Node.js Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Define NodeJS as distinct from browser based JavaScript runtimes.
2. Write a program that reads in a dictionary file using node's FS API and reads a line of text from the terminal input. The program should 'spell check' by putting asterisks around every word that is NOT found in the dictionary.

Git Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Use Git to initialize a repo
2. Explain the difference between Git and GitHub
3. Given 'adding to staging', 'committing', and 'pushing to remote', match attributes that apply to each.
4. Use Git to clone an existing repo from GitHub
5. Use Git to push a local commit to a remote branch
6. Use git to make a branch, push it to github, and make a pull request on GitHub to merge it to master
7. Given a git merge conflict, resolve it
8. Match the three types of git reset with appropriate descriptions of the operation.
9. Use Git reset to rollback local-only commits.
10. Identify what the git rebase command does
11. Use git diff to compare a local 'staging' branch and 'master' branch.
12. Use git checkout to check out a specific commit by commit id

A Tale of Two Runtimes: Node.js vs Browser

Lately, we've been alluding to JavaScript running in the web browser. While we are not quite ready to make that transition yet, the authors of JavaScript really only intended their creation to be used in a browser environment when they originally conceived of the language at Netscape in 1995. To prepare for the coming transition to the browser, let's explore some of the differences between Node.js and browser environments.

When you finish this article, you should be able to:

- identify Node.js as an environment that is *distinct*from web browsers
- list several differences between the Node.JS and browser runtimes

Same specification, different implementation

Since JavaScript is a single programming language, you may be wondering why there are *any*differences between Node.js and browsers in the first place. If they are in fact different, why wouldn't we classify them as different programming languages? The answer is complicated, but the key idea is this: even if we just consider browser environments, different browsers themselves can differ wildly because JavaScript is a *specification*. During the rise of the World Wide Web in the 90s, companies competed for dominance (see The First Browser War). As Netscape's "original" JavaScript language rose to prominence along with their browser, other browser companies needed to also support JavaScript to keep their users happy. Imagine if you could only visit pages as they were intended if you used a certain browser. What a horrible experience it would be (we're looking at you Internet Explorer)! As companies "copied" Netscape's original implementation of JavaScript, they sometimes took creative liberty in adding their own features.

In order to ensure a certain level of compatibility across browsers, the European Computer Manufacturers Association (ECMA) defined specifications to standardize the JavaScript language. This specification is known as *ECMAScript*or *ES* for short. This allows competing browsers like Google Chrome, Mozilla Firefox, and Apple Safari to have a healthy level of competition that doesn't compromise the customer experience. So now you know that when people use the term "JavaScript" they are really referring to the core standards set by ECMAScript, although exact implementation details may differ from browser to browser.

The Node.js runtime was released in 2009 when there was a growing need to execute JavaScript in a portable environment, without any browser.

Did you know?*Node.js is built on top of the same JavaScript engine as Google Chrome (V8). Neat.*

Differences between Node.js and browsers

There are many differences between Node.js and browser environments, but many of them are small and inconsequential in practice. For example, in our*Asynchronous*lesson, we noted how Node's setTimeouthas a slightly different return value from a browser's setTimeout. Let's go over a few notable differences between the two environments.

Global vs Window

In the Node.js runtime, the global object is the object where global variables are stored. In browsers, the window object is where global variables are stored. The window also includes properties and methods that deal with drawing things on the screen like images, links, and buttons. Node doesn't need to draw

anything, and so it does not come with such properties. This means that you can't reference `window` in Node.

Most browsers allow you to reference `global` but it is really the same object as `window`.

Document

Browsers have access to a `document` object that contains the HTML of a page that will be rendered to the browser window. There is no `document` in Node.

Location

Browsers have access to a `location` that contains information about the web address being visited in the browser. There is no `location` in Node, since it is not on the web.

Require and module.exports

Node has a predefined `require` function that we can use to import installed modules like `readline`. We can also import and export across our own files using `require` and `module.exports`. For example, say we had two different files, `animals.js` and `cat.js`, that existed in the same directory:

```
// cat.js

const someCat = {
  name: "Sennacy",
  color: "orange",
  age: 3
};

module.exports = someCat;
```

```
// animals.js

const myCat = require("./cat.js");

console.log(myCat.name + " is a great pet!");
```

If we execute `animals.js` in Node, the program would print `'Sennacy is a great pet!'`.

Browsers don't have a notion of a file system so we cannot use `require` or `module.exports` in the same way.

What you've learned

In this reading, we covered a few differences between Node and browser environments, including:

- Node's `global` and browser's `window`
- the browser specific `document` and `location` objects
- the Node specific `require` and `module.exports`

The Ins and Outs of File I/O in Node

We have previously identified some differences between Node.js and browser environments. One difference was the use of `require` to import different node modules. It is often the case that these modules provide functionality that is totally absent in the browser environment. While browsers support deliberate file download or upload to the web, they typically don't support arbitrary file access due to security concerns. Let's explore a node module that allows us to read, write, and otherwise manipulate files on our computer.

When you finish this article, you should be able to use the `fs` module to perform basic read and write operations on local files.

The fs module

Node ships with an fs module that contains methods that allow us to interact with our computer's **F**ile **S**ystem through JavaScript. No additional installations are required; to access this module we can simply `require` it. We recommend that you code along with this reading. Let's begin with a `change-some-files.js` script that imports the module:

```
// change-some-files.js

const fs = require("fs");
```

Similar to what we saw in the `readline` lesson, `require` will return to us a object with many properties that will enable us to do file I/O.

Did you know? *I/O is short for input/output. It's usage is widespread and all the hip tech companies are using it, like appacademy.io.*

The `fs` module contains tons of functionality! Chances are that if there is some operation you need to perform regarding files, the `fs` module supports it. The module also offers both synchronous and asynchronous implementations of these methods. We prefer to not block the thread and so we'll opt for the asynchronous flavors of these methods.

Creating a new file

To create a file, we can use the writeFile method. According to the documentation, there are a few ways to use it. The most straight forward way is:

```
// change-some-file.js

const fs = require("fs");

fs.writeFile("foo.txt", "Hello world!", "utf8", err => {
  if (err) {
    console.log(err);
  }
  console.log("write is complete");
});
```

The code above will create a new file called `foo.txt` in the same directory as our `change-some-file.js` script. It will write the string `'Hello world!'` into that newly created file. The third argument specifies the encoding of the characters. There are different ways to encode characters; UTF-8 is the most common and you'll use this in most scenarios. The fourth argument to `writeFile` is a callback that will be invoked when the write operation is complete. The docs indicate that if there is an error during the operation (such as an invalid encoding argument), an error object will be passed into the callback. This type of error handling is quite common for asynchronous

functions. Like we are used to, since `writeFile` is asynchronous, we need to utilize *callback chaining* if we want to guarantee that commands occur *after* the write is complete or fails.

