

**Prerequisites:** You already have a GitHub account, and are using a computer that has **git** installed.

**Note:** This lab involves a lot of steps that must be done carefully, and all the different **git** commands may feel overwhelming at first. DON'T PANIC! Instead of just typing things, take time to understand \*conceptually\* what you are doing and why you are doing it. If it's confusing, ask questions!

This lab should be done **individually**, because everyone needs practice with this. But help each other!

If you run into problems after class, post them to the Q&A: <http://lovelace.augustana.edu/q2a/>

### Part A: Set up "personal access token" for GitHub (probably only needed on Mac?)

Due to heightened security, GitHub no longer lets you use your main GitHub password for pushing/pulling code from the cloud to your local git repositories. There are a variety of login methods, but one option is to create a "personal access token" to use in place of a password.

Go to settings on GitHub and click "Developer Settings", then click "Personal Access Token" "Tokens (classic)", and "generate new token". Give it a note like "csc305 token 1".

You can set the access period to "No expiration" (unless you're paranoid about security)

For scopes, select **repo** (which will automatically check all the boxes in that category).

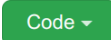
Generate token, and copy the token (long sequence of random characters!). This is a password, so you should store it somewhere safe. However, it's also a pain to type, so you might store it in a text file somewhere, so that you can copy/paste it in when it's needed.

If you lose your token, you can go through these steps again to generate a new one.

### Part B: Basic git commands and GitHub usage, with text files.

Step B1: Go to GitHub.com, log in, and create a new public repository named precisely "TestRepo". Put in a **description** that says "GitHub testing lab for CSC 305 by <YOUR NAME>."

Settings: **public**, **Initialize:"Add a Readme"** Then click "Create repository" button.

Step B2: Click the green Code  button, and copy the https:// URL to your clipboard.

Step B3: Create a new folder on your **H:** drive called **H:\git\** (or on your laptop, perhaps **C:\git** on Windows or **/Users/yourusername/git** on a Mac, or another folder you can easily find,)

Step B4: Start "Git Bash" from the Start Menu or Desktop. (OR open a "Terminal" if you're on a Mac)

Type: **cd /h/git** (*changes working directory to the H:\git folder you made*)

Type: **git clone <PASTE>** ← where <PASTE> is the URL for TestRepo you copied above

e.g.: **git clone https://github.com/YOUR\_GITHUB\_USER\_NAME/TestRepo.git**

You now have a **local** repository that is a clone of the repository you created on GitHub.

Use Windows folders/explorer to open the H:\git\TestRepo folder, and look inside.

You should see a **.git** subfolder and a **README.md** (markdown text file).

(If you don't see the ".git" folder, enable the "hidden files" check mark in the View toolbar/tab.)

The **.git** folder contains all the special innards of the git database for tracking all past history

of the repository, etc. LOOK BUT DON'T TOUCH! If you manually modify the contents of this folder, you'll probably break your repository.

Step B5: In the Git Bash terminal, type **cd TestRepo** to go inside the TestRepo folder.  
Then type **git status**, and read the result. (*Ask for help if you see an error*).

#### **IMPORTANT NOTE: How to continue this lab later!**

You can safely pause this lab at most steps. To continue it later, you will need to re-open git bash (or the terminal) and **cd** (change directory) so that you are working inside the TestRepo folder.

On lab computers, use: **cd /h/git/TestRepo**

On your own computer, **cd /full/path/to/wherever/you/put/TestRepo**

If you don't, "git status" (and most git commands) will show an error "fatal: not a git repository"

#### Step B6. Modifying an existing file.

On Windows, right click on the **README.md** file and open with *Geany* (or any decent editor, like *VS Code* or *Notepad++*). (If you use *TextEdit* on a Mac, be sure to use [plain text mode and avoid RTF](#), which stands for "rich text format" and includes codes for bold/italic/fonts, etc.). Add a blank line then a new line that says **"Beware the wrath of the hungry emu"**, and Save.

