Learn To Code

Command Line and Git

Student Workbook #1-A

Version 3.0 Y

Table of Contents

Module 1

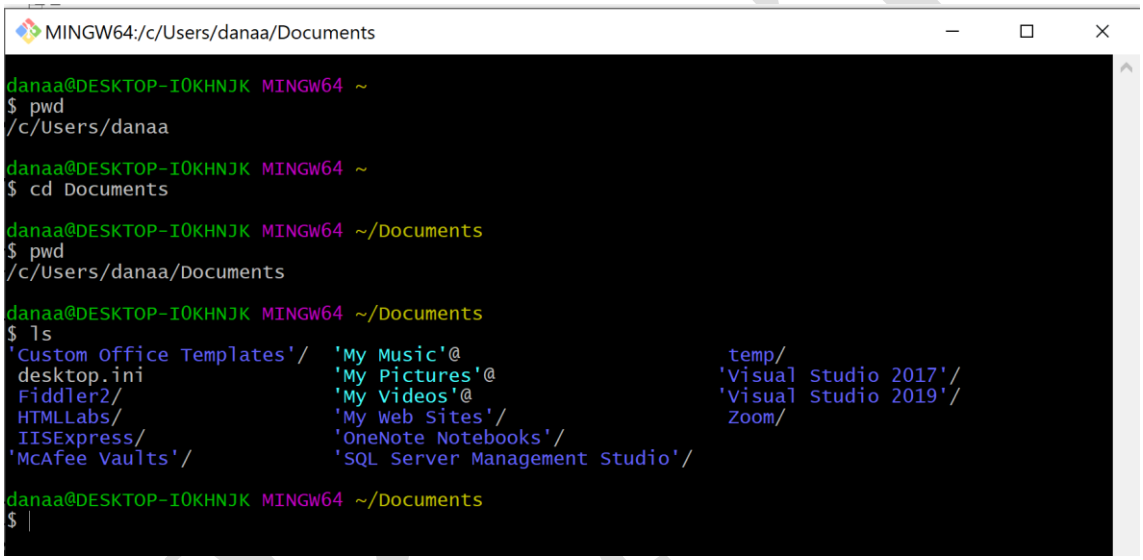
The Command Line

Section 1–1

The Command Line

The Command Line

- **The command line is a powerful, text-based interface that developers accomplish certain tasks**
  - Before operating systems had sophisticated user interfaces like those in Windows or macOS, the command line was all we had
    - Think back to DOS and early Linux systems



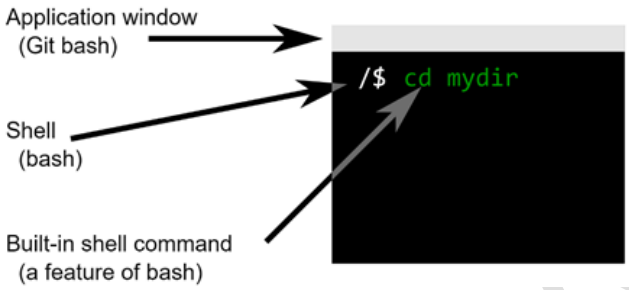
- **There are things you can do from the command line that you can't do with point-and-click user interfaces**
  - So, as a programmer, it is a skill that you eventually have to conquer
  - It takes a while to get used to, but with practice you catch on

CLIs and IDEs

- **If watch old sci-fi from the ’80s or a modern show with hackers working, you see:**
  - a command line presented as a black window with white text that scrolls by quickly
  - computer experts typing cryptic commands
- **The program that processes these cryptic commands is called a *Command-Line Interface (CLI)***
- **If you are a programmer in the 2020s, you may accomplish some tasks using the CLI, but you also use more advanced software development tools to get the job done**
- **Tools like Visual Studio Code (Web programming), Visual Studio (C#/.NET), and Eclipse (Java) allow programmers write and run their code using a single tool**
  - These tools are called IDEs, or Integrated Development Environments
- **However, sometimes the IDE doesn’t allow us to perform the task we need to, or it doesn’t make it easy, so we leave the IDE and use the CLI**

Anatomy of a Command Line

- **Windows provide a program called a *shell* that processes the commands we enter in the command line interface**
  - The Windows operating system has two native command shells: The Command Shell and PowerShell
  - However, we will install a ***different command shell*** (using Git Bash) named bash



- **We can type the name of the program `date` in the terminal window and it will print out the current date**

Example

```
$ date

Sat, Jun 12, 2021  6:12:17 PM
```

Section 1–2

Built-In Shell Commands for Files and Directories

`pwd` - Where am I? (print working directory)

- You have to learn to build a visual tree in your mind of the file system

- At any point, if you need to ask "Where am I?", you can enter the `pwd` command

Example

Find out what folder you are in

```
$ pwd
/c/Users/danaa
$ cd documents
$ pwd
/c/Users/danaa/Documents
```

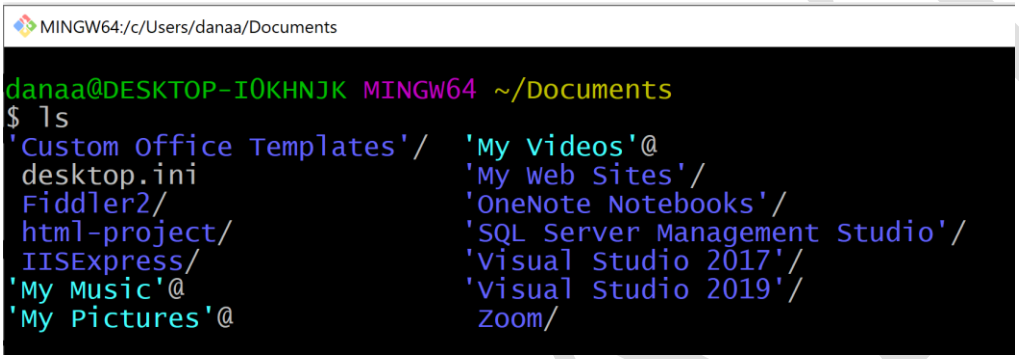
- We will explore the `cd` command soon

`ls` - List Files and Subdirectories

- To list the files and directories of the current working directory, use the ls command

Example

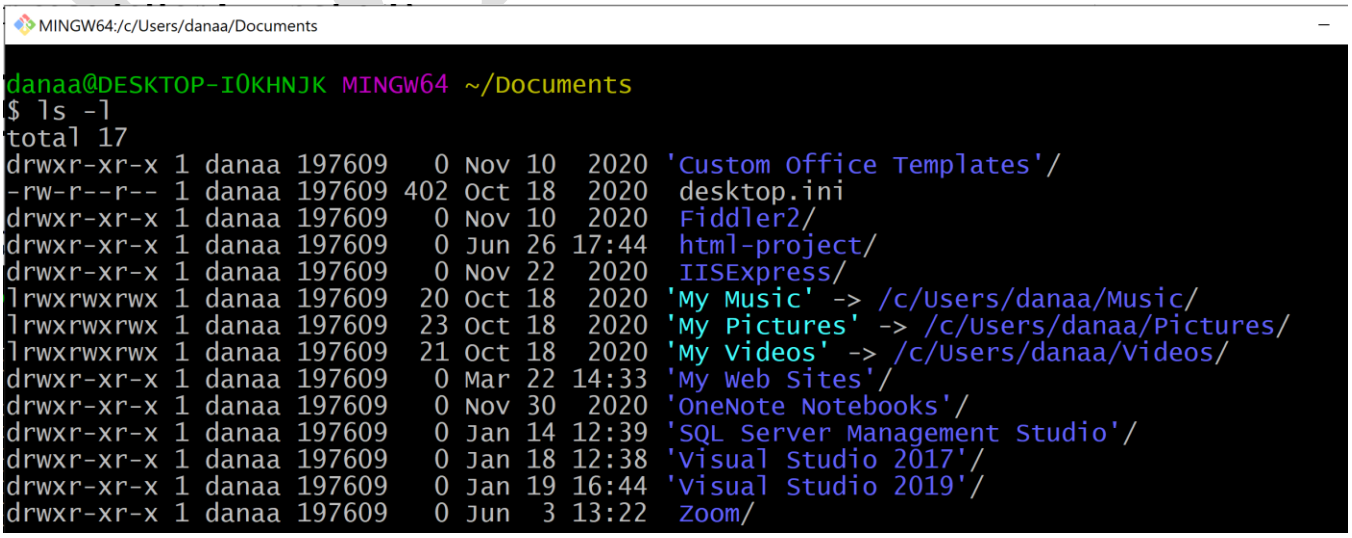
List files and subdirectories (doesn't show hidden files/directories)



- If you want more information, specify the `-l` option (the lower case letter L -- not a 1)

Example

List details of files and subdirectories (doesn't show hidden files/directories)

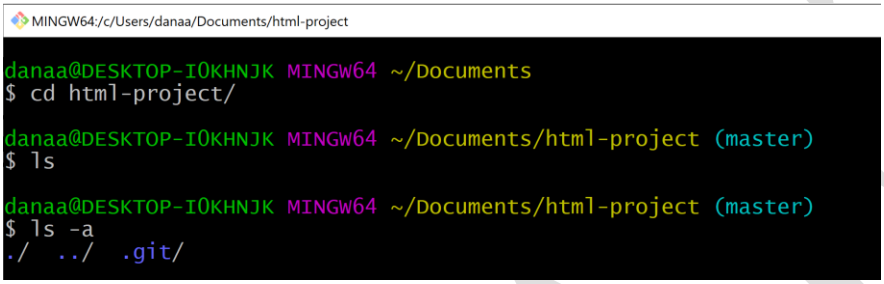


- **In most Unix-like shells, files that begin with a period are hidden and won't appear in `ls` listings**

- **If you want to see *all* files, use the `-a` option (the 'all' flag)**

Example

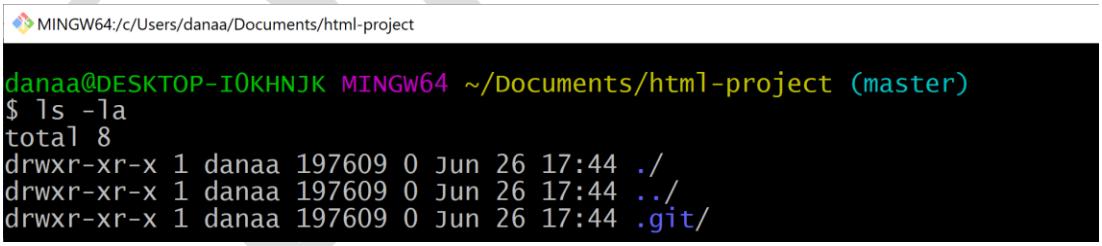
List all files and subfolders. In the `html-project` folder, the `ls` command shows no files, but the `ls -a` command shows a hidden `.git` folder



- **You can combine options to see all of the files and their details**
  - You might see this written as `-la` or `-al`

Example

List details of all files and subfolders



`cd` - Changing the Working Directory

- In the following examples, assume the subdirectories of the project directory look like this:

```
.
|   index.html
└── css/
    └── main.css
└── images/
    └── nature/
        └── tree.png
└── scripts/
    └── index.css
```

- To change the directory, use `cd`

Example

Go into the `images` directory from the project folder

```
$ cd images
```

- You can change directories one at a time, or combine the command

Example

Go into the `images/nature` directory from the project folder

```
$ cd images
$ cd nature
```

– or, you could do it all at once

Example

Go into the `images/nature` directory from the project folder

```
$ cd images/nature
```

- **To change the directory up one level, you would use `cd ..`**

Example

Go up one level in the directory tree

```
$ cd ..
```

NOTE: `..` refers to the parent directory and `.` refers to the current directory

- **If you were in the `images` directory and wanted to change the directory back to the `scripts` directory, use this**

Example

Go up one level in the directory tree and then down into the a subdirectory

```
$ cd ../scripts
```

- **If you were in the `nature` directory and wanted to change the directory back to the top level of the project, use this**

Example

Go up two levels in the directory tree

```
$ cd ../..
```

Go Home!

- **There is a really useful shortcut that takes you back to your "Home" directory**
  - Your "home" directory is the place where your user's files are stored
  - On modern Windows, that's normally at the path `/c/Users/«your user name»`
    * For example, `/c/Users/Danaa`

- **You can use the bash shortcut tilde `~` to get to the "home" directory in**

Example

Go to the home directory

```
$ cd ~
```

- **Or you could just type `cd`**

Example

Go to the home directory

```
$ cd
```

Exercise

Exercise 1

Your instructor will upload a ZIP file containing a set of folders. Unzip it to your `Documents` folder. It will unzip to a folder at the root of drive C:\ named `VirtualWorld`.

DO NOT OPEN THE DIRECTORY USING FILE EXPLORER!

Launch Git Bash. Use a **`cd`** command to navigate to `c:\VirtualWorld` and use the **`pwd`** command to make sure you are in the correct directory.

Using the **`ls`** and **`cd`** commands, answer the following questions:

1. What are the names of the subfolders under `VirtualWorld`?
2. What types of food are there in this virtual world? Hint: find the Foods folder and look inside of it.
3. What kinds of pets are there? Hint: find the Pets folder and look inside of it.
4. What parks are in California?
5. What kinds of BBQ are there?
6. How many different farm animals are there?
7. What Mexican foods are there?
8. What parks are in Texas?

Now, go back to your HOME directory. What files and folders are located there?

