CMPS 12L
Introduction to Programming
Lab Assignment 2

The purpose of this assignment is threefold: get a basic introduction to the Unix operating system, to learn how to create and edit text files using either the Vi or Emacs text editors, and to learn to compile and run a Java program.

Preparation

Before attempting this assignment, begin reading one of the Unix tutorials linked on the course website, such as https://docstore.mik.ua/orelly/unix2.1/lrnunix/index.htm. You need not complete the tutorial, but find one that you like, start it and bookmark it for future reference. Vi and Emacs are two very popular text editors available on all Unix systems. Also start reading one of the Vi tutorials, or the Emacs tutorial that are linked on the course website.

Unix

Follow the instructions at https://its.ucsc.edu/unix-timeshare/tutorials/how-to-connect.html to get logged into the remote Unix server unix.ucsc.edu. Note that if you use the recommended Bitvise program on windows machines, when you login it will open both a command line (ssh) window and a file transfer (sftp) window.
If all this was done correctly you are now logged on to the Unix Timeshare. You will see a prompt that most likely looks like: `bash-4.1$`. This is the Unix command prompt, indicating that the command interpreter is waiting for you to type a command. In the examples that follow, I will represent this prompt by the single character: `%`. If any of the above steps failed and you cannot logon, you'll need to attend a lab session and get help.

Type `ls` to list the contents of your home directory. Use the command `mkdir` to create a new directory called `cs12a` in which you can place work for this class. Type `ls` again to see the new `cs12a` directory listed. Make `cs12a` your current working directory by typing `cd cs12a` at the command prompt.

```
% ls
(listing of files in home directory appears here)
% mkdir cs12a
% ls
(listing now includes cs12a)
% cd cs12a
```

You can see where you "are" now by typing `pwd` which is short for Print Working Directory. This is what I get.

```
% pwd
/afs/cats.ucsc.edu/users/s/mcdowell/12a
%
```

Remember that `%` here represents the Unix command prompt and you do not type it. You can learn about any Unix command by typing `man` at the command prompt. Try:

```
% man mkdir
% man ls
% man cd
% man man
```

Man pages are notorious for being cryptic and even impenetrable, especially for beginners. Typically, they assume a great deal of background knowledge. Nevertheless, you must get used to reading them since they are an invaluable resource. Use the man pages in conjunction with the tutorial to build up your vocabulary of Unix commands. Also try using Google to find Unix commands. For instance, a Google search on the phrase "unix copy" brings up a reference to the `cp` command. Research the following Unix commands, either through the tutorial, or man pages, or Google: `man`, `ls`, `pwd`, `cd`, `mkdir`, `more`, `less`, `cp`, `cat`, `rm`, `rmdir`, `mv`, `echo`, `date`, `time`, `alias`, `history`. You can also try just typing the command and see what happens. Create a subdirectory of `cs12a` called `lab1` and `cd` into it, then type `pwd` to confirm your location.

```
% mkdir lab2
% cd lab2
% pwd
```

The output of the last command should look something like

```
/afs/cats.ucsc.edu/users/f/cruzid/cs12a/lab2
```

where `cruzid` is your CruzID and the letter `f` may be different for you. This is the full path name of your current working directory. See http://docstore.mik.ua/orelly/unix2.1/lrnunix/ch03_01.htm for more on the Unix directory structure.

Editors

Using either the Vi or Emacs text editor create a file in your `lab2` directory called `HelloWorld.java` containing the following lines.

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

This is a Java *source file*. Close the editor and type `more HelloWorld.java` at the command prompt to view the contents of the file.

Java

In order to run the program, we must first compile it. A *compiler* is a program that translates *source* code into *executable code*, which is what the computer understands. To compile the above program type

```
% javac HelloWorld.java
```

You should see the unix prompt (`%`) disappear for a few seconds while it works, then reappear. List the contents of `lab2` again to see the new file `HelloWorld.class`. This is a Java *executable file*. You can now run the program by typing

```
% java HelloWorld
```

This command should cause the words

```
Hello, world!
```

to be printed to the screen, followed by a new command prompt on the command line.

Open up your editor and change the body of the program so that it prints out your name:

```
Hello, my name is Foo Bar.
```

where `Foo` is your first name and `Bar` is your last name. Compile the new program and run it. If it does not compile, i.e. if you get error messages when you run `javac`, look for some stray character that you might have inserted into the file inadvertently, or perhaps a required character you failed to type.