In the Git Bash terminal, type **git status**, and read the result.

You have modified a FILE in your "working directory", but this hasn't changed your local git repository yet, *i.e. the version control system (VCS) database lurking in the .git folder*.

First we must tell git that it should include this change as part of the next "change set" that will be added to the VCS database.

Type: **git add README.md**

Note: a version of README.md was already in the repo, so we're not adding the whole file, but we are ADDing the new changes to the git *staging area* (also sometimes called the *index*).

Type: **git status**

At this point, your change is "staged" – ready to be added with all of the other staged changes the next time you **commit**.

The rationale for this multi-stage process is so we can "stage" a number of changes before committing them all together. Each commit should contain a related set of changes (often across multiples files). For example, if you renamed a method in one .JAVA file, you'd have to change the name of the method in all of the other .JAVA files that call that method, and you'd want to **commit** all of those file changes to the repo at the same time (as ONE *change set*), so that your project would compile properly before and after the change.

So, instead of **commit**-ing this change now, let's make another change first.

### Step B7: Adding a new file to the repo.

Within TestRepo create a new file named “**emu.py**” and open it up and add the text **print("CHOMP CHOMP!")** in it. Save it. (You can use Thonny, or any text editor.)

Run **git status** again and note the result.

(emu.txt is “untracked” and the modifications to README.md are still staged for committing.)

Type: **git add emu.py** and then **git status** again. Both changes should be “staged”.

### Step B8: Committing a change to the repository.

No changes you've made are officially included in the git repo until you “commit” them.

Before you can commit (the very FIRST time for this repo) you need to tell git who you are, so it can record WHO made these changes to the repo.

Type: **git config user.email "you@augustana.edu"** // match *GitHub* email

Type: **git config user.name "YOUR NAME"** // not "username", displayed name

This sets the email and name for the current repository. However, if you're NOT on the lab computers, you should add the **--global** option above (*git config --global user.email ...*) to set your email & name for \*all\* future usage of git. (On the lab computers, that isn't effective because DeepFreeze resets the hard drive after every reboot.)

*Side trip:* As long as we're configuring things, if you're on a Mac OR on your own computer and you didn't remember to choose "nano" as the default editor when you first installed git, then run:

**git config --global core.editor nano**

Now, we can finally commit your “staged” changes. Every commit you do should have a clear and accurate commit message describing the changes you made to the repository.

Type: **git commit -m "Added hungry emu and appropriate warning"**

Finally, type: **git status** again to see that the working directory is “clean” again, because there are no uncommitted changes.

### Step B9: Pushing changes up to GitHub.

At this point, your LOCAL repository (the database inside the ".git" subfolder) is now storing your new changes (along with the old version of the project prior to your changes). However, the "remote repository" on the GitHub server doesn't know anything about your changes.

You need to **push** your committed changes back up to GitHub.

Your local repository knows that it was downloaded from a remote repository (which by default is named “*origin*”, because that's where this repo was cloned from.)

You can list the remote repos that the local repo knows about by typing: **git remote -v**

Your local repo COULD also have multiple “branches” (you can and should read more about git branches<sup>1</sup> online!), but right it only has ONE branch named “*main*”. (*Because it's the*

- 1 **Branches** are one of the powerful features of git, and your team may want to take advantage of them. For instance, someone working on one new feature may want to do that work on a branch for a while, so as not to disturb the other team members, until that work is ready to merge with the “main” branch. However, if branches get edited separately

*main/primary branch – not related to Java's main() method!).*

To get your changes back to GitHub, we need to PUSH them to the remote “origin” repo, coming FROM the main branch of the local repo:

**git push origin main**

To complete this step, you'll need to authenticate your GitHub credentials. If a web browser pops up, you may be able to use your regular GitHub username/password. If it asks for your username/password in the "git bash"/terminal, you should enter your GitHub username and your **PERSONAL ACCESS TOKEN** (from Part A above) as the password.