`mkdir` - Create a Directory

- **You can create a directory using the `mkdir` command with the name of the directory that you want to create**
  - NOTE: The folder is created as a subfolder of wherever you are

Example

Create a new directory

```
$ mkdir html-project2
```

`touch` - Create a File

- **The `touch` command is the easiest way to create a new, empty file**

Example

Create a new file

```
$ touch index.html
```

Example

Create a three new files

```
$ touch index.html about-us.html contact-us.html
```

- **`touch` can also be used to change the timestamps on existing files and directories**
  - A timestamp is the recording of the date/time of the most recent access or modification on a file or folder

Example

Update the timestamp on a file if it already existed

```
$ touch index.html   <----- since the file exists, it updates the timestamp
```

`rm` and `rmdir` - Delete a File or Directory

- **The `rm` command is used to delete one or more files or directories**

Example

Remove a single file

```
$ rm index.html         <---- you can remove any file
```

Example

Remove an empty directory

```
$ mkdir test-project
$ rmdir test-project    <---- the directory must be empty for this to work
```

- **The `rm` command has the powerful (*dangerous?*) option `-r`**
  - It's the recursive option that says to delete that folder, any files it contains, any sub-folders it contains, and any files or folders in those sub-folders, all the way down

Example

Remove a directory and all of its contents

```
$ rm -r test-project
```

Exercise

Exercise 2

Continue exploring the `VirtualWorld`.

Using just the commands **`mkdir, touch, rm`** and **`rmdir`** commands, perform the tasks below:

1. Create a new state (folder) to hold parks in Ohio
2. Add a new file named `Enchilada.txt` as a Mexican food option
3. Add a new state (folder) to hold parks in New York
4. Add three new parks in New York. Remember, they are just files with the park name and a `.txt` file extension.
5. Add an Iguana as a new type of pet
6. Add a new state to hold parks in Iowa
7. Add three new parks from Iowa
8. Remove goldfish as a pet
9. Remove all parks in Ohio
10. List the parks in Iowa
11. Remove all park in Iowa

`cp` - Copy a File or Directory

- You can use the `cp` command to copy files or directories

Example

Copy the file `index.html` to a new directory (`Desktop` folder) in my home directory (~) and rename it to `index_copy.html`

```
$ cp index.html ~/Desktop/index_copy.html
```

`mv` - Move Files and Directories

- **Use the `mv` command to move a file or directory to a new location**
  - In many ways, this is like rename… except that it can also change the location of where the newly renamed file or directory resides

Example

Rename a file

```
$ mv index.html contact_us.html
```

Example

Move the file `index.html` to a new directory (`Documents` directory) in my home directory (~)

```
$ mv index.html ~/Documents
```

`cat` - View the Contents of a File (and other things)

- The `cat` (concatenate) command has several uses, but one of the most common is to display the contents of a file in the terminal window

Example

View the contents of a file

```
$ cat index.html


<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home</title>
</head>
<body>
   <!-- markup goes here →
</body>
```

- More examples of the `cat` command can be found here: https://en.wikipedia.org/wiki/Cat_(Unix)#Examples

Exercise

In this exercise, we will create the file structure for an HTML/CSS/JavaScript project. This exercise will simulate the basic workflow you will use on a daily basis for starting new projects.

Step 1: The following diagram shows what this project directory will look like by the end of this exercise. Take a few minutes to study it.

```
basic-project/
├── index.html
└── css/
    └── styles.css
└── images/
└── scripts/
    └── index.js
```

Note: Anything in the "tree" view above that ends with a forward-slash (`/`) is a *directory*, and everything else is a *file*.

Step 2: Create your project directory

Open Git Bash. Now, navigate (with `cd`) to your HCA directory where you will keep all of your projects. Confirm you are in the proper directory by running the `pwd` command.

Make sure you are in your HCA code folder first. Then, reate a new subdirectory within your HCA coding directory for this project named `basic-project`. Finally, `cd` into the project directory.

```
$ pwd
$ mkdir basic-project
$ cd basic-project
```

NOTE: If you see a `$` at the beginning of a line like this in this course, on the Web, or in a book, it refers to your command prompt. You should not *type* that symbol yourself. Just type whatever follows the `$`.

Step 3: `touch` (cireate) your first file!

Now that you are in the `basic-project` directory, create an `index.html` file using the `touch` command. You might use the `pwd` command first to make sure you are in the correct folder.

```
$ pwd
$ touch index.html
```

NOTE: It is always good to use `pwd` and `ls` frequently as you change directories, touch files, or make directories, so that you can keep track of where you are and what changes you've made.

```
$ pwd
$ ls
```

Step 4: Make more directories!

Make the rest of the directories described in Step 1. You will need three more directories: `css`, `images`, and `scripts`.

```
$ mkdir css
$ mkdir images
$ mkdir scripts
```

Use `ls` to examine the changes.

The file structure should now look something like this:

```
basic-project/
├── index.html
└── css/
└── images/
└── scripts/
```

Step 6: Add a `styles.css` file to the `css` directory.

Use the `cd` command to change into the `css` folder. Then, create `styles.css`.

```
$ cd css
$ pwd
$ touch styles.css
```

Step 7: Add an `index.js` file to the `scripts` directory.

Use the `cd` command to change into the `scripts` folder. You will need to navigate back to the project folder first. Then, create `styles.css`.

```
$ cd ..
$ pwd
$ cd scripts
$ pwd
$ touch index.js
```

Again, we will want to use `ls` to examine the changes.

Step 8: Prepare to code by opening Visual Studio Code.

You should be able to launch Visual Studio Code from the command prompt and have it open the current folder with the following command. Make sure you are at the root of the project folder first.

```
$ pwd
$ cd ..
$ pwd
$ code .
```

Module 2

Version Control and Git Basics

Section 2–1

Overview of Git

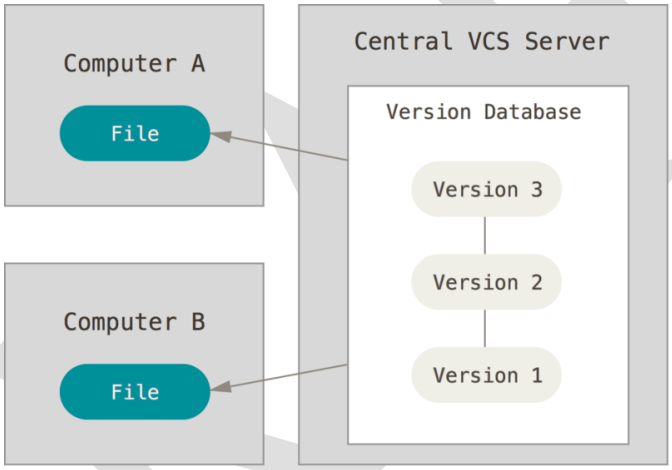
What is Version Control?

- **Version control systems help developers keep track of changes to their source code over time**
  - Commonly referred to as SCM (***Source Code Management***)
- **The idea is to keep track of every modification made to the code base**
- **You "commit" your changes and store them in a repository**
  - Each time you add "something that works" you generally do a commit
  - This might be a several times per hour if you are productive
- **SCM features allow you to roll back to a previous commit if needed**
  - You can also "roll forward" if you've already rolled back
- **Tools also let you compare code in different commits**
  - It can help you figure out what changes might have caused a bug
- **Version control systems can be:**
  - centralized
  - distributed

Centralized Version Control

- **Centralized Version Controls systems store a single "central" copy of the versions of your project on a server somewhere**
  - A popular centralized version control system is SVN (Subversion)

- **Developers "commit" their changes to this centralized version control system**
  - If the server isn't available (no internet access? the server is down?), the developer is not able to save changes to their work within the Version Control System



Typical Workflow for CVC

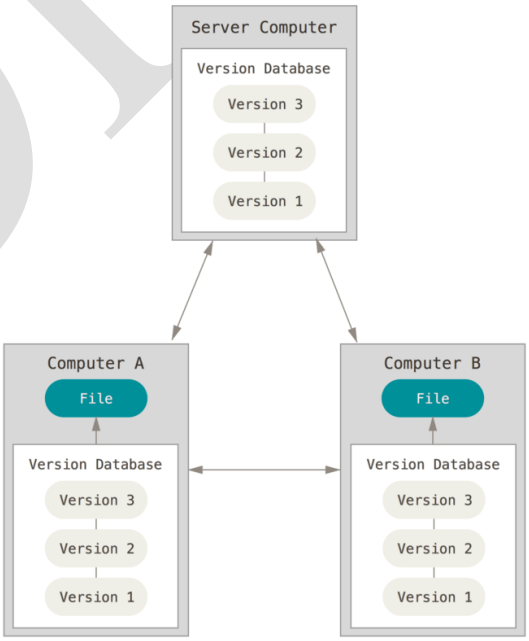
- Developer checks out code from version control or pulls down the latest changes to code already checked out
- Developer makes some changes and tests their work
- Developer commits changes back to a central repository
  - Once changes are committed, others can pull them into their local copies

Distributed Version Control

- **Distributed Version Control systems do *not* rely on a central server to store all the versions of a project's files**
  - Each developer has a copy of the repository and all of the versions
  - A popular Distributed Version Control system is Git

- **However, cloud based services like GitHub, GitLab and BitBucket can provide a centralized (additional) copy of a repository**
  - To work on a project, each developer could create a repository on their local machine or "clone" an existing repository from the cloud service

- **The developer's local repository would contain the full history of the project**
  - A developer can make changes to the code and commit those changes to their local repository
  - When they are ready to share changes with other developers, they "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 the latest changes to code already checked out
- Developer makes some changes and tests their work
- Developer makes a local "commit" that represents the changes made
- Developer repeats this cycle locally until they are ready to share their changes with other developers
- Developer commits changes back to the central repository for others to "pull" into their local copies or provide a patch file to another developer if they choose not to use a service like GitHub

Section 2–2

Git Basics and the Local Repository

Git

- **Git is a free, open source distributed version control system**
  - It can handle both small and very large projects
- **It is easy to learn and has a tiny footprint**
  - Most importantly, it is very fast
- **Before you start using Git, install it on your developer machine from `https://git-scm.com/downloads`**
  - Note: Your machine should have been configured before the class started and Git will already be on it
- **To run Git commands, you can use:**
  - Git Bash
  - Command Prompt window
- **NOTE: We will use Git Bash throughout session 1**
  - Afterwards, you can use Git Bash or the Command Prompt window -- whichever you prefer

Basic Command : `git`

- The top-level Git command is `git`
- The `git` command is followed by an action you want Git to execute or a set of flags
- Run the following command to check the version of Git installed

```
$ git --version
git version 2.8.1
```

Git Setup

- **Git stores configuration information in `~/.gitconfig`**
  - This global configuration contains the settings for all of your repositories

- **Each repository also has a local configuration file named `.git/config` and it affects only the repository in which it is located**

- **You can set key/value pairs in the config file using the command `git config`**

Setting the user's name

```
$ git config --global user.name "Dana Wyatt"
```

Setting the user's email

```
$ git config --global user.email "danawyatt@myemailprovider.com"
```

- **NOTE: It is possible that this will already be configured on your machine**

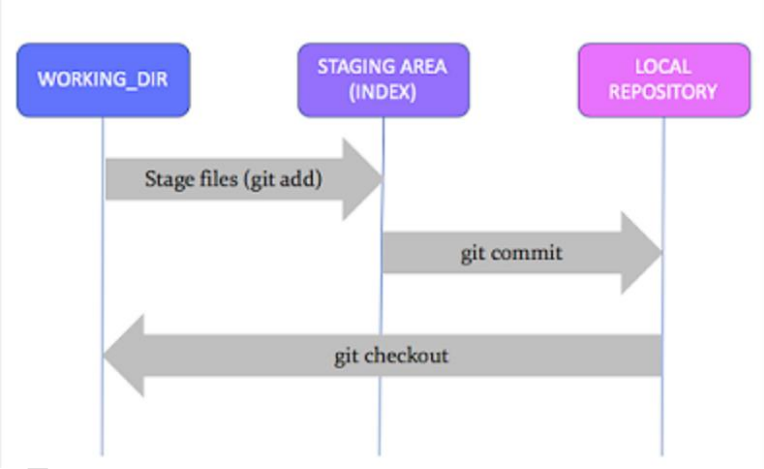
- **To get a list of all of the config values, use the `--list` flag**

```
$ git config --list
```

Git Areas

- **There are three core areas where files and folders are maintained within Git:**
  - The Working Tree is the folder(s) where you are currently working on your source code
    - Files here are called *untracked* files
  - The Staging Area (also known as Index) is where you place things you plan to commit to the repository
    - It allows you to "batch" or "box" a set of changes into one commit
  - The Local Repository is a collection of your checkpoints or commits
    - It is the area where everything is saved

- **Learning Git is mostly about learning how and why to move changes from one Git area to another, or learning to query what changes have been committed**



Creating a Repository: `git init`

- **The `git init` command creates a Git repository that the directory you are in when this command is run should be tracked by Git**
  - It creates the staging area and the local repository in a HIDDEN folder named `.git`
  - The hidden folder is also referred to as the git internals directory

Creating a Local Repo in the current folder

```
$ mkdir Repo1
$ cd Repo1
$ git init
```

