd run git log and read through your logs to find the SHA of the most recent commit before you started working on your ill-fated feature. Then you would run git reset <the-sha-of-that-commit>.
* 
* The chart above depicts what happens when you use git reset. Recall that git checkout <SHA> points HEAD to a different commit, which we said is like turning to a different page in a photo-book — when you do that, the individual pages don't change.
* In contrast, when you run git reset <SHA>, it's like you turn back to an earlier page in the photo-book and then rip out all the pages that come after. This is why we say that git reset *rewrites history*.
* To get a better grasp of how this works, let's add an additional commit to our learn-git repo. Let's add two empty JavaScript files. Run touch foo.js bar.js. Running git status, you should see foo.js and bar.js appear as new files but no other changes. We can stage both files at once by running git add ., which will stage any unstaged changes (that is, changes to existing files and new files) in the current directory and its children. Next, run git commit -m 'added js files' to commit your changes.
* Now run git log and at the very top you should see the new commit we just made. Now let's pretend that we're in the scenario we just discussed: we've fully developed foo.js and bar.js, but we've realized that was a bad idea, and now we want to rewrite our commit history so it's like we never went down this path.
* To do that, run git reset HEAD~1, which will move us back to one commit earlier than the most recent commit. After entering that command, nothing obvious happens. There are no logs, so we'll have to investigate a bit to see what happened.
* Run git log again, and this time you'll see that the commit we just made ("added js files"). Is no longer in our commit history.
* commit 477f29291342f378a3266fcf5a7ae5bb95000f83
* Author: Benjamin White <benjamin.e.white1@gmail.com>
* Date: Tue May 9 10:06:38 2017 -0400
* updated html **and** css
* commit 1c2febba9ebdcab9d182e83a47b795a8ac05245f
* Author: Benjamin White <benjamin.e.white1@gmail.com>
* Date: Mon May 8 16:52:51 2017 -0400
* added content to files
* commit a46297929ba8d13af0955bfd310dcb5e468b6594
* Author: Benjamin White <benjamin.e.white1@gmail.com>
* Date: Mon May 8 16:23:06 2017 -0400
* initial commit
* (END)
* Next, run git status and you'll see
* On branch master
* Untracked files:
* (**use** "git add <file>..." **to** **include** **in** what will be committed)
* bar.js
* foo.js
* **nothing** added **to** **commit** but untracked files **present** (**use** "git add" **to** track)
* So what happened? Our commit history has been rewritten, but the files we added are still in the repo, and they're *unstaged*. When you run git reset, by default it rewrites your commit history, but it does not change the contents of the directory. Instead, the most recent commit in your history is the one you just reset to, and whatever changes you had made up until that point will appear as unstaged changes.
* Let's see this again with one other example. In your text editor, add a new paragraph to index.html and save it. Then, back in the command line, run git add . to stage the re-added foo.js and bar.js and stage the changes to index.html. Run git commit -m 'added js, updated html'.
* Run git reset HEAD~1 to reset to the previous commit. This time, you'll see the following log:
* Unstaged **changes** after reset:
* M index.html
* While our commit history no longer knows about the new paragraph we created in index.html, Git is aware that the file has unstaged changes. If you run git diff index.html, you'll see that the unstaged change is the paragraph we added.
* Running git status, you'll see that foo.js and bar.js are still in the project folder, but they again appear as untracked files.
* So the default behavior of git reset is to change the commit history, but leave the files in the project themselves unchanged. But what if you want to both rewrite your commit history and reset your files to their earlier state?
* To do this, you need to include the --hard flag when you run git reset. Let's try that out. Run git add . to restage our changes (i.e., the new paragraph in index.html and the new files foo.js and bar.js), and then run git commit -m 'updates to html and js'.
* We're ready to reset our commit history *and* our project folder. Run the command git reset --hard HEAD~1. This time you'll see a message that looks like this:
* HEAD **is** now **at** 477f292 updated html **and** css
* HEAD is now pointing at the previous commit. If you run git log, you'll find that the 'updates to html and js' commit is no longer in the commit history. Finally, if you run, git status, you'll see that the working directory is clean, and if you run ls, you'll see that foo.js and bar.js have been removed from the directory.
* **In summary**
* We've covered a lot of ground in this reading and you may be feeling a bit overwhelmed with all the new commands. That's normal: Git is intrinsically complex, and it takes a while to build confidence with it.
* That said, the key to staying sane with Git is keeping the basic mental model in mind. All the commands we've covered in this reading have to do with taking snapshots of your code, comparing snapshots (vs. other snapshots, or vs. "reality"-aka-the-current-state-of-your-repo) and moving between snapshots.

