Learn to Code

Command Line and Git

Workbook 1-b

Version 6.1 Y

Table of Contents

Module 1

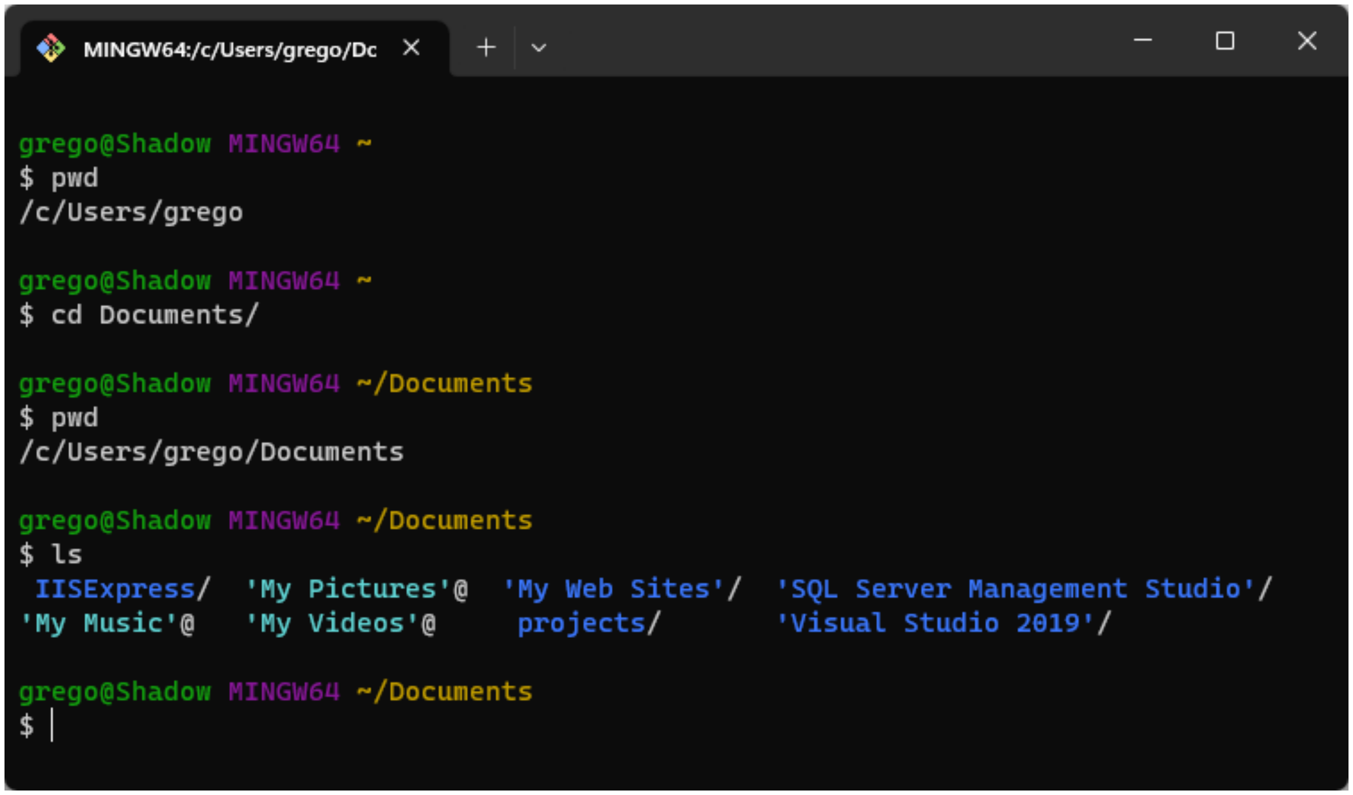
The Command Line

Section 1–1

The Command Line

The Command Line

- **The command line is a powerful, text-based interface that helps developers run programs and control the computer**
  - Before operating systems had sophisticated user interfaces like those in Windows or macOS, the command line was all we had
    - Think of old-school DOS and early Linux systems



MINGW64:/c/Users/grego/Dc

```
grego@Shadow MINGW64 ~
$ pwd
/c/Users/grego

grego@Shadow MINGW64 ~
$ cd Documents/

grego@Shadow MINGW64 ~/Documents
$ pwd
/c/Users/grego/Documents

grego@Shadow MINGW64 ~/Documents
$ ls
 IISExpress/  'My Pictures'@  'My Web Sites'/  'SQL Server Management Studio'/
'My Music'@   'My Videos'@     projects/       'Visual Studio 2019'/

grego@Shadow MINGW64 ~/Documents
$ 
```

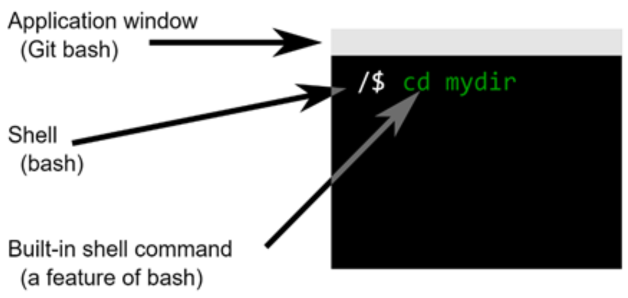
- **There are *still* things you can do from the command line that you *can't* do with point-and-click user interfaces**
  - As a programmer, it is a skill that you eventually *must* conquer
  - It takes a while to get used to, but with practice you catch on

CLIs and IDEs

- **If you 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 have more friendly software development tools to get the job done**
- **Powerful tools like Visual Studio Code (Web programming), Visual Studio (C#/.NET), and IntelliJ (Java) allow programmers to write and run their code using a single tool**
  - These tools are called Integrated Development Environments (IDE)
- **If an IDE doesn't allow us to perform the task we need to, or it doesn't make it easy, we can leave the IDE and use the CLI**

Anatomy of a Command Line

- **A program called the *shell* processes the commands we enter in the command line interface**
  - The Windows operating system has two native command shells: The Command Shell (`cmd`) and PowerShell
  - Plus, we've installed another ***different command shell*** (using Git Bash) named `bash`



```
/$ cd mydir
```

- **We can type the name of a *command* (or program) to run it in the terminal window**

- **The `date` command prints out the current date**

Example

```
$ date
Thu Apr  6 21:22:08 EDT 2023
```

Section 1–2

Built-In Shell Commands for Files and Directories

The File System

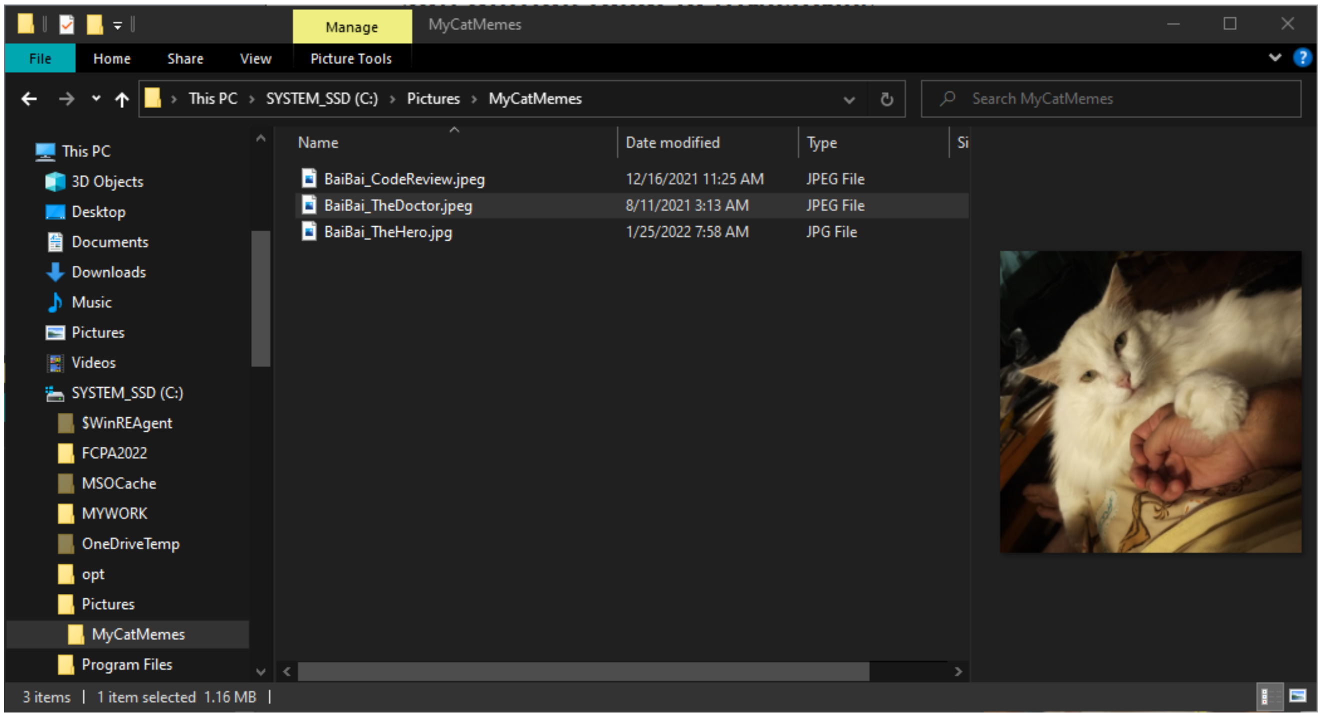
- **In persistent storage devices, a chunk of related data is stored in a *file***
  - A file holds bytes
  - Has a specific location on the drive and a size (in bytes)
    - May be large or small (even 0 bytes), depending on contents
  - Has a name (hopefully, one that makes sense to a person)
- **A file usually represents something important to the user**
  - PDF document
  - Photograph
  - Animated GIF
  - Address book
  - A Java source file (!)
- **Some files store parts of the operating system itself**
  - Without them, your computer is a brick!
- ***We would hate to lose any of our files accidentally, so two things are really important:***
  - Organizing files so you can find them when you want them
  - Backing them up to more than one persistent storage location

Directories (Folders)

- **A set of related files is indexed by a *directory***
  - A directory is just a special kind of file that is only used to organize other files
- **Logically, a *directory* holds files, so you can also call it a *folder***
  - Has a name (again, hopefully, one that makes sense to a human)
  - Has zero or more files called "children"
  - Usually, a child is an "ordinary" file full of bytes, however...
- **A directory can be a child of another directory, called its "parent"**
  - Directories can organize other directories, in a family tree (called a *directory tree*)
  - A child directory is called a *subdirectory* or a *subfolder*
  - Only a "root" directory (e.g. the folder named "/") has no parent
- **A root directory, together with all of its children, grandchildren, great-grandchildren, etc.) is called a *File System***

File Names

- **We organize our files by giving them (meaningful) names, and putting them in directories**



