Git and Command Line

Lab 7
Wednesday, October 25, 2017
CSC 062: Fall, 2017

Introduction

This is a rather long lab, therefore we will split it in two parts. Part A will introduce you to `git`, while Part B will familiarize you with basic command-line tools. Due to the unusually long nature of this lab, you can submit your work by Wednesday, November 1.

Part A: `git`

`Git` is a version-control system, an essential tool in everyday software development, especially when multiple people are working on a project at once.

Accordingly, you must work with a partner on Part A of this lab (or in a group of three if you have to), as you will be learning techniques to collaborate. Additionally, you and your partner will have to work on a lab computer for this lab in order for sharing things to work properly.

Part B can be completed independently.

Background

`git` is a tool for "version control." This means that it keeps track of multiple versions of files, and allows you to switch between them freely. In general, we have two sources of file versions. First, over time, files change, so we have historical versions. Second, when multiple people work on the same file, they make different changes, so we have per-user versions, which may have different histories but at some point may be merged into one version again. `git` keeps track of both of these kinds of versions for us, and can even automatically merge changes in some cases!

For Part A of this lab, we will work in pairs and make diverging edits to a file, but then use `git` to produce a combined version, all the while tracking the history of changes we have made.

Setting up `git`

`git` is primarily a command-line tool, so we will start by opening up the terminal. Both partners should go ahead and use `cd` to get into your Eclipse workspace directory (possibly somewhere like `Documents/cs062/workspace` depending on what you did at the beginning of the semester; if you go to Project – Properties in Eclipse the `Location` item should tell you where this is). Now we will have one partner set up the project: leave the terminal there and create a new Java project in Eclipse called "Lab07." It is important that only one partner does this step! (The other partner should follow along for a bit.)

Whoever created the Lab07 project should add a file called `Random.java` and then go back to the terminal and use `cd` to enter the newly created Lab07 directory. In the terminal from within that directory, type:

```
git init
```

This command tells `git` that we want to start a new version-controlled repository in this folder. At this point, `git` may ask you to set up some basic information, like an author name. Follow the instructions it gives to do this. Once you are done with this step. it should say something about "Initialized empty `git` repository in?" Having a repository is nice, but we also need to tell `git` which files it should keep track of. To see the state of our repository we can use:

```
git status
```

This should print out something like the following:

```
On branch master

Initial commit
```

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

    .classpath
    .project
    bin/
    src/

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

This status report is telling us that a `git` repository exists here, but it has not been told to keep track of anything yet. To fix this, we will type:

```
git add .
```

Here the '.' means "the current directory" and by extension, "everything in the current directory." Afterwards, if we run `git status` again it should give us a big list of new file stuff. At this point, we have told `git` it should keep track of those files, but we have not yet told it that it should take a snapshot of them. I said earlier that `git` tracks versions over time, but it would not make sense for it to keep track of every individual letter that you type, so `git` waits for you to tell it when to capture a snapshot. We do this by using:

```
git commit
```

This will take all of the added files (that have tracked using `git add`) and remember their current state so that we can get it back later if we want. It will also ask you for a "commit message" which it will record to help you remember what was going on. Note that `git` uses the `EDITOR` environment variable to pick an editor for you to edit your commit messages. This may be an editor you have never used before. For now, you can just use the '-m' argument to specify a message on the command line, for example: `git commit -m ''My message goes here''`. Alternatively, enter "Initial commit" as your commit message and save/quit the editor so that `git` takes a snapshot.

At this point, we have saved a version of our project and our partner can now get a copy of it. We have one more step before that point: we need to make our repository accessible to our partner. To do this, we will create a link to the repository from our home directory, and change permissions to let anyone access it. First, from the project directory (where you already are in the terminal), type:

```
chmod -R 775 .
```

This will set permissions on this folder and all included files to 775, which enables anyone in the same group as us to have full permissions (including write permissions!) to those files. This isn not normally something you would set up, but it is useful for this lab. Next, we need to make an accessible path from our home directory all the way to the project directory. To do this, use the following commands (or similar commands if your workspace is in a different place):

```
chmod 750 ~
chmod 750 ~/Documents
chmod 750 ~/Documents/cs062
chmod 750 ~/Documents/cs062/workspace
```

If your workspace is somewhere else, use the correct path to the Lab07 project directory, which you can find by typing pwd into the other terminal (keep that one around as we will be using it again).