TIP: since we only have one branch (main) and one remote (origin), in the future just type:

**git push**

Assuming you don't see any error messages, your push has succeeded and your changes are now safely stored on the repo on GitHub's server too, which has two BIG benefits:

a) there's an off-site backup so in case your hard drive crashes, your source code (and the full history of changes to that source code) is safe

b) Multiple computers can talk to GitHub. For example, if you are working on a lab computer now, after you go home you can clone the GitHub repo onto your home computer and continue working on the project there, make some changes, commit and push them back up to GitHub, come back to the lab tomorrow, and pull down those changes & edit some more, etc.

Even better, multiple PEOPLE can talk to GitHub, so everyone on your team can be working on a project in parallel, and pulling and pushing changes to/from the GitHub repository. (Of course, there can be trouble... a.k.a. “conflicts”... if two different people try to change the same files at the same time... but git also has some tools to help resolve those conflicts, which we'll learn more about soon!)

### Step B10: Pulling changes down.

There is a corresponding **git pull** command that will take the changes from the GitHub repo, and bring them to your local repo. Currently no one has made changes to the TestRepo on GitHub that your local repo doesn't know about.

However, we can make changes to files in the GitHub repo directly via the GitHub website. In your web browser, click on the **emu.py** file, and then the little “pencil” icon for editing it. Change it to print "**CHOMP CHOMP CHOMP!**" Click “commit changes”. For the commit message put “**Now the emu bites three times**”. (Leave the "extended description" blank.)

Also in the web browser, edit **README.md** to say "**very hungry**" emu, instead of just "**hungry**". For the commit message, put "**Made the emu even hungrier**".

Now back in Git Bash/terminal, you can type: **git pull**

It should say "Fast-forward" and show that both files were changed.

To make sure, you should re-open **README.md** and **emu.py** on your local hard drive, and observe that it has been updated to the version from GitHub.

without merging for too long, the merging process can be more difficult. There's a balance here...

### Step B11: Looking back at the repo history

a) Type **git log** to see the full list of commits for the repository.

Type **git log README.md** to see the list of commits that made changes to that file.

Type **git log -p README.md** to see the actual changes to the file for each commit.

*Note: If you get stuck in log viewer, type "q" to exit.*

b) The command **git diff** by itself should show *no changes* right now, since its default behavior is to show you what's different between your working directory files versus the latest version that's been committed to the repository (the *HEAD* version).

Try **git diff HEAD~1** , which will show all differences between your files and the version that was *one commit earlier than the HEAD version*.

c) Type **git blame README.md** to see *who* last modified each line of the file (and when)...  
(*handy if you want to figure out which one of your team members caused some bug!*)

Note: there are many more things you can do with **git log**, and other commands to dig through the repository history! You can also view/browse the history graphically on GitHub.

d) Go back to your repo website on GitHub, click on README.md or emu.py, and click "History". You can look at changes and browse through previous versions of the file.

e) Click the "Insights" button at the top, and choose "Network". This shows you the sequence of commits. Mouse hover and/or click on the dots. This graph will become more interesting once you have more commits, more people working on the project, etc.

## Part C: Using git/GitHub with IntelliJ

### Step C1: Create an IntelliJ project

- Launch IntelliJ. For simplicity, we'll make a plain old Java project instead of a JavaFX project, so choose "New Project" (instead of "Maven Archetype").
- Name it **SillyBirdProject**.
- For Location, choose "**H:\CSC305IntelliJ**" (or elsewhere, if not on a lab computer)
- Switch the "**Build system**" to **Maven**
- Check "**Add sample code**" but UNCHECK "**Generate code with onboarding tips**"
- Under "Advanced Settings" set GroupID to "**edu.augustana**". Create!

**WAIT!** Do you want to know what's *really* going on under the hood? Of course you do! Good computer scientists are naturally inquisitive!