Name	Date modified	Type
BaiBai_CodeReview.jpeg	12/16/2021 11:25 AM	JPEG File
BaiBai_TheDoctor.jpeg	8/11/2021 3:13 AM	JPEG File
BaiBai_TheHero.jpg	1/25/2022 7:58 AM	JPG File

- **When you use the File Explorer, it shows you a visual representation of the file system**
  - That's nice, but…

- **When you use the CLI, your commands may affect files in your current working directory, but you just have to remember where you are in the file system (!)**

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

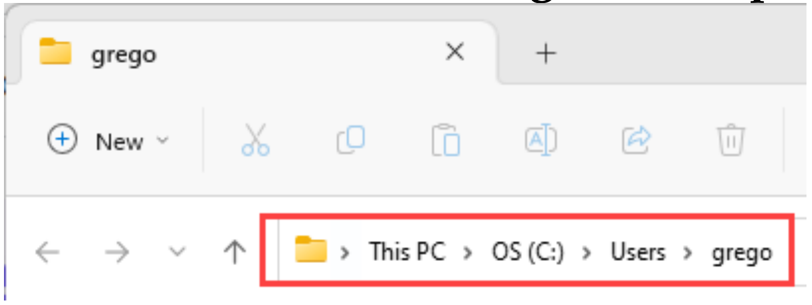
- **As a CLI programmer, you learn to build a visual tree in your mind of the file system**
- **At any point, if you wonder, "Where am I?", you can enter the `pwd` command**

Example

Find out what directory you are in

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

- **This is similar to looking at the explorer navigation bar**



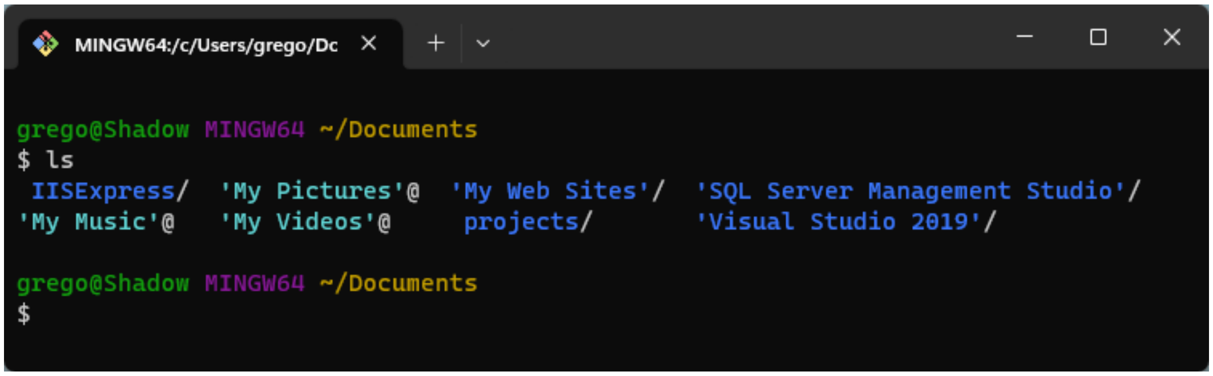
- **We will explore the `cd` command soon**

`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)

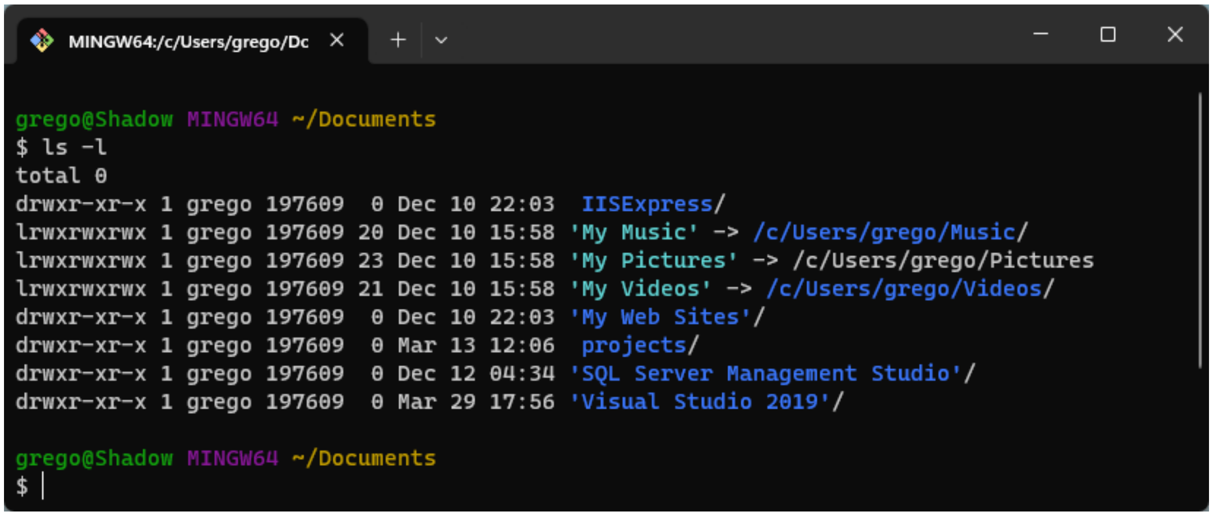


```
grego@Shadow MINGW64 ~/Documents
$ ls
 IISExpress/  'My Pictures'@  'My Web Sites'/  'SQL Server Management Studio'/
'My Music'@   'My Videos'@     projects/       'Visual Studio 2019'/

grego@Shadow MINGW64 ~/Documents
$
```

- **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)



```
grego@Shadow MINGW64 ~/Documents
$ ls -l
total 0
drwxr-xr-x 1 grego 197609  0 Dec 10 22:03  IISExpress/
lrwxrwxrwx 1 grego 197609 20 Dec 10 15:58 'My Music' -> /c/Users/grego/Music/
lrwxrwxrwx 1 grego 197609 23 Dec 10 15:58 'My Pictures' -> /c/Users/grego/Pictures
lrwxrwxrwx 1 grego 197609 21 Dec 10 15:58 'My Videos' -> /c/Users/grego/Videos/
drwxr-xr-x 1 grego 197609  0 Dec 10 22:03 'My Web Sites'/
drwxr-xr-x 1 grego 197609  0 Mar 13 12:06  projects/
drwxr-xr-x 1 grego 197609  0 Dec 12 04:34 'SQL Server Management Studio'/
drwxr-xr-x 1 grego 197609  0 Mar 29 17:56 'Visual Studio 2019'/

grego@Shadow MINGW64 ~/Documents
$ |
```

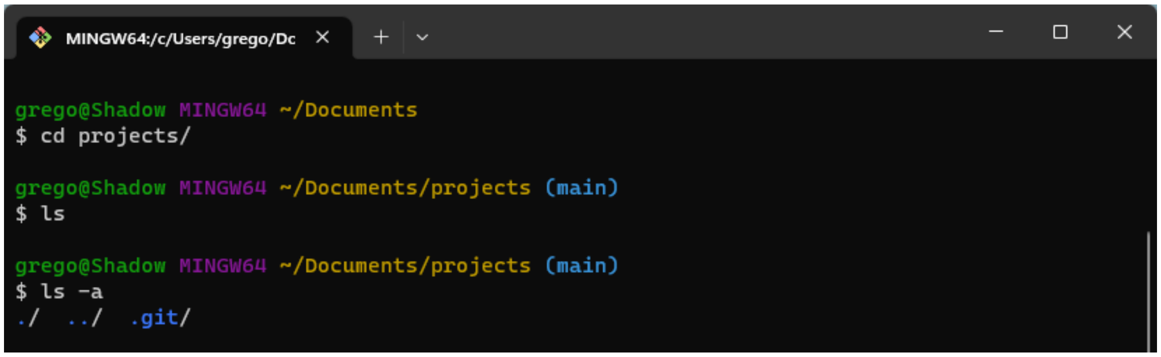
"Hidden" Files

- **In many shells, file names that begin with a period "." are *hidden* and won't appear in `ls` listings**

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

Example

List all files and subdirectories. In the `projects` folder, the `ls` command shows no files, but the `ls -a` command shows a hidden `.git` folder

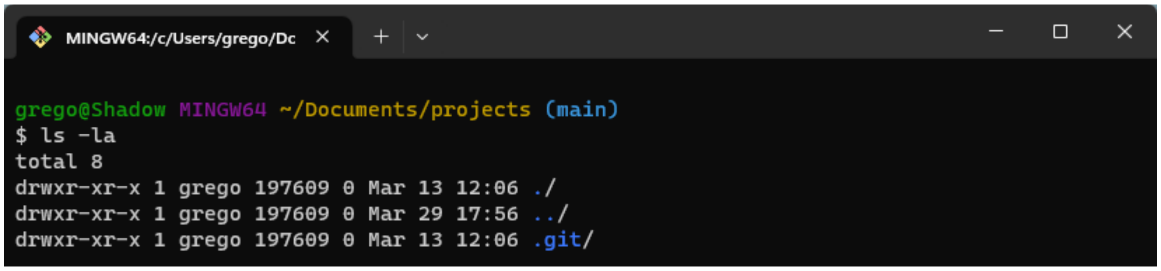


```
grego@Shadow MINGW64 ~/Documents
$ cd projects/

grego@Shadow MINGW64 ~/Documents/projects (main)
$ ls

grego@Shadow MINGW64 ~/Documents/projects (main)
$ ls -a
./  ../  .git/
```

- **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 subdirectories



```
grego@Shadow MINGW64 ~/Documents/projects (main)
$ ls -la
total 8
drwxr-xr-x 1 grego 197609 0 Mar 13 12:06 ./
drwxr-xr-x 1 grego 197609 0 Mar 29 17:56 ../
drwxr-xr-x 1 grego 197609 0 Mar 13 12:06 .git/
```

`cd` - Change the Working Directory

- **In the following examples, assume you have the following directory tree:**

```
projects/           <== You start out here, in the projects directory
└── projects.txt
└── kitchen-remodel/
|   └── description.txt
|   └── budget/
|        └── remodel-budget.csv
└── photography/
    └── food/
    |    └── apples.jpg
    └── nature/
         └── trees.jpg
```

- **To change your current directory, use `cd`**

Example

Go down into the `photography` directory from the project directory

```
$ cd photography
```

- **You can change directories one at a time, or combine directory names into a single *relative path***

Example

Go down into the `photography/nature` directory in two steps:

```
$ cd photography
$ cd nature
```

or do it all at once

```
$ cd photography/nature
```

Going up to the Parent Directory

- **To change the directory up one level, 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 are in the directory `photography/nature` and want to go back to the `food` directory, you can use a path**

  Example

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

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

- **If you are in the `nature` directory and want to go back to the top level of the project, use this**

  Example

  Go up two levels in the directory tree

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

Go Home!