Beware! If the file name specified to `writeFile` already exists, it will completely overwrite the contents of that file.

We won't be using the `foo.txt` file in the rest of this reading.

Reading existing files

To explore how to read a file, we'll use VSCode to manually create a `poetry.txt` file within the same directory as our `change-some-file.js` script. Be sure to create this if you are following along.

Our `poetry.txt` file will contain the following lines:

```
My code fails
I do not know why
My code works
I do not know why
```

We can use the readFile method to read the contents of this file. The method accepts very similar arguments to `writeFile`, except that the callback may be passed an error object and string containing the file contents. In the snippet below, we have replaced our previous `writeFile` code with `readFile`:

```
// change-some-file.js
const fs = require("fs");

fs.readFile("poetry.txt", "utf8", (err, data) => {
  if (err) {
    console.log(err);
  }
  console.log("THE CONTENTS ARE:");
  console.log(data);
});
```

Running the code above would print the following in the terminal:

```
THE CONTENTS ARE:
My code fails
I do not know why
My code works
I do not know why
```

Success! From here, you can do anything you please with the data read from the file. For example, since `data` is a string, we could split the string on the newline character `\n` to obtain an array of the file's lines:

```
// change-some-file.js
const fs = require("fs");

fs.readFile("poetry.txt", "utf8", (err, data) => {
  if (err) {
    console.log(err);
  }

  let lines = data.split("\n");
  console.log("THE CONTENTS ARE:");
  console.log(lines);
  console.log("The third line is " + lines[2]);
});
```

Running this latest version would yield:

```
THE CONTENTS ARE:
[ 'My code fails',
  'I do not know why',
```

```
  'My code works',
  'I do not know why' ]
The third line is My code works
```

Fancy File I/O

Now that we have an understanding of both `readFile` and `writeFile`, let's use both to accomplish a task. Using the same `poetry.txt` file from before:

```
My code fails
I do not know why
My code works
I do not know why
```

Our goal is to design a program to replace occurrences of the phrase 'do not' with the word 'should'. This is straightforward enough. We can read the contents of the file as a string, manipulate this string, then write this new string back into the file. We'll need to utilize callback chaining in order for this to work since our file I/O is asynchronous:

```
const fs = require("fs");

fs.readFile("poetry.txt", "utf8", (err, data) => {
  if (err) {
    console.log(err);
  }

  let newData = data.split("do not").join("should");

  fs.writeFile("poetry.txt", newData, "utf8", err => {
    if (err) {
      console.log(err);
    }
    console.log("done!");
  });
});
```

Executing the script above will edit the `poetry.txt` file to contain:

```
My code fails
I should know why
My code works
I should know why
```

As a bonus, we might also refactor this code to use named functions for better readability and generality:

```
const fs = require("fs");

function replaceContents(file, oldStr, newStr) {
  fs.readFile(file, "utf8", (err, data) => {
    if (err) {
      console.log(err);
    }
    let newData = data.split(oldStr).join(newStr);
    writeContents(file, newData);
  });
}

function writeContents(file, data) {
  fs.writeFile(file, data, "utf8", err => {
    if (err) {
      console.log(err);
    }
    console.log("done!");
  });
}

replaceContents("poetry.txt", "do not", "should");
```

What you've learned

In this reading we explored the `fs` module. In particular, we:

- learned about basic file I/O via `readFile` and `writeFile`
- utilized callback chaining to edit a file based on its existing content

"Gitting" Started With Git!

Good software is never limited to "right now"! Your code grows and changes over time, and the people who work with it may come and go. How can you be certain you're preserving your code's legacy? *Version control*lets us keep track of your changes over time. You'll discuss version control with *Git*, the tool of choice for modern development teams.

After reading, you'll be able to:

- Explain the role of version control in the programmer's toolkit
- Create and manage a Git repository
- Share and collaborate on Git repositories via Github

A little history

Think back to the dark ages of web development: a world of beeping modems and marquee text. If you were going to build a web application in 1995, how might you have done it? You'd start with an empty directory and add some JavaScript and HTML files. As you made changes, you'd save them directly to your directory. There's no history of the changes you've made, so you'd have to keep excellent notes or have an incredible memory to revert your application to a previous state.

What if you have teammates working with you on this project, or a client who wants to review your work? Now each teammate needs a copy of the project directory, and you need a way to share your work with clients. This results in numerous copies of the same files and a lot of extra work keeping those files in sync. If one file gets out of line, it could spell disaster for the whole project. Yikes!

Instead of suffering from these problems, programmers designed a solution:*Version Control Systems (VCS)*. VCS tools abstract the work of keeping projects and programmers in sync with one another. They provide a shared language with which you can discuss changes to source code. They also allow you to step back in time and review your work. VCS tools save you hours of work each day, so learning to use them is a great investment in your productivity.

Git?

You'll be using Git (pronounced similarly to 'get' in English) as your VCS. Git is the most popular VCS today and provides a good balance of power and ease of use. It was created in 2005 by Linus Torvalds (who you may also recognize as the creator of Linux) to address a number of shortcomings VCS tools of that time had, including speed of code management and the ability to maintain workflow when cut off from a remote server. Git is well-known for being reliable and fast, and it brings with it an important online community you can leverage for sharing your code with a wider audience.

Git basics

Like many disciplines, learning Git is just a matter of learning a new language. You'll cover a lot of new vocabulary in this lesson! Remember that the vocabulary you'll learn will allow you to communicate clearly with other developers worldwide, so it's important to understand the meaning of each term.

It's also important to note that Git is a complex and powerful tool. As such, its documentation and advanced examples may be tough to understand. As your knowledge grows, you may choose to dive deeper into Git. Today, you'll focus

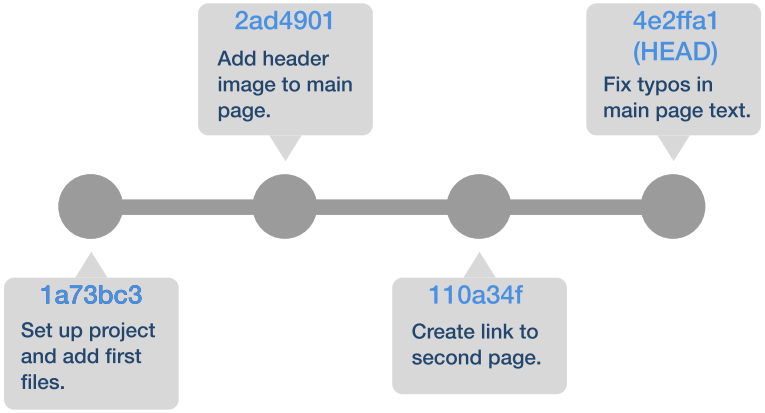
on the commands you'll use every day - possibly for the rest of your programming career! Get comfortable with these commands and resist the urge to copy/paste or create keyboard shortcuts as you're getting started.