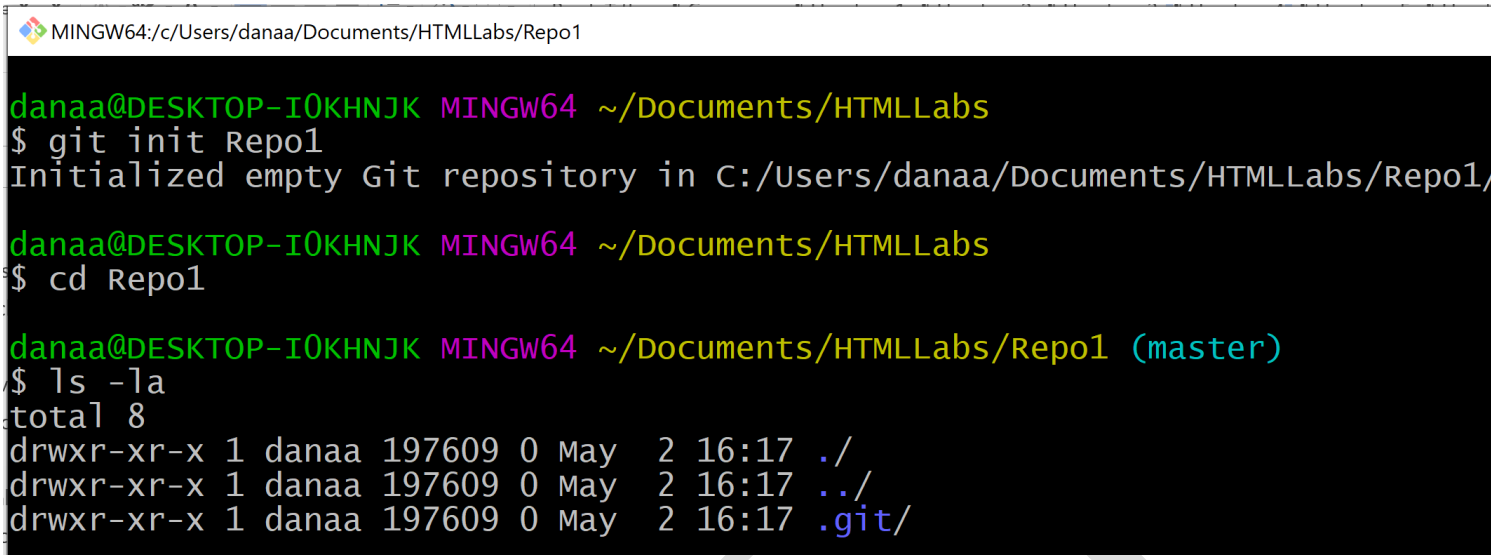
- **The `git init` command can create the repository in a subfolder of the current folder by using the new repo's name after `init`**

Creating a Local Repo in a specified folder

```
$ git init Repo1
Initialized empty Git repository in /Path/to/repo/.git/
$ cd Repo1
```

- **When you initialized the Git repo, it created a hidden folder named `.git`**
  - The `.git` folder in turn contains a set of files and directories that hold the repository's commits and history

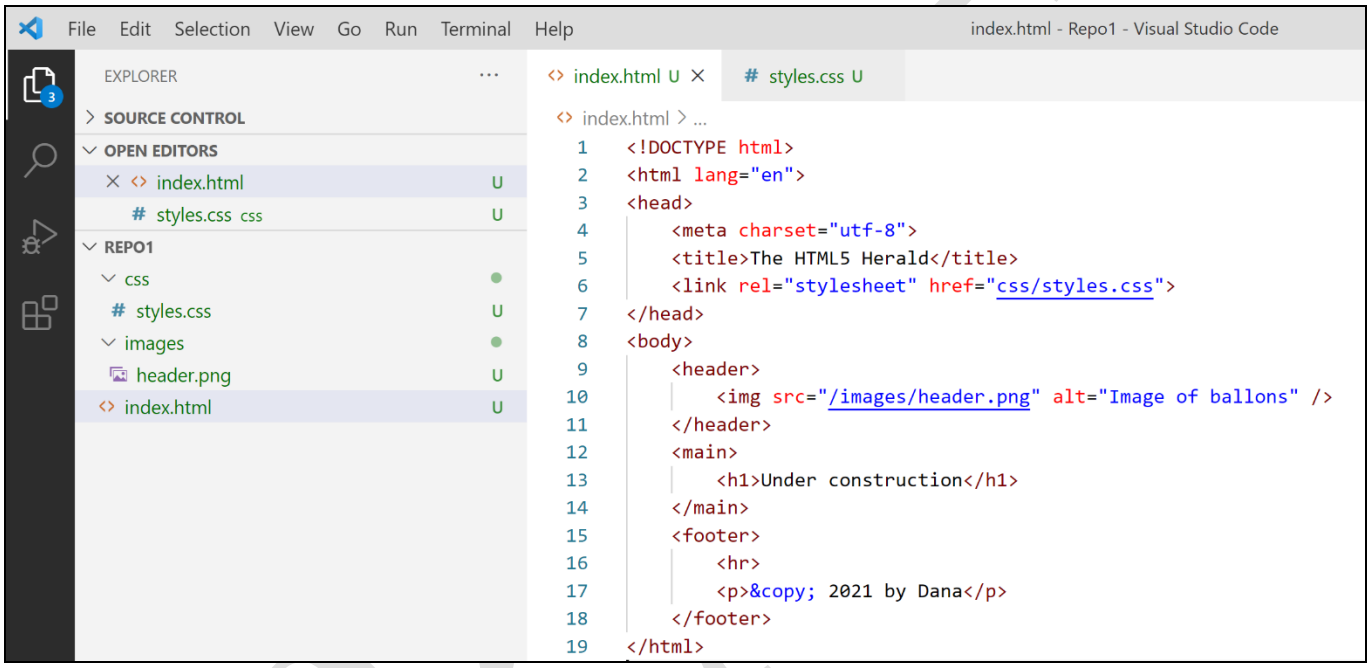
- **You can see the directory using `ls -la`**



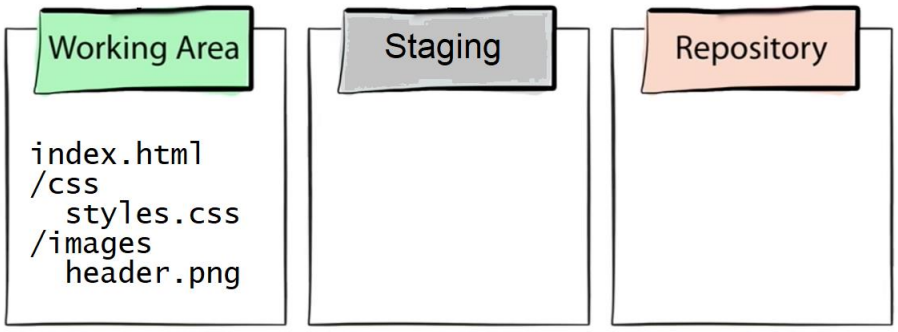
  - We won't concern ourselves with the physical structure of `.git` -- suffice to say it holds the staging area and local repository

Working on a Project

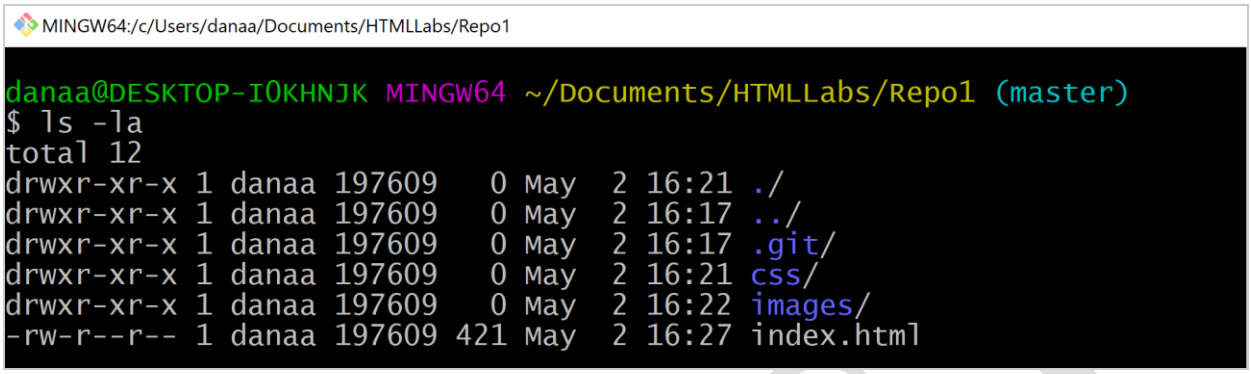
- **At this point, open Visual Studio Code in the `Repo1` folder and begin to work on your website**
  - Typically, you get your project structure set up and get ready to do the heavy coding and then pause to commit your starting code



- **The folder you created in the `Repo1` folder are in the repository's Working Directory**



- **Nothing special has happened yet… they are just files in a folder!**



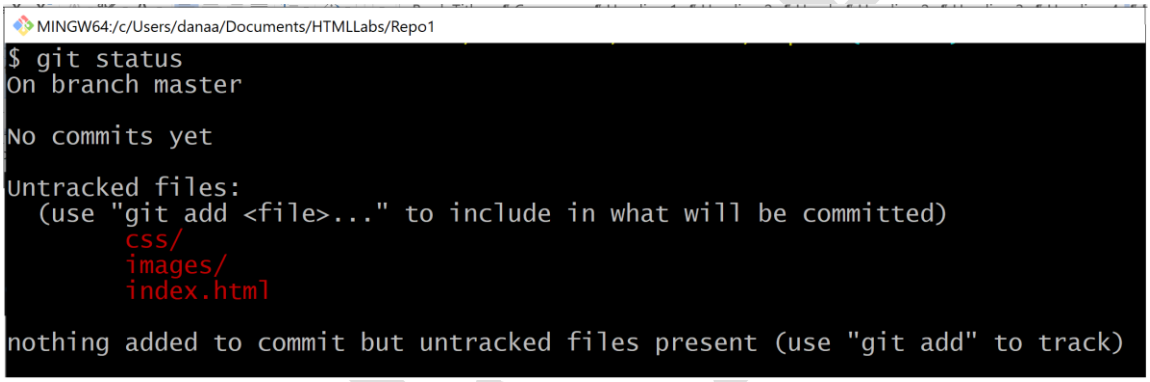
Checking the Status: `git status`

- **The `git status` command displays the state of the working directory and the staging area**
  - It shows you which changes have been staged and which haven't
  - It also shows you which files aren't being tracked by Git

```
$ git status

... response varies based on status ...
```

Checking status before staging



- **The red coloring catches our attention and helps us realized these files/folders aren't tracked**

Adding Files to the Staging Area: `git add`

- **When you are ready to commit the current "version" of your code, use the `git add` command followed by a path to one or more files to tell Git to copy them to the staging area**
  - This let's Git know that the files should be ***tracked*** by version control and staged for the next commit

You can stage a single file

```
$ git add index.html
```

You can stage many files using wildcards

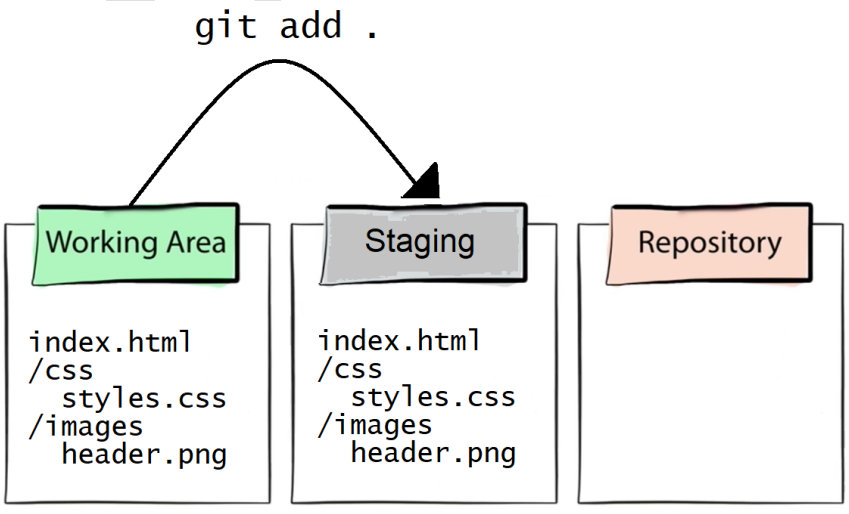
```
$ git add *.html
```

You can stage a whole folder

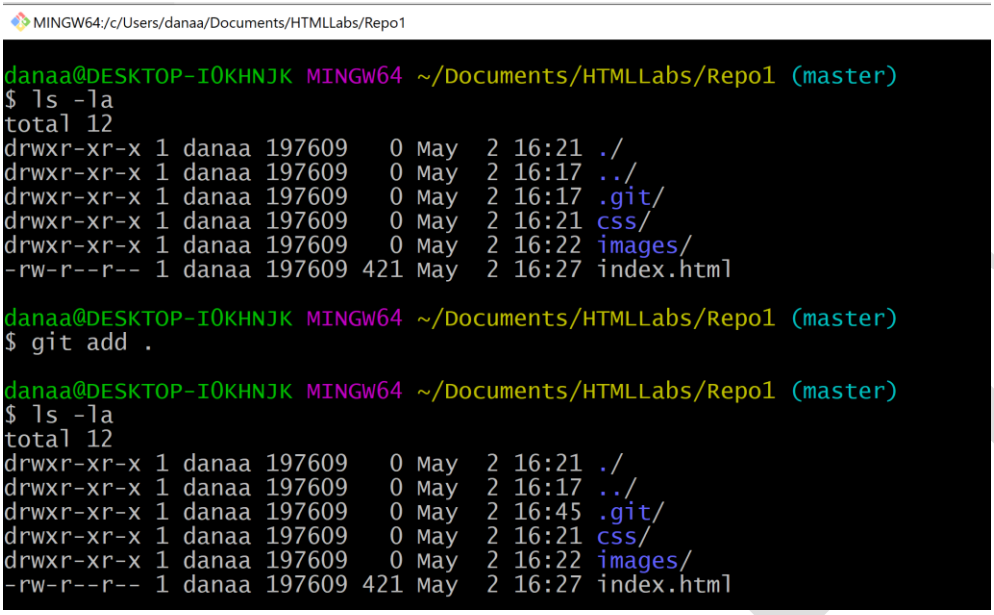
```
$ git add images
```

You can stage everything in Working Directory

```
$ git add .
```

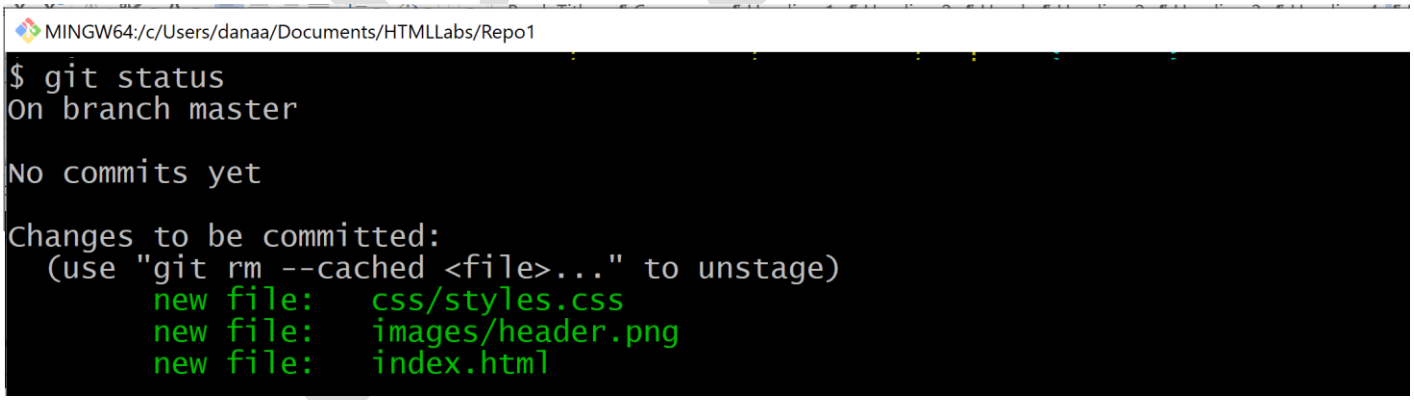


- **From a simple glance, it doesn't look like `git add` did anything because you don't see any changes in the project's "visible" folder**



