CSCI241 Spring 2022: Lab 1

Overview

The purpose of this lab is to give you basic familiarity with some of the tools we'll be using in this class, namely **git** and **gradle**.

Git

Git is a *version control* system that is used to keep track of changes you make your code. We will use git in conjunction with Github, a web-based service that hosts git repositories.

A git repository is a place where your code lives. The internal data needed to track versions of your code is stored in a hidden folder (called `.git` at the base directory ("root") of your repository, but you don't need to worry about that - you'll use the `git` command-line tool to work with the version control system. Github hosts a *remote* copy of your code on their servers, which other people (e.g., me when I go to grade your work) can access and create separate copies of ("clone").

The basic workflow for using git and Github is to (1) make some changes to your code, (2) **commit** those changes to the repository to put them in the official record of version history, and when you've reached a stopping point, (3) **push** to Github to update the hosted repository to reflect the changes you've committed to your local repository ("repo", for short). When you've followed this workflow and gradually made changes over time, git makes it easy to move forward and backward in history, for example to roll back to an old version of your code to find a version before a bug was introduced.

Gradle

Gradle is a *build system* that is used to take care of a lot of the logistics of building, running, and testing your code. It was created for Java, but it can be used for many languages. Once things are properly set up (we'll largely take care of this for you in this course), you compile your code with `javac` and run it with `java`, but instead use commands like `gradle build` and `gradle test` to compile, run, and execute tests on your code.

Keep in mind, this lab is just getting you started; you are not expected to become an expert in all the nuances of these tools yet—expertise will come with time and experience. The goal here is to get you ready to work on future labs and assignments.

Walkthrough - Git

Complete the following steps.

Setup

1. Assignments in this class will be done in GitHub repositories orchestrated by GitHub Classroom. Log into Canvas, find the Lab 1 assignment, and click the GitHub Classroom invitation link found there. You will need to log into your GitHub account if you haven't already.

2. Once you have accepted the GitHub Classroom invitation, you will be given a link to your repository for this assignment. This link should be in the form of:

   ```
   https://github.com/csci241-22s/lab-1-username
   ```

   You should be able to click this link and see the repository on GitHub.

3. The git repository now exists on GitHub's servers, but you still need to **clone** a local copy of the code so you can work on it. You will need to choose a location for the local lab1 repository and working copy. You may choose whatever location you like; here we will assume you choose `~/csci241/`. Note that the ~ indicates that the csci241 directory lives inside your *home directory* (usually /home/username, on linux systems), which is where you should be when you open a new terminal window. To create this directory, type `mkdir csci241`. Enter the directory using the cd command: `cd csci241`. You can list the contents of the directory (it should be empty) using `ls`.

4. Create a working copy of your code by cloning the repository from GitHub. You can copy the command from the green button that says "clone or download" on GitHub, or type it yourself:

   ```
     # replace username with your github username
     git clone https://github.com/csci241-22s/lab-1-username
   ```

   This should clone the repository into a new directory called `lab1-username`, which you should now see if you type `ls`.

Basic Git Operations

5. Before you use Git commands for the first time, you need to tell it a little bit about yourself. You'll only need to do this once for each computer you use git on. Change directory into your freshly cloned repository and run the following commands, supplying your full name and email address:

   ```
   # sub in your name
   git config --global user.name "Your Name Goes Here"

   # replace username w/ your wwu username
   git config --global user.email username@wwu.edu
   ```

6. Now you will a writeup file and tell git to track it (that is, include it in your repository). Note that spelling, spacing and capitalization matter in the following commands:

   ```
   touch writeup.txt
   git add writeup.txt # run "git help add" for details
   ```

   This `git add` does two things: 1) it begins "tracking" `writeup.txt` and 2) it "stages" the changes to the file (in this case, its creation) so it will be incorporated in the next commit.

7. **Commit** your changes to the local repository:

   ```
   git commit -m "Added empty writeup"
   ```

   The text in quotes is the commit message; aim for it to be concise yet specific. Bad commit messages include `"Made some changes"`, `"stuff"`, and `"more edits"`. At this stage your changes have been stored on the local repository but not in the "remote" repository stored on Github. You can confirm this by browsing to the Github URL for your repository:

   ```
   https://github.com/wehrwein-teaching/lab-1-username
   ```

   You should see that writeup.txt is not listed among the files on GitHub.

8. To synchronize your local repo with the version on GitHub, push your changes from the local repository to the original one:

   ```
   git push # sometimes it wants you to be more specific: git push origin main
   ```

   Now refresh the Github URL, and you should see writeup.txt file.

