GROUP 0301 REPORT

// Student1: Abhinav Chaudhary
// UTORID username: chaud 349
// UT Student #: 1002707733
// Author: Abhinav Chaudhary

// Student2: Alexandru Andros
// UTORID username: androsal
// UT Student #: 1004354263
// Author: Alexandru Andros

// Student3: Balaji Babu
// UTORID username: babubala
// UT Student #: 1003354871
// Author: Balaji Babu

// Student4: Zhi Zhong Huang
// UTORID username: huang472
// UT Student #:1002671094
// Author: Zhi Zhong Huang

-Design Patterns-

During the course of assignment 2A and 2B we have used: Singleton, Reflection, Factory methods, Abstract classes, Dependency Injection. We used the singleton design pattern so that we have a common filesystem that the commands affect in addition to having multiple windows/instances of the program open and they all interact with one file-system.This could be see in ***line 68 Jshell.java*** . Reflection was used to find the specific command, given the path depending on src. There are dictionaries created that were made that map the name of the command to its class. Then the forName method is then used to find a constructor for the class, and then is executed. This can be seen in **Command Manager RunCommand private method**. The abstract class was used as a form of a blueprint, for every other command. This way the reflection was able to work, since every possible constructor was defined in the abstract class. We used Dependency Injection in utilizing mock objects to make our testing much easier. For example we created a file called MockJS which would implement a mock filesystem as many commands are dependant on the file-system. Now with Mock JShell we can utilize it to test without affecting the main file-system. We created other mock classes for which we used a dependency injection model, they include MockMan and MockParse. We also used factory methods, specifically in the Directory class to be able to circumvent the need for the Directory class constructor to call it's superclass' constructor initially, which didn't allow for exception throwing ***(Line 92 Directory.java)****.* Adding a factory method for the Directory class allowed for us to better anticipate the possible ways the user might attempt to create a directory and to place restrictions on the user based on the stipulations we were given in the assignment.

-Design transition from 2A to 2B-

We changed echo to be more modular, by adding helper methods which would help with redirection. We added a class location and generics to Commands to make testing easier. Instead of using static variables in J-Shell we created a class Location to handle these static variables, this made testing easier as we could create a mock location. So in situations while testing, where we needed directory information such as the current directory, it was easier to obtain through mock location utilization. Our design is better now and can accommodate new customer requirements as adding new commands is easy because we have used abstract classes that hold key methods etc that every command share, and so there is minimal work in adding a new command for example. If we were to add a new command, we would simple create a new class for it, implement its functionality and that's it. Our driver and main back-end file-system does not need modification**.** The use of generics in our assignment allowed us to be flexible with the implementation of abstract methods in our Commands superclass ***(Line 37 Commands.java)***. As every command which inherits the Commands superclass either returns some string output (i.e. tree or ls) or doesn't return any output (i.e. mkdir and cp) we can use generics to differentiate between the two different return types (String and Void respectively) while also preserving the similar behaviour that all commands have, which is the ability to be executed.