- **The staging area is implemented in the hidden `.git` folder**

- **However, the `git status` command confirms that the Working Directory has been staged**



- **Git won't track empty directories**
  - This might be a problem if you want to add an empty folder (ex: images) where you are going to add files later on

- **The solution to this problem is to place an empty file in the folder that you won't really use**
  - People usually name this file `.gitkeep` or `.keep`

- **By starting the file name with a dot (`.`), most people won't confuse it with a code file in the project**
  - In fact, files that start with a dot are hidden when you list the content of the folder using the `ls` command in most operating systems

Committing Changes to the Repo: `git commit`

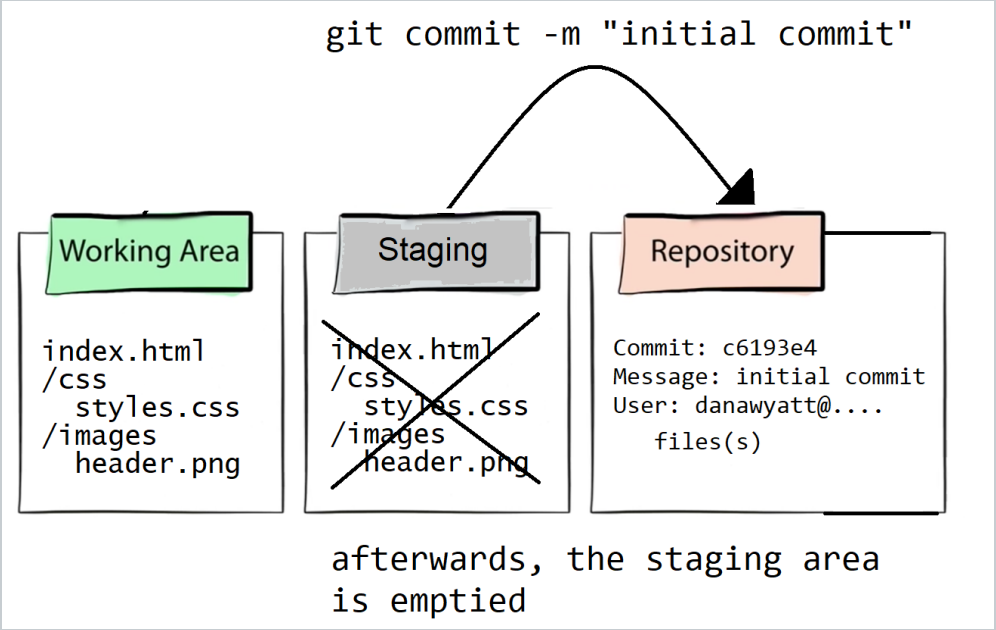
- **The `git commit` command takes the staged files and commits them to version control**
  - It creates a point in the version control history that can be referenced later

- **You can add a short message to the commit**
  - This means when you look at the commit history later, you will understand what is "inside" the commit
  - Although your files in the Working Directory are saved when you use the File -> Save menu in Visual STudio Code, you can think of this as the moment you are saving your ***changes***.

Committing changes

```
$ git commit -m "initial commit"
[master (root-commit) c6193e4] initial commit
 3 files changed, 19 insertions(+)
 create mode 100644 css/styles.css
 create mode 100644 images/header.png
 create mode 100644 index.html
```

  - Git commits are assigned a hex number that can be used to find and query the commit



- **After the commit, the status shows that there is nothing to commit**

```
MINGW64:/c/Users/danaa/Documents/HTMLLabs/Repo1

danaa@DESKTOP-IOKHNJK MINGW64 ~/Documents/HTMLLabs/Repo1 (master)
$ git status
On branch master
nothing to commit, working tree clean
```

- **You can't see the commits unless you look in the hidden `.git` folder**

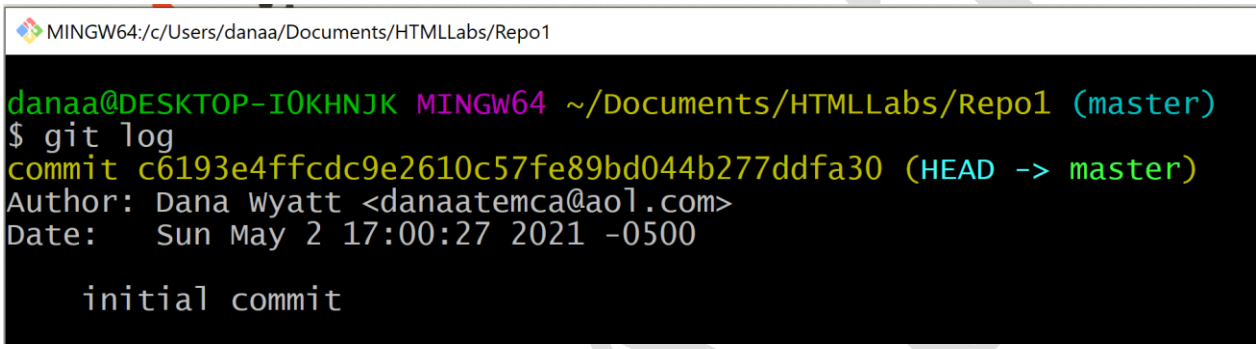
```
MINGW64:/c/Users/danaa/Documents/HTMLLabs/Repo1

danaa@DESKTOP-IOKHNJK MINGW64 ~/Documents/HTMLLabs/Repo1 (master)
$ ls -la
total 12
drwxr-xr-x 1 danaa 197609   0 May  2 16:21 ./
drwxr-xr-x 1 danaa 197609   0 May  2 16:17 ../
drwxr-xr-x 1 danaa 197609   0 May  2 16:17 .git/
drwxr-xr-x 1 danaa 197609   0 May  2 16:21 css/
drwxr-xr-x 1 danaa 197609   0 May  2 16:22 images/
-rw-r--r-- 1 danaa 197609 421 May  2 16:27 index.html
```

Checking the Commits: `git log`

- The `git log` command displays information about the previous commits

```
$ git log
  ... shows a list of previous commits ...
```



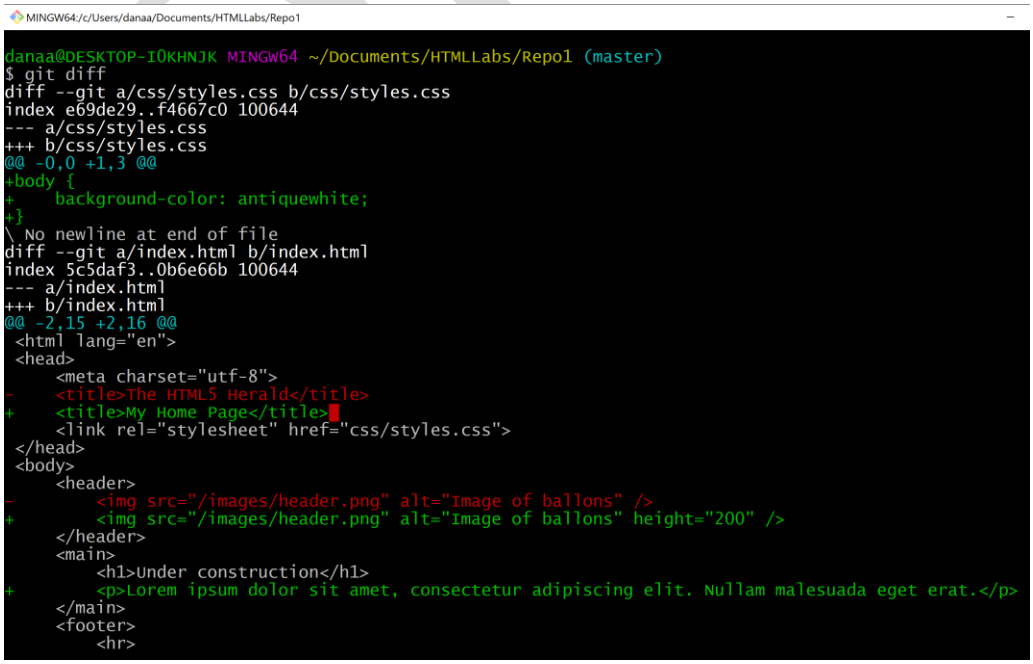
Comparing Differences: `git diff`

- **The `git diff` command displays line-by-line differences between files in your repo compared to the last commit**
  - By default, it displays them to the console
  - You can configure a graphical tool to show the differences in an easier to read manner

- **Before we can see the diff command in action, let's make some changes to the code in our current Working Directory so that it differs from the last commit**
  - We change the title in `index.html`
  - Added a `height` attribute to the image in `index.html`
  - We added a paragraph to `index.html`
  - We added a new page named `contact_us.html`
  - We added a `body` style rule to `styles.css`

```
$ git diff

  ... shows the difference between files in the working directory and the last commit ...
```

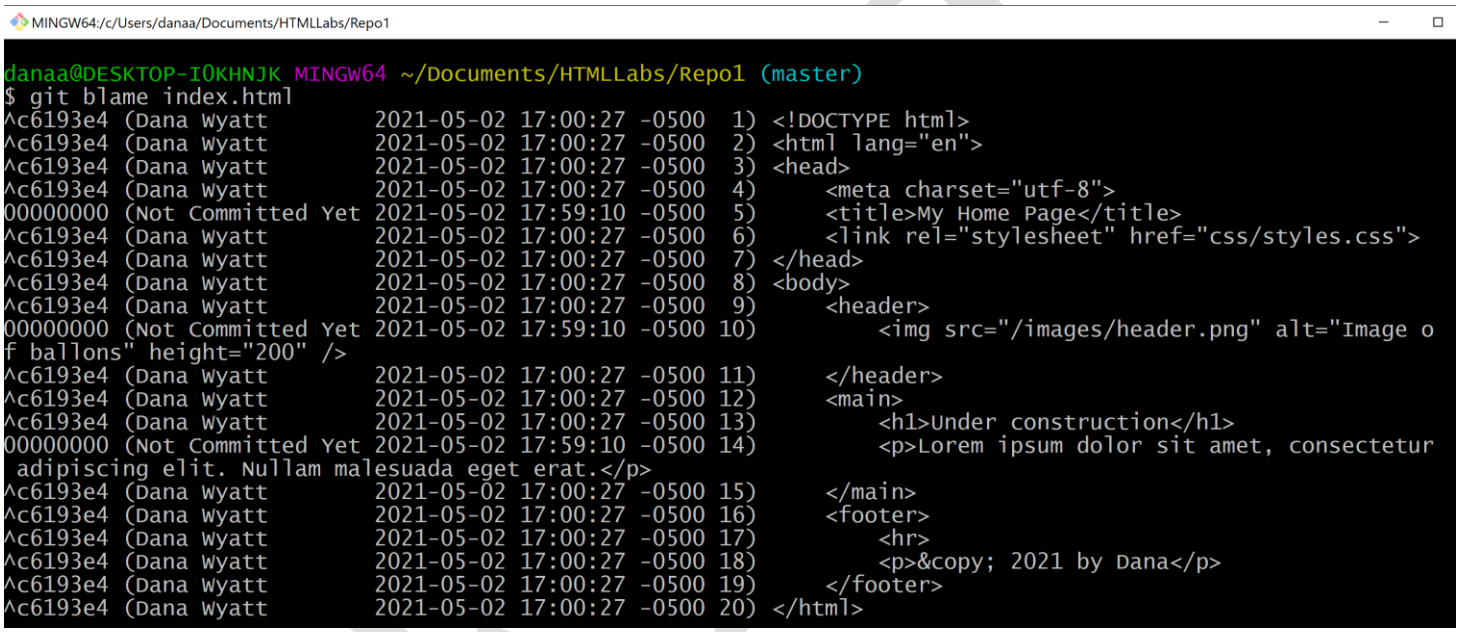


Figuring Out Who Made Changes: `git blame`

- The `git blame` command shows what revision and author are related to each line in a file tracked by version control

```
$ git blame index.html

... responds with a line-by-line list of index.html
```



Ignoring Files

- **Sometimes, your project has files that you want Git to intentionally ignore**
  - In a JavaScript project, this often means Node.js modules
  - In a C/C++/C# project, this might mean object and compiled files
- **We want Git to ignore these files because we can reinstall/rebuild them**
  - Node.js modules can be ***reinstalled*** using NPM
  - C/C++/C# object and compiled files can be ***rebuilt*** by compiling the project
- **By not tracking these very large files, the commit process and the process of pushing the repository to a cloud service like GitHub or BitBucket is much faster**
  - It also makes the cloning or pulling from a remote repository faster too!
- **To ignore files, add a file named `.gitignore` to root of project**
  - Within `.gitignore`, identify the files to be ignored
- **The `.gitignore` can list individual files, or include wildcards and regular expressions**

```
# Node modules
node_modules/
#Logs
logs/*.log
# Other files
*.pdf
```

- **When you put `.gitignore` at the root, it applies to the whole project**
  - You can also put a `.gitignore` file in individual folders for finer grain control over the process

- **To learn more about `.gitignore`, see: `https://git-scm.com/docs/gitignore`**

- **To see examples of `.gitignore` files, see: `https://github.com/github/gitignore`**

References

- A cool Git cheat sheet:
  `https://education.github.com/git-cheat-sheet-education.pdf`

- Official docs on Git Internals:
  `https://git-scm.com/book/en/v2`

- Quick explanation of Git internals:
  `http://gitready.com/advanced/2009/03/23/whats-inside-your-git-directory.html`

Exercise

In this exercise, we will create our first repository using common commands and point out some quirks like how Git works with empty directories.