This is for git checkout ouput from command line; check the portion upward.

souleymanes-MBP:learn-git desolo$ git checkout bcfe0fd7e297df92c6a8cb773164ec974f04bf5e

M index.html

Note: switching to 'bcfe0fd7e297df92c6a8cb773164ec974f04bf5e'.

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.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -c with the switch command. Example:

git switch -c <new-branch-name>

Or undo this operation with:

git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at bcfe0fd addedcontent to file

souleymanes-MBP:learn-git desolo$

Working with branches (6)

**Objective:** By the end of this checkpoint, you can create branches in a Git repo.

In this checkpoint, we'll learn about **branches** in Git. A branch is a parallel version of a repository. For instance, you might create a new branch called feature/signup-form when adding a signup form to a site. The starting point for this branch would be the most recent commit on master. Once you're on your branch, you can add, change, and remove files and commit these changes without affecting master. By that same token, master can change without affecting your feature branch. Then, when you've got a working version of your signup form that you're ready to release to production, you *merge* your feature branch into master and then push that code out to your production server.

In this reading, we'll get you up to speed on the Git commands you need to know to work with branches. Specifically, we'll cover:

* git branch: used to create a new branch
* git checkout: used to switch to a branch
* git merge: used to merge changes from one branch into another

In our discussion of git merge, we'll touch on **merge conflicts**, which arise when code from the branch you want to merge into has changed since you branched out — specifically, code that your feature branch also changed. In this situation, Git can't automatically decide which set of changes should be in the final commit, and you have to decide. It's not uncommon for newcomers to Git to panic when they encounter merge conflicts, thinking they've "broken" their repo. In reality, merge conflicts are just evidence that Git is working correctly, but you do need to understand how to resolve them.

As with the previous checkpoint, you should code along as you read to practice using these new commands.

**Key Terms**

* Branches
* Merge conflicts

## Creating and switching between branches

We're going to continue working with the learn-git project/repo we created in the previous assignment. cd into that directory, and then run git status to verify that it's clean and that there are no unstaged changes.

Let's imagine that we've been tasked with adding a signup form to this project and that the code that is currently in learn-git is live in production. To work on our new feature, we'll create a new branch and do our work there. This is the flow we recommend you follow from now on whenever you're working on a project that uses Git. Reserve master for your production-ready code, and when it's time to add new features or fix existing ones, do your work in a separate branch, then when your work is done, merge it into master.

To create a new branch, there are two options. The first is to use the git branch <branch-name> command. Let's try that. Run the command git branch feature/signup-form. Nothing obvious will happen (aka, there are no logs), but if you run git branch -a, you'll get a list of all the branches in the local repo. When you run that you should see something like this:

feature/signup-**form**

*\* master*

There are now two branches in our repository: master and feature/signup-form. The star next to master indicates that this is the current branch.

To move from master to the new branch, you can use git checkout <branch-name>. Run git checkout feature/signup-form, and you should see the message: Switched to branch 'feature/signup-form'.

So we first created a new branch (git branch feature/signup-form) and then checked it out. That's two steps, and everyone knows that programmers like efficiency gains so let's see how we can do this in a single step.

First, let's go back to master by running git checkout master. Next, let's delete feature/signup-form so we can recreate it using the quicker way. Run git branch -D feature/signup-form and you should see a message that looks like this:

Deleted branch feature/signup-form (was 477f292).

To create this branch and check it out in a single step run git checkout -b feature/signup-form. You'll see a message like this:

Switched to **a** new branch 'feature/signup-form'

When you run git checkout -b <branch-name>, the -b flag allows you to indicate a new branch to be created before switching over to it.

## Merging a feature branch

Inside index.html, add this code to the bottom of the body element:

<form name="signup" action="/signup">

<div>

<label for="username">Username</label>

<input type="text" name="username" required>

</div>

<div>

<label for="password">Password</label>

<input type="password" name="password">

</div>

<input type="submit">

</form>

Save these changes in your text editor, then stage them (git add index.html) and commit them (git commit -m 'finished signup form').

Now if this was a real project, you might have several other commits before you'd be ready to merge this code into master. But for the purposes of learning Git, let's pretend that this is all we needed to do to complete this new feature.

