HTML Basics, Version Control Intro

HTML Intro

During week 1, we will be introduced to HTML and tackle some coding together. Next week, we will dig into more advanced knowledge of HTML and CSS and even create our own sites.

Let's get started with HTML!

HTML, CSS, and JavaScript are very important like we discussed on day 1!

Recap:

- HTML5 - Essential Structure
- CSS - Page Design
- JS - Page behavior

Atom, Sublime Text, and Brackets are great editors for web development. Here are the links so you can download the editor of your choice.

Of course, if these editors aren't your speed, you can development in whatever environment you choose!

Atom Sublime Text Brackets

HTML Document

HTML stands for Hypertext Markup Language, and it's the most popular language for writing pages for the web.

Hypertext refers to the way in which Web pages (HTML documents) are linked together.

As its name refers to, HTML is a Markup Language which means you use HTML to "mark-up" a text document with tags that tell a Web browser how to structure it to display.

Originally, HTML was developed with the intent of defining the structure of documents like headings, paragraphs, lists, and so forth to facilitate the sharing of scientific information between researchers.

Now, HTML is being widely used to format web pages with the help of different tags available in HTML language.

In its simplest form, the following code block is an example of an HTML document:

```
 Live Demo
<!DOCTYPE html>
<html>
```

```
   <head>
      <title>This is document title</title>
   </head>

   <body>
      <h1>This is a heading</h1>
      <p>Document content goes here.....</p>
   </body>

</html>
```

doctype - is supposed to do is tell the browser or user agent which version of HTML that you'll be using so it knows which version of the syntax to use in parsing

document head - the head element is where you'll place information about the document, references to things like external scripts and styles, and additional instructions to the browser about how the document should be rendered.

body - where all of your pages visible content (not nested in the head tag)

tags act like containers. They tell you something about the information that lies between their opening and closing tags. Tags are often referred to as elements.

```
<br /> tag for line break
<hr /> tag for horizontal rule
```

Heading Tags

Any document starts with a heading. You can use different sizes for your headings. HTML also has six levels of headings, which use the elements h1, h2, h3, h4, h5, and h6.

While displaying any heading, the browser adds one line before and one line after that heading.

HyperText is text displayed on a computer or device that provides access to other text through links, also known as hyperlinks.

HTML Structure

HTML is organized as a collection of family tree relationships. When an element is contained inside another element, it is considered the child of that element. The child element is said to be nested inside of the parent element.

```
<body>
  <div>
    <h1>Sibling to p, but also grandchild of body</h1>
    <p>Sibling to h1, but also grandchild of body</p>
  </div>
</body>
```

In this example, the element is the parent of the div element. Both the h1 and p elements are children of the div element. Because the h1 and p elements are at the same level, they are considered siblings and are both grandchildren of the element.

Understanding HTML hierarchy is important because child elements can inherit behavior and styling from their parent element. You'll learn more about webpage hierarchy when you start digging into CSS.

Divs

One of the most popular elements in HTML is the div element. <div> is short for "division" or a container that divides the page into sections. These sections are very useful for grouping elements in your HTML together.

Divs can contain any text or other HTML elements, such as links, images, or videos. Remember to always add two spaces of indentation when you nest elements inside of divs for better readability.

Attributes

If we want to expand an element's tag, we can do so using an attribute. Attributes are content added to the opening tag of an element and can be used in several different ways, from providing information to changing styling. Attributes are made up of the following two parts:

- The name of the attribute
- The value of the attribute

```
<div id="intro">
  <h1>Introduction</h1>
</div>
```

paragraph tag vs span

paragraphs tags contain a block of plain text.

span contains short pieces of text or other HTML. They are used to separate small pieces of content that are on the same line as other content.

Styling Text

- The em tag will generally render as italic emphasis.
- The strong tag will generally render as bold emphasis.

HTML’s line break element is **<br>**. You can use it anywhere within your HTML code and a line break will be shown in the browser.

Unordered Lists

The unordered list tag (<ul>) is used to create a list of items in no particular order

The <li> or list item tag is used to describe an item in a list.

```
<ul>
  <li>Limes</li>
  <li>Tortillas</li>
  <li>Chicken</li>
</ul>
```

Ordered Lists

Ordered lists (<ol>) are like unordered lists, except that each list item is numbered.

```
<ol>
  <li>Preheat the oven to 350 degrees.</li>
  <li>Mix whole wheat flour, baking soda, and salt.</li>
  <li>Cream the butter, sugar in separate bowl.</li>
  <li>Add eggs and vanilla extract to bowl.</li>
</ol>
```

The <img> tag allows you to add an image to a web page. Most elements require both opening and closing tags, but the <img> tag is a self-closing tag.