Step 1: Create the following directory structure

Create the following project structure in your HCA code

```
git_exercise/
    ├── index.html
    └── css/
    └── images/
    └── styles/
```

Step 2: Initialize the repository in this existing project.

Navigate to the root of the project directory. Then run `pwd` to confirm you are there.

Run `git init` to initialize your repository

```
$ pwd
$ git init
```

Step 3: Check the "status" of your repository

Run `git status` and you should see output similar to the following:

```
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  index.html
nothing added to commit but untracked files present (use "git add" to track)
```

Notice that it shows `index.html` as an untracked file. Where are your directories? Git does not keep track of empty directories.

Step 4: Handle empty directories

Add a `.gitkeep` file to your `css`, `images` and `scripts` directories. This allows them to be tracked by Git.

Step 5: Check the "status" of your repository using the proper git command.

You should now see your directories being tracked.

```
On branch master

Initial commit

Untracked files:

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

  css/

  images/

  styles/

  index.html

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

Step 6: Stage the files so that they can be committed

Run `git add .` to add our files to our staging area to be committed. `git add` is the command to stage files for a commit. `.` is everything in the current directory.

Step 7: Check the "status" of your repository

Run `git status` to see the status of your repository. You no longer have untracked files. You have a list of files to be committed.

```
On branch master

Initial commit
```

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   css/.gitkeep
  new file:   images/.gitkeep
  new file:   scripts/.gitkeep
  new file:   index.html
```

Step 8: Create our first commit

Run `git commit -m "Initial commit"`. This command will take all the files to be committed (staged files) and create a point in history related to these changes.

When you create a commit, it should encompass all the changes related to a certain task or logical set of changes

Step 9: Check the log

If you run `git log` , you will see your commit history

```
commit bddf8d41d0f8f6d0c9ebdb0677e2913b55da9311
Author: Dana Wyatt <dana.wyatt@danawyatt@myemailprovider.com>
Date:   Wed Feb 15 16:36:20 2017 -0500
    Initial commit
```

Step 10: Edit `index.html`

Open the project in Visual Studio Code3.

Then, open `index.html` and add the basic HTML structure to the page

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Home</title>
</head>
<body>


</body>
</html>
```

Step 11: Check the "status" of your repository

Run `git status` to see that you have modified the `index.html` file since your last commit

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

Step 12: View the differences between now and your last commit

Run `git diff` to see what has changed in the file(s) that have been modified. Lines added to the file will have a **+** next to them. Lines removed from the file will have a **-** next to them.

Step 13: Stage the new changes for commit and review repo status

Let's stage these changes using `git add .`. Run `git status` to now see that the files are listed as changes to be committed.

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

```
  modified:   index.html
```

Step 14: Commit the changes to `index.html`

Commit your changes with a meaningful message.

```
git commit -m "added basic html to index.html"
```

Step 15: Check the commit log

Run `git log` and see that we have another entry in the log for our new commit

```
commit 789b028b04bfe0efc080827855dd00f79d20433d
Author: Dana Wyatt <dana.wyatt@myemailprovider.com>
Date:   Wed Feb 15 16:47:32 2017 -0500

    added basic html to index.html

commit bddf8d41d0f8f6d0c9ebdb0677e2913b55da9311
Author: Dana Wyatt <dana.wyatt@myemailprovider.com>
Date:   Wed Feb 15 16:36:20 2017 -0500

    Initial commit
```

Step 17: Compare commits with 'git diff'

Run `git diff` to see what was different between commits. You will need to provide the actual hash from the commits you are comparing. You can find them in the commit log.

```
$ git diff INITIAL_COMMIT_HASH SECOND_COMMIT_HASH
```

Module 3

Branching and Merging with Git

Section 3–1

Working with Branches

Branching

- **Branching allows us to pursue a new line of development**
  - We can keep it separate and distinct from the *master* branch

- **By doing this, we can:**
  - We can keep code in development separate from code in production
  - Add a new feature while keeping the master branch stable
  - Test out an idea that we aren't sure we want to keep

- **In the diagram you just saw, the Develop branch is used during the coding phase**
  - New features branch off from there

- **When a new feature is completed, it can be merged back in the Develop branch**

- **When it is time to release the product, you can merge it into a Release branch**
  - By keeping code here, you know EXACTLY what code went live

- **The "master" copy of working code is merged back into the master branch**

- **When the next set of changes need to be made to the program, you can branch from Release**

Practical Example of Branching

- **You are going to create a new feature in a project that will take some time to complete**
  - Make a new branch and start coding the new feature
- **Suddenly, an emergency change comes in that has to be pushed live**
- **Switch back to master (or whatever branch has the issue)**
  - Create another branch for the emergency changes
  - Make the changes
  - Commit the changes
  - Merged the emergency change branch back to master
- **Switch back to your feature branch and continue working on the task**

Section 3–2

Merging and Branching Commands

Creating a Branch: `git branch <name>`

- The `git branch` command creates a branch with the specified name

Creating a branch

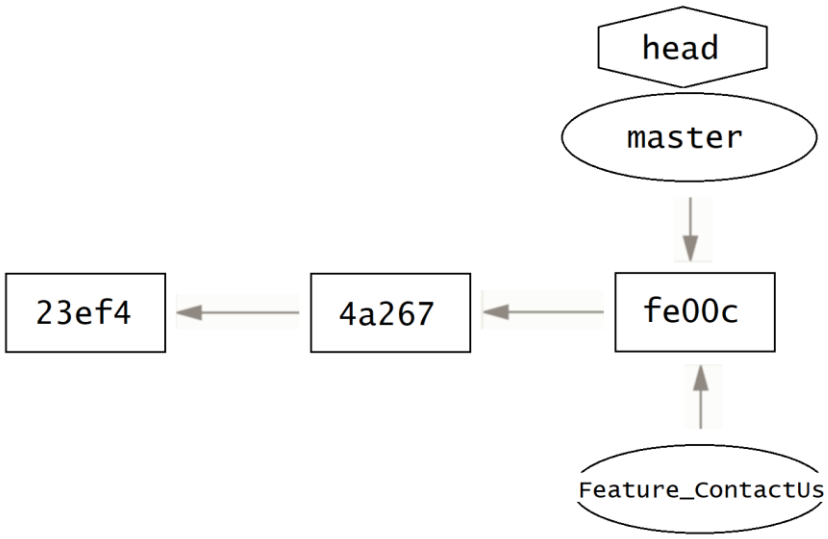
```
$ git branch Feature_ContactUs
  ... still in previous branch ...
```

- However, it doesn't checkout (switch) to the branch
  - You must checkout the branch in a separate command

What Happens When You Create a Branch?

- **Under the hood, Git maintains a reference called "head" to the branch you are working on**

- **Creating a new branch creates a new reference to the branch you are working on**
  - However, it doesn't switch the head reference to the new branch



Checking Out a Branch: `git checkout`

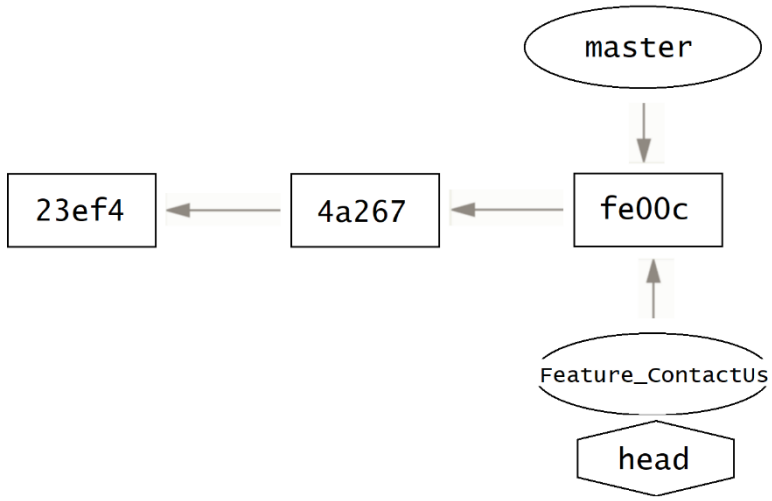
- **Checking out a branch with the `git checkout` command switches the head reference to the checked out branch**
  - The code in the Working Directory is replaced with the code in the branch

Checking out a branch

```
$ git checkout Feature_ContactUs

Switched to branch 'Feature_ContactUs'
```

- **The command can also be used to checkout a specific commit, tag, or file**
  - A tag is essentially a named commit



Creating a Branch and Checking it Out in One Command: `git checkout -b <name>`

- The `git checkout` command with the `-b` flag creates a branch with the specified name and checks it out at the same time

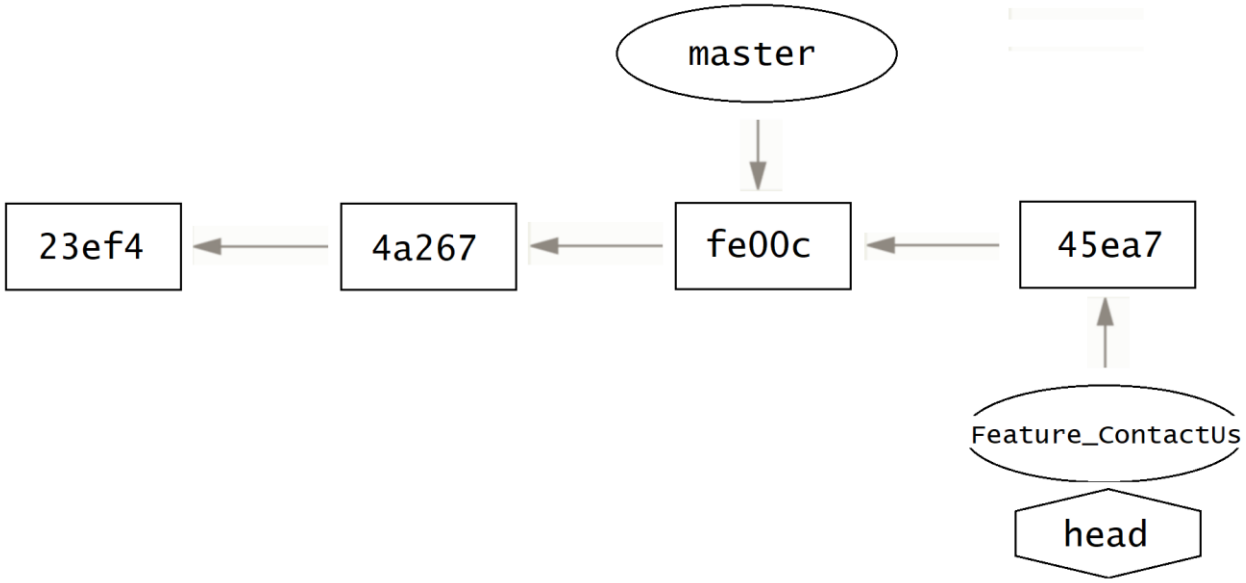
Creating and checking out a branch - One Step!

```
$ git checkout -b Feature_ContactUs

Created and switched to branch 'Feature_ContactUs'
```

Commits on a New Branch

- **When you have a commit on a new branch, your graph resembles:**



Switch -- a new command

- **Git 2.23 added the `git switch` command as a substitute for checking out a branch**
  - The working tree and staging area are updated to match the branch
  - It is essentially checkout with a new name!

Switching to a branch

```
$ git switch Feature_ContactUs
Switched to branch 'Feature_ContactUs'
```

- **The `-c` flag lets you create a branch and then switch to it**

Creating and switching to a branch - One Step!

