6.005 Elements of Software Construction
Fall 2008

Lab 3: Networking

6.005 Elements of Software Construction
Fall 2008

In this lab, you will get a blazing-fast, learn-by-doing introduction to aspects of Java's I/O, networking, and threading API, to help you get started building network applications. We will incrementally build a simple chat server application that you can use as a starting point for implementing your instant-messaging system.

Before Lab

Before coming to lab, please do the following:

- Read both Project 3 and this lab handout.
- Check out the `friendly` project from your personal SVN repository.

Friendly

Our starting point will be a small program called Friendly. The program takes input that the user types and prints out canned responses, until the user types `bye', at which point the program terminates. Here is an example run of the program (lines preceded by ">" represent input typed by the user)

```
Hello, my friend! How are you today?
> Just ok.
I see. Why is that?
> Hard day at work.
I see. Why is that?
> Is that all you ever say?
Interesting question.
> Bye.
Well, bye then.
```

 **Run `friendly.Main`.** The program will start running. You can interact with the program in Eclipse's `Console` window. Try out the program by going through an interaction like the one shown above.

Inspect the source code of `friendly.Main`. Notice that to read the input from the console, we use `java.io.BufferedReader`. Buffered readers will come in handy later, and also during your project, so take a moment to see how they are used in Friendly. In particular, note the following usage pattern:

Pattern for repeatedly reading text lines from some source using a BufferedReader.

```
// Create a reader.
BufferedReader reader = new BufferedReader(...);

try {

    // Read first line.
    String line = reader.readLine();

    while (line != null) { // line == null means stream end reached

        // Process the line
        ...

        // Read next line.
        line = reader.readLine();

  } catch (IOException e) {
    // Error while reading. Deal with the error.
}
```

Note that a common mistake when applying this pattern is to forget to put `readLine()` at the end of the loop, which leads to an infinite loop.

Adding more canned responses

This task will augment Friendly's conversational repertoire as follows:

User types...	Friendly responds...
I think it's going to rain today.	Nonsense! Let's go to the beach.

What's happening?	We need to talk about your TPS reports.
I think I'm gonna lose it.	Uh-oh. Sounds like somebody's got a case of the Mondays.

We could hardcode the responses as a bunch of if-then clauses (similarly to the if-clauses that Friendly uses to differentiate questions from non-questions). A better way of doing it is using a map.

 **Read the API for `java.util.Map`.**

 **Add the above canned responses to Friendly.** Use a `java.util.LinkedHashMap` (one of the implementations of `Map` that Sun provides) to store the input/response pairs. Test that the program works as intended. Below is a template that you can use to follow your code.

```
package friendly;
.....
public class Main {

  public static void main(String[] args) {

      Map<String,String> cannedResponses = new LinkedHashMap<String, String>();
      cannedResponses.put(...);
      cannedResponses.put(...);
      cannedResponses.put(...);
      .....

         while (line != null) {
         .....

         // TODO: If a canned response matches line, output response.

         .....

         }
  }
}
```

Sockets: Friendly over the network

You can't imagine living without Friendly, and now you want to share it with your friends. You decide to extend Friendly so that anyone can access it over the network.

The basic logic of the program will stay unchanged. The difference is that instead of reading input from the console (`System.in`), the program will read its input from a network **socket**, and instead of writing its output to the console (`System.out`), it will write to the socket.


Read the page What Is a Socket? in the Java tutorial. You don't need to read beyond the first page, unless you want to.


Take a look at the program `socketexample.Main`. It is a simple program that listens in a specific port of your machine for a connection and echoes back the first line of input it receives.


Run `socketexample.Main`. The program will print the following message to the console, and then wait for an incoming connection.

```
Accepting connections...
```

Without terminating the server, open a command prompt in your operating system. Connect to the server as follows:

```
> telnet localhost 4444
Connected to server. Type a line of input followed by the ENTER key.
> Hello there
You said: Hello there
Closing connection... Connection to host lost.
```


Using `socketexample.Main` as a guide, modify Friendly so that it works over a network connection. You can have Friendly listen on port 4444 on your machine. You should be able to have an interaction with Friendly as before, except this time over a network connection:

```
> telnet localhost 4444
Hello, my friend! How are you today?
> Just ok.
I see. How come?
> Hard day at work.
```

```
I see. How come?
> Is that all you ever say?
Interesting question.
> Bye.
Well, bye then.

Connection to host lost.
```

In real use, your friends would use the hostname or IP address of your machine rather than `localhost` (which refers to your machine). In addition to an IP address, you would need to tell your friends the port which the server uses, e.g. 4444.