At this point, your partner, on their own computer, should be able to do the following (from their Eclipse workspace directory; `cd` there if necessary):

```
git clone /home/partner/Documents/cs062/workspace/Lab07
```

Here, replace "partner" with your partner's username. The `git clone` operation creates a copy of a repository, but it remembers the original source, so that you can send back updates later. At this point, both you and your partner have a copy of the same (empty) `Random.java` file, and in fact identical histories of that file (albeit with only a single entry each). In other words, `git clone` did not just copy the files, it copied their entire histories. If you are curious about efficiency, rest assured that `git` is pretty smart: instead of storing actual copies of different versions of files, it only stores the differences between versions, which allows it to reconstruct different versions when it wants to without storing too much data.

For the remaining sections of Part A, go ahead and **work separately from your partner, each person picking one of the two tasks below** and working on their own copy of the Lab07 project in Eclipse. Do not forget that you may have to tell Eclipse to "refresh" on occasion in order to see the changes that `git` makes. Right after cloning the repository, you may also have to tell Eclipse that you want to open the project. Use `Import > Existing Projects` and select Lab07.

Task 1: Pseudo-Random Numbers

Before starting this task, use `cd` to get into the project directory, and then execute the following commands:

```
git branch nextint
git checkout nextint
```

This will create a new "branch" for working on this feature, so that our changes won't interfere with our partner. The second command switches over to the new branch.

For our `Random.java` file, we would like to implement a dead-simple pseudo-random number generator. We will need a constructor, which will store an integer instance variable that is passed in as a parameter, and a `nextInt` method:

```
private int state;

public Random(int seed) {
    this.state = seed;
}

public int nextInt() {
    // code here
}
```

Go ahead and add these methods, and then implement the following algorithm for the `nextInt` method:

1. To get the next "random" integer, we will first add a reasonably large prime number to our current state value (four digits should suffice, you can use Wolfram alpha to find a large prime by giving a query like "prime 1000").
2. Next, we will take our state value and multiply it by another largish prime (three digits should be good enough).
3. That's it, we just return the state value.

This random integer algorithm is not very random, especially for the first few results when given a small seed value, but it could be used in some applications where we don't care much about quality.

When you've finished the `nextInt` function go ahead and commit your changes using `git commit` with an appropriate message, such as "Implemented Random constructor and Random.nextInt." Remember you have to add your changes to be tracked first using `git add` (in this case we could just say `git add src/Random.java` since we have only changed a single file). If you are lazy, you can give `git commit` the '-a' switch to tell it to automatically add all already-tracked files. Now you can wait for your partner to be done with their changes.

Task 2: Choose an Item

Before starting this task, use `cd` to get into the project directory, and then execute the following commands:

```
git branch choose
git checkout choose
```

This will create a new "branch" for working on this feature, so that our changes will not interfere with our partner. The second command switches over to the new branch.

For our `Random.java` file, let's assume we have a working `nextInt` function that takes no arguments and returns an integer, and implements a `choose` function which picks an item out of a `List`. Go ahead and add a method:

```
public Object choose(List l) {
    // code here
}
```

Since our `nextInt` method might return any integer, positive or negative, we can just use a modulus operation (%) to constraint it to the range we want (`l.size()`). We can use `l.get()` to get the appropriate element as an Object and return it.

Once you have finished `choose`, go ahead and commit your changes (see the last paragraph from task 1) and wait for your partner to be done.

Merging

Okay, now that both partners have made inconsistent changes to the `Random.java` file, we would like `git` to merge these into one version. Luckily, our changes were inconsistent at a high level, but conceptually they are not really opposed to each other: we worked on separate methods, and if you just put all the code together, things should work. In this case, `git` should be able to figure that out and merge automatically. To do this, **whoever did the `git clone` operation** should try to do:

```
git push
```

The `push` operation says: take my local changes, and push them back up to the place I got the code from in the beginning. There is an analogous `git pull` operation that pull versions from the original repository. In this case, because the origin does not know about the new branch we created, `git` will tell us to do the following (where `<branch>` is whichever branch we worked on):

```
git push --set-upstream origin <branch>
```

After doing this once, the "upstream" repository will know about the branch we created, and we can just do a normal `git push`.

