**Lab 13:  Version Control with git**

The goals of these lab activities are to learn the basics of using git and GitHub together.

1. Install git on your laptop.  See instructions below.
2. Create an account on [github.com](https://github.com/) if you do not already have one. Use your HPU email ID,
3. Download a copy of the [Git Cheatsheet](https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf). You'll want to refer to this during lab, so you may want to print a copy and bring it to lab. Otherwise be prepared to look at it on your computer.
4. Review the slides we covered in the lecture on git and version control.
5. Look over one of the two suggested readings on git:
   * “Git Immersion” <http://gitimmersion.com/>
   * Simple tutorial of the basics: [http://rogerdudler.github.io/git-guide/](http://rogerdudler.github.io/git-guide)
6. Complete the activity listed below and submit the screen-capture described there to BlackBoard Week 14, assignment

**Upon Completing the Lab Activities:** Submit the output of a *git lo*g command showing your work in the lab to the BlackBoard assignment called *Week 14*.

**Installing git on your Laptop**

Linux and OS X users may have git installed already (type *git* to see).

You can download a version of git from GitHub and other providers, but we recommend you download it from the main git site at this URL: <http://www.git-scm.com/downloads>   
**Note: This is particularly important for Windows users,** since this download comes with a bash-shell (like you find on Linux) called **git-bash** that we ask you to use.  We've seen git commands fail mysteriously when run from the Windows command-prompt. Don't use that! **Use git-bash!** It's a bash shell command line environment, just like Linux, with git commands and also basic Unix commands installed, plus some other useful things in its /bin folder: ssh, scp, perl, curl, vi, vim.  If you need to do UNIX shell kinds of things on your Windows machine, this is a nice thing to have.  (FYI, if you know cygwin, it's like that but lighter-weight, since it installs less.)

Follow the directions to install git.  Here are some notes for Windows users.  When running the installer, I kept all the defaults, except you need to make sure you choose the following:

1. In the window *Select Components*, I chose
   * Under *Additional Icons*, I checked *On the Desktop*(This puts the Git Bash icon on your desktop.)
   * Under *Windows Explorer*integration, I checked *Get Bash Here* (This lets your right-click on a folder and open Git Bash in that folder.)
2. In the window *Adjusting your PATH environment*, choose *Use Git from Git Bash only*  
   (Experience in previous classes suggests that you should only use Git from Git Bash.)
3. In the window *Configuring the line ending conversions*, keep the default.

**Initial configuration (one time only, for all platforms):**  
  
Start a shell window to use Git (on Windows use Git Bath, on Linux or OS X open a "terminal" window), in that shell window, set some configuration values.  Type the following but update the final arguments:  
*git config --global user.name "Your Name"  
git config --global user.email "your\_email@whatever.com"*  
  
There are times a git command will bring up an editor window. The default is *vi*, but if you want *emacs* instead, type:  
*git config --global core.editor emacs*

First, create a repository on GitHub:

1. Sign into www.github.com with your account.  (Don't have one? See the top of this page.)
2. Create new repository.  (See instructions they give, or just go for it by clicking the plus-sign.)
3. Name the repo *csci2916-labdemo*, and make it public.
4. Check the box to initialize this with a README file.
5. Keep gitignore as none, and choose the MIT License.
6. Now click Create Repository.

GitHub will show the main page for your new repo.   
Near the bottom-right, you'll see a text-box with the text *HTTPS clone URL* right above it, and an icon like a clipboard with an arrow to its right. This textbox contains the URL that you can use with a git-command on your local machine to clone the directory. To do that:

1. Click on the clipboard icon to copy the URL to your clipboard.
2. Go to your shell window on your local machine, and *cd* to a folder where you want to keep your projects in folders. (Create such a folder if you don't have one.). **We will use our current folder so, you do not have to create a folder.**
3. Type *git clone* and then paste in the URL from your clipboard and hit return.  It should look something like this:  
   *git clone https://github.com/mlsmithHPU/csci2916lab-demo.git*  
   (Windows users:  in Git-Bash, you might find copy-and-paste commands by right-clicking in your window.  If not, see [this link](https://www.udacity.com/wiki/ud775/git-bash-copy-paste) for some more info on how to cut and paste.)
4. This will download all the files from the remote repo into a directory (named after your repo) in the current directory.