```
$ git switch -c Feature_ContactUs
Created and switched to branch 'Feature_ContactUs'
```

- **Under the hood, it is the same as:**

```
$ git branch Feature_ContactUs
$ git checkout Feature_ContactUs
```

Viewing all Branches: `git branch`

- **The `git branch` command shows you what branches exist in your repository and what branch you are on**
  - Your current branch is marked with a `*`

Viewing the Branches

```
$ git branch
  Feature_ContactUs
  Feature_ProductsPage
  Feature_CheckoutPage
  Feature_ContactManufacturer
  * master
```

Delete a Branch: `git branch -d`

- If you create an experimental branch and decide you don't want to keep it, you can delete the branch using the `git branch -d` command

Deletes a local branch from your local repository

```
$ git branch -d Feature_ContactManufacturer

Deleted branch Feature_ContactManufacturer (was a6adcc2).
```

Section 3–3

Merging Branches

Merge Branches: `git merge <name>`

- The `git merge` command 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
  - Assuming there are no merge conflicts!

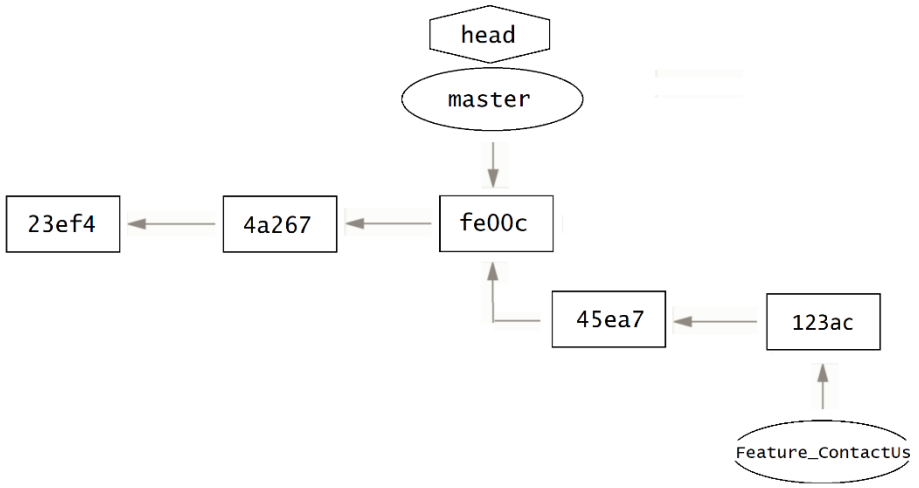
Merging a branch into master

```
$ git switch master
$ git merge Feature_ContactUs
Updating a6adcc2..a71b4f2
Fast-forward
 contact.html | 9 +++++++++
 1 file changed, 9 insertions(+)
```

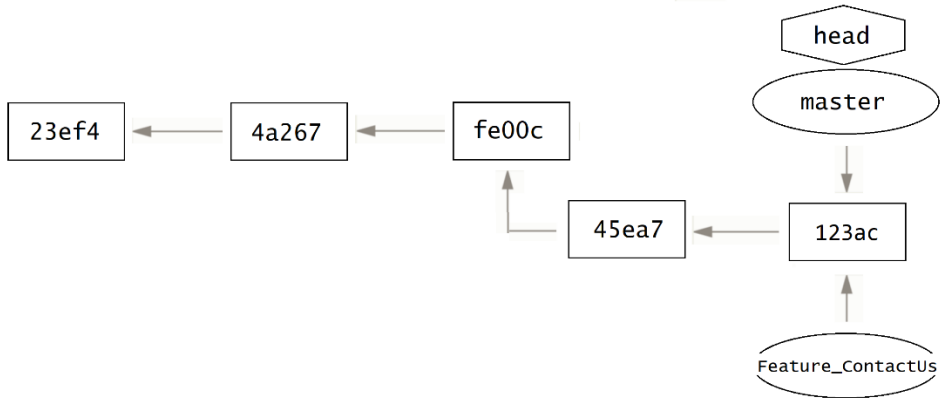
Fast-Forward Merge

- **A fast-forward merge occurs when the path from the current branch tip to the target branch is linear**
  - To merge the branches, all Git has to do to integrate the histories is move (i.e., "fast forward") the current branch tip up to the target branch tip

BEFORE MERGE



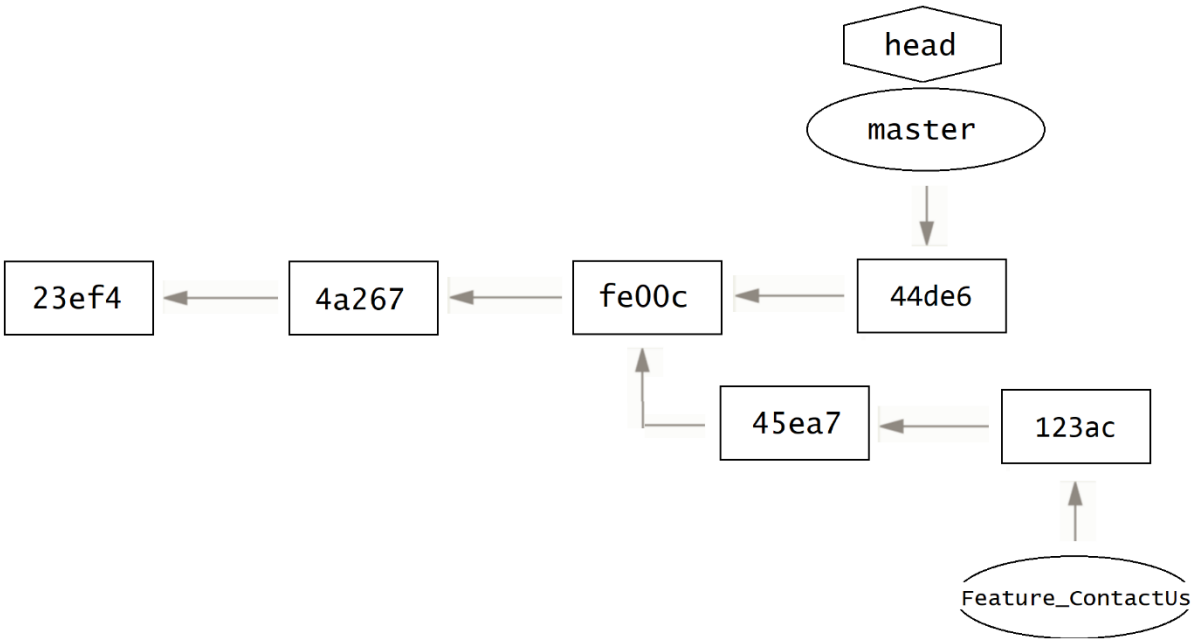
AFTER MERGE



Merge Conflicts

- **A merge conflict can happen when you try to merge two branches that have made edits to the same line in a file**
  - It can also happen when a file has been deleted in one branch but edited in the other

- **Conflicts usually happens when many developers are working on the same project**
  - In the diagram below, someone has done a commit on the master branch that contains changes that the Feature_ContactUs branch doesn't know about



- **In many cases, Git will figure out how to integrate new changes (it is incredibly intelligent!)**
  - But sometimes you have to resolve the conflict yourself

- **When Git can't resolve the conflict, it actually edits the conflicted files and injects markers to help you see/resolve the conflicts**
  - Use the `cat` command to see the file or examine it in Visual Studio Code

Using git status to show conflicts

```
$ git status
# On branch contact-form
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:   contact.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Resolving Merge Conflicts

- **When you issue the merge command the Git discovers conflicts, it marks the problem area *in the file* by enclosing it in "<<<<<<< HEAD" and ">>>>>>> [*other_branch_name*]"**

Example of file contents showing merge conflict markers

```
1  <<<<<<< HEAD
2  This line was committed while working in the "login-box" branch.
3  =======
4  This line, in contrast, was committed while working in the "contact-form" branch.
5  >>>>>>> refs/heads/contact-form
```

- **Examine contents of the conflicted file(s)**
  - The contents after the <<<< angle brackets represent the current working branch (HEAD)
  - A line with "=======" separates the two difference
  - The name of the branch where the conflicts came from is listed after the >>>> angle brackets

- **To fix the conflicts, open the file in an editor and figure out what changes you need to make**
  - This might mean typing new code, editing code, or even removing code!
  - You may also have delete files or restore files!

- **Resolving merge conflicts can take a few minutes or they can take days if there are extensive conflicts!**

- **Fix the conflicts and commit the new changes**

```
$ git add .
$ git commit -m "Merged branch Feature_ContactUs"
```

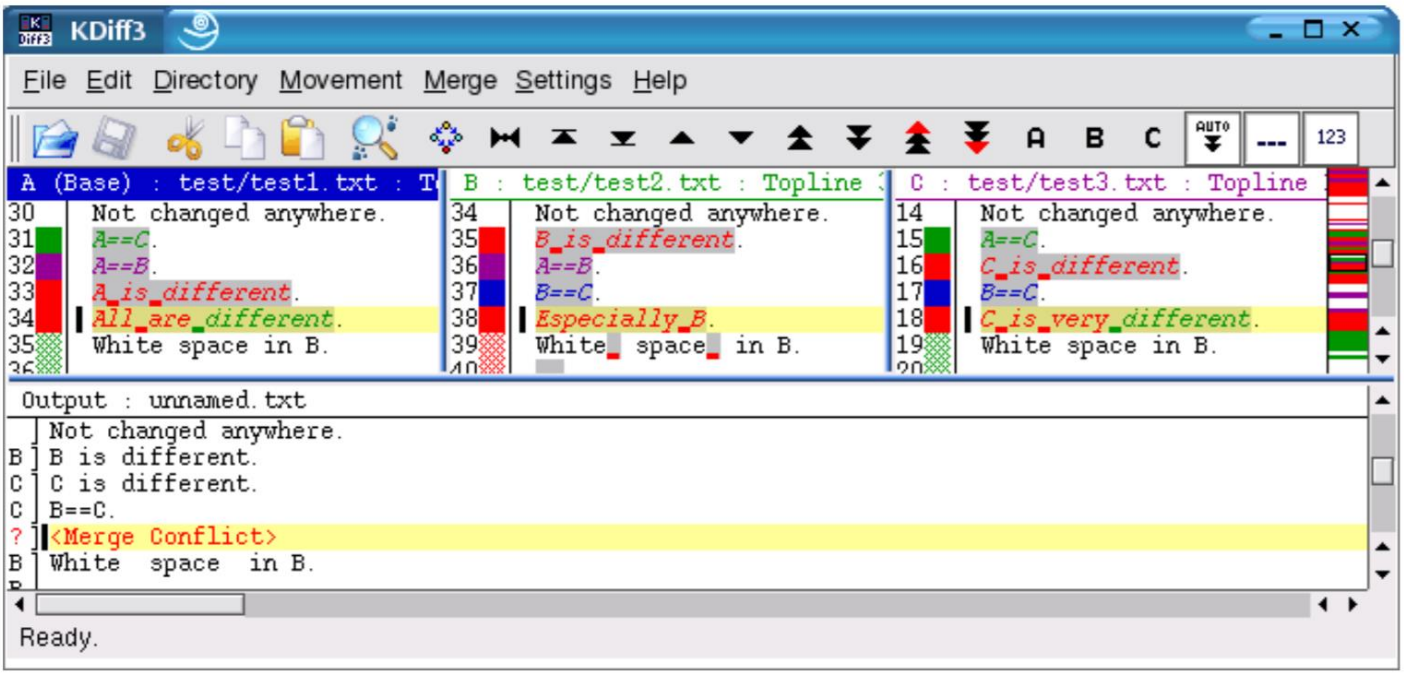
- **If the conflicts are non-trivial, you may want to abort the merge to give you a chance to fix all of the errors**

```
$ git merge --abort
```

Merge Tools

- **You can configure Git merge tools with a sophisticated UI that makes the process easier**
  - One list of tools can be found at: `https://www.git-tower.com/blog/diff-tools-windows/`
  - Examples of how to configure them can be found at: https://stackoverflow.com/questions/137102/whats-the-best-visual-merge-tool-for-git

- **Example (KDiff3):**



Exercise

In this exercise, you will practice branching and merging inside our repository. We will also be a little less specific with directions as you get more comfortable with using Git.

Step 1: Create a branch for a feature

We will add a header to out HTML file and create a branch for this feature. In reality, this is probably too small of a task to have its own branch - but we are just practicing.

Run `git checkout -b AddHeader` to create a branch called AddHeader and checkout that branch

Run `git status` and notice that you are now on the AddHeader branch

```
On branch AddHeader

nothing to commit, working directory clean
```

Step 2: Edit `index.html`

Open up `index.html` and add a header using the `<h1>`

Save your changes.