So far, your IntelliJ actions have created a project folder (perhaps inside H:\CSC305IntelliJ) named SillyBirdProject and put a bunch of files inside it. See →

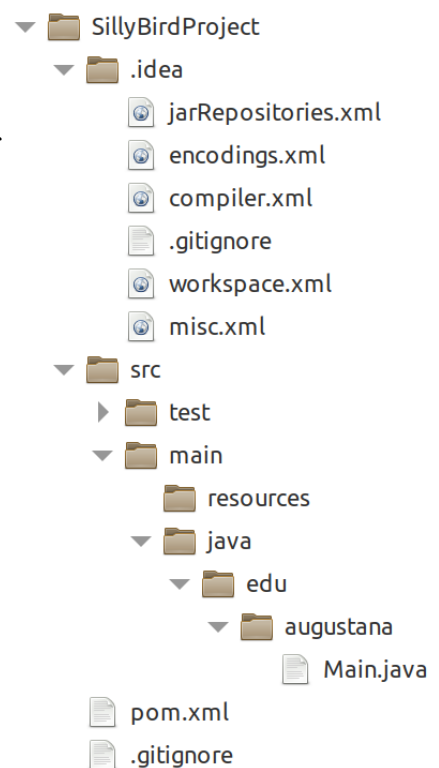
The **.idea** folder stores IntelliJ project settings & preferences. You usually shouldn't need to edit these files manually.

The "**src/main/java**" folder will contain your actual Java code, including **Main.java** which is part of the "**edu.augustana**" package, so it's inside the /edu/augustana folder structure.

(The **/src/test** folder is a place you can put code to automate testing your code to ensure it is bug-free... we'll learn more about that later.)

The **pom.xml** file stores the maven dependencies (e.g. which third-party libraries, such as JavaFX, that your project needs to have in order to build. (Maven will automatically download these for you.)

The **.gitignore** files tell git which files/folders to ignore (i.e. the files that should NOT get added/committed to a git repository).

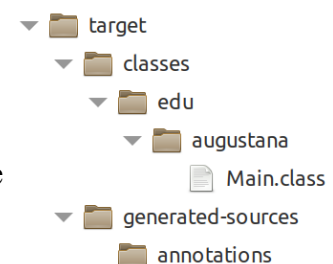


Open up the **SillyBirdProject/.gitignore** file (not **.idea/.gitignore**, which only affects that subfolder). The very first line says "**target/**", which means that git commands will avoid adding/committing anything inside the target subfolder. Why? Because once you build/run the project, IntelliJ will compile all your **.java** source files into JVM-executable **.class** files, and put them inside the target folder. See →

We don't want to store these generated files in git.

We only want to store source files (preferably in plain text format, because text is what git does best). Storing a bunch of binary (.class) files in git is inefficient, and can bog things down.

Also note that it's set up to ignore any **.DS\_Store** files, which is good because Mac computers randomly create these hidden files in folders all over, and if they get included it's annoying!



### Step C2: Moving your new IntelliJ project into your git repository folder

- Right click on SillyBirdProject and "Open In" Explorer (or Files).
- Now, close your IntelliJ project.
- Move the SillyBirdProject folder into your H:\git\TestRepo folder (or wherever you cloned your TestRepo to earlier)
- Re-open IntelliJ, choose "Open Project", and browse to find SillyBirdProject at its new location.

### Step C3: Committing the new IntelliJ project

- Back in your Git Bash/terminal, type **git status** again, and you should see that the SillyBirdProject folder is "untracked".
- You could add it with **git add SillyBirdProject**, BUT a nice shortcut to add ALL changes (new files, modifications, and deletions) to the staging area is: **git add -A**
- **git status** again should show 7 files:

```

new file:   SillyBirdProject/.gitignore
new file:   SillyBirdProject/.idea/.gitignore
new file:   SillyBirdProject/.idea/encodings.xml
new file:   SillyBirdProject/.idea/misc.xml
new file:   SillyBirdProject/.idea/vcs.xml
new file:   SillyBirdProject/pom.xml
new file:   SillyBirdProject/src/main/java/edu/augustana/Main.java

```

- **git commit -m "Added new IntelliJ project"**
- **git push**

To double check, open GitHub in the browser and make sure you can see the new files up there. Browse to find the actual Java source files!