- **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/grego`
- **You can use the bash shortcut tilde `~` to cd to your "home" directory**

  Example

  Go to your home directory

```
$ cd ~
```

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

  Example

  Go to your home directory

```
$ cd
```

Exercise

In this first exercise you will create the folder structure that you will use to do your work throughout this coding academy. Each week you will receive a new workbook which will contain your lab exercises. All of your exercises should be organized into the appropriate workbook folder. For these first exercises, we will use the `command-line` directory

```
C:/
└── pluralsight/
    └── command-line/
        └── VirtualWorld/
```

EXERCISE 1

Create a directory at the root of drive `C:\` named `pluralsight`. You will do your work in this directory, unless otherwise specified.

First add a new directory named `command-line`. This will be where you'll save all your projects for the first week.

Then, navigate into `command-line` and unzip the `VirtualWorld.zip` file to that directory. After unzipping your folder structure should match the diagram above.

Note: We will use the words *directory* and *folder* interchangeably throughout this course; they mean the same thing.

DO NOT OPEN THE `VirtualWorld` DIRECTORY USING FILE EXPLORER!

Launch Git Bash.

Use a **`cd`** command to navigate to

`C:/pluralsight/command-line/VirtualWorld`.

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?

Section 1–3

Creating and Deleting Files and Directories

`mkdir` - Create a Directory

- **You can create a directory using the `mkdir` command, followed by the name of the directory that you want to create**
  - NOTE: The directory is created as a subdirectory of wherever you are (!)

Example

Create a new directory

```
$ mkdir landscape-yard
```

`touch` - Create a File

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

  Example

  Create a new file

```
$ touch description.txt
```

  Example

  Create three new files

```
$ touch description.txt budget.csv tasks.csv
```

- **`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 directory

  Example

  Update the timestamp on a file if it already exists

```
$ touch description.txt   <----- 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 test.txt         <---- 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 directory, any files it contains, any subdirectories it contains, and any files or directories in those subdirectories, all the way down

  Example

  Remove a directory and all of its contents

```
$ rm -r landscape-yard
```

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 2 new files named `Adams-Lake-State-Park.txt` and `Caesar Creek-State-Park`.txt to Ohio
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 Mexican meal with the name `Enchilada.txt`
7. Add a new state to hold parks in Iowa
8. Add three new parks from Iowa
9. Remove goldfish as a pet
10. Remove all parks in Ohio
11. List the parks in Iowa
12. Remove all parks in Iowa

Section 1–4

Copying and Moving Files and Directories

`cp` - Copy a File or Directory

- **Use the `cp` command to copy files or directories**

Example

Copy the file `budget.csv` to a new directory (`Desktop`) in my home directory (~) and rename it to `landscape_budget.csv`

```
$ cp budget.csv ~/Desktop/landscape_budget.csv
```

- **You must use the `-r` flag if you want to include all of the contents of the directory you are copying**

Example

```
$ cp kitchen-remodel bathroom-remodel
```

This will create a copy of the `kitchen-remodel` directory, but the new `bathroom-remodel` directory will be empty

Example

```
$ cp kitchen-remodel bathroom-remodel -r
```

This will create a copy of the `kitchen-remodel` directory and because of the **-r** flag (recursive) all of the contents of the original folder will also be copied to the `bathroom-remodel` directory

`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 description.txt landscape_project_description.txt
```

Example

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

```
$ mv budget.csv ~/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 HelloWorld.java


public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello world!");
    }

}
```

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

Exercise

EXERCISE 3

Continue working in the `VirtualWorld` directory.

Using only the commands **`cp`** and **`mv`** commands, perform the following task:

1. Create a new `American` cuisine (file) in the `Foods` directory by copying the BBQ folder (directory) and all of its contents
2. In the `American` cuisine (directory), rename the following files:
   ```
   Brisket.txt  -> Hamburgers.txt
   Ribs.txt     -> Ribeye.txt
   Sausage.txt  -> BLT.txt
   ```
3. Create a new meal (file) by copying the `Hamburgers.txt` file. Name the new file `Fries.txt`

EXERCISE 4

Create a new directory in the `C:/pluralsight/command-line` directory. Name it `first-java-app`.

Using GitBash commands create the following folder structure:

```
first-java-app/
├── src/
│   └── main/
│   │   └── java/
│   │       └── HelloWorld.java
│   └── test/
│       └── java/
│           └── .gitkeep
└── pom.xml
```

1. Using Notepad open `pom.xml` file and add the following text.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.pluralsight</groupId>
    <artifactId>first-java-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

2. Close Notepad and using the `cat` command in GitBash, view the contents of the `pom.xml` file.

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

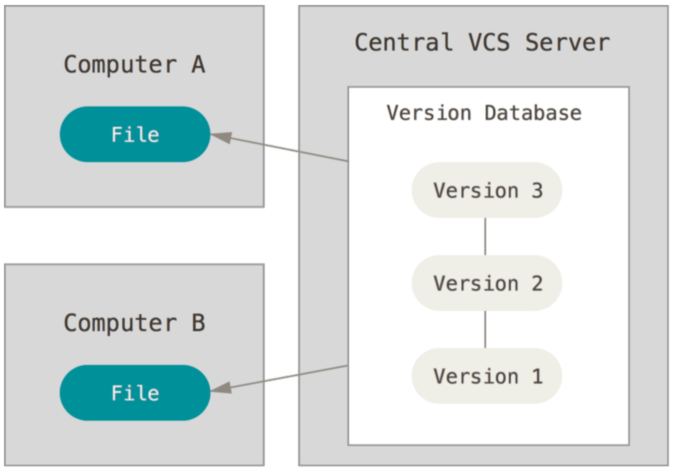
- **Tools also let you compare code in different commits**
  - It can help you figure out what changes might have caused a bug

- **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



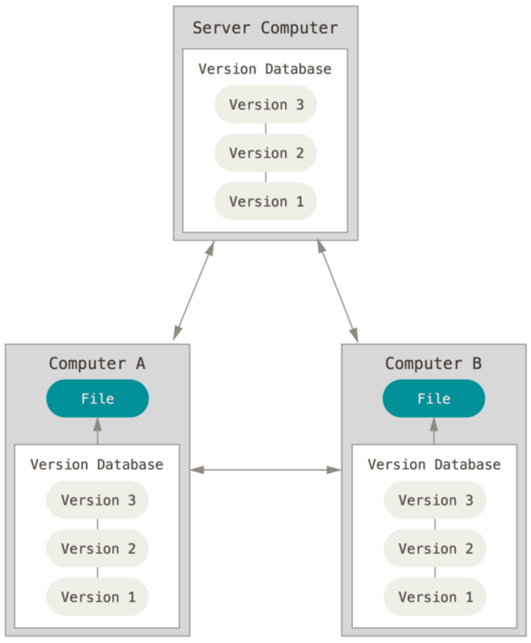
- **Typical Workflow for CVC**
  - Developer checks out code from version control or pulls down the latest changes to code already checked out
    - The file will be locked when someone checks it out, so no one else can check out until it is checked back in
  - Developer makes some changes and tests 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 shell
  - or the Windows 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 and/or a set of flags**
- **Run the following command to check the version of Git installed**
  - Your version may be different than what is shown here

```
$ 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` which affects only the repository in which it is located**

- **You can set key/value pairs in a config file using the command `git config`**
  - `user.name` and `user.email` are used when work is committed to keep a history of who made changes

Example

Setting the user's name (global)

```
$ git config --global user.name "Sallie Sheppard"
```

Setting the user's email (global)

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

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

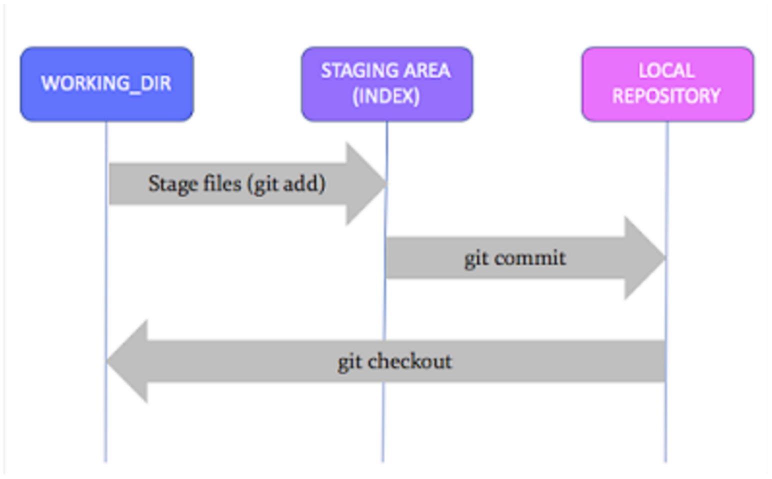
- **To get a list of all current config values, use the `--list` flag**

```
$ 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 from the directory you are in when the command is run**
  - Files in the directory can now be tracked by Git

Example

Create a local repo using the current folder

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

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

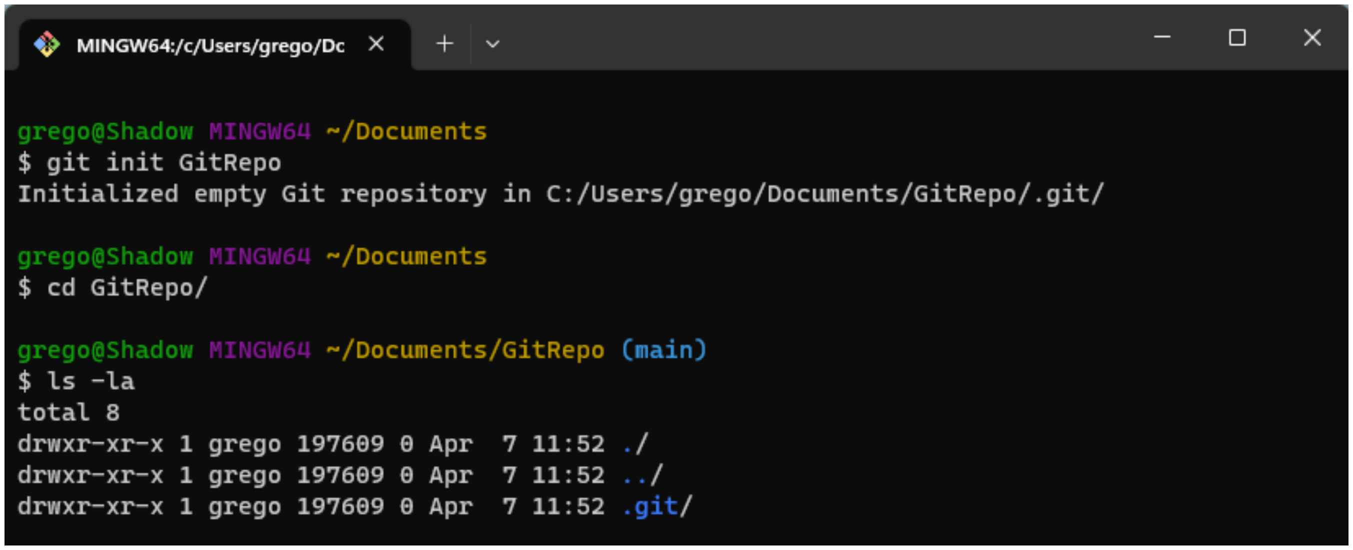
Create a local repo in a specified subfolder