Run `git status` to show that we are on our branch and that the `index.html` file has been modified.

```
On branch AddHeader

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

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

Step 3: Commit the changes

Run `git add index.html` to stage this file for commit

Run `git commit -m "added a heading to index.html"`

Run `git log` and now you can see that a new commit has been added

Step 4: Explore the differences in the master branch and your feature branch

Checkout the master branch with `git checkout master`

Now, look at the contents of `index.html`. You will notice the header you added is not there! This is because that change only lives on the AddHeader branch.

Checkout the AddHeader branch, `git checkout AddHeader`, and you should see your header again.

Step 5: Merge the feature branch into master

Now checkout master, `git checkout master`.

Merge the AddHeader branch into the master branch by running the command: `git merge AddHeader -m "merged AddHeader branch"`.

NOTE: If you forget the `-m` and your message when committing, Git may popup a text editor for you to provide a message for the merge commit. If this happens, (1) hit `i` on your keyboard and enter the message "merged AddHeader branch", (2) hit `esc` twice, (3) hold `shift` and hit `z` twice.

Now open the `index.html` file while on the master branch and notice the header has been added.

You can check out the commit history and you will see how your project has changed over time.

Some organization then delete the feature branch. Other organizations leave it for history purposes. To delete a branch, use: `git branch -d AddHeader`

Module 4

GitHub - Collaborating with Others

Section 4–1

GitHub

GitHub - A Remote Repository Service

- **GitHub is a cloud-based service that allows can be used to store and manage remote repositories**

- **In addition to providing cloud storage for your repositories, it lets you to make your Git repositories available to other developers**

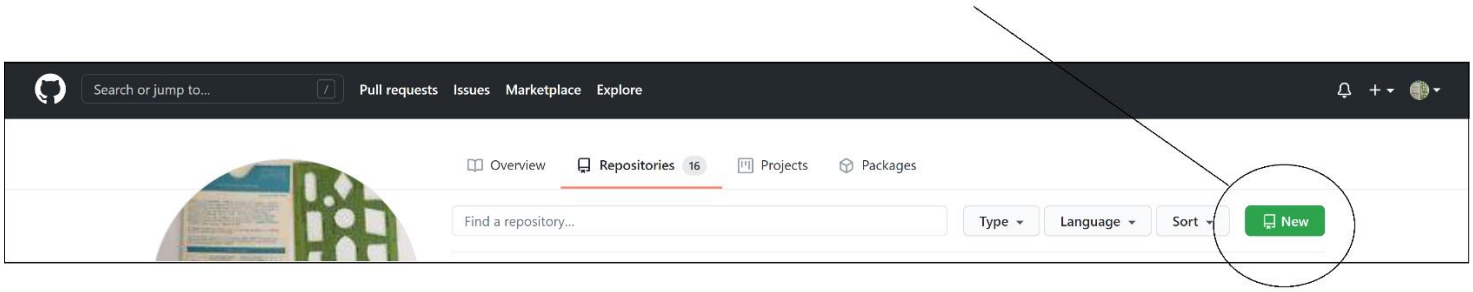
- **Anyone can sign up at GitHub and host public code repositories for free**
  - This makes GitHub very popular as a hosting site for open-source projects
  - Sign up for an account at: `https://github.com`

- **GitHub is a for-profit company and makes money by selling hosted private code repositories and other team-based development services**

- **Many organizations, like The Hartford, host their own internal GitHub cloud service so that they have complete control over its visibility**
  - This is what we will use in class

GitHub User Interface

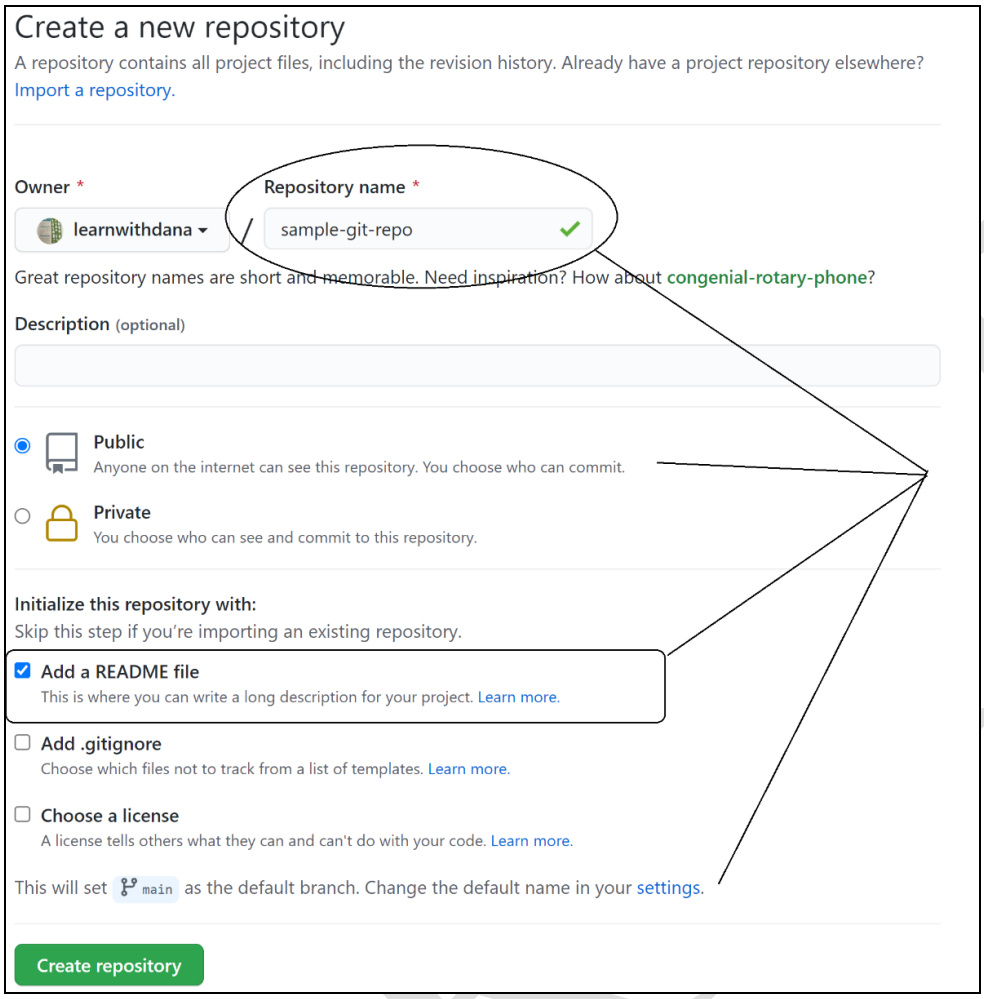
- **GitHub has a web-based graphical interface**
  - It allows you to create new repositories easily



- **It has opinions on how to grant access to your code as well as how people can contribute to your code**

- **It provides several collaboration features for your project, such as:**
  - basic repo management
  - wikis

Creating a Remote Repository



- **After clicking the New button, you can:**
  - Name the repo
  - Set the visibility of the repo (public / private)
  - Add a README file and/or .gitignore file
  - Specify licensing
  - Accept the default master branch name as 'main' or change it to something else

    * Until recently, it defaulted to 'master'

- **There may be small differences on your internal GitHub**

Common GitHub Terminology

- **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

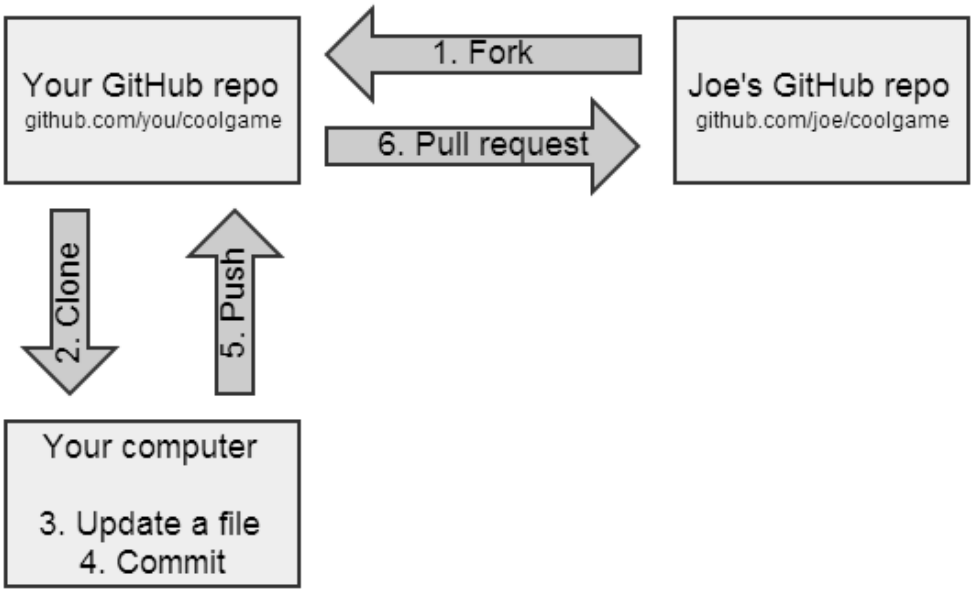
- **Cloning**
  - Cloning a GitHub repository creates a local copy of the remote repo
  - This allows you to make all of your edits locally rather than directly in the source files of the remote repo

- **Forking**
  - Forking is the process of creating you're a copy of a repo from an existing one
  - 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 to your own repo
  - You can then release the revised project as your own version
    * Or you can request that your changes be merged into the original project using a Pull Requests
  - If the original repository that you forked gets updated, you can easily add those updates to your current fork

- **Pull requests**
  - A pull request lets you tell others about changes you've pushed to a branch in a repository on GitHub
    * It is GitHub's way of making a merge request
  - It allows others to provide feedback and have collaboration before the merge happens

* If you make changes in a branch or a forked repository, you 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 review your changes and merge those changes if all is good



Source: https://www.dataschool.io/simple-guide-to-forks-in-github-and-git/

Section 4–2

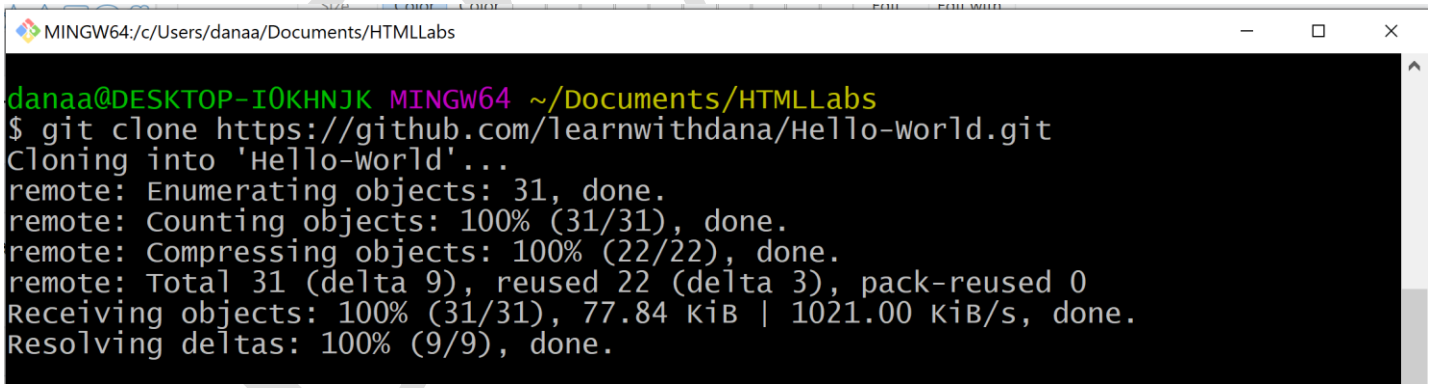
Common Git Commands Used With Remote Repositories

Cloning: `git clone`

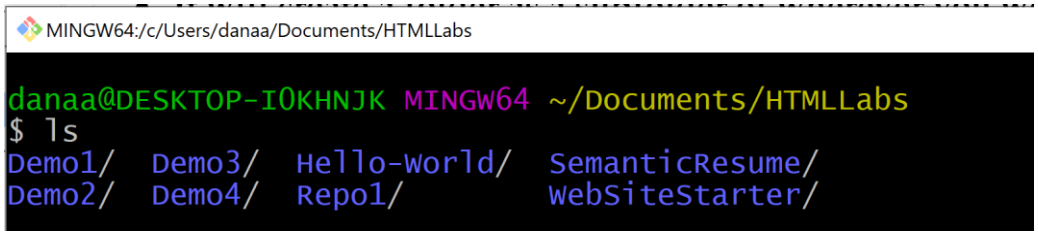
- **The `git clone` command is used to "clone", or make a local copy, of a remote repository**
  - When issued, it is followed by the URL of the remote repository
- **It will create a folder as a subfolder of wherever you were when you issued the command**
- **Depending on the authentication needs, you might:**
  - use HTTPS and specify a username and password
  - Use SSH keys
  - Specify a Personal Access Token

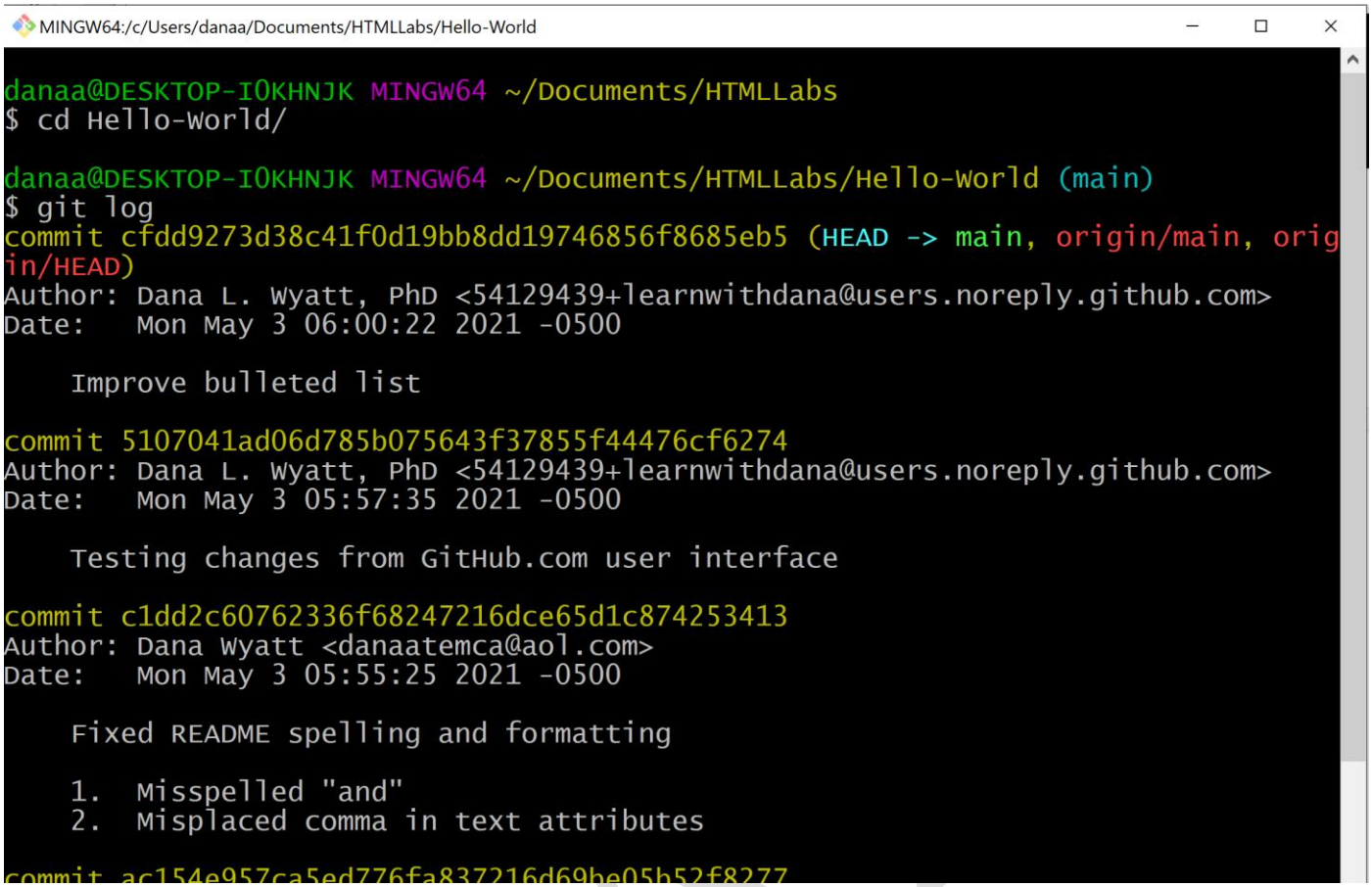
Cloning a repo

