Pipelines Communication with Other Java Processes

Purpose:

To gain familiarity with the instructor supplied Java pipeline class. This class is a necessary part of the CPSC 101 team term project for 2017.

Due Date:

This assignment is due Friday, 2017-02-24 *at the beginning of class*.

Background Reading Assignment:

⇒ Find an online description of UNIX pipes, such as

- `http://www.december.com/unix/tutor/pipesfilters.html`, *or*
- `http://en.wikipedia.org/wiki/Pipeline_%28Unix%29`

What does "`ls | wc`" at the command-line prompt accomplish?

The `JavaPipe` class

JAVA has the ability to execute separate processes and communicate with them via pipes. In order to simplify this process for the 2017 team term project, Dr. Casperson has created a `JavaPipe` class for student use.

This class is custom designed for communicating with another subservient JAVA program. For instance, suppose that you have created a program `Reverse.class` that reads strings from `System.in` and writes the reversed strings to `System.out`:

```
1  import java.util.Scanner ;
2  import ca.unbc.cpsc101.JavaPipe ;
3  //...
4      JavaPipe myReverser = new JavaPipe("Reverse");
5      myReverser.start() ; // causes the program to start and
6                           // and the readers and writers to be set up.
7      Scanner pipeScanner = new Scanner(myReverser.getInputReader()) ;
8      myReverser.getOutputWriter().println("derF") ;
9      System.out.println(pipeScanner.nextLine()) ;
10     myReverser.destroy() ;
```

Figure 1: A program using JavaPipe

Assuming that all of the defaults are set correctly, you can set up a pipeline to this program as shown in Figure **??**. Figure **??** shows schematically how the two programs are connected together.

Here is a more detailed explaination of the program in Figure **??**.

The import on line 2 shows that the `JavaPipe` class is found in package `ca.unbc.cpsc101`. The constructor on line 4 creates a `JavaPipe` object, but it does *not* start the process.

The `myReverser.start()` command on line 5 starts up a second Java process that runs the main method in the `Reverse.class`. However, this second Java process does not have `System.in` and `System.out` connected to the keyboard and display. Instead they are connected by pipes to `myReverser.getOutputWriter()` and `myReverser.getInputReader()` (see Figure **??**).

Thus when, on line 8, the main Java program writes ""derF"" to `myReverser.getOutputWriter()`, this appears on `System.in` of the secondary Java process.

Assuming that the `Reverse` program then writes `"Fred"` to its `System.out` this becomes available as input through `myReverser.getInputReader()`, and can be read through a `java.util.Scanner` in the standard way (lines 7 and 9).

Downloads

A Jar file containing the class can be found via `http://web.unbc.ca/~casper/Semesters/2017W/101-project.php`, (and possibly here) as can a link to docu-