Images

```
 <img src="image-location.jpg" />
```

The src attribute must be set to the image’s source, or the location of the image. In this case, the value of src must be the uniform resource locator (URL) of the image.

Videos

```
 <video src="newVid.mp4" width="320" height="240" controls>
</video>
```

Let's talk about displaying content

Block vs. Inline Elements

Most elements are either **block** or **inline**.

What's the difference?

Block-level elements begin on a new line, stacking one on top of the other, and occupy any available width. Block-level elements may be nested inside one another and may wrap inline-level elements. We'll most commonly see block-level elements used for larger pieces of content, such as paragraphs.

Inline-level elements do not begin on a new line. They fall into the normal flow of a document, lining up one after the other, and only maintain the width of their content. Inline-level elements may be nested inside one another; however, they cannot wrap block-level elements.

A <div> is a block-level element that is commonly used to identify large groupings of content, and which helps to build a web page's layout and design. A <span>, on the other hand, is an inline-level element commonly used to identify smaller groupings of text within a block-level element.

Display

Exactly how elements are displayed—as block-level elements, inline elements, or something else—is determined by the display property. Every element has a default display property value; however, as with all other property values, that value may be overwritten. There are quite a few values for the display property, but the most common are block, inline, inline-block, and none.

We can change an element's display property value by selecting that element within CSS and declaring a new display property value. A value of block will make that element a block-level element.

```
 p {
  display: block;
}
```

The Space Between Inline-Block Elements

One important distinction with inline-block elements is that they are not always touching, or displayed directly against one another. Usually a small space will exist between two inline-block elements. This space, though perhaps annoying, is normal.

What Is the Box Model?

According to the box model concept, every element on a page is a rectangular box and may have width, height, padding, borders, and margins.

That's worth repeating: Every element on a page is a rectangular box.

Every element on every page conforms to the box model, so it's incredibly important. Let's take a look at it, along with a few new CSS properties, to better understand what we are working with.

Working with the Box Model

Every element is a rectangular box, and there are several properties that determine the size of that box. The core of the box is defined by the width and height of an element, which may be determined by the display property, by the contents of the element, or by specified width and height properties. padding and then border expand the dimensions of the box outward from the element's width and height. Lastly, any margin we have specified will follow the border.

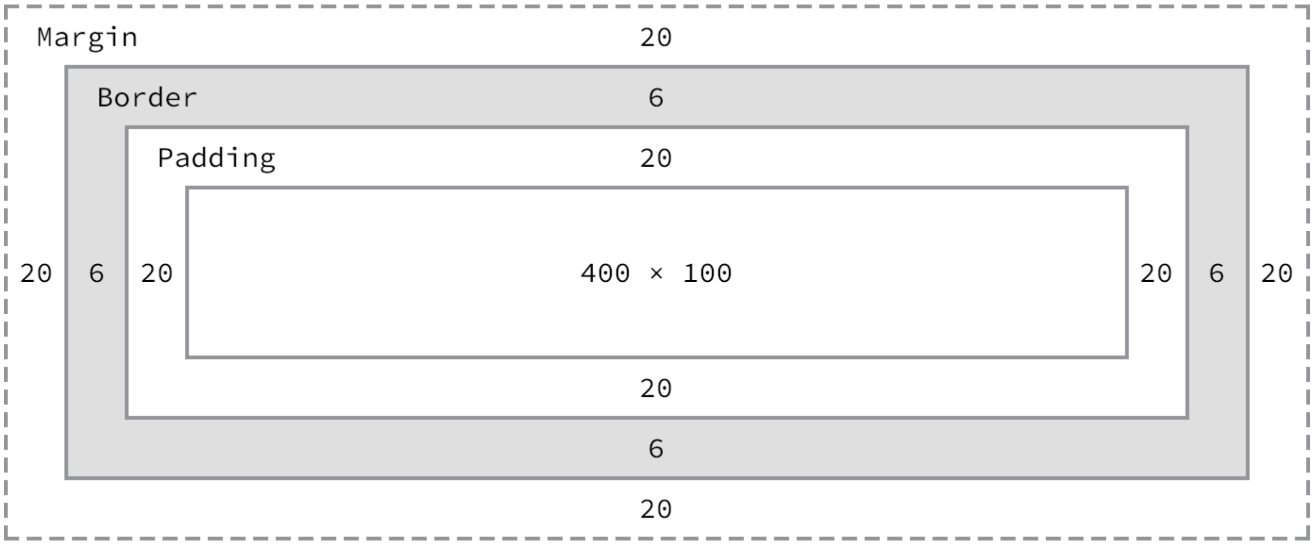
Each part of the box model corresponds to a CSS property: width, height, padding, border, and margin.