See the world through Git's eyes

Before you look at any practical examples, let's talk about how Git works behind the scenes.

Here is your first new word in Git-speak: *repository*, often shortened to *repo*. A Git repo comprises all the source code for a particular project. In the "dark ages" example above, the repo is the first directory you created, where work is saved to, and which acts as the source for code shared to others. Without a repo, Git has nothing to act on.

Git manages your project as a series of *commits*. A commit is a collection of changes grouped towards a shared purpose. By tracking these commits, you can see your project on a timeline instead of only as a finished project:



Notice the notes and seemingly random numbers by each commit? These are referred to as *commit messages*and *commit hashes*, respectively. Git identifies your commits by their hash, a specially-generated series of letters and numbers. You add commit messages to convey meaning and to help humans track your commits, as those hashes aren't very friendly to read!

A Git hash is 40 characters long, but you only need the first few characters to identify which hash you're referring to. By default, Git abbreviates hashes to 7 characters. You'll follow this convention, too.

Git provides a helpful way for us to "alias" a commit in plain English as well. These aliases are called *refs*, short for "references". A special one that Git creates for all repositories is `HEAD`, which references the most recent commit. You'll learn more about creating your own refs when you learn about "branching".

Git maintains three separate lists of changes: the *working directory*, the*staging area*, and the *commit history*. The working directory includes all of your in-progress changes, the staging area is reserved for changes you're ready to commit, and the commit history is made up of changes you've already committed. You'll look more at these three lists soon.

Git only cares about changes that are "tracked". To track a file, you must add it to the commit history. The working directory will always show the changes, even if they aren't tracked. In the commit history, you'll only have a history of files that have been formally tracked by your repository.

Tracking changes in a repository

Now, let's get practical!

You can create a repository with `git init`. Running this command will initialize a new Git repo in your current directory. It's important to remember that you only want a repository for your project and not your whole hard drive, so always run this command inside a project folder and not your home folder

or desktop. You can create a new repo in an empty folder or within a project directory you've already created.

What good is an empty repo? Not much! To add content to your repository, use `git add`. You can pass this command a specific filename, a directory, a "wildcard" to select a series of similarly-named files, or a `.` to add every untracked file in the current directory:

```
# This will add only my_app.js to the repo:
> git add my_app.js

# This will add all the files within ./objects:
> git add objects/

# This will add all the files in the current directory ending in `.js`:
> git add *.js

# This will add everything in your current directory:
> git add .
```

Adding a file (or files) moves them from Git's working directory to the staging area. You can see what's been "staged" and what hasn't by using `git status`:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   index.html

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:   app.js

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        styles.css
```

In this example, "Changes to be committed" is your staging area and "Changes not staged for commit" is your working directory. Notice that you also have "Untracked files", Git's way of reminding us that you may have forgotten to `git add` a file to your repo. Most Git commands will include a bit of help text in the output, so always read the messages carefully before moving forward. Thanks, Git!

Once you're happy with your files and have staged them, you'll use `git commit`to push them into the commit history. It's significantly more work to make changes after a commit, so be sure your files are staged and exactly as you'd like them before running this command. Your default text editor will pop up, and you'll be asked to enter a commit message for this group of changes.

Heads Up: You may find yourself in an unfamiliar place! The default text editor for MacOS (and many variants of Linux) is called *Vim*. Vim is a terminal-based text editor you'll discuss in the near future. It's visually bare and may just look like terminal text to you! If this happens, don't worry - just type `:q` and press your "return" key to exit.

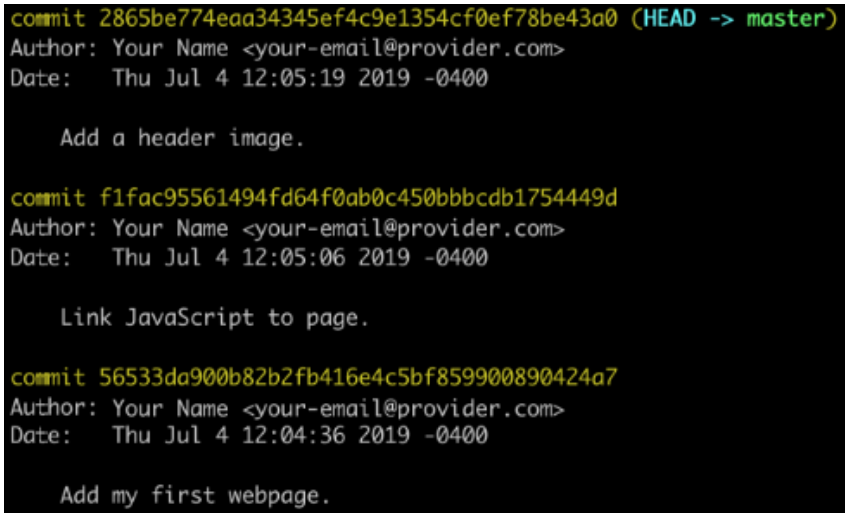
You'll want to ensure that future commit messages open in a more familiar editor. You can run the following commands in your terminal to ensure that Visual Studio Code is your `git commit`editor from now on:

```
> git config --global core.editor "code --wait"
> git config --global -e
```

If you experience any issues, you may be missing Visual Studio Code's command line tools. You can find more details and some troubleshooting tips on Microsoft's official VS Code and macOS documentation.

Once you close your editor, the commit will be added to your repository's commit history. Use `git log` to see this history at any time. This command

will show all the commits in your repository's history, beginning with the most recent:



Like many Git commands, `git commit` includes some helpful shorthand. If you need a rather short commit message, you can use the `-m` flag to include the message inline. Here's an example:

```
> git commit -m "Fix typo"
```

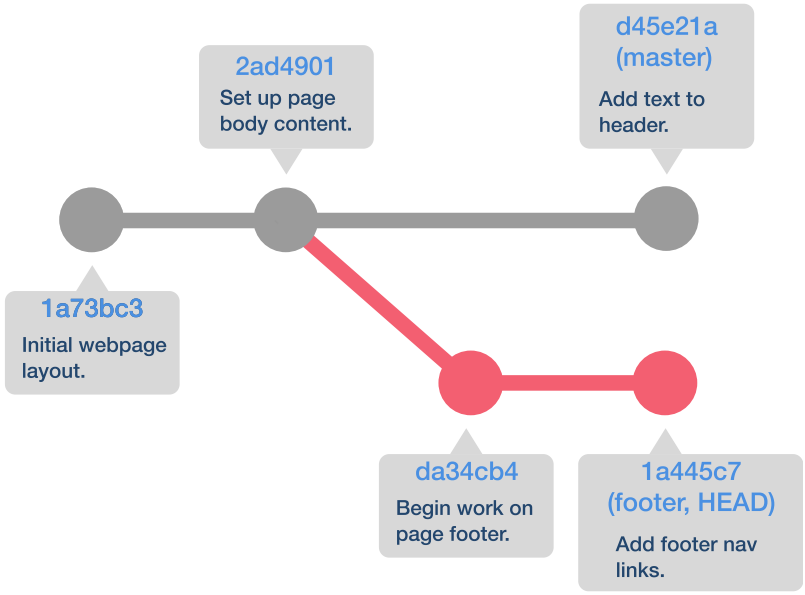
This will commit your changes with the message "Fix typo" and avoid opening your default text editor. Remember the commit messages are how you make your project's history friendly to humans, so don't use the `-m` flag as an excuse to write lame commit messages! A commit message should always explain why changes were made in clear, concise language. It is also best practice to use imperative voice in commit messages (i.e. use "Fix typo" instead of "Fixing the typo" or "Typo was fixed").

Branches and workflow

You've seen what a project looks like with a linear commit history, but that's just scratching the surface of Git's utility. Let's explore a new realm with *branches*. A branch is a separate timeline in Git, reserved for its own changes. You'll use branches to make your own changes independently of others. These branches can then be *merged* back into the main branch at a later time.

Let's consider a common scenario: a school project. It's a lot of extra hassle to schedule time together and argue over exactly what should be done next! Instead, group members will often assign tasks amongst themselves, work independently on their tasks, and reunite to bring it all together as a final report. Git branches let us emulate this workflow for code: you can make a copy of what's been done so far, complete a task on your new branch, and merge that branch back into the shared repository for others to use.

By default, Git repos begin on the `master` branch. To create a new branch, use `git branch <name-of-your-branch>`. This will create a named reference to your current commit and let you add commits without affecting the `master` branch. Here's what a branch looks like:



Notice how your refs help to identify branches here: `master` stays to itself and can have changes added to it independently of your new branch (`footer`). `HEAD`, Git's special ref, follows us around, so you know that in the above diagram you're working on the `footer` branch.

You can create a new branch or visit an existing branch in your repository. This is especially helpful for returning the `master` branch or for projects you've received from teammates. To open an existing branch, use `git checkout <name-of-branch>`.

Bringing it back together

Once you're happy with the code in the branch you've been working on, you'll likely want to integrate the code into the `master` branch. You can do this via `git merge`. Merging will bring the changes in from another branch and integrate them into yours. Here's an example workflow:

```
> git branch my-changes
> git add new-file.js
> git commit -m "Add new file"
> git checkout master
> git merge my-changes
```

Following these steps will integrate the commit from `my-changes` over to `master`. Boom! Now you have your `new-file.js` on your default branch.

As you can imagine, branching can get *very* complicated. Your repository's history may look more like a shrub than a beautiful tree! You'll discuss advanced merging and other options in an upcoming lesson.

Connecting with the world via GitHub

Git can act as a great history tool and source code backup for your local projects, but it can also help you work with a team! Git is classified as a "DVCS", or "Distributed Version Control System". This means it has built-in support for managing code both locally and from a distant source.

You can refer to a repository source that's not local as a *remote*. Your Git repository can have any number of remotes, but you'll usually only have one. By default you'll refer to the primary remote of a repo as the `origin`.

You can add a remote to an existing repository on your computer, or you can retrieve a repository from a remote source. You can refer to this as *cloning* the repo. Once you have a repository with a remote, you can update your local code from the remote by *pulling* code down, and you can *push* up your own code so others have access to it.

Collaboration via Git and GitHub

While a remote Git server can be run anywhere, there are a few places online that have become industry standards for hosting remote Git repositories. The best-known and most widely-used Git source is a website called GitHub. As the name suggests, GitHub is a global hub for Git repositories. It's free to make a Github account, and you'll find literally millions of public repositories you can browse.

GitHub takes a lot of work out of managing remote Git repositories. Instead of having to manage your own server, GitHub provides managed hosting and even includes some helpful graphical tools for complicated tasks like deployment, branch merging, and code review.

Let's look at a typical workflow using Git and GitHub. Imagine it's your first day on the job. How do you get access to your team's codebase? By cloning the repository!

```
> git clone https://github.com/your-team/your-codebase.git
```

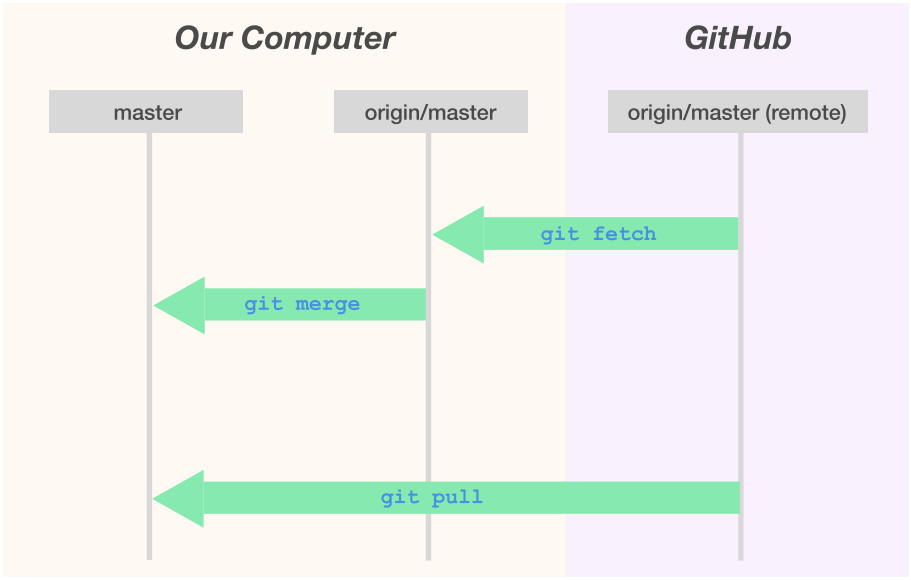
Using the `git clone` command will create a new folder in your current directory named after the repo you're cloning (in this case, `your-codebase`). Inside that folder will be a git repository of your very own, containing the repo's entire commit history. Now you're ready to get started.

You'll likely start on the `master` branch, but remember that this is the default branch and it's unlikely you want to make changes to it. Since you're working on a team now, it's important to think of how your changes to the repository might affect others. The safest way to make changes is to create a new branch, make your changes there, and then push your branch up to the remote repository for review. You'll use the `git push` command to do this. Let's look at an example:

```
> git branch add-my-new-file
> git add my-new-file.js
> git commit -m "Add new file"
> git push -u origin add-my-new-file
```

Notice how you used the `-u` flag with `git push`. This flag, shorthand for `--set-upstream`, lets Git know that you want your local branch to follow a remote branch. You've passed the same name in, so you'll now have two branches in your local repository: `add-my-new-file`, which is where your changes live after being committed, and `origin/add-my-new-file`, which keeps up with your remote branch and updates it after you use `git push`. You only need to use the `-u` flag the first time you push each new branch - Git will know what to do with a simple `git push` from then on.

You now know how to push your changes up, but what about getting the changes your teammates have made? For this, you'll use `git pull`. Pulling from the remote repo will update all of your local branches with the code from each branch's remote counterpart. Behind the scenes, Git is running two separate commands: `git fetch` and `git merge`. Fetching retrieves the repository code and updates any remote tracking branches in your local repo, and merge does just you've already explored: integrates changes into the local branches. Here's a graphic to explain this a little better:



It's important to remember to use `git pull` often. A dynamic team may commit and push code many times during the day, and it's easy to fall behind. The more often you `pull`, the more certain you can be that your own code is based on the "latest and greatest".

Merging your code on GitHub

If you're paying close attention, you may have noticed that there's a missing step in your workflows so far: how do you get your code merged into your default branch? This is done by a process called a *Pull Request*.

A pull request (or "PR") is a feature specific to GitHub, not a feature of Git. It's a safety net to prevent bugs, and it's a critical part of the collaboration workflow. Here's a high-level of overview of how it works:

- You push your code up to GitHub in its own branch.
- You open a pull request against a *base branch*.
- GitHub creates a comparison page just for your code, detailing the changes you've made.
- Other members of the team can review your code for errors.
- You make changes to your branch based on feedback and push the new commits up.
- The PR automatically updates with your changes.
- Once everyone is satisfied, a repo maintainer on GitHub can merge your branch.
- Huzzah! Your code is in the main branch and ready for everyone else to `git pull`.

You'll create and manage your pull requests via GitHub's web portal, instead of the command line. You'll walk through the process of creating, reviewing, and merging a pull request in an upcoming project.

What you've learned

This lesson included lots of new lingo and two new tools you can take advantage of immediately: Git and GitHub.

- You should feel comfortable exploring both of these more on your own.
- You should understand that a VCS is a critical part of your development workflow.
- You able to get started tracking your own projects and sharing code across a team.

If it's still a little hazy, never fear! Git will be a core part of your projects going forward. You'll get lots of practice `pull`ing, `push`ing, and `clone`-ing in the coming months.

Browsing Your Git Repository

Repositories can feel intimidating at first, but it won't take you long to navigate code like you own the place - because you do! Let's discuss a few tools native to Git that help us browse our changes over time.

We'll be covering:

- Comparing changes with `git diff`
- Browsing through our code "checkpoints" with `git checkout`

Seeing changes in real time

Git is all about change tracking, so it makes sense that it would include a utility for visualizing a set of changes. We refer to a list of differences between two files (or the same file over time) as a *diff*, and we can use `git diff` to visualize diffs in our repo!

When run with no extra options, `git diff` will return any tracked changes in our working directory since the last commit. **Tracked** is a key word here; `git diff` won't show us changes to files that haven't been included in our repo via `git add`. This is helpful for seeing what you've changed before committing! Here's an example of a small change:

```
diff --git a/App.js b/App.js
index 0ee3657..fbe8c52 100644
--- a/App.js
+++ b/App.js
@@ -1,3 +1,3 @@
 const myAppOptions = {
-  "primaryColor": "???"
+  "primaryColor": "green"
 }
```

Let's break down some of the new syntax in this output.

- The diff opens with some Git-specific data, including the branch/files we're checking, and some unique hashes that Git uses to track each diff. You can skip past this to get to the important bits.
- `---` & `+++` let us know that there are both additions and subtractions in the file "App.js". A diff doesn't have a concept of inline changes - it treats a single change as removing something old and replacing it with something new.
- `@@` lets us know that we're starting a single "chunk" of the diff. A diff could have multiple chunks for a single file (for example: if you made changes far apart, like the header & footer). The numbers in between tell us how many lines we're seeing and where they start. For example: `@@ +1,3 -1,3 @@` means we'll see three lines of significant content, including both addition & removal, beginning at line one.
- In the code itself, lines that were removed are prefixed with a `-` and lines that were added are prefixed with a `+`. Remember that you won't see these on the same lines. Even if you only changed a few words, Git will still treat it like the whole line was replaced.

Diff options

Remember that, by default, `git diff` compares the current working directory to the last commit. You can compare the staging area instead of the working directory with `git diff --staged`. This is another great way to double-check your work before pushing up to a remote branch.

You're also not limited to your current branch - or even your current commit! You can pass a base & target branch to compare, and you can use some special characters to help you browse faster! Here are a few examples:

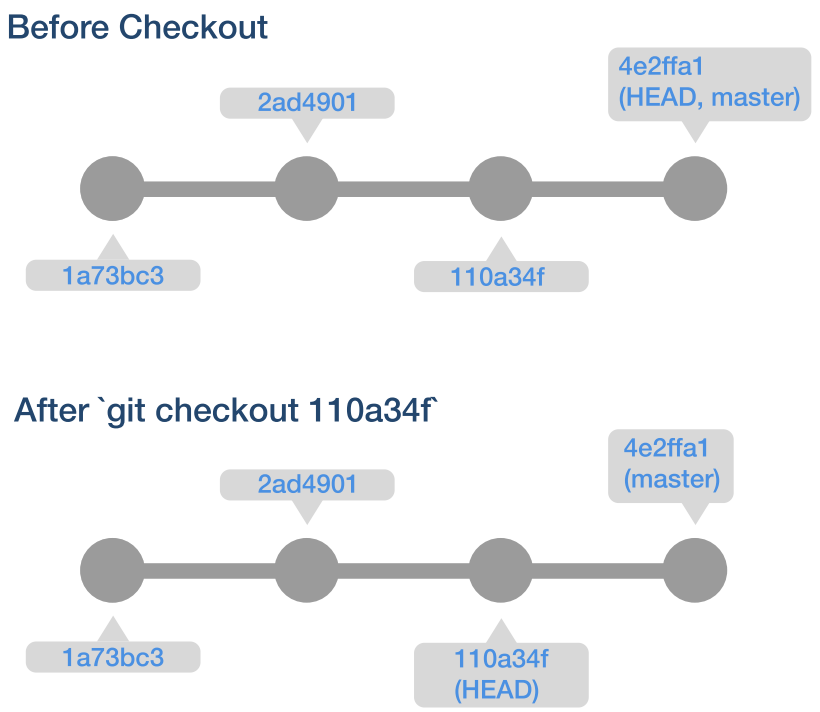
```
# See differences between the 'feature'
# branch and the 'master' branch.
> git diff master feature