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

- **When you initialize a repo, git creates the staging area and the local repository in a HIDDEN subfolder named .git**
  - The hidden folder is also referred to as the git internals directory

After Initialization

- You can see the `.git` directory using `ls -la`



MINGW64:/c/Users/grego/Dc

```
grego@Shadow MINGW64 ~/Documents
$ git init GitRepo
Initialized empty Git repository in C:/Users/grego/Documents/GitRepo/.git/

grego@Shadow MINGW64 ~/Documents
$ cd GitRepo/

grego@Shadow MINGW64 ~/Documents/GitRepo (main)
$ ls -la
total 8
drwxr-xr-x 1 grego 197609 0 Apr  7 11:52 ./
drwxr-xr-x 1 grego 197609 0 Apr  7 11:52 ../
drwxr-xr-x 1 grego 197609 0 Apr  7 11:52 .git/
```

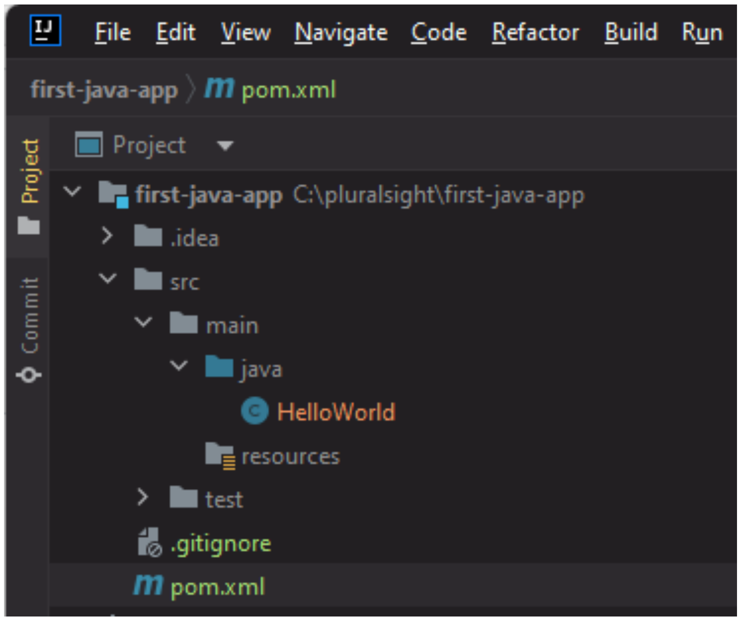
- We won't concern ourselves with the physical structure of `.git` -- suffice to say it holds the staging area and local repository

Working on a Project

- **You can work on a Java project in any text editor**
  - After creating the file structure from the Module 1 exercise we could open this project in IntelliJ

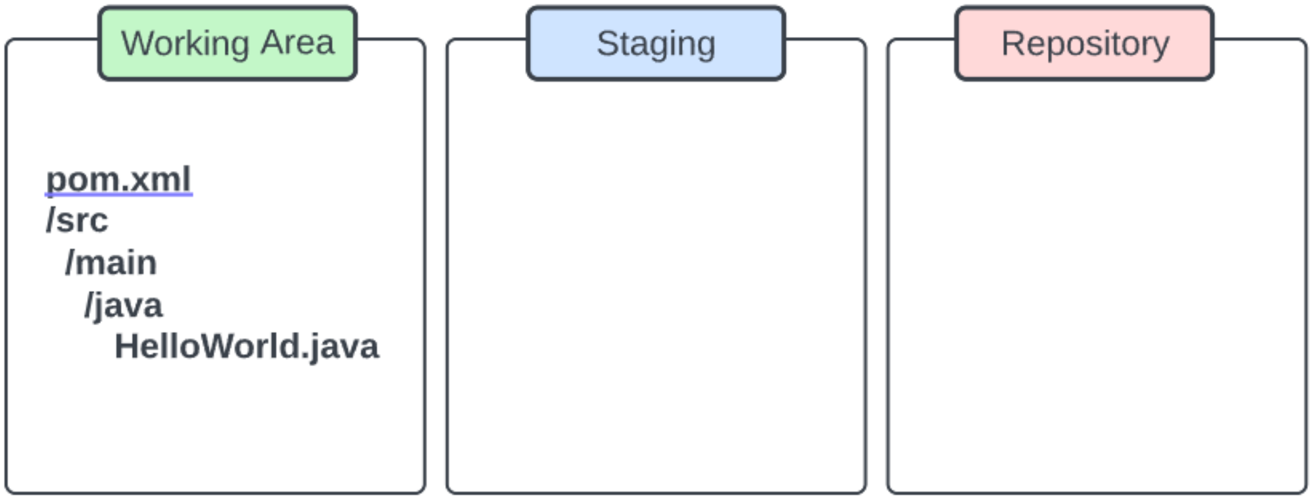
```
first-java-app/
├── src/
│   └── main/
│   │   └── java/
│   │       └── HelloWorld.java
│   └── test/
│       └── java/
│           └── .gitkeep
└── pom.xml
```

  - None of the files have any content in them yet

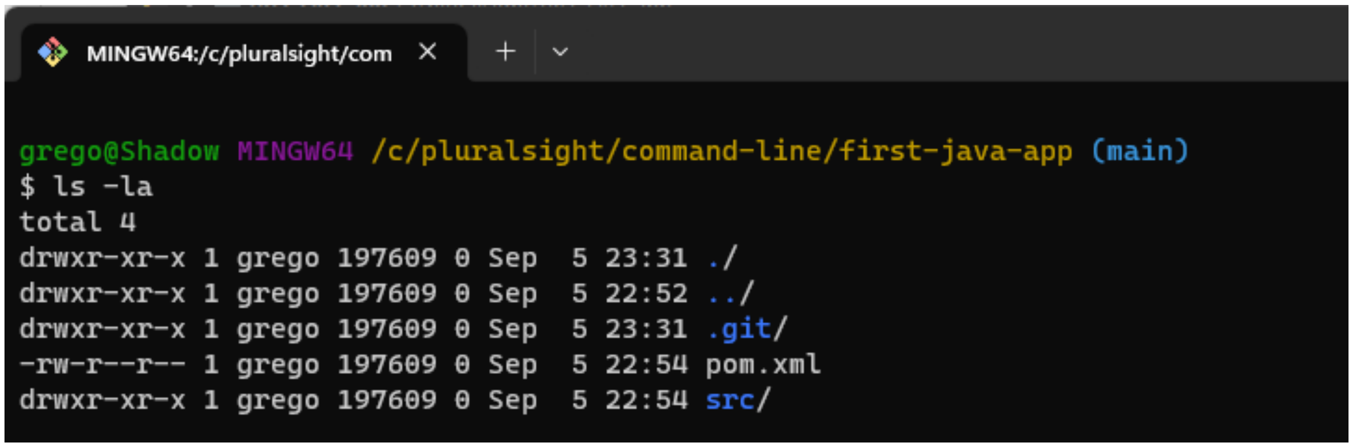


Working on a Project *cont'd*

- **The folders and files you create in the `GitRepo` folder are in the repository's Working Area**
  - The files are not actually tracked by git until they are *staged* and *committed*



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



```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ ls -la
total 4
drwxr-xr-x 1 grego 197609 0 Sep  5 23:31 ./
drwxr-xr-x 1 grego 197609 0 Sep  5 22:52 ../
drwxr-xr-x 1 grego 197609 0 Sep  5 23:31 .git/
-rw-r--r-- 1 grego 197609 0 Sep  5 22:54 pom.xml
drwxr-xr-x 1 grego 197609 0 Sep  5 22:54 src/
```

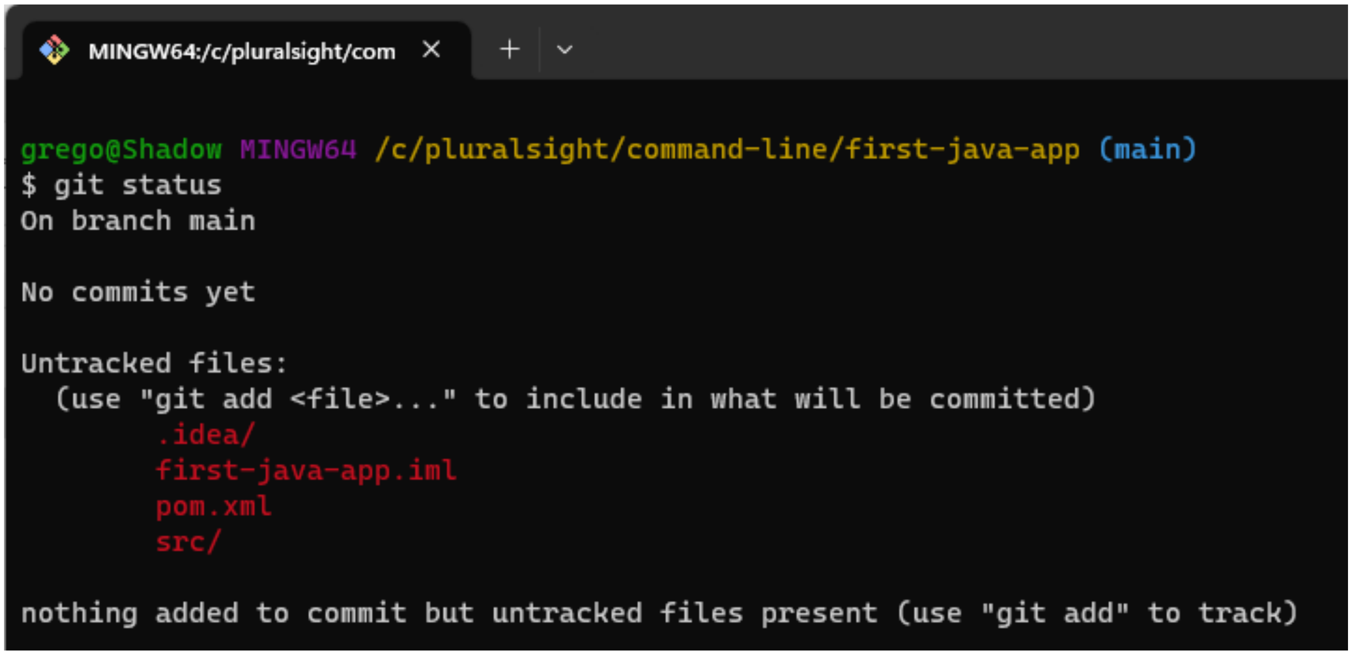
Checking the Status: `git status`

- **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

Example

```
$ git status
... response varies based on status ...
```

Checking status before staging:



```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .idea/
        first-java-app.iml
        pom.xml
        src/

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

- **The red coloring catches our attention and helps us realize these files/folders aren't tracked**
  - The `.idea/` folder and the `.iml` file were created by IntelliJ when we opened the project

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

Example

You can stage a single file:

```
$ git add pom.xml
```

You can stage many files using wildcards:

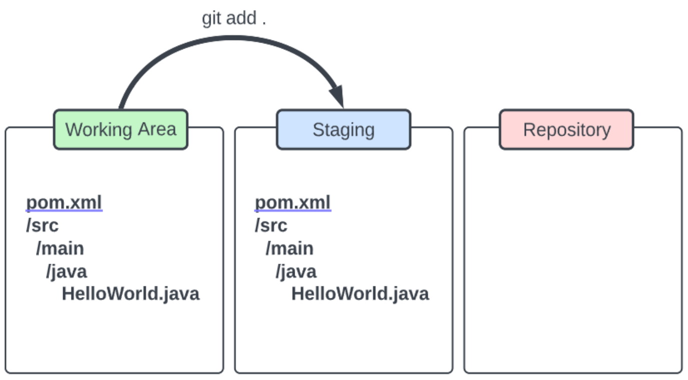
```
$ git add *.java
```

You can stage a whole folder:

```
$ git add src
```

You can stage everything in the Working Directory:

```
$ git add .
```

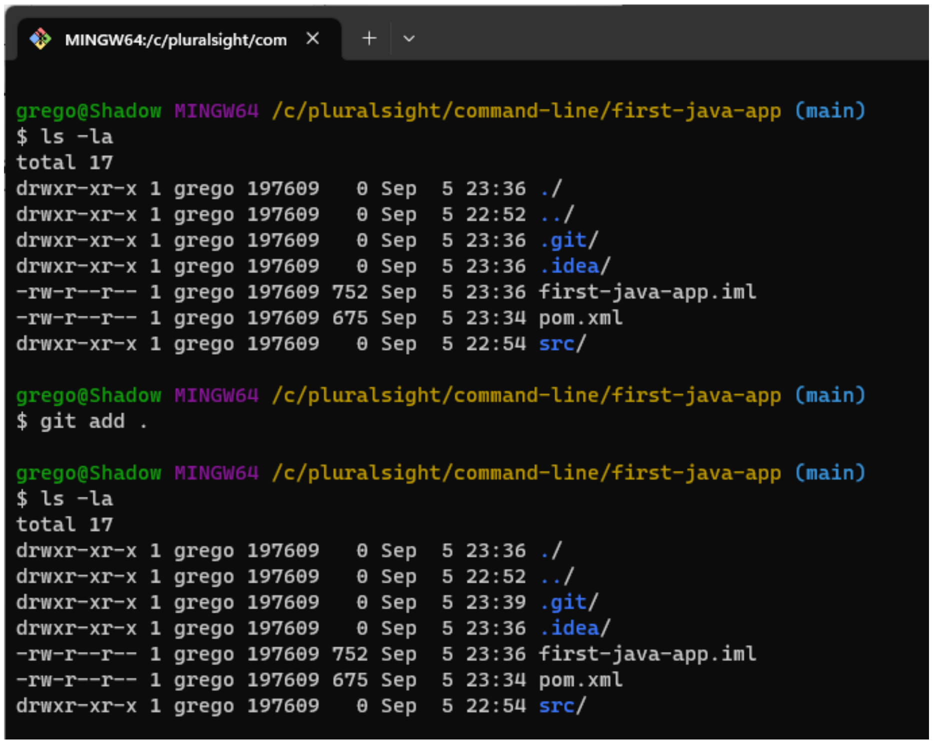


You can stage all changes in the entire repo from any subdirectory:

```
## current path c:/pluralsight/command-line/first-java-app/src/test/java

$ git add -A
```

After Staging

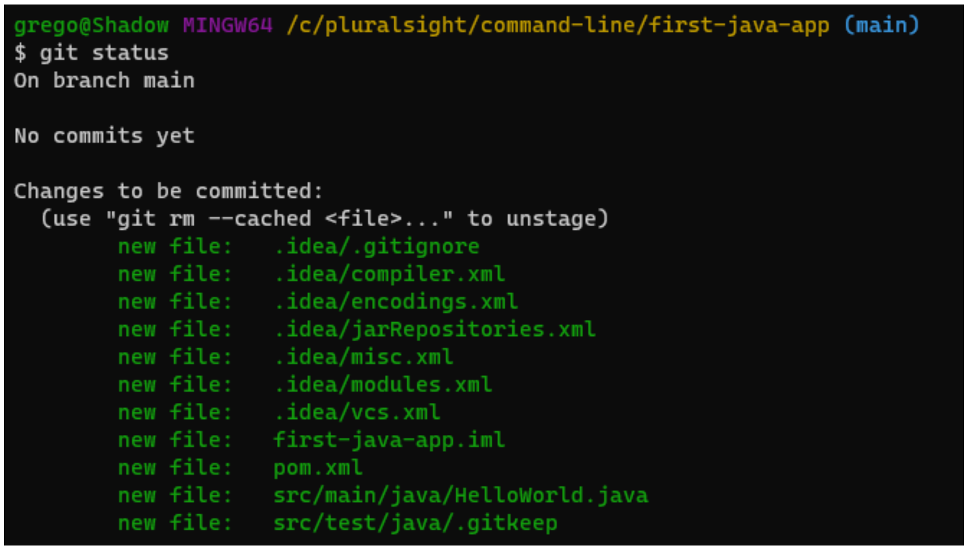
- From a simple glance, it doesn't look like `git add` did anything because you don't see any changes in the project's "visible" folder



MINGW64:/c/pluralsight/com

```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ ls -la
total 17
drwxr-xr-x 1 grego 197609   0 Sep  5 23:36 ./
drwxr-xr-x 1 grego 197609   0 Sep  5 22:52 ../
drwxr-xr-x 1 grego 197609   0 Sep  5 23:36 .git/
drwxr-xr-x 1 grego 197609   0 Sep  5 23:36 .idea/
-rw-r--r-- 1 grego 197609 752 Sep  5 23:36 first-java-app.iml
-rw-r--r-- 1 grego 197609 675 Sep  5 23:34 pom.xml
drwxr-xr-x 1 grego 197609   0 Sep  5 22:54 src/

grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ git add .

grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ ls -la
total 17
drwxr-xr-x 1 grego 197609   0 Sep  5 23:36 ./
drwxr-xr-x 1 grego 197609   0 Sep  5 22:52 ../
drwxr-xr-x 1 grego 197609   0 Sep  5 23:39 .git/
drwxr-xr-x 1 grego 197609   0 Sep  5 23:36 .idea/
-rw-r--r-- 1 grego 197609 752 Sep  5 23:36 first-java-app.iml
-rw-r--r-- 1 grego 197609 675 Sep  5 23:34 pom.xml
drwxr-xr-x 1 grego 197609   0 Sep  5 22:54 src/
```

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



```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .idea/.gitignore
        new file:   .idea/compiler.xml
        new file:   .idea/encodings.xml
        new file:   .idea/jarRepositories.xml
        new file:   .idea/misc.xml
        new file:   .idea/modules.xml
        new file:   .idea/vcs.xml
        new file:   first-java-app.iml
        new file:   pom.xml
        new file:   src/main/java/HelloWorld.java
        new file:   src/test/java/.gitkeep
```

Placeholder Directories

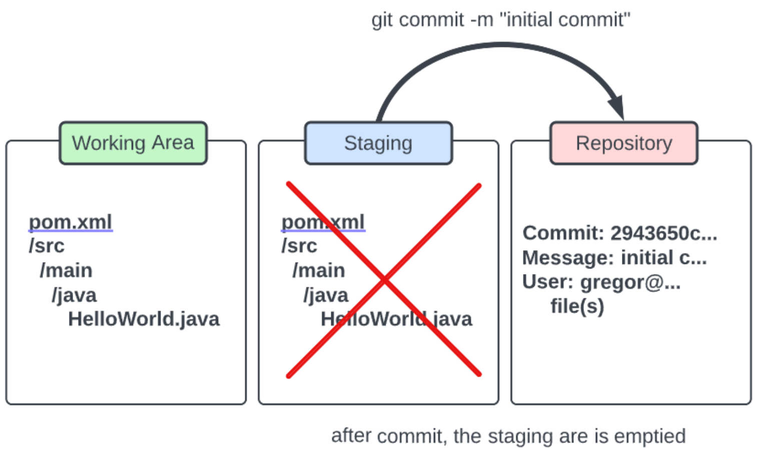
- **Git won’t track empty directories**
  - This might be a problem if you want to keep an empty folder (ex: `src/test/java`) where you are going to add files later on

- **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 all to version control *as a single transaction***
  - It creates a point in the version control history that can be referenced later
  - The changes can be rolled back as a unit



- **You must supply a short message for the commit**
  - Later, when you look at the project history, this should remind you of what is "inside" the commit, and why the files were changed

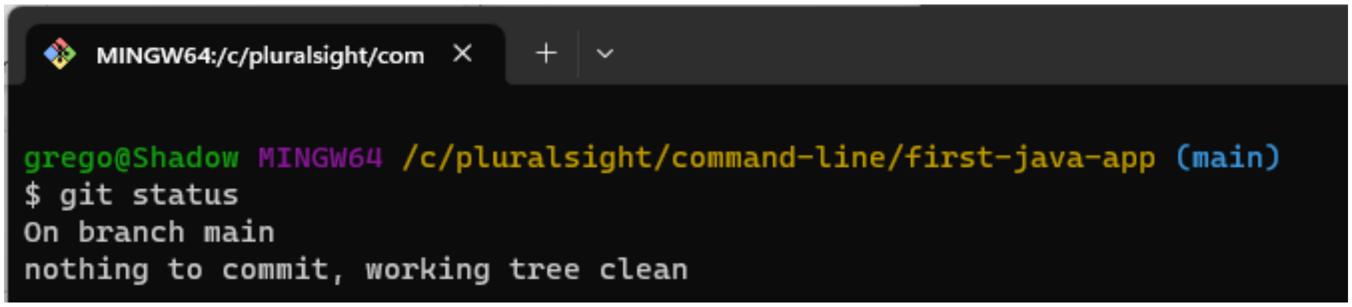
Example

```
$ git commit -m "initial commit"

