CSC 207 Assignment1

Due Date: 13th October @ 11:59pm

Introduction:

This assignment has you **design** a set of classes for maintaining a *mock* file system and interacting with it in a program that operates like a Unix shell. **You must work with a partner. Submitting this alone is not an option; if you do, we will give you a mark of 0.**

IMPORTANT:

1) Your Assignment1 SVN repo is of the following form:

https://142.1.44.22/svn/csc207h/groupzzz

NOTE: The three digit groupNumber i.e. zzz can be found by checking your email. I had emailed this out on 4th October evening.

2) This Assignment consists of two tasks. i.e. **Task1** and **Task2**.

3) You are **NOT REQUIRED** to implement any commands in Assignment1 (except for the `exit` command). The implementation of these commands happens in Assignment2. YOU WILL BE SUBMITTING ONLY 1 JAVA FILE (I.E. JShell.java).

Remember, this assignment is out of 10 marks. The break up of 10 marks is:

8 of these marks are allocated towards **Task1** and

2 of these marks are allocated towards **Task2**.

Hence spend more time towards *Task1* in order to get a good mark for this assignment. The code that is expected from you is very minimal and clearly outlined in *Task2* of this assignment. Again you are not asked to implement any of the commands except for the exit command.

What's a shell?

Before the GUI, there was the command line. All interactions with the file system were done using commands, much like what we have been seeing in lectures (*Lecture1*, your instructor using the *svn client* via the terminal) and in your *future labs*. Rather than double-clicking on windows containing pictures of the contents of a hard drive, people would type commands:

- ***mv*** <u>*oldfile*</u> <u>*newfile*</u> to rename a file or move it to a different directory
- ***cd*** to change directories
- ***cat*** <u>*filename*</u> to display the contents of a file called *filename*.

A ***command*** consists of the command name followed (optionally) by a space and then <u>arguments</u> for the command.

A shell is just another program. Most are written in the programming language C. Shells interact with the operating system.

Java's virtual machine is like an operating system. You'll be writing a shell program for it, and you'll be writing the complete functional code (only in `Assignment2` and not `Assignment1`) to manage files and directories. These will all be Java objects; you do not need to save them to disk. <u>The expected code from you for `Assignment1` is outlined in *Task2* of this assignment.</u>

When a user runs your program, your program will start a shell that allows the user to interact with an initially empty file system. As long as your shell is running, your program will keep track of the files the user has created, allowing them to move around the directory structure they've built and allowing them to display the contents of any files they have created.

Your program needs to support the equivalents of these bash commands (*italics* indicate the names of files and directories or something else that the user chooses). Everything is case sensitive.

`exit`

Quit the program.

`mkdir` ***`DIR`***

Create a directory DIR, which may be relative to the current directory or may be a full path.

`cd` ***`DIR`***

Change directory to DIR, which may be relative to the current directory or may be a full path. As with Unix, **`..`** means a parent directory and **`.`** means the current directory. The directory separator must be /, the forward slash. The root of the file system is a single slash: **/**.

`ls`

Print the names of files and directories in the current directory, with a new line following each of them.

`pwd`

Print the current working directory (including the whole path).

`mv` ***`OLDFILE NEWFILE`***

Move file OLDFILE to NEWFILE. Both OLDFILE and NEWFILE may be relative to the current directory or may be full paths.

`cp` ***`OLDFILE NEWFILE`***

Like mv, but don't remove OLDFILE.

`cat` ***`FILE`***

Display the contents of FILE in the shell.

`get` ***`URL`***

URL is a web address. This command retrieves the file at that URL and adds it to the current directory.

echo *STRING* > *OUTFILE*

Put STRING in file OUTFILE. STRING is a string of characters surrounded by quotation marks. This creates a new file if OUTFILE does not exist, and erases the old contents if OUTFILE already exists. In either case, the only thing in OUTFILE should be STRING.

echo *STRING* >> *OUTFILE*

Like the previous command, but appends instead of overwrites.

Task1: [8 marks]

Create a set of **CRC cards** that could be used to implement the shell described. Instead of handing in your index cards, there is a directory named ***crcCards*** in your subversion repository. In ***crcCards***, for each card that create, create a text file (with a .txt suffix) with the name of the class. Each CRC Card (or each .txt file) contains the following format:

```
Class name: Classname
Parent class (if any): Classname
Subclasses (if any): List all the subclasses
separated by a comma.

Responsibilities:
*g
*h
*i
Collaborators:
*j
*k
*l
```

Do a thorough job on the CRC analysis. There may be hidden responsibilities that are not listed, but are necessary in order to write the program.

Make sure your responsibilities are clearly written: the grader (one of the TAs) should easily be able to understand each of them.