# Compare two different commits
> git diff 1fc345a 2e3dff

# Compare a specific file across separate commits
> git diff 1fc345a 2e3dff my-file.js
```

Time travel

`git diff` gives us the opportunity to explore our code's current state, but what if we wanted to see its state at a different point in time? We can use *checkout*! `git checkout` lets us take control of our `HEAD` to bounce around our timeline as we please.

Remember that `HEAD` is a special Git reference that usually follows the latest commit on our current branch. We can use `git checkout` to point our `HEAD` reference at a different commit, letting us travel to any commit in our repository's history. By reading the commit message and exploring the code at the time of the commit, we can see not only what changed but also why it changed! This can be great for debugging a problem or understanding how an app evolved.

Let's look at a diagram to understand what `checkout` does a little better:



Notice that we haven't lost any commits, commit messages, or code changes. Using `git checkout` is entirely non-destructive.

To browse to a different commit, simply pass in a reference or hash for the commit you'd like to explore. `git checkout` also supports a few special characters & reserved references:

```
# You can checkout a branch name.
# You'll be using this particular branch a lot!
```

```
> git checkout master

# You can also use commit hashes directly
> git checkout 7d3e2f1

# Using a hyphen instead of a hash will take
# you to the last branch you checked out
> git checkout -

# You can use "HEAD~N" to move N commits prior
# to the current HEAD
> git checkout HEAD~3
```

Once you're done browsing the repo's history, you can use `git checkout <your-branch-name>` to move `HEAD` back to the front of the line (your most recent commit). For example, in our diagram above, we could use `git checkout master` to take our `HEAD` reference back to commit `42ffa1`.

Why checkout?

Most of Git's power comes from a simple ability: viewing commits in the past and understanding how they connect. This is why mastering the `git checkout` command is so important: it lets you think more like Git and gives you full freedom of navigation without risking damage to the repo's contents.

That said, you'll likely use shortcuts like `git checkout -` far more often than specifically checking out commit hashes. Especially with the advent of user-friendly tools like GitHub, it's much easier to visualize changes outside the command line. We'll demonstrate browsing commit histories on GitHub in a future lesson.

What we've learned

We're building our skill set for navigating code efficiently, and we're starting to got more accustomed to seeing our projects as a series of checkpoints we can review instead of a single point in time.

- You should be able to compare code changes between commits or files.
- You should have confidence when switching between branches or commits.
- You should understand the role of `git checkout` as fundamental to how Git organizes our projects.

Git Do-Overs: Reset & Rebase

Git is designed to protect you - not only from others, but also from yourself! Of course, there are times where you'd like to exercise your own judgement, even if it may not be the best thing to do. For this, Git provides some helpful tools to change commits and "time travel".

Before we talk about these, a warning: **The commands in this lesson are destructive!**If used improperly, you could lose work, damage a teammate's branch, or even rewrite the history of your entire project. You should exercise caution when using these on production code, and don't hesitate to ask for help if you're unsure what a command might do.

After this lesson, you should:

- Be able to roll back changes to particular commit.
- Have an understanding of how rebasing affects your commit history.
- Know when to rebase/reset and when **not**to.

Resetting the past

Remember how our commits form a timeline? We can see the state of our project at any point using `git checkout`. What if we want to travel back in time to a point before we caused a new bug or chose a terrible font? `git reset` is the answer!

*Resetting*involves moving our `HEAD`ref back to a different commit. No matter how we reset, `HEAD`will move with us. Unlike `git checkout`, this will also destroy intermediate commits. We can use some additional flags to determine how our code changes are handled.

Starting small: Soft resets

The least-dangerous reset of all is `git reset --soft`. A soft reset will move our `HEAD`ref to the commit we've specified, and will leave any intermediate changes in the staging area. This means you won't lose any code, though you will lose commit messages.

A practical example of when a soft reset would be handy is joining some small commits into a larger one. We'll pretend we've been struggling with "their", "there", and "they're" in our app. Here's our commit history:

```
7a78d33 (HEAD -> master) That looks good! Finish writing our homepage text.
0694c96 Still wrong. Use 'their' now.
6f0ec56 Try 'there' instead.
38f8524 Try using 'they're' on the homepage.
9c5e2fc Set up our homepage.
```

Those commit messages aren't great: they're not very explanatory, and they don't provide a lot of value in our commit history. We'll fix them with a soft reset:

```
git reset --soft 9c5e2fc
```

This moves our `HEAD`ref back to our first commit. Looking at our commit log now, we might be worried we've lost our changes:

```
9c5e2fc (HEAD -> master) Set up our homepage.
```

However, check out `git status`:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   homepage.html
        new file:   their.txt
        new file:   there.txt
        new file:   theyre.txt
```

You'll see that our changes are still present in the staging area, ready to be re-committed when we're ready! We can use `git commit` to re-apply those changes to our commit history with a new, more helpful message instead:

```
861b5f9 (HEAD -> master) Add all of our homepage text, including 'their'.
9c5e2fc Set up our homepage.
```

Notice that the new commit has a totally new hash. The old commit messages (and their associated hashes) have been lost, but our code changes are safe and sound!

Getting riskier: Mixed resets

If soft resets are the safest form of `git reset`, mixed resets are the most average! This is exactly why they're the default: running `git reset` without adding a flag is the same as running `git reset --mixed`.

In a mixed reset, your changes are preserved, but they're moved from the commit history directly to the working directory. This means you'll have to use `git add` to choose everything you want in future commits.

Mixed resets are a good option when you want to alter a change in a previous commit. Let's use a mixed reset with our "their", "there", "they're" example again.

We'll start with "they're":

```
6f7499e (HEAD -> master) Try using 'they're' on the homepage.
9c5e2fc Set up our homepage.
```

Instead of pushing ahead, we'd like to revoke that change and try it again. Let's use a mixed reset:

```
git reset 9c5e2fc
```

Now you'll see that your changes are in the working directory instead of the staging area:

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        theyre.txt

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

You can edit your files, make the changes you'd like, and use `git add` and `git commit` to add a new commit to your repo:

```
936c89e (HEAD -> master) Add 'their' to the homepage text.
9c5e2fc Set up our homepage.
```

Notice again that you don't lose your code with a mixed reset, but you do lose your commit messages & hashes. The difference between `--soft` and `--mixed` comes down to whether you'll be keeping the code exactly the same before re-committing it or making changes.

Red alert! Hard resets

Hard resets are the most dangerous type of reset in Git. Hard resets adjust your `HEAD` ref and *totally destroy any interim code changes*. Poof. Gone forever.