[main (root-commit) 2943650] initial commit

 3 files changed, 3 insertions(+)
 create mode 100644 pom.xml
 create mode 100644 src/main/java/HelloWorld.java
 create mode 100644 src/test/java/.gitkeep
```

  - Each Git commit is assigned a unique hex number that can be used to find and query the commit later on

After Commit

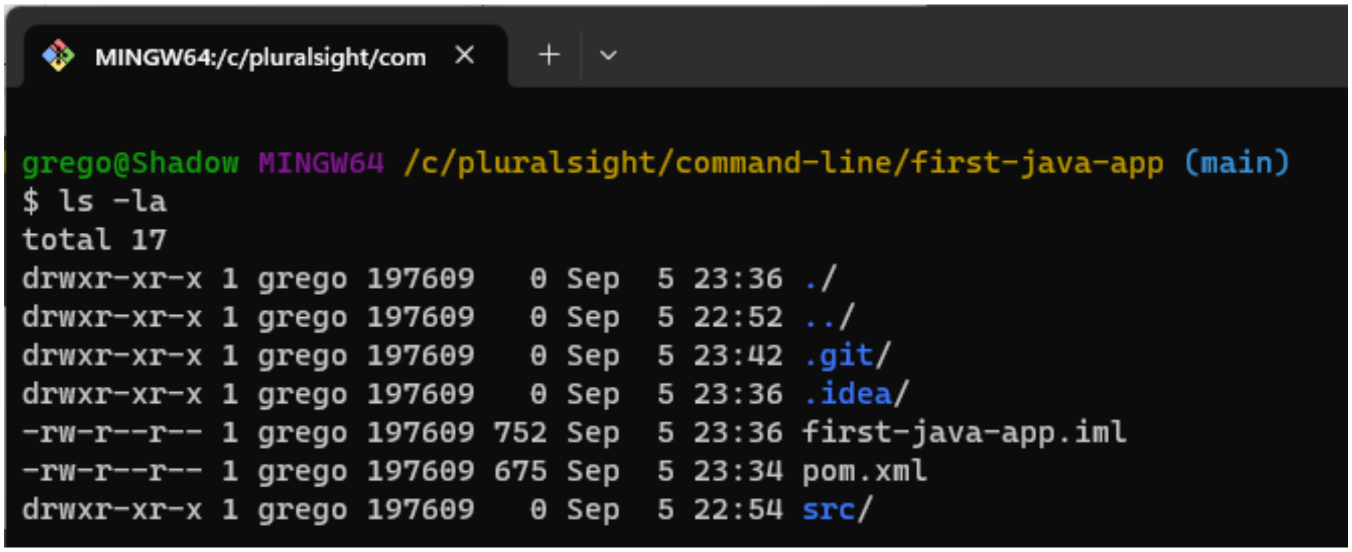
- After the commit, `git status` shows that there is nothing left in the staging area to commit



MINGW64:/c/pluralsight/com

```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ git status
On branch main
nothing to commit, working tree clean
```

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



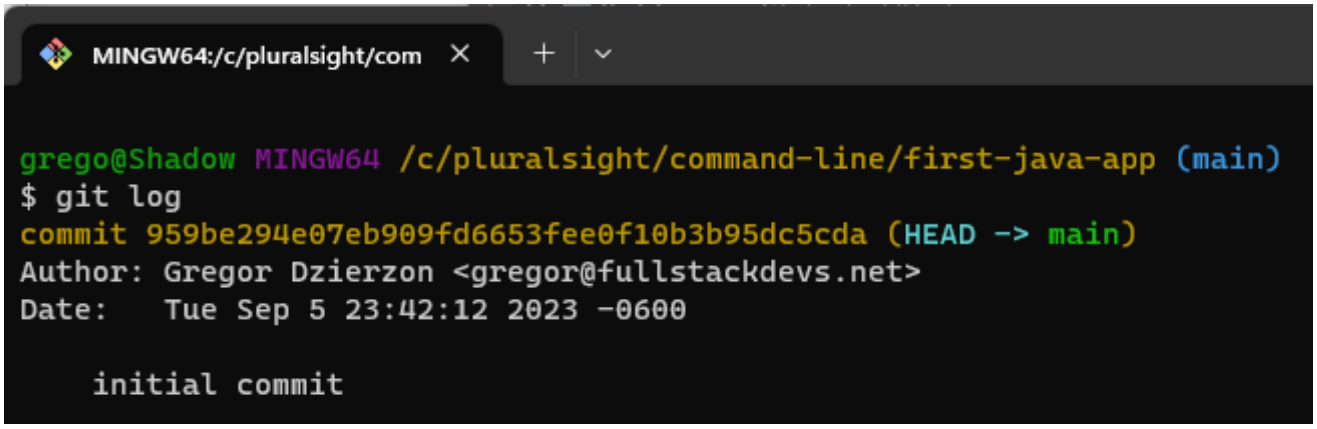
MINGW64:/c/pluralsight/com

```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ ls -la
total 17
drwxr-xr-x 1 grego 197609   0 Sep  5 23:36 ./
drwxr-xr-x 1 grego 197609   0 Sep  5 22:52 ../
drwxr-xr-x 1 grego 197609   0 Sep  5 23:42 .git/
drwxr-xr-x 1 grego 197609   0 Sep  5 23:36 .idea/
-rw-r--r-- 1 grego 197609 752 Sep  5 23:36 first-java-app.iml
-rw-r--r-- 1 grego 197609 675 Sep  5 23:34 pom.xml
drwxr-xr-x 1 grego 197609   0 Sep  5 22:54 src/
```

Checking the Commits: `git log`

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

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

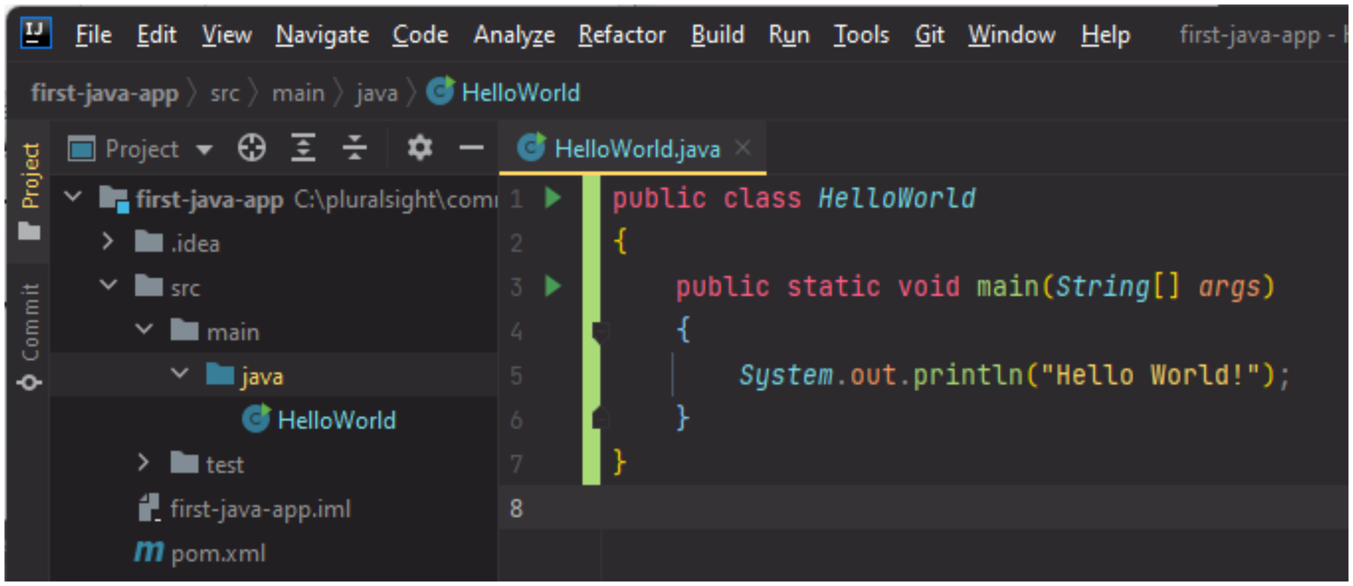


MINGW64:/c/pluralsight/com

```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ git log
commit 959be294e07eb909fd6653fee0f10b3b95dc5cda (HEAD -> main)
Author: Gregor Dzierzon <gregor@fullstackdevs.net>
Date:   Tue Sep 5 23:42:12 2023 -0600

    initial commit
```

Modifying Files in IntelliJ

- **After you make changes to your files in IntelliJ, you must *save*, *stage* and *commit* those changes**
  - We will discuss the actual code changes later


File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help first-java-app -
first-java-app 〉 src 〉 main 〉 java 〉 HelloWorld
Project
HelloWorld.java
first-java-app C:\pluralsight\comm
.idea
src
main
java
HelloWorld
test
first-java-app.iml
pom.xml
Project
Commit

```java
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

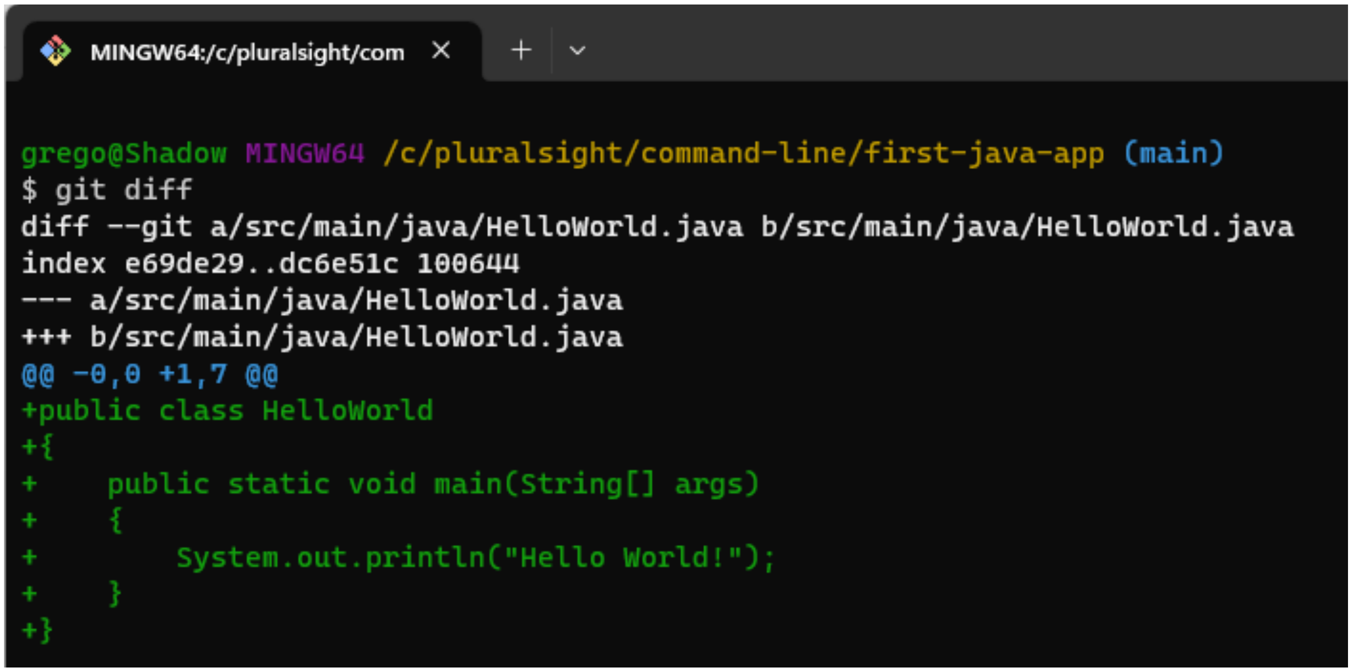
- **Before you add and commit, you may want to browse the changes you made**