Let's look these properties inside some code:

```
div {
  border: 6px solid #949599;
  height: 100px;
  margin: 20px;
  padding: 20px;
  width: 400px;
}
```

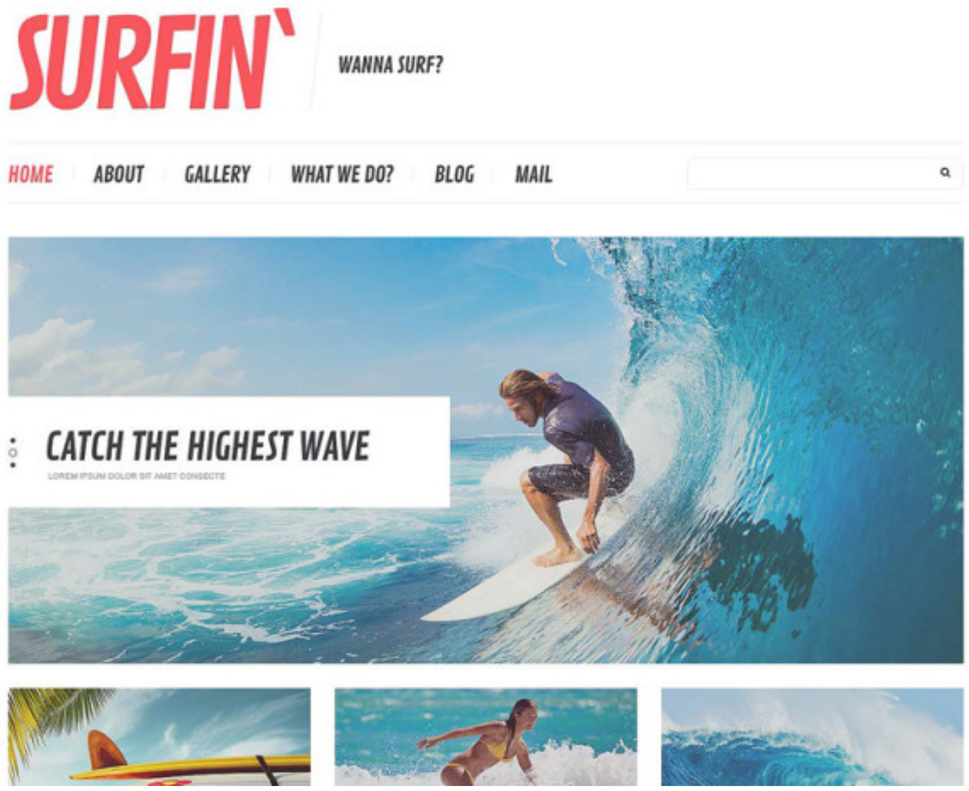
According to the box model, the total width of an element can be calculated using the following formula:

```
margin-right + border-right + padding-right + width + padding-left + border-left
+ margin-left
```



Lab Activity

With display in mind, develop a simple website called 'uniqueName.html'



Create headers and add elements to your page!

Examples of website you could create:

1. Your favorite recipe
2. A tutorial on how to surf
3. All the reasons you love to code

GitHub

Version Control

What is Version Control and why is it an important part of the development lifecycle?

What is version control?

Version control systems help development teams keep track of changes to their source code over time. It is also commonly referred to as SCM (Source Code Management). Version Control Systems keep track of every modification made to the code base and allows you to go back and forward in time as it relates to changes in your source code. It also facilitates comparing code at different points in time to help fix any bugs or issues that may have been the result of a specific change.

Useful Resources

Git Website Official Git Online Tutorial Git Command Cheat Sheet Sourcetree GUI client The Free Git book Atlassian Git Tutorials Git Reference

Centralized vs Distributed Version Control

For this course we are going to focus on Distributed Version Control with Git but it is worth discussing the difference between Distributed and Centralized Version Control.