There are very few good uses for a hard reset, but one is to get yourself out of a tight spot. Let's say you've made a few changes to your repository but you now realize those changes were unnecessary. You'd like to move back in time so that your code looks exactly as it did before any changes were made. `git reset --hard` can take you there.

It's our last round with "their", "there", and "they're". We've tried it all three ways and decided we don't need to use that word at all! Let's walk through a hard reset to get rid of our changes.

We'll start in the same place we began for our soft reset:

```
7a78d33 (HEAD -> master) That looks good! Finish writing our homepage text.
0694c96 Slill wrong. Use 'their' now.
6f0ec56 Try 'there' instead.
38f8524 Try using 'they're' on the homepage.
9c5e2fc Set up our homepage.
```

It turns out that we'll be using a video on our homepage and don't need text at all! Let's step back in time:

```
git reset --hard 9c5e2fc
```

Our Git log output is much simpler now:

```
9c5e2fc (HEAD -> master) Set up our homepage.
```

Take a look at `git status`:

```
On branch master
nothing to commit, working tree clean
```

It's empty - no changes in your working directory and no changes in your staging area. This is major difference between a hard reset and a soft/mixed reset: you will lose *all your changes* back to the commit you've reset to.

If your teammate came rushing in to tell you that the boss has changed their mind and wants that homepage text after all, you're going to be re-doing all that work! Be very confident that the changes you're losing are unimportant before embarking on a hard reset.

Rebase: An alternative form of time travel

Sometimes we want to change more than a few commits on a linear timeline. What if we want to move multiple commits across branches? `git rebase` is the tool for us!

Rebasing involves changing your current branch's base branch. We might do this if we accidentally started our branch from the wrong commit or if we'd like to incorporate changes from another branch into our own.

You're probably thinking "Gee, this sounds familiar! Can't we accomplish those tasks with `git merge`?" In almost all cases, you'd be right. Rebasing is a dangerous process that effectively rewrites history. There's a whole slew of movies, books, and TV shows that explain why rewriting history is a bad idea!

Okay, rebasing is risky! Show me anyway.

Let's look at a situation where we might be tempted to rebase. We've added a couple commits to a feature branch while other team members have been merging their code into the `master` branch. Once we're ready to merge our own branch, we probably want to follow a tried-and-true procedure:

```
> git pull origin master
```

This will fetch our remote `master` branch and merge its changes into our own feature branch, so it's safe to pull request or `git push`. However, every time we do that, a merge commit will be created! This can make a big mess of our Git commit history, especially if lots of people are making small changes.

We can use `git rebase` to move our changes silently onto the latest version of `master`. Here's what the `git log` history of our two example branches looks like:

`master`

```
c52b936 (HEAD -> master) Change footer background color.
45d17bc Added 'Contact Us' link to footer.
9c5e2fc Set up our homepage.
```

`working-on-the-header` **(our feature branch)**

```
a5360ec (HEAD -> working-on-the-header) Animate the logo.
d929978 Add a cool dropdown menu.
9c5e2fc Set up our homepage.
```

Notice that both branches start at `9c5e2fc`. That's our common ancestor commit, and is where `git merge` would start stitching these branches together! We're going to avoid that entirely with a rebase. We'll run this command while we have `working-on-the-header` checked out:

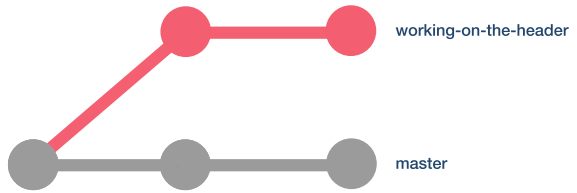
```
git rebase master
```

Here's our new commit history:

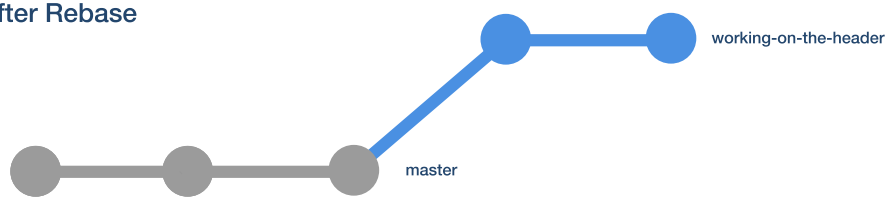
```
e009e89 (HEAD -> working-on-the-header) Animate the logo.
c93e706 Add a cool dropdown menu.
c52b936 (master) Change footer background color.
45d17bc Added 'Contact Us' link to footer.
9c5e2fc Set up our homepage.
```

And a diagram of what just happened:

Before Rebase



After Rebase



See how we changed the color of our commits after the rebase? Take a close look at the commit history changes as well. Even though our commits have the same content, they have a new hash assigned, meaning they're entirely new commits! This is what we mean by "rewriting history": we've actually changed how Git refers to these changes now.

One last warning & the "Golden Rule of Git"

These tools can all feel pretty nifty, but be very wary of using them too much! While they can augment your Git skills from good to great, they can also have

catastrophic side effects.

There's a "Golden Rule of Git" you should know that directly relates to both `git reset` and `git rebase`:

Never change the history of a branch that's shared with others.

That's it! It's simple and to the point. If you're resetting or rebasing your own code and you make a mistake, your worst case scenario is losing your own changes. However, if you start changing the history of code that others have contributed or are relying on, your accidental loss could affect many others!

What we've learned

What a wild trip we've been on! Watching commits come and go as we `git reset` and `get rebase` can get a little confusing. Remember that while these tools are unlikely to be part of your everyday workflow, they're great topics for technical interviews. You should:

- Understand how `git reset` differs from `git checkout`.
- Be able to list the 3 types of `git reset` and differentiate them.
- Be comfortable comparing and contrasting `git rebase` and `git merge`.
- Remember the "Golden Rule of Git": never rewrite history after a `git push`!

Git Merge Conflicts & You

Welcome to the arena! Let's discuss what you'll need to do when attempting to merge two conflicting Git branches. You'll get to make the final say in which code enters...and which code leaves!

In this lesson, we'll discuss:

- The `git merge` process in-depth
- Managing conflicting code across branches
- Common workflows to help know how & when to merge

What is a "merge conflict"?

Whoa there - maybe we dove in a little too fast. Let's discuss what a merge conflict is and how we can resolve it.