Comparing Differences: `git diff`

- **The `git diff` command displays line-by-line differences between files in your working area 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 added code to `HelloWorld.java`

```
$ git diff

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

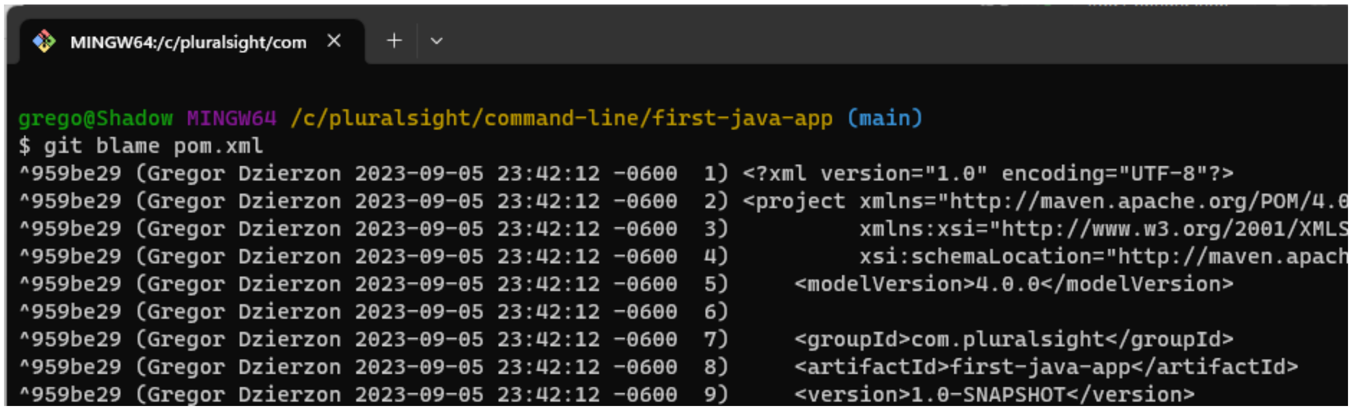


```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ git diff
diff --git a/src/main/java/HelloWorld.java b/src/main/java/HelloWorld.java
index e69de29..dc6e51c 100644
--- a/src/main/java/HelloWorld.java
+++ b/src/main/java/HelloWorld.java
@@ -0,0 +1,7 @@
+public class HelloWorld
+{
+    public static void main(String[] args)
+    {
+        System.out.println("Hello World!");
+    }
+}
```

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**
  - While **`blame`** has a negative connotation, this is not intended as a tool to set blame to a developer for changes they made
    - It's just a funny little Linus Torvalds programming joke
  - It is intended to help quickly discover where a change originated, so if you have questions you know who to ask

```
$ git blame pom.xml

... responds with a line-by-line list of pom.xml
```



```
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ git blame pom.xml
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  1) <?xml version="1.0" encoding="UTF-8"?>
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  2) <project xmlns="http://maven.apache.org/POM/4.0
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  3)          xmlns:xsi="http://www.w3.org/2001/XMLS
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  4)          xsi:schemaLocation="http://maven.apach
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  5)     <modelVersion>4.0.0</modelVersion>
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  6)
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  7)     <groupId>com.pluralsight</groupId>
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  8)     <artifactId>first-java-app</artifactId>
^959be29 (Gregor Dzierzon 2023-09-05 23:42:12 -0600  9)     <version>1.0-SNAPSHOT</version>
```

Ignoring Unimportant Files

- **The really important files in your project are the ones that were built with the hands (and tears) of programmers**
  - These are precious, and would be hard to replace
- **Sometimes, your project has files that you want Git to intentionally ignore**
  - In a JavaScript project, this often means Node.js modules
  - In a Java project, this might mean compiled class files
- **We want Git to ignore these files because we frequently reinstall or rebuild them automatically**
  - Node.js modules are simply ***reinstalled*** using NPM
  - Java class files are always ***rebuilt*** by compiling the project
  - Log files are *generated* each time you run your program
  - These files are cheap and not worth tracking
- **We also want git to ignore large binary files that are hard to compare and are not written directly by programmers**
  - 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 cloning or pulling from a remote repository faster too!

.gitignore file

- **To ignore files, add a text file named `.gitignore` to the root of project**
  - Within `.gitignore`, identify the files to be ignored

- **`.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 `.gitignore` files in individual folders for finer grained 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, you will create a local git repository using common commands and we will point out some quirks like how Git works with empty directories. You will create the repo and commit changes similar to the demos in the preceding pages.

EXERCISE 1

First you need to initialize your `first-java-app` folder as a git repository.

Step 1: Navigate to the `first-java-app` folder. Run `pwd` to confirm you are there (`C:/pluralsight/command-line/first-java-app`).

Run `git init` to initialize your repository

```
$ pwd

$ git init
```

Step 2: Create an initial commit to "save" all of the work that we have done to this point.

NOTE: we will discuss the following commands in more detail in exercise 2

Run `git add .` to stage your work, then run `git commit -m "initial commit"` to commit it

```
$ git add .

$ git commit -m "initial commit"
```

EXERCISE 2

In this exercise you will make changes to the `HelloWorld`.java file and commit the changes..

Step 1: Using Notepad (or IntelliJ) open `C:/pluralsight/command-line/first-java-app/src/main/java/HelloWorld.java` file and add the following code:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

Step 3: Check the "status" of your repository

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

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/main/java/HelloWorld.java

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

Notice that it shows `HelloWorld.java` as a modified file.

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

Run `git add .` to add the changed 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 main

Initial commit

Changes to be committed:

  (use "git restore --staged <file>..." to unstage)

        modified:   src/main/java/HelloWorld.java
```

Step 8: Commit the changes to the java file

Run `git commit -m "Added HelloWorld code"`. 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 d4e2f53651cb3d9d5debf6102ce80d1ed8a1ff84 (HEAD -> main)
Author: Gregor Dzierzon <gregor@fullstackdevs.net>
Date:   Wed Sep 6 00:02:07 2023 -0600


    Added HelloWorld code


commit 959be294e07eb909fd6653fee0f10b3b95dc5cda
Author: Gregor Dzierzon <gregor@fullstackdevs.net>
Date:   Tue Sep 5 23:42:12 2023 -0600