Transferring (upload/download) files

You will sometimes need to move files between the file system on your local computer and the remote server. If you are using Bitvise on a Windows machine then you can simply drag and drop files between the two panes of the sftp window that opened up when you logged in. There are similar programs for a Macs but you can also copy files using the command line program scp. To move a file from the unix server to your local machine, first use cd to get into the directory where you want the file to end up, then type:

%scp cruzid@unix.ucsc.edu:pathToFile .

Don't forget the last "." (dot). That says put the file in the current directory. For example to move the HelloWorld.java program created above in my lab2 folder to the current local directory I would type:

%scp mcdowell@unix.ucsc.edu:12a/lab2/HelloWorld.java .

File Redirection

Unix programs (invoked via a command line) generally make use of "standard input" (the keyboard by default) and "standard output" (the console window by default). It is possible to "redirect" the output into a file or "pipe" the output of one program into the input of another. It is also possible to "redirect" the program to use a file as "standard input" instead of the keyboard. Some examples should make it clear how this is done. Remember the "%" is representing the unix command prompt and is not something you type. Typing

```
%cat in.txt
```

will print the contents of the file in.txt to the standard output. In this case that is the console (screen). You can instead send the output into another file like this

```
%cat in.txt > out.txt
```

Now in.txt and out.txt will contain the same data. This is not the recommended way to copy a file but it works for text files. Here is another example. Typing

```
%ls
```

prints the directory listing. Typing

```
%ls > myfiles
```

creates a file `myfiles` and puts the listing in the file. The ">" symbol redirects standard output into the file that follows the ">". Redirecting standard input is the same but uses "<". The program "spell" reads standard input and echoes any words that are misspelled. Here is a brief interactive session with spell.

```
%spell this is a tst
type control-d when done entring data
^D
entring tst
%
```

We could also spell check an entire file with:

```
spell < somefile
```

You can of course combine both in a single command so

```
spell < file2check > errors
```

will put the misspelled words from `file2check` into errors. If you wanted to know how many words were misspelled you could then run `wc` on the errors file.

```
wc errors
```

But wc can also read from standard input so the above could also be written

```
wc < errors
```

This then sets us up for the use of a Unix "pipe" indicated with the symbol "|". The "pipe" connects the standard output of one program to the standard input of another. Using "|" we could count the number of misspelled words in `file2check` with a single command.

```
spell < file2check | wc
```

The redirects `file2check` to be the standard input for spell, and then pipes the standard output of spell into the standard input of `wc`.

File redirection with your Java programs.

Of course file redirection works just fine with Java programs. This allows you to, for example, type up some sample input once for your program, then run the program repeatedly with the same input without having to retype it. It also allows you to capture the output in a file. One important note is that it is important that if you are using a Scanner that you create only one Scanner, otherwise you can get some unexpected behavior because the Scanner will buffer ahead all of the input from a redirected input file. The result being that the second Scanner, when it is created, never sees any input. See section 6.12 in the text for an example demonstrating the problem. It doesn't generally come up until you begin creating your own methods (chapter 6).

Use vi or emacs to enter this program on the unix server, or download it from the Files folder in Canvas.

```
/*
  Print the number of words found on each line of input.
  For now, a word is simply any group of characters seperated by
  a space, tab or newline.
 */
import java.util.*;

class ScannerTest {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        while (in.hasNextLine()) {
            // read one line and count the words on the line
            int count = 0;
            String line = in.nextLine();
            // Scanners can be used with Strings
            Scanner scanLine = new Scanner(line);
            while (scanLine.hasNext()) {
                scanLine.next(); // ignore the actual word read - but need to read it
                count++;
            }
            System.out.println(count);
        }
    }
}
```

Compile and run the program, typing some sample input at the command line. You indicate you are done typing by typing control-D (that's hold down the control key while typing the d key). Try using file redirection to run the program on itself:

```
%java ScannerTest < ScannerTest.java
```

Try it once more but saving the output to a file:

```
%java ScannerTest < ScannerTest.java > out.txt
```

What to turn in

For this lab you will submit a text file generated by the script program on unix that demonstrates you have completed the steps above. To do this, do the following:

1. log into unix if not already logged in
2. cd into 12a/lab2 where you created HelloWorld.java
3. type: `script lab2.txt` which will create a log of everything you do in `lab2.txt` until you type exit at the command line.
4. Type the following commands:
   ```
   pwd
   ls
   javac HelloWorld.java
   java HelloWorld
   wc out.txt
   exit
   ```
5. To confirm it created the script, type
   ```
   more lab2.txt
   ```
   and you should see the log of the brief session from pwd through the running of wc (word count) on the out.txt file created from running ScannerTest (the last step in the redirection section).
6. Transfer the file `lab2.txt` to your local machine then submit it under lab2 in Canvas.