Shell Interface Assignment

Creating a Shell Interface Using Java

This assignment 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.

Overview

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

```
jsh> 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 the separate process using the `ProcessBuilder()` object, as illustrated in the section labeled **Process Creation in Java** found later in this assignment description. 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 the code of Figure 1 on the next page. The `main()` method presents the prompt `jsh>` (for java shell) and waits to read input from the user. The program is terminated when the user enters `<Control><C>`.

This assignment is organized into three parts: (1) creating the external process and executing the command in that process, (2) modifying the shell to allow changing directories, and (3) adding a history feature.

```
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));

    // we break out with <control><C>
    while (true)
    {
      // read what the user entered
      System.out.print("jsh>");
      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 */
    }
  }
}
```

Figure 1 – Outline of simple shell.

Part 1: Creating an External Process

The first part of this assignment is to modify the `main()` method in Figure 1 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

```
jsh> 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)
```

A `java.util.ArrayList`, which implements the `java.util.List` interface, 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.)

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

Part 2: Changing Directories

The next task is to modify the program in Figure 1 so that it changes directories. In UNIX 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 `/usr/tom` and the user enters `cd music`, the current directory becomes `/usr/tom/music`. Subsequent commands relate to this current directory. For example, entering `ls` will output all the files in `/usr/tom/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 `/usr/tom` invokes the command `cd music`, subsequent processes must set their working directories to `/usr/tom/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");
```

Part 3: Adding a History Feature

Many UNIX 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 pwd
1 ls -l
2 cat Prog.java
```

the history would list `pwd` as the first command entered, `ls -l` as the second command, and so on.

Modify your shell program so that commands are entered into a history. (Hint: The `java.util.ArrayList` provides a useful data structure for storing these commands.)

Your program must allow users to rerun commands from their history by supporting the following three 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 the history. If there is no previous command, output an appropriate error message.
3. When the user enters `!<integer value` *`i`*`>`, run the *i*th command in the history. For example, entering `!4` would run the fourth command in the command history. Make sure you perform proper error checking to ensure that the integer value is a valid number in the command history.

Additional Design Requirements

1. The driver class is the class **SimpleShell** as shown in Figure 1 and it should only contain only one method, main.

2. This assignment used UNIX commands in its description. You might assume your program would only be able to execute in Linux, UNIX, or MacOS. But, your program can also run in Microsoft Windows. If you test your program in Linux, UNIX, or MacOS then type UNIX commands at the java shell (`jsh>`) prompt. If you test your program in Microsoft Windows then type DOS prompt commands at the java shell (`jsh>`) prompt.

   Regrettably, for the external process to execute DOS prompt commands, if your program executes within Microsoft Windows, then your program always has to make the first two elements of the list passed to the ProcessBuilder constructor the following strings: `"cmd.exe"` and `"/c"`. For example, if the user enters the command

   ```
   jsh> type Prog.java
   ```

   the parameters are the strings `"type"` and `"Prog.java"`. The list object passed to the ProcessBuilder constructor has its first element the string `"cmd.exe"`, its second element the string `"/c"`, its third element the string `"type"`, its fourth element the string `"Prog.java"`.

   Luckily, the above situation is not necessary if your program runs within Linux, UNIX, or MacOS. But, you must configure your program to work whether it executes within Linux, UNIX, MacOS or Microsoft Windows. Therefore, your program has to detect the operating system it is running within. Actually, it is easy to determine through the use of the following line of code:

   ```
   String OSName = System.getProperty("os.name");
   ```

   If the string OSName contains “Windows” then your program is running within Microsoft Windows; otherwise, it is running within Linux, UNIX, or MacOS.

3. **Tip:** Make your program as modular as possible, not placing all your code in one .java file. You can create as many classes as you need in addition to the class described above. Methods should be reasonably small following the guidance that "A function should do one thing, and do it well." You will lose a lot of points for code readability if you don’t make your program as modular as possible. But, do not go overboard on creating classes and methods. Your common sense should guide your creation of classes and methods.

4. Do **NOT** use your own packages in your program. If you see the keyword **package** on the top line of any of your .java files then you created a package. Create every .java file in the **src** folder of your Eclipse project

5. Do **NOT** use any graphical user interface code in your program!

Process Creation in Java

When a Java program begins execution, an instance of the Java virtual machine is created. On most systems, the JVM appears as an ordinary application running as a separate process on the host operating system. Each instance of the JVM provides support for multiple threads of control; but Java does not support a process model, which would allow the JVM to create several processes within the same virtual machine. Although there is considerable ongoing research in this area, the primary reason why Java currently does no support a process model is that it is difficult to isolate one process's memory from that of another within the same virtual machine.

It is possible to create a process external to the JVM , however, by using the `ProcessBuilder` class, which allows a Java program to specify a process that is native to the operating system (such as `/usr/bin/ls` or `C:\\WINDOWS\\system32\\mspaint.exe`). This is illustrated in Figure 2 on the next page. Running this program involves passing the name of the program that is to run as an external process on the command line.

We create the new process by invoking the `start()` method of the `ProcessBuilder` class, which returns an instance of a `Process` object. This process will run external to the virtual machine and cannot affect the virtual machine and vice versa. Communication between the virtual machine and the external process occurs through the `InputStream` and `OutputStream` of the external process.