To merge our changes into master, we need to move into master, then merge from feature/signup-form.

Run git checkout master. Before merging the changes from feature/signup-form, let's verify that we'll be getting exactly what we expected. Run git diff feature/signup-form and you'll see the diff between master and the feature branch. As expected, you'll see that from the perspective of master, the difference with feature/signup-form is that master omits the signup form. You may need to quit out of the git diff dialogue with :q (or just hit q if the : is already there).

We're ready to merge. Run git merge feature/signup-form, and you'll see a message like this:

Updating 477f292..aeef5c1

Fast-forward

index.html | 13 +++++++++++++

1 file changed, 13 insertions(+)

This is telling us that our previous commit in master (477f29...) has been updated by merging in the most recent snapshot from feature/signup-form (aeef5c1...). We get a summary of the changes this entailed. In this case, index.html is the only file that changed, and it got 13 lines inserted.

If you run git log, you'll see that there's a new entry at the top:

commit aeef5c13dcbcddc68918f4511e6ba93e0d9a7c47 (HEAD -> master, feature/signup-form)

Author: Benjamin White <benjamin.e.white1@gmail.com>

Date: Tue May 9 14:39:26 2017 -0400

finished signup form

Note that the log makes it look like the commit occurred directly in both the master and feature/signup-form branches — there's no indication that we merged the two.

If you want to retain that information, you need to use the --no-ff flag (standing for "no fast forward"). To try that out, let's first run git reset --hard HEAD~1 to move back to where we were before the merge. Then run git merge --no-ff feature/signup-form, edit the commit message or run :wq to close the commit dialog interface, and then run git log. If you edit the message, you may need to hit esc before :. This time your commit history will look a little different:

commit fed1687118db2d1b743c1fade91c4e41a40cf356 (HEAD -> master)

Merge: 477f292 aeef5c1

Author: Benjamin White <benjamin.e.white1@gmail.com>

Date: Tue May 9 14:44:00 2017 -0400

Merge branch 'feature/signup-form'

commit aeef5c13dcbcddc68918f4511e6ba93e0d9a7c47 (feature/signup-form)

Author: Benjamin White <benjamin.e.white1@gmail.com>

Date: Tue May 9 14:39:26 2017 -0400

finished signup form

*//... other stuff*

Instead of seeing only one new entry in our commit history, we see two: the aeef5c13... commit happened in our feature/signup-form branch, but since we merged that branch into master, we can see any commits we made in feature/signup-form here in master's commit history. We also get a separate commit history item for the merge commit (the one at the top).

We recommend getting in the habit of using the --no-ff flag when merging branches because it makes it easier to see when feature branches were merged when you look at your commit history.

## Handling merge conflicts

The final thing we want to discuss in this reading is merge conflicts. A merge conflict arises when you're merging one branch into another, but the two branches have competing changes. A common scenario would be something like this: It's Monday and you create a new branch you've been tasked with creating by Thursday. On Thursday morning, your work is done. Your feature branch changes the index.html file for the site and adds two new files.

Unbeknownst to you, another developer on your team developed her own new feature Tuesday and merged it into master on Wednesday. Your co-worker's change also alters index.html at precisely the line where you linked to the two new files in your branch.

You get the green light to merge into master, and you run git merge --no-ff feature/my-great-feature only to see a message indicating that the merge failed because of merge conflicts. What do you do?

Well, let's find out! We're going to simulate exactly the scenario we just described. In your learn-git repo, make sure you're in master (git checkout master) and that there are no unstaged changes (git status).

Next, let's create a new branch: git checkout -b feature/foo-js. In this new branch, we're going to create one new JavaScript file and then link to that JavaScript file in index.html. From the command line, run echo "console.log('foo')" >> foo.js. This will create a new file called foo.js in the current directory, with the single JavaScript command console.log('foo') in it. Open index.html in your text editor, and at the bottom of the head element, in the line below the link element, add:

<script type="text/javascript" src="./foo.js"></script>

Save this change, and also **be sure to close index.html in your text editor** (otherwise, you may inadvertently save code from this branch in a different branch we'll be creating in a moment). Run git add index.html, git add foo.js, and git commit -m 'added foo.js' to take a snapshot of these changes.

Now to ensure we'll get a merge conflict we need to go back to our master branch (git checkout master), check out a separate branch, commit changes in it that alter one or more lines that feature/foo-js also alters, then merge those changes back into master.