    initial commit
```

Section 2–3

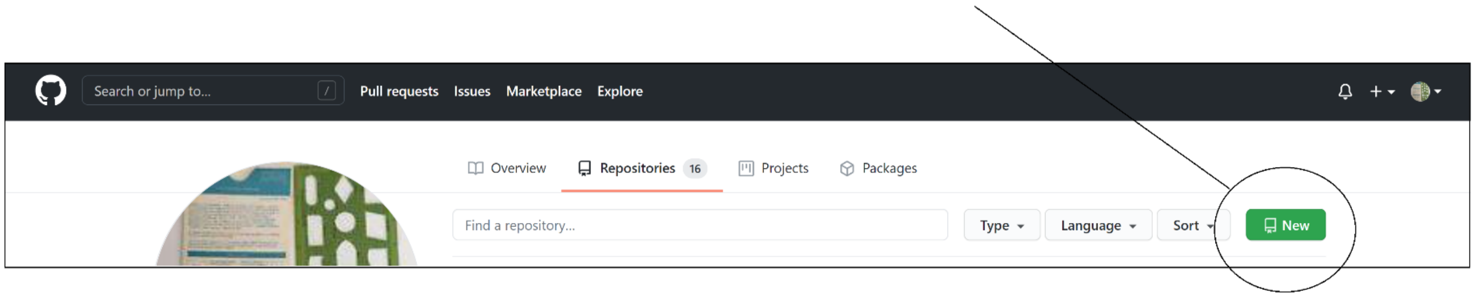
GitHub

GitHub - A Remote Repository Service

- **GitHub is a cloud-based service that can be used to store and manage remote git repositories**
  - Think of it as a backup for your local repo (!)
- **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 host their own internal GitHub cloud service so that they have complete control over its visibility**
- **However, we will use the public GitHub in this class**

GitHub User Interface

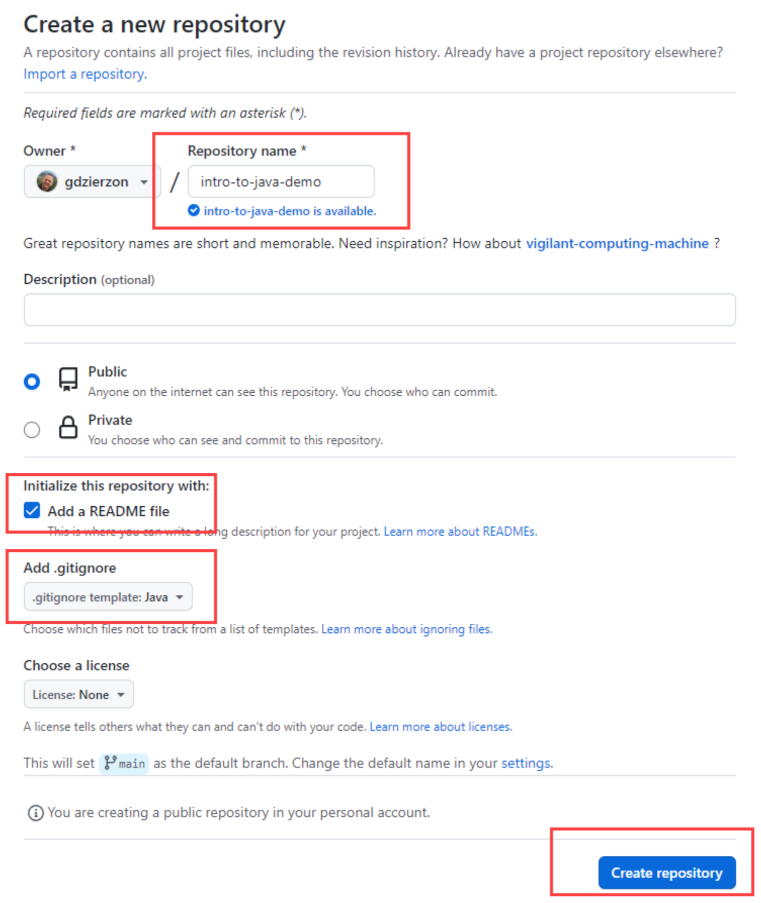
- **GitHub has a web-based graphical interface**
  - It allows you to create new repositories easily



- **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



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

Owner * gdzierzon / Repository name * intro-to-java-demo

intro-to-java-demo is available.

Great repository names are short and memorable. Need inspiration? How about vigilant-computing-machine ?

Description (optional)

[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Initialize this repository with:

[✓ Add a README file] This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

.gitignore template: Java

Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license

License: None

A license tells others what they can and can't do with your code. Learn more about licenses.

This will set main as the default branch. Change the default name in your settings.

You are creating a public repository in your personal account.

- **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 main 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**

Exercise

In this exercise you will create a GitHub repository that you will use to store and submit your work throughout this coding academy. You will organize all of your work throughout this coding academy within this repo.

EXERCISE 2

Step 1: In your browser navigate to `https://github.com` and create a new repository with the following settings:

- Name: `workbook-1`
- Create a `README` file
- Add a `.gitignore` file with a `java` template

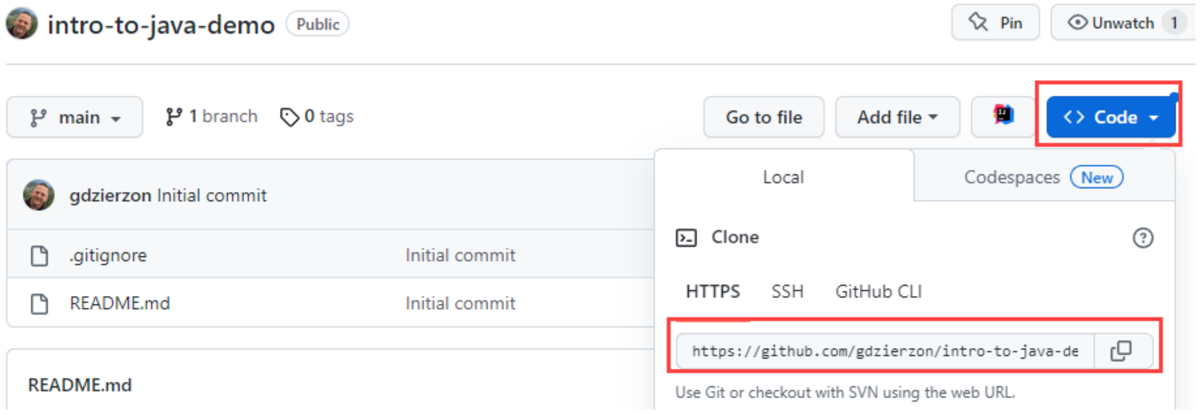
Step 2: Send a link to your GitHub repo to your instructor

Section 2–4

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 new repository as a subfolder of wherever you were when you issued the command**
- **Depending on the remote server, you may need to present some authentication credentials:**
  - Use HTTPS and specify a username and password
  - Use SSH keys
  - Specify a Personal Access Token
- **You can locate the URL of your git repo on the repo page**



gdzierzon Initial commit

.gitignore Initial commit

README.md Initial commit

README.md

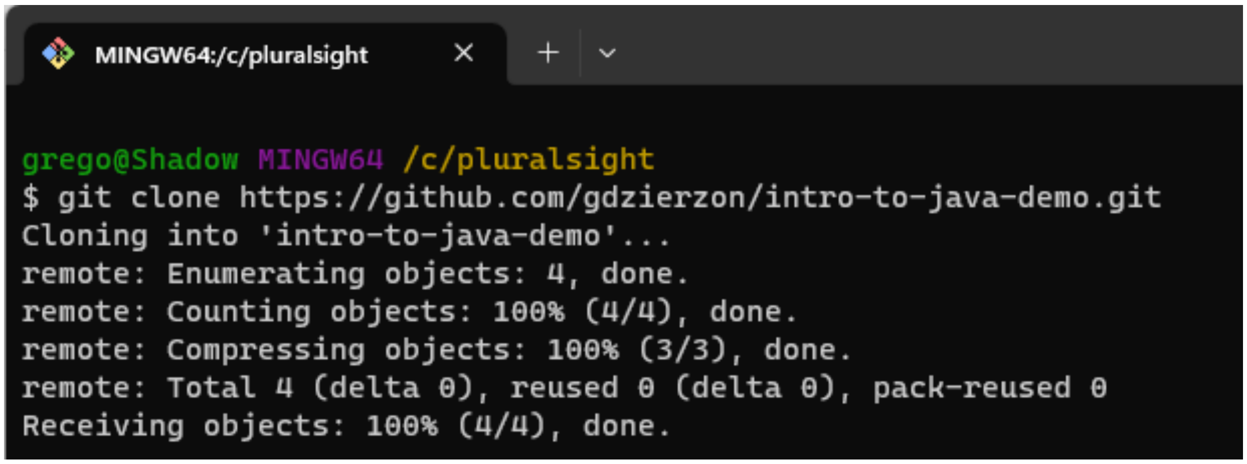
https://github.com/gdzierzon/intro-to-java-de

Use Git or checkout with SVN using the web URL.

Example

Clone a remote repo to your local system

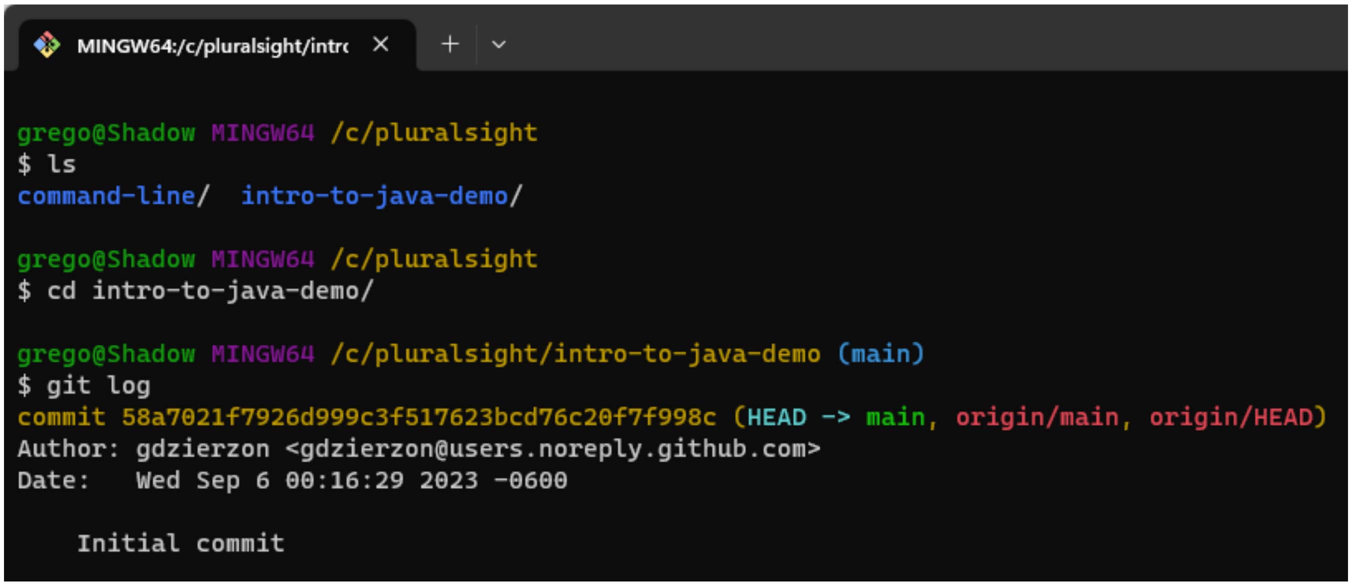
```
$ git clone https://github.com/gdzierzon/intro-to-java-demo.git
```



```
grego@Shadow MINGW64 /c/pluralsight
$ git clone https://github.com/gdzierzon/intro-to-java-demo.git
Cloning into 'intro-to-java-demo'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

Cloning Creates a New Local Repo

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



MINGW64:/c/pluralsight/intr

```
grego@Shadow MINGW64 /c/pluralsight
$ ls
command-line/  intro-to-java-demo/

grego@Shadow MINGW64 /c/pluralsight
$ cd intro-to-java-demo/

grego@Shadow MINGW64 /c/pluralsight/intro-to-java-demo (main)
$ git log
commit 58a7021f7926d999c3f517623bcd76c20f7f998c (HEAD -> main, origin/main, origin/HEAD)
Author: gdzierzon <gdzierzon@users.noreply.github.com>
Date:   Wed Sep 6 00:16:29 2023 -0600

    Initial commit
```

Pushing to the Remote Repository: `git push`

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

Example

Initial push

```
$ git push -u origin main
```

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

- **Development is a constant cycle of:**
  - making local changes to files
  - *staging* the files you want to commit, and *committing* to the local repo
  - *pushing* changes up to the remote repo
  - *pulling* changes back from the remote repo

Pulling from the Remote Repository: `git pull`

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

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

Example

Set the branch you want to update, and pull from the remote repo

```
$ git pull origin main
```

Common Programming Steps

- **Create a remote repository on GitHub first**
- **Clone the repository to your computer**

```
$ git clone <your_github_repo_url>.git
```

- **Create a new Java project in the local repository - commit and push the initial code**

```
$ git add -A

$ git commit -m "initial code commit"

$ git push origin main
```

- **Make changes to your project - commit and push OFTEN**
  - Use `git pull` to "download" any changes made by other team members
  - You should commit and push your changes very frequently - possibly multiple times a day

```
## git pull is generally only required when working as part of a team

$ git pull origin main

## make changes to your code/project

$ git add -A

$ git commit -m "initial code commit"

$ git push origin main
```

Exercise

In this exercise you will clone your GitHub repository to the `C:/pluralsight` directory of your computer and modify the `.gitignore` file.

<u>EXERCISE 3</u>

Step 1: Navigate to your GitHub account and copy the url for your `workbook-1` repo.

Step 2: Open GitBash, navigate to the `C:/pluralsight` directory and clone it

Step 3: Open the newly cloned `workbook-1`directory and view all contents (including hidden files and folders)

You should see following files and folders

```
.git/          (hidden folder)
.gitignore     (hidden file)
README.md
```

Step 4: Open the `.gitignore` file in Notepad to view its contents. This file provides instructions for git to define which files and folders should be ignored as you are making changes in that repository (directory).

Add the following line to the end.

```
# IntelliJ IDEA #

**/.idea/
```

NOTE: Throughout this cohort, you will work on many java projects with IntelliJ. Each time you open a project in IntelliJ it will open or create a folder called `.`idea. This line of code ensures that these `.`idea folders will not get pushed to GitHub.

Step 5: Open the Readme in a text editor and write/save

"## This is a practice repo"

Step 6: Stage, commit and push your changes

```
git add .

git commit -m "completed git setup exercise"

git push origin main
```

NOTE: You should `add`, `push`, and `push` your changes **EACH DAY** (possibly multiple times a day). So, get in the habit of using the previous 3 commands frequently.