9. Edit `writeup.txt`, so that it contains the line `1) First Last` where you replace `First Last` with your first and last names.

10. Stage these changes for commit:

    ```
    git add writeup.txt
    ```

    While `writeup.txt` is already tracked, this will stage the change.

11. Commit your changes:

    ```
    git commit -m "Added part 1 (names) to writeup"
    ```

12. `git status` lets you know the status of your working copy and local repository. Run the command and see what it reports. Now edit `writeup.txt` again, adding the line `2) Hobby: XYZ` where you replace `XYZ` with a hobby of yours. Check the status after making this edit. Check the status again after `git add`ing the writeup. Check the status again after committing, and then again after pushing.

13. Create another file in your repository called `username.txt`. Edit this file to contain exactly the following three items, separated by commas, with no extra spacing:

    - First and last name
    - WWU Username
    - Github username

    For example, Scott's username.txt would read:

    ```
    Scott Wehrwein,wehrwes,swehrwein
    ```

    Commit username.txt to your repository.

14. Learn your way around the following commands (use `git help` or search the web for details). Be sure to try out each command at least once and get familiar with how they work.

    (a) `git checkout` and `git reset` to undo a change to a file
    (b) `git rm` to delete a file from the repository
    (c) `git mv` to rename a file (if you move writeup, move it back after)
    (d) `git blame` to see who edited which lines when
    (e) `git log` to see your commit history
    (f) `git diff` to see unstaged changes to a local file
    (g) `git diff` to see changes between two different committed versions of `writeup.txt`

    Here's a recommended approach:

    (a) After making sure your `writeup.txt` is committed, delete its contents and save it. Then recover the last committed version of the file using `git checkout`.
    (b) Create a new file and commit it. Rename it using `git mv`, commit, then `git rm` the file and commit again.
    (c) Make a change to writeup.txt and run `git diff` to see the latest changes, then use `git diff` with arguments to see the difference between the last committed version and the commit before you added the hobby to your writeup.

    Make sure that by the end of playing around with those commands your `writeup.txt` is back to being named `writeup.txt` and contains the two lines described above, and your `username.txt` also conforms to the specification given above.

Branching and Merging

15. Read through Git's documentation on branching and merging:

    `https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging`

16. Create a branch named `song` and checkout that new branch. Edit your `writeup.txt` to add the line `3) Favorite song: ABC by XYZ` where you replace `ABC` with the track name of your favorite song and `XYZ` by the artist/band of the song. Add and commit your change to the branch, then merge the changes into the `main` branch. Normally, you'd delete a branch after you've merged it into main in order to keep the repository from getting too messy. In this case, go ahead and leave this branch. Sometimes, you'll just use a branch in your local copy and the remote (github) repo never needs to know. However, if you are working on a branch and, say, need to collaborate with someone else on that branch, you'll need to sync up your local branch with a corresponding branch on the remote copy. To do this, you'll need to issue the following command:

    ```
      git push --set-upstream origin song
    ```

    What's happening here? You've seen `git push` before. Here, we're telling git that we want to push the local `song` branch to the `origin` remote repository, and since the branch doesn't already exist up on the remote copy, `--set-upstream` tells git to set up a branch on the remote that *tracks* with your local branch; this way, next time you `git push`, git will know to push your local `song` branch to the corresponding one on github.

Java and Gradle

In this class, we'll be writing our code in java. To facilitate the process of compiling, testing, and running code, we'll be using a *build tool* called Gradle. This replaces the command line workflow (using javac to compile and java to run) or the build-and-run functionality provided by IDEs.

Your repository already contains the directory structure used by Gradle to keep track of source code and compiled programs. You will find a file called Hello.java in the `app/src/main/java/lab1/` directory. This file contains a Java implementation of the canonical "Hello, World" program.

17. Whereas you may be accustomed to compiling and running programs directly using the `javac` and `java` commands, we will instead use Gradle to compile and run the program for us. Gradle needs to be run from the project root directory or the `app` directory, so make sure your terminal is there. Then, you can run the Hello World program with the following command:

    ```
    gradle run
    ```

Note that if the code hasn't been compiled yet (or has been changed since it was last compiled), gradle will automatically compile it for you before trying to run the program.

18. Edit `app/src/main/java/lab1/Hello.java`, modifying the program to print the contents of the 0th command-line argument instead of "world". Recall that the `args` parameter to the main method contains the command line arguments passed to a Java program.