```
$ git clone https://github.com/learnwithdana/Hello-World.git
```



- **It not only brings the source files down, it brings the entire repository history**



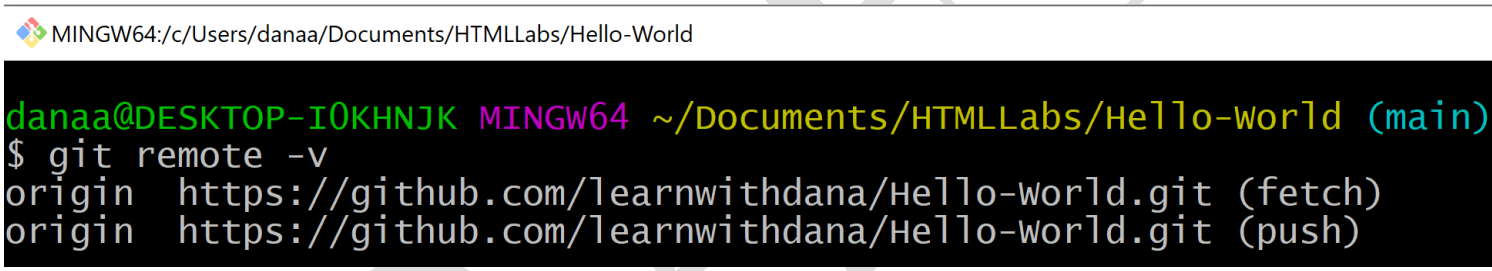

MINGW64:/c/Users/danaa/Documents/HTMLLabs/Hello-World
danaa@DESKTOP-IOKHNJK MINGW64 ~/Documents/HTMLLabs
$ cd Hello-World/
danaa@DESKTOP-IOKHNJK MINGW64 ~/Documents/HTMLLabs/Hello-World (main)
$ git log
commit cfdd9273d38c41f0d19bb8dd19746856f8685eb5 (HEAD -> main, origin/main, orig
in/HEAD)
Author: Dana L. Wyatt, PhD <54129439+learnwithdana@users.noreply.github.com>
Date: Mon May 3 06:00:22 2021 -0500
Improve bulleted list
commit 5107041ad06d785b075643f37855f44476cf6274
Author: Dana L. Wyatt, PhD <54129439+learnwithdana@users.noreply.github.com>
Date: Mon May 3 05:57:35 2021 -0500
Testing changes from GitHub.com user interface
commit c1dd2c60762336f68247216dce65d1c874253413
Author: Dana Wyatt <danaatemca@aol.com>
Date: Mon May 3 05:55:25 2021 -0500
Fixed README spelling and formatting
1. Misspelled "and"
2. Misplaced comma in text attributes

Configuring the Remote Repository Locally: `git remote`

- If you cloned the repository, your local Git knows where the repo came from
- You can see that using the `git remote -v` command

Finding out about the remote repo

```
$ git remote -v
```



- Do you see the word 'origin' at the beginning?
  - When you clone another repository, Git automatically creates a remote named "origin" and points to it.
  - You then refer to 'origin' in order to refer to that remote!
- If you created the repo on your local computer first, you should use the `git remote add` command to create and configure the remote repo
  - You may be prompted for GitHub credentials

Creating a remote repo using Git

```
$ git remote add origin https://github.com/learnwithdana/Remote-Demo.git
```

  - Notice here we are specifically giving the remote the name 'origin'

Pushing to the Remote Repository: `git push`

- **The `git push` command says "push the commits in the local branch to the remote branch"**
  - Once executed, commits since your last push will be available on GitHub (the remote repo)

Initial push

```
$ git push -u origin master
```

  - Use the -u flag the FIRST time you push any new branch to create an ***upstream*** tracking connection
  - `origin master` is the name of the remote and the name of the new branch

- **Development is a constant cycle of:**
  - making local changes
  - committing to the local repo
  - pushing changes to the remote repo
  - pulling changes from the remote repo

- **So, why do you pull changes?**
  - Because others might be working in the same project and have made pushes that you want to refresh in your own local repo

Pulling from the Remote Repository: `git pull`

- **The `git pull` command that says "pull the commits in the remote branch to the local branch"**
  - When executed, any commits made to the remote branch since your last pull become available in your local copy of the repository

Example pull

```
$ git checkout master
$ git pull origin master
```

  - It's a good practice to make sure you are in the right branch before you execute the pull command

Working with Other Branches and a Remote Repo

- **When you create a new branch locally and decide to push it to the remote repo, you will need to use the -u flag on the initial push**
  - Afterwards, the push command is simple

- **A log of a programmer's work might be:**

```
$ git checkout master              > checkout the latest master
$ git pull origin master

$ git checkout -b MyNewFeature1    > create a branch off master
> make changes <

$ git status
$ git add .
$ git commit -m "Some changes occurred"
$ git status

> check everything <

$ git push -u origin MyNewFeature1  > create a remote tracking
                                    > branch
> make more changes <

$ git status
$ git add .
$ git commit -m "More changes occurred"
$ git status

> check everything <

$ git push origin MyNewFeature1     > push additional changes
```

- **Alternatively, you may not push the branch to the remote**
  - You might work in the branch locally and then merge to master

- **Then, push master to the remote**

Section 4–3

`README.md` - How to Create Good Markup Files

README.md

- **A `README.md` file is a text file that describes the project contained in a repo**
  - It resides at the root of the project
  - The Git cloud services GitHub, GitLab, and Bitbucket look for your README and display its information along with the list of files and directories in your project

- **The README contains information such as:**
  - what the project is used for
  - the technologies used in the project
  - how to collaborate with you (if appropriate)
  - license information (if any)

- **It helps the audience understand how to install and use your project**

- **Many people get into the habit of making it the first file you create in a new project**

Markdown Language

- **Most READMEs are often written using the Markdown language**
  - The wiki for markdown is here:
    `https://en.wikipedia.org/wiki/Markdown`

- **It lets you to add some lightweight formatting without using a sophisticated editor**
  - Remember, the README file is a plain text file

- **A quick Google search will lead you to lots of good examples of READMEs and markdown syntax, however the following pages will demonstrate some examples**

Basic Markdown Rules

- **The examples below show different ways to use markdown in a README file**

```
There are two ways to create a BIG heading:

# This is a Big Heading

This is also a Big Heading
==========================

## This is a Sub-heading

This is also a Sub-heading
--------------------------

Paragraphs are just a bunch of sentences separated
by a blank line. Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nullam.

See, this is another paragraph. Lorem ipsum dolor sit
amet, consectetur adipiscing elit. Nullam.

Sometimes, you want to force a line break.  In this case,
put two spaces at the end of a line
and do you see the line break?

You can add a horizontal rule like this:

---

You can also set the font on text.  For example,
_these words are italicized_,
on some systems *these worlds might be italicized*,
**these words are bolded**,
and finally, `the words use a monospace font`.

You can have bulleted lists:

  * HTML
  * CSS
  * JavaScript

You also can have numbered lists:
```

```
1. HTML
2. CSS
3. JavaScript

You can add links.  For more information, click [here](https://en.wikipedia.org/wiki/Markdown)

If you want to add formatted code to markdown, use backticks before and after it like this:

```
<!DOCTYLE html>
<html lang="en">
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <!-- body markup goes here -->
  </body>
</html>
```

And to add an image, try this:
![Image](images/readme-images/dana.jpg "icon")
```

- **To see the markdown above rendered, look at the next page**

Example

This is a Big Heading

This is also a Big Heading

This is a Sub-heading

This is also a Sub-heading

Paragraphs are just a bunch of sentences separated by a blank line. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam.

See, this is another paragraph. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam.

Sometimes, you want to force a line break. In this case, put two spaces at the end of a line
and do you see the line break?

You can add a horizontal rule like this:

You can also set the font on text. For example, *these words are italicized*, on some systems *these worlds might be italicized*, **these words are bolded**, and finally, `the words use a monospace font`.

You can have bulleted lists:

- HTML
- CSS
- JavaScript

You also can have numbered lists:

1. HTML
2. CSS
3. JavaScript

You can add links. For more information, click here

If you want to add formatted code to markdown, use backticks before and after it like this:

```
<!DOCTYLE html>
<html lang="en">
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <!-- body markup goes here -->
  </body>
</html>
```

And to add an image, try this:

A Good README

- **Good READMEs are like good books -- everyone has different opinions**
  - `https://www.makeareadme.com/`
- **Minimally, you should include:**
  - Name
  - Description
    - Consider adding a list of features section
- **You may also want to add screenshots or even a video of your user interface**
- **If your project requires non-obvious installation steps, you will want to include them**
  - You may even include a list of system or software requirements
- **You may include code examples of how to use or extend the software if it is a framework**
- **If you provide support, tell people where they can go to for help**
  - It might be an email address, a web site, a chat room, or even a place to report issues
- **If you are open to contributions, describe your requirements are for accepting them**

- **Perhaps even include an authors and acknowledgment section!**

- **Finally, for open source projects, describe how it is licensed**
  - Types of licenses include MIT, Apache, and BSD
  - `https://snyk.io/blog/mit-apache-bsd-fairest-of-them-all/`

Exercise

In this exercise, we will create a local repository and push it to a remote GitHub repository.

Step 1: Create a local repository

Under your `LearnToCode` folder, create a local Git repository named `first-repo`. Within it, add an `index.html`. (That file can stay empty)

Now, commit the changes with a message of "Initial commit"

Step 2: Create a remote repository

Go to your GitHub profile, click the Repositories tab, then click the **New** button

Name the repo `first-repo` and let everything else can stay at the default values (so do not add a readme!), then click "Create Repository".

Step 3: Push your local repository to the remote

Since we already have a repo, we are going to focus on the instructions underneath that says **…or push an existing repository from the command line**

First, run the `git remote add origin YOUR_URL` as it's shown on the screen

Next, we need to do our initial push with `git push -u origin master` to get our code on GitHub.

Step 4: Examine the GitHub repo

Refresh the GitHub page for your repo and you should now see your code in place of the instructions.

Mini-Project

In this exercise, we will use Git to create a single page web site that lets you practice your HTML and CSS. You will build a single HTML page that provides a resume (real or fake) for someone (you? .Bugs Bunny?)

Step 1: Create a GitHub repo

Create a new GitHub repo and name it "AllAboutMe".

Clone it to your local machine under your HCA code folder.

Step 2: Within the project structure

Create an `index.html` file and three folders (`css`, `images`, and `scripts`) .

Add .gitkeep files to your three folders so that Git will track them.

Stage the files and then create your first commit with the message "Initial commit"

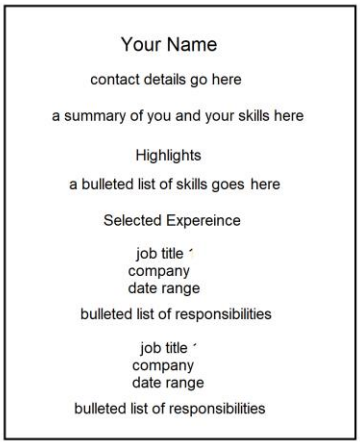
Push the changes to GitHub.

Step 3: Create a branch that will contain the index.html page under development.

Create a new branch named "ResumeDetails".

Switch to that branch.

Step 4: Create a basic resume that resembles the following:



You will need to write a little CSS to center things. Create a file named styles.css and add it to your CSS file.

Then add a `<link>` to the head of the index page to include the CSS file.

Experiment until the page looks sufficient for an academic exercise.

Step 5: Commit the project

Stage the files and then commit with a message similar to "Added resume basics and experience"

Step 6: Add education to the resume page.

Go back to your index page and add another section at the bottom for education. Use any style that you think looks okay.

Step 7: Commit the project and push changes to GitHub

Stage the files and then commit with a message similar to "Added resume education"

Then push the new branch to GitHub

Step 8: Merge your branch into master

Checkout the master branch. Quickly examine the inex.html page. You should notice that this branch does not have any of your changes in it,

Merge your ResumeDetails branch with master.

Push your project to GitHub

Step 9: Examine the code available on GitHub

Take a few moments and examine the code available on GitHub.