Congrats! You now have a local repository on your machine you can work with, plus a separate repository on GitHub where you can upload your files for safekeeping or to share.  Now do the following from the shell window on your local machine:

1. Use *cd* to make sure you're in the *csci2916-labdemo* directory.
2. Type *ls -a* to see what files are there.  Note the hidden files that begin with a "."
3. Create a file *hello.java* that prints "hello".
4. Type *git status* and make sure you understand what that tells you.
5. Type *git add hello.java* to add that file to the staging area.
6. Type *git status* and make sure you understand what that tells you.
7. Add the file to your local repo by typing *git commit -m "Initial commit"*
8. Synchronize your local repository with your remote repo on GitHub by typing *git push*

Go back to your web-browser showing your GitHub repo.  Refresh your browser window, and you should see the hello.java file in the list of files.

**Submission:**

Take a screen-capture of this window, and submit the image file to *Week 14 Assignment* in BlackBoard.  Make sure your image file shows your repo name (which includes your GitHub id) and the list of files.  

**Lab Activity 1:**

Feel free to work on all the activities with a partner. Everyone must do them, but you can talk with your fellow students about how to do them. Helping each other learn in this lab is a good thing!  
  
The goal of this activity is to create some history! That sounds much more dramatic than what you're going to do. You'll enhance your program by making a series of small changes, committing the file to your local repo each time. The changes you are committing here are much smaller and more trivial than you would normally do using version control. Again, the goal here is to get a sequence of commits into your local repo.

1. Edit your file *hello.java* to make a method called *greeting(msg)* that prints out the msg parameter. Use this to print your message "hello".   Test your program
2. Before adding it to the staging area and putting it into the repo, use git's *status* command to see what that tells you about the status of your files.
3. Add it to the staging area, and use git's *status* command to see what that tells you about the status of your files.  
   *git add hello.java git status*
4. Now, do what it takes to commit it into the local repo (with a good commit message).  
   *git commit -m “Added method”*
5. Use git's log command to see the history of your changes.
6. Now, create a file *helper.java* and print out a helpful message
7. Do *git add* again to put it into the staging area.  Before you commit....
8. Question:  What if you change your mind, and decide you do not what this in the staging area?  How do you remove it?  Hint: type *git status* and it will tell you.  Try this.
9. Then do what it takes to put that into the local repo (with a good commit message). Before proceeding, use git's *log* command again to see the history of your changes.  Try using this form:   *git log --oneline*
10. What if you goof up?  Maybe you commit but then decide the message wasn't right? Maybe you forgot to stage a file for this commit?  Maybe you've decided that file you just committed was bad, and you want the version before that commit?  These are easy mistakes to correct.  See [the section "Undoing Things" in the Git Pro book](https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things) on-line if you ever need to do any of these.

At this point, you should have a short set of entries in your project history. Note that in the output of the *log* command you will see each commit identified by an SHA-1 hash (or the first part of that). A hash value is created for each set of files. This is how git identifies a set of files that makes up a commit.

**Lab Activity 2:**

The files in your local repo have changed, but the remote repo has not. If you want the remote repo to match your GitHub repo, it's easy: *git status* will remind you what to do if you forget.  But before we do that, here are some things to know:

* When you cloned the GitHub repo to your local machine, your local git repo remembers this. It keeps track of a **remote** that points to the "remote address" of that remote repo.  Remember this term: a **remote** is a reference to a repository on a remote server.  
  Use this command to see what remotes are active:  *git remote -v* (Try it!)   Very often a remote is given the name *origin*, which is the case here from our use of *git clone* earlier.
* Another term to know is a **branch**. We'll say more about these in the next activity. If you think of your series of commits as a plant growing from a starting point, a branch is a pointer to where the next commit will "grow" from.  In what we've done in this lab so far, right now our series of commits is like a vine, with no branching at all.  Each new commit is added to the end. But it's possible to have more than one branch.
* Each branch has a name. You can give new branches a name you choose.  When you start a new project, the "default" branch is given the name **master**.  There's no special meaning for git; it could have called it *default* or anything.  (It's not in charge of anything!)
* You can see the branches in your local repo by typing:  *git branch* (Try it!)  
  You can see all branches by typing:  *git branch -a*   (Try it!)
* A branch points to a particular commit.  If you are "working on that branch", this means that your working copy (the files you see in your directory) are the file-versions in that particular commit, and that any new commit you make will "grow" from that commit.
* You can switch from one branch to another using the git command *checkout*.  This will change the files in your working copy!
* When talking about branches in git, **HEAD** refers to the current branch that you're working on.  So HEAD is a pointer to a branch.
* Back to the concept of a remote.   When you create a remote to access versions of files on a remote repository, git actually creates a new branch that is identified by *remote/branch*.  So when you typed *git branch -a* above, you see *origin/master*.  This tells there is a 2nd branch call *master* but this one shows the files as they were when you cloned the repo from GitHub.