Centralized Version Control Centralized Version Controls systems like SVN (Subversion) require a single “central” copy of your project typically on a server somewhere. Developers “commit” their changes to this central copy. If this central copy is not available then the developer is not able to save changes to their work within the Version Control System until they have access to the server.

Typical Workflow for CVC

- Developer checks out code from version control or pulls down latest changes to code already checked out.
- Developer makes some changes and tests his work
- Developer commits changes back to central repository for other to pull into their local copies

Distributed Version Control Distrubuted Version Control sytems like Git do not rely on a central server to store all the versions of a project’s files. You can however utilize a service like GitHub to simulate this functionality. Every developer would “clone” an existing repository from a service like GitHub or initiate a repository locally. A copy of a repository and the full history of the project is stored locally with no need to connect to the centralized server to track changes. The developer can make changes to the source code and commit those changes to the local repository on their system. When they are ready to share changes with other developers they can "push" their changes and all the attached history to the remote repository. Once pushed, other developers can now update their local copies with those changes.

Typical Workflow for DVC

- Developer checks out code from version control or pulls down latest changes to code already checked out.
- Developer makes some changes and tests his work
- Developer makes a local "commit" that repesents the changes made
- Developer repeats this cycle until locally until they are ready to share their changes with other developers
- Developer commits changes back to central repository for other to "pull" into their local copies or provide a patch file to another developer if they choose not to use a service like GitHub

Git Basics

This section will cover the basics of working Git on your local machine

Common Basic Commands

- `git` - This is the top level command. You will typicially type `git` and then an action you want git to execute. You may remember this command when we checked what version of git we had installed by running `git --version`
- `git init` - This will tell git that the directory you are in when this command is run should be tracked by git. Your are creating the repository when you run this command
- `git add` - followed by a path will let git know you want to add files at that path to be tracked by version control and staged for commit
- `git commit` - Takes your staged files and commits them to version control and creates a point in your version control history that can be referenced later. You will almost always find the `-m "COMMIT MESSAGE"` flag used to provide a message related to your commit.
- `git status` - Displays the files that have changed in your repo as it relates to the latest commit. It will show what files have been modified and which files are staged for your next commit. It also shows information about which branch you are currently working with.
- `git diff` - Displays the line by line differences between files in your repo compared to the last commit

Branching and Merging

Branching is the concept of going down a different path in the code base from a specific commit. Usually branching off master in it's latest state to make a particular set of changes and not modifying the state of the the original code you branched from. Merging is the concept of applying changes made in one branch to another.

Practical Example

You need to make some changes to a project that may take some time. You make a branch to do those changes. An emergency change comes in that has to be pushed live. Instead of having your changes for your feature in the way, you can switch back to the master branch, create another branch for those changes, merge the emergency changes into master and push it live. Update your feature branch with whats now in master using a merge and continue your work. This avoids your changes that may take some time to complete from being in the way.