First off, what is a *merge conflict*? It's a special state Git presents us with when two branches have code changes that are incompatible with each other. Here's a very simple example:

- You're working on a team trying to design a new app. You're working on the Git branch `my-red-app`, because you've decided that the app's primary color should be red! You add the following code to line 3 of your app's main file, `App.js`:

```
this.primaryColor = red;
```

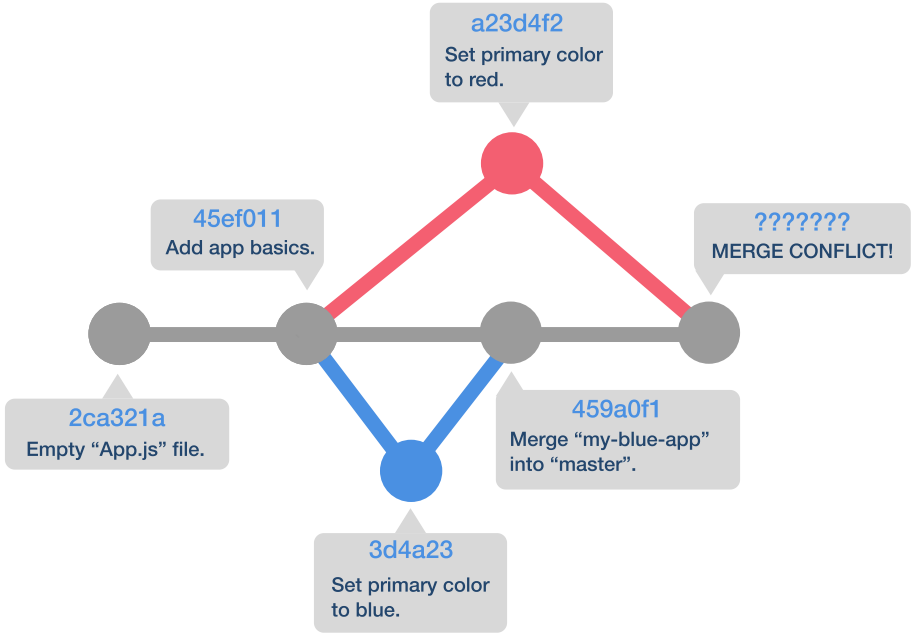
- At the same time, one of your teammates has opened the `my-blue-app` branch with an alternative decision on your app's aesthetics. They add the following code to line of 3 of their `App.js` file:

```
this.primaryColor = blue;
```

- Your friend types a little faster than you and has already merged their code into the `master` branch. You get ready to merge your own code and...**Merge Conflict!**

Git identifies changes to `master` since your branch began and flags similar changes in the same place on your branch. Instead of trying to make a decision on your behalf, it hands you control. A merge conflict is Git's way of saying "I'm not sure what to do - help!".

Here's what a merge conflict looks like between branches:



Yikes! Remember that this looks awfully complicated, but the theory is pretty simple. In the workplace you'll need to resolve much more complicated conflicts than this so now's the time to get your practice in.

Resolving a merge conflict

Git is a complex tool, but it's built to help guide us as much as it can. Merge conflicts are no different. You will find that resolving them is easy once you know what you're looking at.

We'll stick with our "Red vs. Blue" example from above. When you attempt to `git merge`, you'll get a message like the following:

```
Auto-merging App.js
CONFLICT (content): Merge conflict in App.js
Automatic merge failed; fix conflicts and then commit the result.
```

Git is so helpful - it's telling us where to look *and* what to do! Following the instructions here, we'll look at `App.js`, resolve the conflict, and `git commit` with our resolved file(s).

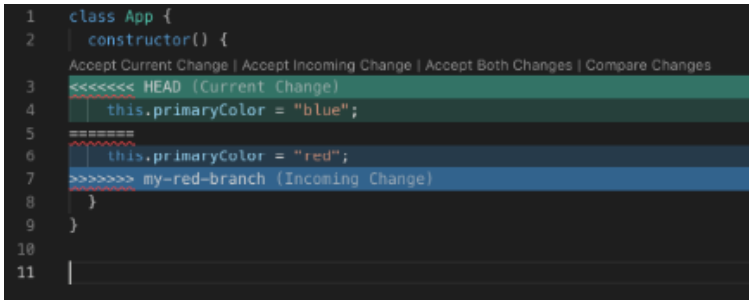
For even more info, check out `git status` during a merge conflict:

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   App.js

no changes added to commit (use "git add" and/or "git commit -a")
```

Notice the `both modified` prefix, reminding us that we have a conflict. "Both" refers to our two branches, `my-red-app` and `master`, which each include changes to the conflicting file. It's up to us to decide what code the file should contain when we complete the merge.

Conflict Resolution

If we open the conflicting file, we'll see some new syntax:



Notice the `<<<<<<` (line 3), `======` (line 5), and `>>>>>>` (line 7). These are special delimiters that Git uses to separate two pieces of conflicting code.

The first piece of code (between `<<<<<<` and `======`) comes from our *base branch* - the branch we're merging in to, which we're currently on. We can see it's labelled "HEAD", and VS Code is helping us out by noting that this is the "Current Change".

The second piece of code (between `======` and `>>>>>>`) comes from our *incoming branch*. VS Code is again helping us out by labelling this as the "Incoming Change".

To resolve this conflict, we need to decide which code to keep and which to get rid of. This is where your communication skills become important! During a merge conflict, you'll need to check in with teammates to decide what's best. Once you've come to a decision, you can edit the file, leaving only the code you want in the base branch when the conflict is over. If we decided to keep "red" in our example, we would delete lines 3, 4, 5, and 7.

You can do this manually in other editors, but VS Code helps us by providing the "Accept" buttons above our conflict. You can click "Accept Incoming Changes" to automatically update the code for us. It will remove the "Current

Changes" and any delimiters related to this conflict, leaving only the "Incoming Changes" we chose to keep.

Back on solid ground

Once you've saved your resolved file, the process is more familiar. You'll save your file, use `git add` to add it to the staging area, and `git commit` to complete the merge. Git will help you out with the commit message: it should say something like "Merge branch 'my-red-branch'", though you can change this during the commit process if you'd like.

There are a few important things to note:

- Not every merge results in a conflict, and not every change between two branches requires resolution! Git is smart about bringing code together when it's in separate files or on different lines, so it won't need your help to resolve every single change. It's only when two pieces of code actually conflict that Git will ask you to get involved.
- `git merge` is the safest way to ensure your code is up to date. We've looked at merging a "finished" branch back into `master` here, but you can also merge `master` into your feature branch while you're working on it! While this may cause conflicts, you're well-equipped to handle them now.
- Merge conflicts will always result in a *merge commit*: an extra commit in your history documenting where code was merged in from other sources. We'll discuss more dangerous ways of merging in code without these commits in an upcoming lesson.

What we've learned

Whew! We've emerged victorious from our merge conflict and can start work on a new branch or feature. Merge conflicts are a nearly daily part of life as a developer.

- You should now be able to identify the difference between a base branch and incoming branch.
- You should understand how `git merge` works and what it does when it's unable to proceed automatically.
- You're now familiar with resolving merge conflicts, both with VS Code's helpers and when in another editor by manually editing code.