Next, whoever set things up originally should go ahead and merge things together. To do this, we will first want to switch back to the "master" branch, by doing:

```
git checkout master
```

Notice that after doing this, all of your code changes are wiped out in Eclipse (you may have to refresh). That's scary, but with `git`, everything you've ever committed is saved, and it is just a question of how to get it out again. In this case, we went back to the master branch which hasn't seen any of our changes yet. To pull them in, we use `git merge`. Since our partner has already pushed their branch, if we type just:

```
git branch
```

We should see all three branches in our repository: `master`, `nextint`, and `choose`. So let's start with `nextint`:

```
git merge nextint
```

This should say something about "fast-forward" and pull in the changes from that branch, so that if we refresh in Eclipse, they're visible. Next, merge the other branch, using:

```
git merge choose
```

This will pull our partner's changes and try to merge them automatically. If it fails, it will show a warning and will put both versions together into the file while marking where they differ, so that we can fix up the disputed area and do a `git commit`.

If `git` asks you to, fix up any overlap areas, (they should be clearly marked) and then do another `git commit`. Now that we have reconciled our differences into the `master` branch, our partner can do a `git pull` to see everything that we've done, followed by a `git checkout master` so that they're also on the `master` branch again and can see all the changes.

The `git` Workflow

This is the usual workflow of `git`: you create a branch for working on a specific feature, commit (perhaps several times) to that branch as you work on it, then eventually merge it back into the master branch, which may take some manual fixing-up (but `git` does a lot for you). No matter how many people are working on the same code, as long as they work mostly on different parts of it, they can all work separately this way, and still synchronize their results periodically as branches merge together.

As a bonus, `git` keeps track of history for you, so it is unlikely that you will ever lose more work than what you have done since the last time you ran `git commit` (for this reason, it is a good idea to commit often). `git` stores all of its data in a hidden `.git` directory, so unless you wipe out that directory (and any clones other people may have made) you will be able to recover your project.

Note that with most development environments, including Eclipse, there are plugins you can add to use `git` or other VCSs from within the IDE. These can be more convenient, but it is useful to know the raw operations so that you know what you are doing when you use those plugins.

Part B: Command Line Tools

We can now move to second part of this lab. Working with the partner from Part A is optional.

Note that the Documentation and Handouts page of the course website has a command-line tools introduction PDF that covers some of the same material. For today, we'll be learning about the basic commands:

```
cd          ls
mv          cp
rm          rmdir
mkdir       man
ssh         scp
```

We will also learn about how to redirect output and some useful commands like:

```
less
wc
grep
xargs
```

Finally, we will be learning the basics of text editing with `vim`, and how to compile and run Java code using `javac` and `java`.

Set up

For Part B, we will be doing all of our work on `little.cs.pomona.edu`, which is one of the department's common servers. Because the server isn't hooked up to a monitor, we'll be forced to use the command line to do everything. Start by connecting to `little.cs.pomona.edu` using the command:

```
ssh yourusername@little.cs.pomona.edu
```

If you are connecting from one of the lab machines, you can leave off the "yourusername" part, as the local username will be the same as the remote one. You will have to type in your password, and then you will have a terminal that is running on the remote machine. Because your home directory (`/home/yourusername`, which can be abbreviated as `~`) is actually mounted remotely from a common file server, the files in your home directory on `little.cs.pomona.edu` are the same as those on the lab machines. You can see this if you run the command:

```
touch ~/Desktop/hello.txt
```

Which will create a file called `hello.txt` on your desktop. The file should show up on the desktop of the lab computer you are using within a few seconds. For this lab, we will be working exclusively through the terminal though.

To start Part B, `cd` into your desktop (`cd ~/Desktop`) and then copy over the starter files using the `cp` command:

```
cp -r /common/cs/cs062/labs/lab07 lab07
```

Notice that like all shell commands, `cp` takes arguments separated by spaces. Some arguments start with - and are called "flags" or "switches"?they alter the behavior of the command in some way. Other arguments are just listed out in order, and they tell the program what to do or operate on. In the case of `cp`, the first argument(s) are the thing(s) to copy, while the last argument is the directory to put them in (or the new name to copy a single source file as).

For `cp`, the '-r' flag forces it to copy entire directories 'r'ecursively, rather than just copying individual files. But what if you wanted to figure that out? The first and most useful command we'll talk about today is called `man`, which stands for manual. `man` can be used to get information about most other programs, so you can type `man cp` to get the manual page for the `cp` program. This will show how the command should be used (in the "SYNOPSIS" section) and list all of the different possible arguments along with what they do). If you want to figure out how to do something with a command, try to use the manual to find what you are looking for.