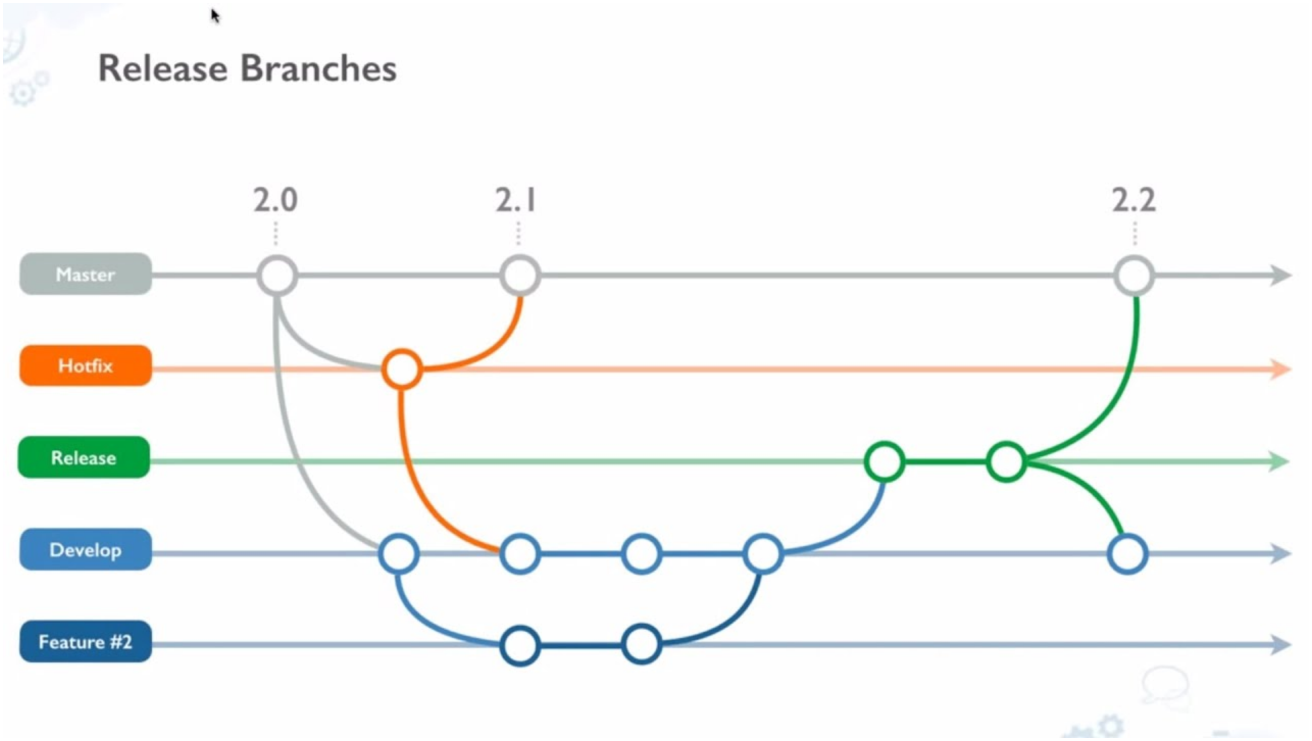
Merging and Branchin Commands

- `git branch` - Shows you what branches exist in your repository and what branch you are on. Your current branch is marked with a `*`
- `git checkout -b BRANCH_NAME` or `git branch BRANCH_NAME` - git checkout -b creates a branch and checks out that branch. git branch with a branch name provided creates the branch but does not check it out.

- `git branch -d BRANCH_NAME` - Deletes a branch.
- `git merge BRANCH_NAME` - Merges the named branch into the branch you are currently on. All the changes that exist in the branch you are merging from now exist in your branch.

Typical Git Workflow

This section will cover a basic git workflow for making changes to a repository tracked with Git.



It is best practices to create a branch for each logical grouping of changes you are making to your project.

Practical Example

You have been assigned a task to add a header to the website. You would create a branch for this task called something like "AddingHeaderToWebsite". You would create your branch of your up-to-date master branch. Once the branch is created you would start working on adding the header to the website. You should make a commit after each meanigful milestone in your development process. Once you are done with the header and tested everything on your branch, you would make sure your master branch is up-to-date incase anyone changes have been made to master since creating your branch. You would then merge master into your branch and handle any conflicts. Once confilicts have been resolved, if any came up, you would then merge your "AddHeaderToWebsite" branch into the master branch. Since you no longer have a need for your "AddHeaderToWebstieBranch" it is safe to delete it. All the history and commits from your branch have been merged into master so you can still go back to all the commits you made in your branch.

GitHub - Sharing your code with the world

Why use something like Git? Say you and a coworker are both updating pages on the same website. You make your changes, save them, and upload them back to the website. So far, so good. The problem comes when your coworker is working on the same page as you at the same time. One of you is about to have your work overwritten and erased.

A version control application like Git keeps that from happening. You and your coworker can each upload your revisions to the same page, and Git will save two copies. Later, you can merge your changes together without losing any work along the way. You can even revert to an earlier version at any time, because Git keeps a “snapshot” of every change ever made.

GitHub is a service that allow you to make your Git repositories available to other developers and the world through a Web-based graphical interface. It has opinions on how to grant access to your code as well as how people can contribute to your code. It also provides several collaboration features, such as a wikis and basic task management tools for every project.

Words People Use When They Talk About Git

Repository A repository is a location where all the files for a particular project are stored, usually abbreviated to “repo.” Each project will have its own repo, and can be accessed by a unique URL.

Forking “Forking” is when you create a new project based off of another project that already exists. If you find a project on GitHub that you’d like to contribute to, you can fork the repo, make the changes you’d like, and release the revised project as a new repo. If the original repository that you forked to create your new project gets updated, you can easily add those updates to your current fork. You can also request that changes you have made can be merged into the original project. This is facilitated through **Pull Requests**.