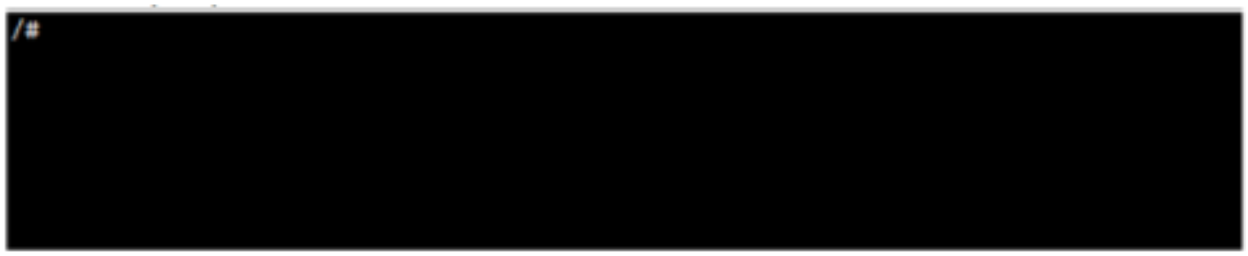
Task2: [2 marks]

Before you begin Task2, I encourage you to watch this video of mine to understand and know what are minimum expectations of this task.

http://youtu.be/Mk5p4X_oho8

Write the start of the shell program. Call your main class ***JShell***. You should not submit any other java files other than `JShell`. Your shell program shall:

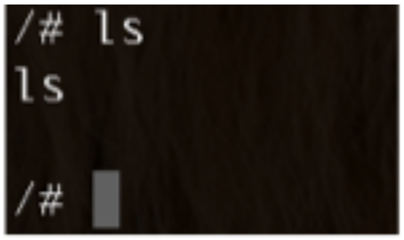
- Print a prompt: the full path for the current working directory (in Assignment1, your current working directory will always be the root or /), followed by the # symbol, followed by a space.



```
/#
```

Figure 1: When your `JShell` is first executed, it must show the above prompt to the user and wait for user input. I am running this example from command line hence have a black background. If you happen to run this via Eclipse, you may have a white (or some other color depending on your Eclipse setting) background.

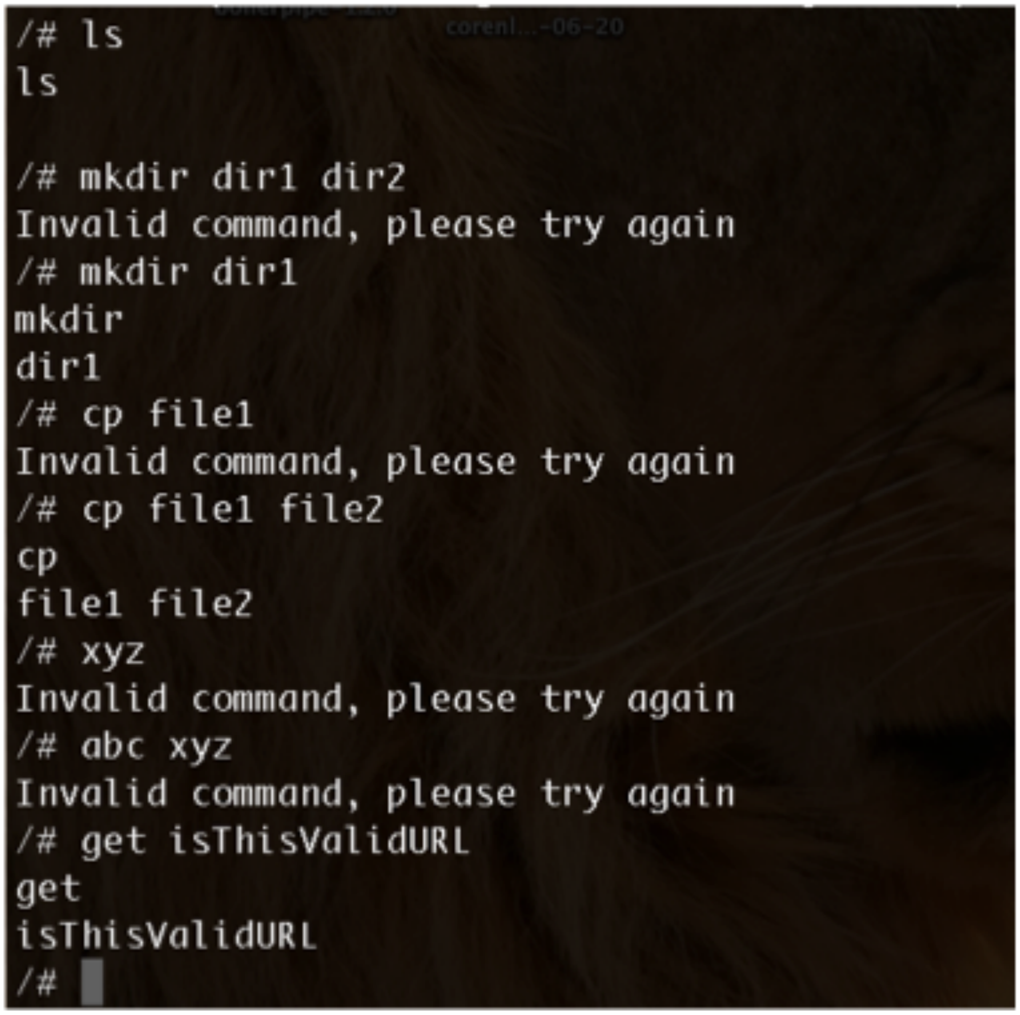
- Repeatedly read a command that the user types and print the following information on two separate lines:
    - The command name (for example, `ls`)
    - The rest of the line (if it contains any arguments), which may be empty.



```
/# ls
ls

/# 
```