```
import java.io.*;

public class OSProcess
{
  public static void main(String[] args) throws IOException
  {
    if (args.length != 1)
    {
      System.err.println("Usage: java OSProcess <command>");
      System.exit(0);
    }

    // args[0] is the command that is run in a separate process
    ProcessBuilder pb = new ProcessBuilder(args[0]);
    Process process = pb.start();

    // obtain the input stream
    InputStream is = process.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);

    // read the output of the process
    String line;
    while ( (line = br.readLine()) != null)
      System.out.println(line);

    br.close();
  }
}
```

Figure 2 – Creating an external process using the Java API.

Grading Criteria

The total assignment is worth 20 points, broken down as follows:

1. If your code does not implement the task described in this assignment then the grade for the assignment is zero.
2. If your program does not compile successfully then the grade for the assignment is zero.
3. If your program produces runtime errors which prevents the grader from determining if your code works properly then the grade for the assignment is zero.

If the program compiles successfully and executes without significant runtime errors then the grade computes as follows:
Followed proper submission instructions, 4 points:

1. Was the file submitted a zip file.
2. The zip file has the correct filename.
3. The contents of the zip file are in the correct format.
4. The keyword **package** should not appear at the top of any of the .java files.

Program execution, 4 points:

- Program input, the program properly reads, processes, and uses the input.
- Program output, the program produces the correct results for the input.

Code implementation, 8 points:

- The driver file has the correct filename, **SimpleShell.java** and contains only the method **main** performing the exact tasks as described in the assignment description.
- The code performs all the tasks as described in the assignment description.
- The code is free from logical errors.

Code readability, 4 points:

- Good variable, method, and class names.
- Variables, classes, and methods that have a single small purpose.
- Consistent indentation and formatting style.
- Reduction of the nesting level in code.

Late submission penalty: assignments submitted after the due date are subjected to a 2 point deduction for each day late.

Submission Instructions

Go to the folder containing the .java files of your assignment and select all (and **ONLY**) the .java files which you created for the assignment in order to place them in a Zip file. The file should **NOT** be a **7z** or **rar** file! Then, follow the directions below for creating a zip file depending on the operating system running on the computer containing your assignment's .java files.

Creating a Zip file in Microsoft Windows (any version):

1. Right-click any of the selected .java files to display a pop-up menu.
2. Click on **Send to**.
3. Click on **Compressed (zipped) Folder**.
4. Rename your Zip file as described below.
5. Follow the directions below to submit your assignment.

Creating a Zip file in Mac OS X:

1. Click **File** on the menu bar.
2. Click on **Compress ? Items** where ? is the number of .java files you selected.
3. Mac OS X creates the file **Archive.zip**.
4. Rename **Archive** as described below.
5. Follow the directions below to submit your assignment.

Save the Zip file with the filename having the following format:
your last name,
followed by an underscore _,
followed by your first name,
followed by an underscore _,
followed by the word **Assignment1**.
For example, if your name is John Doe then the filename would be: **Doe_John_Assignment1**

Once you submit your assignment you will not be able to resubmit it!
Make absolutely sure the assignment you want to submit is the assignment you want graded.
There will be **NO** exceptions to this rule!

You will submit your Zip file via your CUNY Blackboard account.
Follow these instructions:

Log onto your CUNY BlackBoard account.
Click on the CSCI 340 course link in the list of courses you're taking this semester.
Click on **Content** in the green area on the left side of the webpage.
You will see the **Assignment 1 – Shell Interface Assignment**.
Click on the assignment.
Upload your Zip file and then click the submit button to submit your assignment.

Due Date: Submit this assignment by Thursday, October 25, 2018.