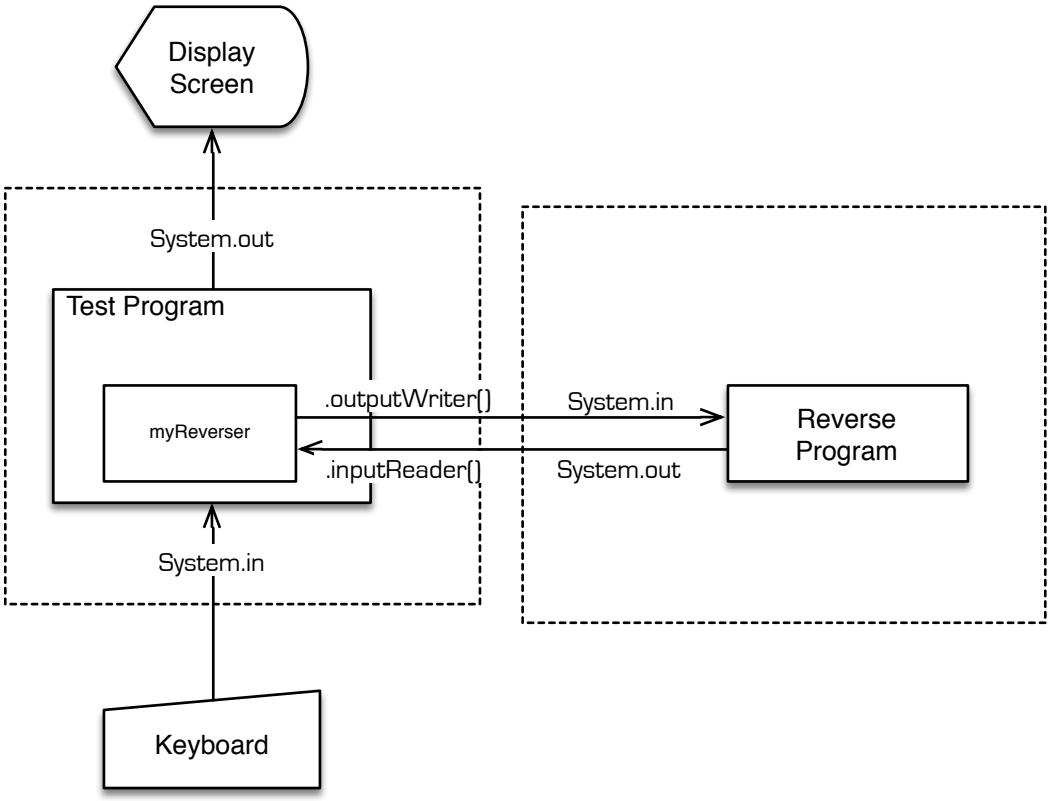

Figure 2: Illustration of a program using `JavaPipe`

mentation for the class.

⇒ Download this file.

Compilation and Runtime Necessities

There are several things that can go wrong related to paths.

The following explanations assumes that you are engaged in command-line compilation using a UNIX-like operating system. If this is not the case *you* are responsible for determining how to get your IDE and/or operating system to coöperate.

Compilation

Firstly, when you *compile* the test program, the compiler (`javac` if you doing things the old-fashioned way) needs to be able to see the `ca.unbc.cpsc101.JavaPipe` class. This class is in the `.jar` file that you downloaded above, so you need to ensure that the `.jar`-file is on the appropriate class path. Using `javac` you can do this in multiple ways:

```
$ javac -classpath :::ca.unbc.cpsc101.JavaPipe.jar TestProgram.java
```

```
$ javac -cp ::ca.unbc.cpsc101.JavaPipe.jar TestProgram.java
```

```
$ # when running /bin/bash as your shell.
$ CLASSPATH="::ca.unbc.cpsc101.JavaPipe.jar"
$ bash-3.2$ export CLASSPATH
$ bash-3.2$ javac TestProgram.java
```

By the way, the actual name of the `.jar` file is not magic. You can rename `ca.unbc.cpsc101.JavaPipe.jar` to `x.jar` and use "`javac -cp ::x.jar TestProgram.java`" if you wish.

Running the Test Program

Secondly, when you *run* the test program, the JAVA runtime system (`java` if you doing things the old-fashioned way) needs to be able to see the `ca.unbc.cpsc101.JavaPipe` class. There are multiple similar ways to do this using `java`:

```
$ java -classpath :::ca.unbc.cpsc101.JavaPipe.jar TestProgram
```

```
$ java -cp ::ca.unbc.cpsc101.JavaPipe.jar TestProgram
```

```
$ # when running /bin/bash as your shell.
$ CLASSPATH="::ca.unbc.cpsc101.JavaPipe.jar"
$ bash-3.2$ export CLASSPATH
$ bash-3.2$ java TestProgram
```

If the class `TestProgram` is contained in a package, things are slightly more complicated, but the above principles apply.

Finding the Reverse Program

Remember that `TestProgram` itself is running `java`. In the simple case where `Reverse.class` is not packaged, and is in the same directory as `TestProgram.class`, the command-line command to run `Reverse` is just `java Reverse`.

However, if you need to set the class path for the `java` command run on your behalf by the `JavaPipe` class, you need to use the `JavaPipe.setClassPath` method. There is currently no provision for passing command-line arguments to the JAVA-program run by `JavaPipe`[1]

Tests

⇒ Write a simple `Reverse` class and test it directly from the command line. Then test it indirectly through a `JavaPipe` somewhat as shown above.

[1] Dr Casperson hopes to change this soon.

Caution The timing of the steps is critical. For instance, if one attempts to read from `.getInputReader()` while the process at the other end of the pipe is also waiting for input, the program will hang with both processes waiting for the other to provide input.

⇒ Write a `Responder` class that reads commands as per the Score 4 specification and sends back syntactically correct responses. Test it from the command line. Then test it indirectly through a `JavaPipe`.