Pull requests A pull request is a way of facilitating a merge through the GitHub interface. If you or someone on your development team has made some changes in a branch or a forked repository, you can create a pull request asking for those changes to be merged into the master branch or any other branch that exists in the repo. The owner of the project can then review your changes and merge those changes if all is good. Pull requests can be created from forked repos as well.

`git remote add` To communicate with the outside world, git uses what are called **remotes**. These are repositories other than the one on your local disk which you can push your changes into (so that other people can see them) or pull from (so that you can get others changes). The command `git remote add NAME_OF_REMOTE URL_OF_REMOTE` creates a new remote called whatever you put in place of NAME_OF_REMOTE located at URL_OF_REMOTE. NAME_OF_REMOTE is usally origin and URL_OF_REMOTE is the link to your GitHub repository. This lets git know that you are connected to a remote repository.

`git push` This is a command that says "push the commits in the local branch to the remote branch". Once this is executed, your commits since your last push will be available on GitHub. Your initial push may looks someghing like `git push -u origin master` which says push my code to my remote named `origin` into the master branch. Once this has been done the first time you can just use `git push` from that branch and dont need the rest of the command.

`git pull` This is a command that says "pull the commits in the remote branch to the local branch". Once this is executed, any commits made to the remote branch since your last pull will be available in your local copy of the repository.

`git clone` `git clone` is usually followed by the url of the remote repository on GitHub you would like to "clone" or make a local copy of on your computer. On any repo you will see a "Clone or download" button. Clicking that button will provide you with the url required to clone that repo.

Command Line: The computer program we use to input Git commands. On a Mac, it's called Terminal. On a PC, it's a non-native program that you download when you download Git for the first time (we'll do that in the next section). In both cases, you type text-based commands, known as prompts, into the screen, instead of using a mouse.

Repository: A directory or storage space where your projects can live. Sometimes GitHub users shorten this to "repo." It can be local to a folder on your computer, or it can be a storage space on GitHub or another online host. You can keep code files, text files, image files, you name it, inside a repository.

Version Control: Basically, the purpose Git was designed to serve. When you have a Microsoft Word file, you either overwrite every saved file with a new save, or you save multiple versions. With Git, you don't have to. It keeps "snapshots" of every point in time in the project's history, so you can never lose or overwrite it.

Commit: This is the command that gives Git its power. When you commit, you are taking a "snapshot" of your repository at that point in time, giving you a checkpoint to which you can reevaluate or restore your project to any previous state.

Branch: How do multiple people work on a project at the same time without Git getting them confused? Usually, they "branch off" of the main project with their own versions full of changes they themselves have made. After they're done, it's time to "merge" that branch back with the "master," the main directory of the project.

git init: Initializes a new Git repository. Until you run this command inside a repository or directory, it's just a regular folder. Only after you input this does it accept further Git commands.

`git config`: Short for "configure," this is most useful when you're setting up Git for the first time.

git help: Forgot a command? Type this into the command line to bring up the 21 most common git commands. You can also be more specific and type "git help init" or another term to figure out how to use and configure a specific git command.

`git status`: Check the status of your repository. See which files are inside it, which changes still need to be committed, and which branch of the repository you're currently working on.

`git add`: This does not add new files to your repository. Instead, it brings new files to Git's attention. After you add files, they're included in Git's "snapshots" of the repository.

`git commit`: Git’s most important command. After you make any sort of change, you input this in order to take a “snapshot” of the repository. Usually it goes git commit -m “Message here.” The -m indicates that the following section of the command should be read as a message.

`git branch`: Working with multiple collaborators and want to make changes on your own? This command will let you build a new branch, or timeline of commits, of changes and file additions that are completely your own. Your title goes after the command. If you wanted a new branch called “cats,” you’d type git branch cats.

`git checkout`: Literally allows you to “check out” a repository that you are not currently inside. This is a navigational command that lets you move to the repository you want to check. You can use this command as git checkout master to look at the master branch, or git checkout cats to look at another branch.

`git merge`: When you’re done working on a branch, you can merge your changes back to the master branch, which is visible to all collaborators. git merge cats would take all the changes you made to the “cats” branch and add them to the master.

Lab

Go through the GitHub Training available at https://try.github.io/

References:

1. www.tutorialspoint.com
2. https://www.codecademy.com/courses/learn-html/lessons/intro-to-html/exercises/structure-html?action=resume_content_item
3. https://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1/`