Notice that at the bottom of the manual display it says "press h for help or q to quit". The command "man" looks up a manual page, but then uses another command, called `less` to display it. `less` is nice because it allows you to search for things: just type / followed by the text you want to search for. So to find out what '-r' does, we can do `man cp` and then type `/-r` to see the part that describes the `-r flag`. You can view any file using `less` by just typing `less <filename>`.

Sorting Files

To help you get familiar with some basic shell commands, the starter files include some simple tasks. The first is to separate out some image and text files. Go ahead and `cd` into the `images-and-text/` directory.

Now is a good time to be reminded that you can use the "Tab" key to auto-complete the names of files when typing in a shell. So you can type `cd im<Tab>` instead of `cd images-and-text`. This both saves you keystrokes and ensures that you don't accidentally misspell the filename. If you just hit "Tab" several times without typing anything, the shell should list out all of the possibilities. You can use this to navigate when using `cd` to go deep into a directory structure.

Now that we are in the `images-and-text/` directory, use `ls` to list the directory contents. Notice that there is a mix of `.jpg` and `.txt` files in this directory. Your first task is to sort these into an `images/` directory and a `text/` directory. Start by creating these directories using the `mkdir` command. Now you can use the `mv` command to move the `.txt` files into the `text/` directory and the `.jpg` files into the `images/` directory.

Moving each file individually is a bit of a chore. Instead, we can ask the shell to handle all files that match a pattern. The * character in a shell argument will match any number of characters, so we can type `mv *.txt text/` to move all of the `.txt` files into the `text/` directory. We can use a second wildcard command to move all the `.jpg` files into the `images/` directory.

Counting Files

How many text files are there? We could count by hand, but instead we will use another shell trick to find out. There is a handy command called `wc` which counts words (and/or lines or characters). Notice that if you run just `wc` nothing happens. In fact, it cuts off your terminal (try typing `ls`)! Luckily, there's a universal shortcut for interrupting the current process: hit `control-C` and the rogue `wc` process will be killed, giving us back our normal prompt.

What happened? Well, if we look at `man wc` we can see that if no "FILE" is given, it will "read standard input." This means that when we ran `wc` it was expecting us to type in its input. That's all fine and good, but how do we tell it that we're done? Well, the terminal has another shortcut for that: `control-D` will tell a program that you're done giving it input (be careful: if you hit `control-D` in the outer shell, it will quit the `ssh` session, and you will have to log in again). So we can type `wc` and then `a<enter>b<enter>c<enter><control-D>` and it should print out `3 3 6` which is the line-count, word-count, and character-count of the input, respectively. Notice from `man wc` that we can give it the `-l` flag to count just lines.

But how can we get `wc` to count files? It can read what we type or something from a file, but we want it to read the output from the `ls` command. In the shell, there are two ways to do this. First, we can redirect output into a file, using the `>` operator. So we could say `ls > list` and then `wc -l` list to count the number of files in the current directory. However, this would leave around an extra list file every time we did it. Instead, we can redirect output directly from one program into another using the `|` operator. So to count how many text files there are, assuming we have moved them all into the `text/` directory, we can do:

```
ls text | wc -l
```

This should print out "10." Note in this case that we did not give `wc` any arguments (besides the flag), so it works in "read from standard input mode." However, because of the `|` redirection (also called a "pipe") the standard input came from the output of the `ls` command, instead of from us typing. This pattern of redirection is quite useful, and can be repeated multiple times, with several programs feeding their output into each other.

Searching for Patterns

Now that we have sorted our text and image files, let's move over to the programs directory. To do this, use `cd ../programs` (assuming you are still in the `images-and-text/` directory). In this directory are a few random Java programs that were downloaded from Pastebin. However, the directory is messy: it also has some `.class` files in it from compiling the `.java` files.

We want to find out which files contain drawing code. To do this, we will use a command called `grep`, which searches for patterns in its input. To search for the pattern "draw" we can use:

```
grep draw *.java
```

Recall that `*.java` matches all files in the current directory that end in `.java`. The entry for `man grep` tells us that it expects a "PATTERN" followed by one or more "FILEs," so this command will search all of the matching files for the pattern "draw." By default, it prints out each line that matches. But in this case, we only want to know which files contain that pattern, not how many times or what their content is. Luckily, `grep` has a `-l` flag which does exactly that: 'l'ists files that match. So we'll use:

```
grep -l draw *.java
```

Notice that `grep` does not care about the order of the `-l` flag: you can put it first, second or third, and the other arguments will be interpreted the same way.

