Creating a Shell Interface Using Java

This project consists of modifying a Java program so that it serves as a shell interface that accepts user commands and then **executes each command in a separate process external to the Java virtual machine.** You are to implement a simple shell (command interpreter) that behaves similarly to the UNIX shell. When you type in a command (in response to its prompt), it will create a process that will execute the command you entered.

The basic lifetime of a shell

A shell does three main things in its lifetime.

Initialize: In this step, a typical shell would read and execute its configuration files. These change aspects of the shell's behavior.

Interpret: Next, the shell reads commands from stdin (which could be interactive, or a file) and executes them.

Terminate: After its commands are executed, the shell executes any shutdown commands, frees up any memory, and terminates.

Basic loop of a shell

So we've taken care of how the program should startup. Now, for the basic program logic: what does the shell do during its loop? Well, a simple way to handle commands is with three steps:

Read: Read the command from standard input.

Parse: Separate the command string into a program and arguments.

Execute: Run the parsed command.

Example

A shell interface provides the user with a prompt, after which the user enters the next command. The example below illustrates the prompt mysh> and the user's next command: *cat Prog.java*. This command displays the file *Prog.java* on the terminal using the UNIX cat command.

```
mysh> cat Prog.java
```

Perhaps the easiest technique for implementing a shell interface is to have the program first read what the user enters on the command line (here, *cat Prog.java*) and then **create a separate external process that performs the command**. We create a separate process using the ***ProcessBuilder()*** object. In our example, this separate process is external to the JVM and begins execution when its ***run()*** method is invoked.

A Java program that provides the basic operations of a command-line shell is supplied in ***code1*** below. The ***main()*** method presents the prompt **mysh>** (for java shell) and waits to read input from the user.

This project is organized into the following parts:

1. Open Eclipse, create a new project and new java file as MyShell.java
2. provides the user with a prompt **mysh>**
3. If they entered a return, just loop again
4. Parse the input to obtain the command and any parameters
5. Unlike the real shell, your program doesn't have to know how to run arbitrary programs, only the following commands.
   Dir/ls, pwd, cd, history, help, exit
6. If other commands or words entered, sent a message to the user and loop again.
7. creating the external process and executing the command in that process

Task1: Creating an External Process

The first part of this project is to modify the *main()* method in *code1* so that an external process is created and executes the command specified by the user. Initially, the command must be parsed into separate parameters and passed to the constructor for the *ProcessBuilder* object. For example, if the user enters the command

mysh> cat Prog.java

the parameters are (1) *cat* and (2) *Prog.java*, and these parameters must be passed to the *ProcessBuilder* constructor. Perhaps the easiest strategy for doing this is to use the constructor with the following signature:

public ProcessBuilder (List<String> command)

An ArrayList can be used in this instance, where the first element of the list is **cat** and the second element is **Prog.java**. This is an especially useful strategy because the number of arguments passed to UNIX commands may vary (the cat command accepts one argument, the cp command accepts two, and so forth).

For the Windows system, add two elements to your list. So your first element is *cmd*, and the second element is */c*, the next one is what you get from the command line.

```
input.add("cmd");
input.add("/c");
```

If the user enters an invalid command, the *start()* method in the *ProcessBuilder* class throws a java.io.IOException. If this occurs, your program should output an appropriate error message and resume waiting for further commands from the user.

```
import java.io.*;

public class SimpleShell {
        public static void main(String[] args) throws java.io.IOException {
                String commandLine;
                BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
                while (true) {
                        // read what the user entered
                        System.out.print("mysh>");
                        commandLine = console.readLine();
                        // if the user entered a return, just loop again
                        if (commandLine.equals(""))
                                continue;
                        /**
                         * The steps are:
                                (1) parse the input to obtain the command and any parameters
                                (2) create a ProcessBuilder object
                                (3) start the process
                                (4) obtain the output stream
                                (5) output the contents returned by the command
                         */
                }
        }
}
```

Code1: Outline of the simple shell.

Task 2: Changing Directories

The next task is to modify the program in task 1 so that it changes directories. In systems, we encounter the concept of the *current working directory*, which is simply the directory you are currently in. The ***cd*** command allows a user to change current directories. Your shell interface must support this command.

For example, if the current directory is /user/test and the user enters cd music, the current directory becomes /user/test/music. **Subsequent commands relate to this current directory.** For example, entering ls will output all the files in /user/test/music.

The ProcessBuilder class provides the following method for changing the working directory:

public ProcessBuilder directory(File directory)

When the start() method of a subsequent process is invoked, the new process will use this as the current working directory. For example, if one process with a current working directory of /user/test invokes the command cd music, subsequent processes must set their working directories to /user/test/music before beginning execution.

It is important to note that your program must first make sure the new path being specified is a valid directory. If not, your program should output an appropriate error message.

If the user enters the command **cd**, change the current working directory to the user's home directory. The home directory for the current user can be obtained by invoking the static getProperty() method in the System class as follows:

System.getProperty("user.dir");

Task3: Adding a History Feature

Many shells provide a history feature that allows users to see the history of commands they have entered and to rerun a command from that history. The history includes all commands that have been entered by the user since the shell was invoked. For example, if the user entered the ***history*** command and saw as output:

```
0 dir
1 cd test
2 dir
```

the ***history*** would list *dir* as the first command entered, *cd test* as the second command, and so on. Modify your shell program so that commands are entered into history. Your program must allow users to rerun commands from their history by supporting the following two techniques:

1. When the user enters the command ***history***, you will print out the contents of the history of commands that have been entered into the shell, along with the command numbers.

2. When the user enters **!!**, run the previous command in history. If there is no previous command, output an appropriate error message.