Run git checkout -b feature/bar-js. Then run echo "console.log('bar')" >> bar.js. After that, re-open index.html in your text editor. You should no longer see the script element for foo.js since feature/bar-js branches from master. On the same line you put the script element for foo.js a moment ago, now put:

<script type="text/javascript" src="./bar.js"></script>

Save this change, then close your text editor. Back on the command line, run git add bar.js, git add index.html, and then git commit -m 'added bar.js'.

At this point we have three branches: master, feature/foo-js, and feature/bar-js. We're going to merge feature/bar-js into master, and then after that merge feature/foo-js, but when we try to do that we'll get a merge conflict.

Run git checkout master to get back in your master branch. Now let's merge feature/bar-js by running git merge --no-ff feature/bar-js. You'll see a message that looks like this:

Merge made by the 'recursive' strategy.

bar.js | 1 +

index.html | 1 +

2 files changed, 2 insertions(+)

create mode 100644 bar.js

This tells you that the bar.js file, as well as the change to index.html, were merged into master.

Now we're ready to try merging feature/foo-js into master. Run git merge --no-ff feature/foo-js, and you'll get a message that looks like this:

Auto-merging **index**.html

**CONFLICT** (content): Merge **conflict** **in** **index**.html

Automatic merge failed; fix conflicts **and** **then** **commit** the result.

This message is telling you that a merge conflict occurred when Git tried to automatically merge the changes to index.html from feature/foo-js into master. The reason this conflict arose is that index.html in master has changed (and changed at precisely the line that feature/foo-js also changed) since we created the feature/foo-js branch as an offshoot of master.

Now we need to manually resolve the merge conflict. To do this, open index.html in your text editor. Inside, you'll find some odd (aka, non-HTML) text that you've probably never seen before:

<!DOCTYPE html>

<html lang="en">

<head>

<title>learn git</title>

<link rel="stylesheet" type="text/css" href="main.css">

<<<<<<< HEAD

<script type="text/javascript" src="./bar.js"></script>

=======

<script type="text/javascript" src="./foo.js"></script>

>>>>>>> feature/foo-js

</head>

Git is telling us that there is a conflict between HEAD (which is the current commit of master) and the most recent commit from feature/foo-js (which we're trying to merge in). Specifically, this conflict occurs between <<<<<<< HEAD and >>>>>>> feature/foo-js. Everything between <<<<<<< HEAD and ======= is what we have in the most recent snapshot of master. And everything between ======= and >>>>>>> feature/foo-js is the conflicting code we have in feature/foo-js.

In this case, what we ultimately want is our index.html to link to both ./bar.js and /foo.js. That means we just need to delete the <<<<<<< HEAD, =======, and >>>>>>> feature/foo-js lines. Go ahead and do that and then save index.html.

Note that depending on the specifics of your merge conflict, you might want to keep both sets of changes, as we have done here, only those from the most recent version of master, or only those from the most recent version of your feature branch.

Also note that it's possible to have multiple merge conflicts in the same file and multiple files with merge conflicts. You'll know you have multiple merge conflicts in the same file if you scroll through it and see more than one block that looks like

// ... other code

If you have conflicts in multiple files, that will be noted in the conflict message that printed to the command line when you first merged the two branches.

Once you've cleaned up the merge conflicts, you need to stage the changes, then complete the merge commit. Run git add index.html to stage the changes we just made to that file. Then run git commit, this time without providing a commit message. When you get taken to the commit interface, it will be pre-populated with a commit message about merging, and you don't need to alter this message. Enter :wq to write and quit. You should see a message that looks like this if all went well:

[**master** **93dc247**] Merge branch 'feature/foo-js'

Congrats! You just resolved your first merge conflict. Moving forward, if you encounter merge conflicts in the wild, first recognize that this is not an indication something has gone wrong with Git or that you've done something wrong. Instead, this is a confirmation that Git is working correctly! Second, carefully inspect the files that Git tells you have conflicts, looking for the <<<<<<< HEAD, =======, and >>>>>>> feature/my-feature markers. When you encounter such blocks, think through which set of changes you ultimately want to see in your project once the merge goes through. Alter the file accordingly to resolve the conflict, save your changes, then stage and commit them.

## In summary

In this reading, we explored how to create branches, move between them, and merge changes from one branch into another. Moving forward, make sure that you get into the practice of developing new features on separate feature branches, and then merging them into master when they're complete and you're ready to deploy your new feature to production.