19. Run your modified code and see if it works as expected. To pass command-line arguments to your program via gradle, use the `--args` flag as follows:

```
gradle run --args="your args here"
```

A sample invocation of the modified program should look something like this:

```
$ gradle run --args="241"

> Task :run
Hello, 241!

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Stage, commit, and push your changes to Hello.java:

```
git add app/src/main/java/lab1/Hello.java
git commit -m "Hello now prints command line arg"
git push
```

Running Unit Tests

In this final section, you'll make one final modification to your main program, and learn how to run **unit tests** in the process. A unit test is a piece of code that verifies that some small piece of your program's code behaves as expected. We'll talk more about unit tests in the next lab, but for now you'll learn the basics of running and debugging using unit tests in gradle. Our goal in this section is to make one more modification to the Hello world program so that instead of "Hello", it prints the time-appropriate greeting "Good morning", "Good afternoon", or "Good evening". Notice that there's a method in `Hello.java` called `getGreeting` that currently just returns `"Good morning"`. If you're working on this in the morning, that's the correct behavior!

20. I've written some unit tests for you to make sure that all of the cases are covered correctly. If you open the file `app/src/test/java/HelloTest.java`, you'll find that it contains three methods that pass different `LocalDateTime` objects to the `getGreeting` method to make sure it gives the correct response. You can run these tests using gradle by entering the command `gradle test`. When you do so, you should find that your code currently fails two of the three tests, giving you a helpful message, the relevant part of which looks something like this:

    ```
    lab1.HelloTest > test01afternoon FAILED
        org.junit.ComparisonFailure: expected:<Good [morning]> but was:<Good [afternoon]
            at org.junit.Assert.assertEquals(Assert.java:115)
            at org.junit.Assert.assertEquals(Assert.java:144)
            at lab1.HelloTest.test01afternoon(HelloTest.java:41)

    lab1.HelloTest > test02evening FAILED
        org.junit.ComparisonFailure: expected:<Good [mor]ning> but was:<Good [eve]ning>
            at org.junit.Assert.assertEquals(Assert.java:115)
            at org.junit.Assert.assertEquals(Assert.java:144)
            at lab1.HelloTest.test02evening(HelloTest.java:48)

    3 tests completed, 2 failed

    FAILURE: Build failed with an exception.

    * What went wrong:
    Execution failed for task ':test'.
    > There were failing tests. See the report at: file:///Users/wehrwes/Documents/2220/
    ```

    From this, we can see that two of the three tests failed. Furthermore, you can see the stack trace including the specific line of each test that failed, which can be immensely helpful for debugging.

21. Modify the `getGreeting` method to change the greeting based on the `LocalDateTime` object passed as an argument to the method. The documentation for this class can be found here: `https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html`. When you're done, re-run `gradle test` and see if your tests pass. If a test is still failing, find the line that's triggering the test failure to see if that gives you any insight into what's going wrong. Debug your `getGreeting` method until it passes all three tests.

22. The unit tests are intentionally focused on just the `getGreeting` method; the behavior of your main program should be unchanged up to this point. Now that you know `getGreeting` is correct, use it in your main program to print the appropriate greeting in place of "Hello". For instance, as I write this around 2:30pm, entering `gradle run`

`--args "241"` on the command line results in my program printing `Good afternoon, 241!"`.

You can get a current `LocalDateTime` object using the static `LocalDateTime.now()` method. Use `gradle run` to check that you get the correct output for the current time of day. You won't be able to test the rest of the cases unless you wait a while, but that's why we had unit tests—to make sure all the cases are handled correctly. As usual, commit and push your changes to github when you have things working.

Submission

Make sure that you've followed all of the steps above, all of your latest changes are committed to your repository (check this using `git status`), and push your submission to github. In this and every other git-based assignment in this course, that's all you have to do! We'll clone your code from github and provide you with feedback in the "Feedback" pull request (see the syllabus for details on how to view your feedback).

Grading

This lab is worth 10 points, assigned as follows:

- 5 points: All steps are completed and your repository's state is as specified; we will check this using `git log` and `git blame`.
- 2 points: `username.txt` contains name, username, and github username as specified
- 1 point: `writeup.txt` contains name, hobby, and song as specified.
- 1 point: `getGreeting` passes all three unit tests
- 1 point: `Hello.java` prints the appropriate greeting followed by the 0th command-line argument, followed by "!"

Acknowledgments

This lab is based on materials developed and refined by Tanzima Islam, Brian Hutchinson, Filip Jagodzinski, Qiang Hao, Nicholas Majeske, and several past TAs.