 **Run the `FriendlyTest` JUnit test.** Make sure you run your Friendly server first. This test acts as a client, connecting to port 4444, sending some messages to the server, and checking Friendly's replies.

> **Checkpoint.** Find a TA or LA or another member of the course staff, and demonstrate your working Friendly server to get checked off on this part of the lab.

Constantly Friendly

Word has spread about Friendly. You start getting complaints from users that try to use Friendly but are denied a connection. The problem with the current implementation is that it after it services one user, the server terminates. If someone then tries to connect using the port, it gets a message like the following.

```
> telnet localhost 4444
Connecting to localhost... Could not open connection to the
host, on port 4444: Connection failed.
```

 **Modify Friendly so that after a user types `bye', it closes the connection to the user *but* continues accepting new connections, until the server is forcefully terminated.** You'll know you succeeded if you can have two sequential sessions without restarting the server, as shown below. Implementation hint: wrap the server's functionality in a while-loop, but avoid closing the server socket on each iteration.

```
> telnet localhost 4444
```

```
Hello, my friend! How are you today?
> Bye.
Well, bye then.

Connection to host lost.
> telnet localhost 4444
Hello, my friend! How are you today?
```

Note that this new version of Friendly never terminates by itself, and two processes can't listen to 4444 at the same time. So whenever you want to run Friendly, you will have to make sure any running Friendly process is stopped first. In Eclipse, you can stop a running process by clicking on the little red Stop button on the Console view toolbar. You can remove terminated processes from the console window by pressing the 'X' button to the right. If you don't close terminated processes, you might have running processes hiding underneath them.

Threads: Multi-user Friendly

As popularity mounts, you get a new kind of complaint. Users connect to Friendly but wait a long time to get any response. You can reproduce the problem as follows. Start the Friendly server. Use telnet to open a connection to it.

```
> telnet localhost 4444
Hello, my friend! How are you today?
```

Now, open a second command shell, and try to connect to the server (this mimics a second user trying to connect to Friendly). You will get no error message, but also no response, because the server is busy interacting with the first telnet session. Now, exit the first telnet session (i.e. type `bye'). Friendly will finally present the greeting message to the second user.

Friendly can currently support only one user at a time. In this task, you will augment Friendly to support multiple users *concurrently,* through the use of **threads.**

Read the first four pages in Sun's Java concurrency tutorial. Do not read beyond the section "Defining and Starting a Thread" (unless you want to).

You will now finish implementing a multi-threaded version of Friendly. This version is in package `friendly.multi`.

 **Inspect `friendly.multi.Main`.** Notice that, like the multi-run version of Friendly, it also has a loop that calls `ServerSocket.accept()` and then processes a user session. The key difference is in that instead of handling the session itself, the method creates a new thread object with a `FriendlyClientHandler`, and starts the thread, which goes off and manages the user interaction, leaving the main thread free to accept new connections:

```
.....

while (true) {
  // Wait until someone connects.
  Socket socket = serverSocket.accept();

  // Hand off the work to a new thread.
  FriendlyClientHandler handler = new FriendlyClientHandler(socket);
  Thread t = new Thread(handler);
  t.start();
}
.....
```

 **Fill in the body of the run method in class `friendly.multi.FriendlyClientHandler`.** This class implements the Runnable interface, used to create objects that run in separate threads. Basically, you can think of the run method as the new entry point for a Friendly user session. Move the required functionality into the run method.

 **Compile and run the new server.** Make sure that the new program works as expected. You will know that it works if you are able to run two concurrent telnet sessions of Friendly.

> **Checkpoint.** Find a TA or LA or another member of the course staff, and demonstrate your working multi Friendly server to get checked off on this part of the lab.

Conclusion

You have learned some aspects about I/O, networking, and threads in Java. The implementation of the IM server will be similar to the implementation of Friendly where networking and multi-

threading is concerned.

This lab was partly inspired by Sun's Java sockets tutorial. If you would like a second source to learn more about sockets, we recommend taking a look at it.

Commit Your Solution

This is the end of the lab. Be sure to commit your solutions to your personal repository.