Filtering Files

Now that we know which files contain some kind of drawing code, let's move them all to a new directory. Create a new directory called `draws/` using `mkdir`. Now all we need to do is issue an `mv` command targeting all of our `grep -l` files and putting them into this directory. Unfortunately, our `grep -l` command produces

output (which we could move around with a '—' or put in a file with '>') but what we want those things to be are arguments to the `mv` command. We could redirect the output to a file using '>' and then edit that file to add "mv" before them and "draws/" after them and then run that file as a script, but there's a better way: a command called `xargs` exists just for this purpose.

As `man xargs` says, `xargs` is designed to take input and turn it into arguments. Normally, these arguments are passed to the target program after any default arguments, but in our case, we want the variable arguments to come before the destination argument of `mv`, so we need to use `xargs '-i'` flag to tell it where to put the variable arguments. So we'll construct a command line that does the following:

1. Use `grep -l draw *.java` to generate output.
2. Pipe that output into `xargs -iZZ`
3. Give `xargs` additional arguments to tell it to execute the `mv` command with our variable arguments followed by `'draws/'`. Use the 'ZZ' pattern that we gave to the '`-i`' command to tell `xargs` where to put the additional arguments.
4. The command at the end should look like `grep -l draw *.java | xargs -iZZ mv ZZ draws`

If you execute the command properly, you should be able to move all of the `.java` files that contain the text 'draw" into the `draws/` folder. Note that there's another way to do this: in the shell, the "` character (a.k.a. the "backtick") can be used to do command substitution, where the output of a command is used as part of another command. With this method, we write our `mv` command, and just use `grep -l draw *.java` where we want the extra arguments to appear.

javac

Since we've got these Java programs sitting here, let's see if we can get any to run. First, let's create a new directory `has_main/` and put all of the java files which declare a `public void main` method somewhere in them into that directory. We can use the same technique as we did with "draw", with one caveat: we want to search for the pattern "`public void main`", but that pattern has spaces in it. Spaces are also used to separate arguments, but only one argument can be the pattern. If we just say:

```
grep -l public static void main *.java
```

the first three lines will look like:

```
grep: static: No such file or directory
grep: void: No such file or directory
grep: main: No such file or directory
```

To tell `grep` that we really want one big argument including spaces, we will use double quotes, like this:

```
grep -l "public static void main" *.java
```

Use this command with the pattern above to isolate all of the `.java` files which have main methods into a `has_main/` directory.

Now let's try to compile them. To compile Java files, you use the command `javac` which produces `.class` files as output. Let's start by trying to compile all of the `.java` files:

```
javac *.java
```

Unfortunately, one of our `.java` files has an error in it (well, they were randomly downloaded from `Pastebin`, so what can you expect). Let's skip that file for now by just changing its name (using `mv` to have an extra 'a' in the `.java` part, so that our `*.java` won't pick it up). Now the command above should work. Note that you can query the return value of a command by running `echo $?`. A value of "0" indicates success, any other value indicates some type of failure. The details of shell if statements and variables are too complex to go into in depth for this lab however

java

To run a Java program, we use the command `java`. However, `java` is a bit quirky: we need to tell it the name of a class to run, not just the name of a `.class` file. Luckily in Java, the name of a `.class` file should always be the name of the class it defines. Unluckily, Java requires that classes which are in packages be in a folder with the name of that package. To see which classes need packages, use `grep` to print out lines containing the pattern "package" from `.java` files in the current directory.

Now pick a non-packaged file and run the class, using `java <classname>`. Depending on which you ran, it should either print a few numbers or hang. If it hangs, use `<control-C>` to kill it. Try running a few different classes.

The Classpath

When compiling or running Java code, you often need to tell it where to find external code that your code depends on. For example, if your code depends on the `structure5` library, you need to tell it where to find that code (at least the `.class` files). Both `javac` and `java` commands require this information whenever you use import to import something that isn't part of the Java standard library. You can give Java a list of places to look by using the `--classpath` flag (short version: `-cp`) followed by a colon-separated list of directories and/or `.jar` files to search. So if we want to import `structure5` along with other `.java` files in the `src/` directory, we'd give it:

```
-cp /common/cs/cs062/bailey.jar:src/
```

If we also wanted to use `JUnit`, we'd get:

```
-cp /common/cs/cs062/bailey.jar:/common/cs/cs062/junit.jar:src/
```

As a final convenience, you can separate the `.class` files that `javac` produces from their `.java` files by giving the `'-d'` flag followed by a directory to put output in. Using `-d bin` as an argument along with `-cp`, we can exactly duplicate what Eclipse does when we tell it to run our project. First, we'll run `javac` with the `-cp` and `-d` arguments as above, and then we'll run java with just the `-cp` argument. If all goes well, you should be able to run this way any assignment that doesn't require some kind of graphics, which are unavailable while ssh'd onto another machine.