Figure2: For example, if the user types *ls*, then the first line of output would contain *ls* and the second line would be empty; your shell will then print another prompt waiting for the user input.

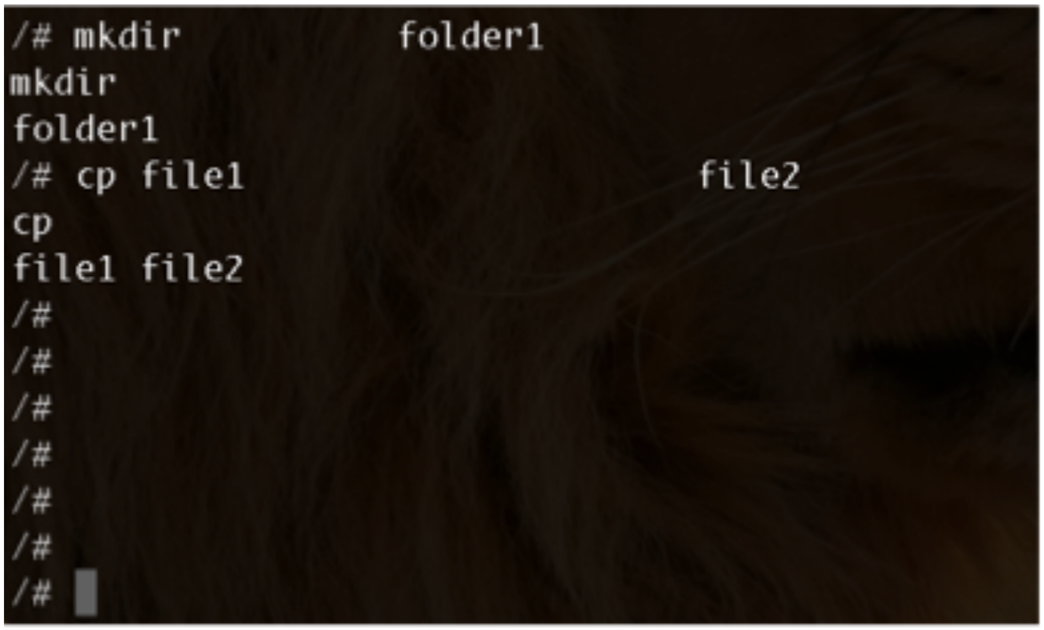


```
/# ls
ls

/# mkdir dir1 dir2
Invalid command, please try again
/# mkdir dir1
mkdir
dir1
/# cp file1
Invalid command, please try again
/# cp file1 file2
cp
file1 file2
/# xyz
Invalid command, please try again
/# abc xyz
Invalid command, please try again
/# get isThisValidURL
get
isThisValidURL
/# 
```

Figure 3: Few more examples that the user may type on your `JShell` and the expected output of your Assignment1. In the above example, ***cp file1*** is considered as invalid command because ***cp*** expects two arguments. Likewise ***mkdir dir1 dir2*** is considered as invalid command in `JShell` because it expects one argument.

There may be one or more space between a command and its arguments. You should ignore all the extra spaces after the end of the command. If the user types only spaces and press enter, your `JShell` should print the prompt again.



```
/# mkdir          folder1
mkdir
folder1
/# cp file1                       file2
cp
file1 file2
/#
/#
/#
/#
/#
/#
/# 
```

Figure 4: Few more examples dealing with white spaces.

Note:

1) YOUR PROGRAM SHOULD NOT CRASH. IF THE USER ENTERS INVALID SEQUENCE OF CHARACTERS, PRINT THE FOLLOWING ERROR MESSAGE:

```
Invalid command, please try again
```

2) You do not need to make any of the commands work; this is only to get you started writing the shell before `Assignment2` begins. Your main focus for this `Assignment1` is towards the CRC cards.

Marking:

Please follow the grading scheme for `Assignment1` on Blackboard.

Checklist:

Have you...

. 1) Committed your files?

. 2) Made sure all your changes were committed by checking out a new copy of your repository?

. 3) Thanked your assignment partner for their effort?

. 4) Have your team of 4 ready for `Assignment2`?