### Step C4: Editing the project in IntelliJ while using the git terminal

- Back in IntelliJ, change Main.java to print out "Hello **bird** world!".
- Run it to make sure it works!
- **git status**
- **git add -A**
- **git commit**

If you don't provide the **-m "message"** option, then git will open up an editor window for you to enter one. Above all the # comments, type in "**Made the world more birdy**" and save/exit the editor. (Assuming you're using **nano** as the default editor, you can save it with Control+O, hint ENTER to accept the file name, then exit with "Control+X".

*If you're stuck in the **vi** editor, type **:q!** to exit, and go back to the "Side trip" in Step B8!).*

- **git push**

### Step C5: Git Operations from within IntelliJ

I expect everyone to learn the terminal-based git commands, because the command line is more powerful, and these same commands will work across any development project, regardless of programming language or IDE.

*That said...* IntelliJ does have nice built-in GUI support (much of it under the "git" menu) for common git commands, like adding files, committing changes, pushing, pulling, etc, and sometimes it will be more convenient to interact with git in this way. Let's try it that way too.

a) Right click on the edu.augustana package and create a new class named Chicken (which will create a new file Chicken.java). IntelliJ will pop up a message asking if you want to add the new Chicken.java to git. Say NO for now.

b) *Note that IntelliJ is coloring Chicken.java RED, showing that it is an "untracked" file for git.*

c) Right click on Chicken.java, **git** → **Add**. Now Chicken.java is GREEN (new file in git)

d) Write a public method **dance()** inside the Chicken class which prints "*Chicken dance!*". Save it.

e) Edit Main.java to create a new chicken object, and call the dance() method on it.

*(Note that Main.java is BLUE to show it has been modified versus the git repo version) .*

f) Run & test your code.

g) If using the command line, we would still need to "add" (stage) the changes to Chicken and Main before committing, but the IntelliJ IDE will let us be lazy and if we choose commit, it assumes we want to commit all changes to the files.

h) git menu, **Commit**. It should show you that both files have changed. Here we could uncheck any files that we don't want to commit, but in this case we want to commit both.

In the textbox above the "commit" button, write message "**Now with dancing chicken**".

Click **Commit and Push**. IntelliJ may need to get you to authenticate to GitHub as well.

i) You can also use IntelliJ to compare files to older versions from the repository, etc. Try this out by **right clicking on Main.java** and choosing **git** → **Compare With Revision**, and choose the earlier commit ("**Made the world more birdy**"). This will open up a "diff" viewer and show you the differences between different versions of your code.

j) Try **git** → **Show history** for another way to browse through all prior versions of a source file.

Feel free to use the IntelliJ GUI method for working with git when it's more convenient, as long as you ALSO know how to accomplish tasks on the command line.

### Step C6: Using .gitignore to avoid adding all files of a certain type.

Suppose that your program regularly created .PDF files when it was run, and you didn't want these files getting stored into git all the time. Open the **SillyBirdProject/.gitignore** file, and edit it to add a new line at the bottom containing **\*.pdf**. Save your .gitignore file.

Copy a random **.pdf** file (maybe this Lab3 pdf?) into your SillyBirdProject folder.

Check **git status** – it should show that .gitignore has changed, but it should NOT list the PDF file that you put in the folder as an untracked file!



Step C7: Verify your changes were pushed, back on the GitHub website for your "repo".

a) Click on the text where it shows the number of commits.

 6 commits

This will show you a list of all the commit messages, and if you click on one of them, it will display the changes that took place in that commit.

b) Also, look inside the SillyBirdProject/src folder to ensure your code's all there. (If not, fix!)

**Submission, due before class on Thursday Sept 26.**

No need to submit on Moodle, since I should be able to find your public **TestRepo** on GitHub. (Also, all git commits are timestamped, so I can tell whether it was turned in late).