Editing Text

Now that you can move files around and even compile and run Java code from the terminal, what if you want to edit a file? As hinted at above, you can use `less` to view the contents of a file and even search within it, but it doesn't allow you to make changes.

To edit files on the command line, there's a simple editor called `nano` that lets you type text and do basic operations. It even displays available commands at the bottom of the screen. However, for this lab, you'll be learning a tiny bit of how to use `vim`, a more powerful and efficient editor.

`vim` basics

To open a file with `vim`, type `vim <filename>`. In this case, `cd` into the edit directory from the starter materials and type `vim ed<TAB>`. Now that you're in `vim`, the most important thing to remember is how to quit: type '`:`' followed by '`q`' and then the enter/return key to quit. This works because in `vim`, the '`:`' key initiates a command, and '`q`' is short for the 'quit' command.

`vim` is a modal editor, which means that most keys on the keyboard execute commands instead of just inserting characters. To insert characters, you use the '`i`' key to get into *insert mode* where typing produces text in the file you're editing. When you're done typing, use the escape key to get back to normal mode. Although this is confusing at first, if you get used to it it's considerably more efficient when you enter lots of commands (as opposed to just typing text). In particular, `vim` has a powerful set of commands for moving the cursor, which we can use to our advantage.

In the `edit-me.txt` file, you'll see a line that says "Delete me." Let's move to this line by using the '`/`' command, which starts a search. Type "`/elete<enter>`" and the cursor should move to the first 'e' of the word "Delete." Now you can type "`dd`" to delete the current line. At any point, if you want to undo a command, just type '`u`' (you can also use control-R to redo something you undid).

Next, move down to the line that says "Add a line after this", either with a '`/`' command, or using the arrow keys or the h/j/k/l keys, which move the cursor around in `vim`. Use the '`o`' command to start inserting on a new line after the current line, and type something in. Then hit escape to get back to normal mode.

Next, go to the line that asks you to add to its end. Here, use the '`A`' command (shift-a) to append to the end of the line (while '`i`' and '`a`' 'i'nsert and 'a'ppend respectively, 'I' and 'A; do so at the beginning or end of the current line instead of at the cursor). Again, type something and hit escape.

Finally, let's replace all of those targets with some other text. To do this, start by using a '`/`' command to find the first target. Then notice that you can use the '`n`' and '`N`' commands to move forwards and backwards among matches to the last pattern. Now, we'll use the '`c`' command (for 'c'orrect). The '`c`' command is special, because it wants to be told what to correct. After '`c`', you can give `vim` a "movement command" to correct text from the cursor to the end of the movement. So "`cl`" would correct just the character under the cursor ('`l`' moves the cursor right one character), while "`cj`" would correct two entire lines ('`j`' moves down a line). We'll use a special movement command "`iw`" that means "the current word." So our whole command will be "ciw", after which we'll type some replacement text and then hit escape.

Conveniently, this `ciw` command, up to the point we hit escape, counts as a single command. And `vim` happens to have a shortcut for "repeat last non-movement command:" the "`.`" command. So to replace all of the targets quickly, we can now just hit "`n`" followed by "`.`" several times. This "`.`" command makes `vim`'s other commands much more useful, because it's easy to repeat them (`vim` also has a full-fledged macro-creation system).

Saving your work

One other important command in `vim` is the "`:w`" command, which 'w'rites out (saves) the current file. Once you're done editing `edit-me.txt`, you can save it and quit it: "`:wq`" in an abbreviation that does both save and quit at once.

What to Hand In

For Part A, you should export your Eclipse project, rename the folder appropriately and include an `lab07.json` file (you can copy and rename any previous .json file).

For Part B, fill the `answers.txt` file with your username(s) and the answers to the questions in that file. Make sure you put your answers exactly where the example answers are. It may be useful to know that the "D" command deletes text from the cursor position to the end of the current line.

When you are done, rename use the `mv` command to move the `answers.txt` file to the `Lab07_Names` directory and submit it:

```
/common/cs/cs062/bin/submit cs062 lab07 Lab07_Names
```

If you want extra practice, try each putting your own name into `lab07.json` and using `git` to merge the different versions. If you have worked on Part B, specify it the json file.