Do the following:

Make sure we know what's in our current files.  Print out or view hello.java so you see what's in it .  Also, type *git log --oneline --decorate* which shows the history of this branch and shows the branch names.

1. Let's change branches, and move to the branch origin/master.  Type:  *git checkout origin/master*  
   Now see what's in *hello.java* and type *git log --oneline --decorate* again.
2. What's going on?  Do you understand what's happened?  (Note:  when you did the checkout, you got to see some messages about the fact that you're now working on a branch that shouldn't be changed, since it's supposed to reflect what's on your remote repo.)
3. You can go back to your local master branch.  Type:  *git checkout master*  
   Now see what's in *hello.java* and type *git log --oneline --decorate* again.
4. Let's send all our changes in our local repo to the GitHub repo.  Once we've done that, it will store the same versions of the files there as we have here locally.  Just type: *git push*  
   Look at the messages.  Type git status and note that it says "Your branch is up-to-date with 'origin/master'." **Note**: you may have to provide your Github account name and Github password.
5. Go back to GitHub, do a refresh.  Is all as you expect?
6. While on GitHub, edit your README.md there in the web-interface.  Change an existing line.
7. Go back to your local machine, and edit README.md by changing that exact same line in a different way.  We're trying to force a conflict here, so do what it takes to create something that cannot be merged automatically.  Put this version into your local repo.
8. Now try pushing your files to GitHub.  It should report a problem, right?
9. Now try pulling your GitHub files down to your local machine.  You should get a conflict situation.  Use your editor to fix to resolve this conflict (as you see fit), and then push this back to GitHub.  Check out the state of things on GitHub.  Is all as you expect it to be?

**Lab Activity 3:**

Let's practice branching.  Our goal here is for you to create two different working versions of your Java program.

1. On your local machine, create a new branch with the *git checkout -b* command.  Name your branch *develop*.   - *git checkout -b develop*
2. Switch to that branch.   Now, create a new .java file that does something simple, perhaps changing or using your existing files.   Feel free to edit your existing files too.  We want some changes to this new branch.
3. Do what it takes to put those changes into your local repo.
4. Switch back to your master branch.  Look at the files in your directory.  Do you see the new file you added?   Why or why not?
5. Again, create a new .java file here in the master branch, and change some existing files too.  Do what it takes to put these into the local repo. Look at the files in your directory.  Now, switch to the other branch and look again.  Is all as you expect?
6. Switch back to the master branch, and now merge that other branch into this branch.  Resolve any conflicts.  Look at all your files.  Is all as you expect?
7. Type this command:  *git log --graph --oneline --decorate*  
   This will show a graph-like version of your branches, with info on which commit is in which branch.  Does its output make sense to you?
8. Finally, from here in the master branch, push to the GitHub repo.

This scenario mimics a common work-flow.  You want to keep track of the stable version of your files.  You use the branch *master* for that.  New development can continue on the *develop* branch, where you make changes and test them, but wait to add them to the stable version of your code-base (i.e. the *master* branch).  Once you're satisfied they can be added to the stable-code base, you switch to the *master* branch and then merge the *develop* branch into it.  (And, if you happened to make changes to the *master* branch too, those will be merged as well.)  
  
See [the section "Branching Workflows" in the Pro Git book](http://www.git-scm.com/book/en/v2/Git-Branching-Branching-Workflows) on-line for more info on this idea.  

**Submission:**

**By the end of lab**, submit the output of this command from your master branch:  *git log --oneline --decorate –graph* in the text-field of the BlackBoard lab assignment for Week 14.