Workbook 1

In this workbook you will learn how to write a simple Java messaging client which will allow you to send and receive plain text messages to and from fellow classmates over the Internet.

Important

An on-line version of this guide is available at:

https://www.cl.cam.ac.uk/teaching/current/FJava

You should check this page regularly for announcements and errata. You might find it useful to refer to the on-line version of this guide in order to follow any provided web links or to cut 'n' paste example code.

Connecting to the Internet

Many distributed systems use the *client-server* metaphor for communication. This style of communication will be explored in more detail in the Concurrent and Distributed Systems courses, as well as the Computer Networking course. A server is a piece of software which runs indefinitely, accepting requests from multiple clients and providing those clients with any information or other services they request. For example, a web server runs indefinitely, responding to requests from many web browsers. Last year you wrote a very simple client when you used an instance of the `URL` class to create a `URLConnection` object, and thus connect to a web server and download descriptions of board layouts for Conway's Game of Life.

The Java programming language provides several libraries to support the construction of distributed systems. In this course we are going to explore writing clients and servers with the `Socket` and `ServerSocket` classes respectively. These classes use the TCP/IP communication protocol, which won't be explained in detail here, but is taught in the Computer Networking course. For this course it suffices to know that a TCP/IP connection provides a method of transmitting a stream of bytes between a client and a server (and back again) and that the TCP/IP protocol guarantees *reliable* and *in order* delivery of any bytes sent. A TCP/IP connection is initiated by a client connecting to a server running on a particular machine and port. For example, the Computer Laboratory website runs on a machine called `www.cl.cam.ac.uk` on port number 80. Port numbers allow multiple servers and clients to run on a single machine.

The Java programming language also supports other types of communication which we will not have time to explore in detail. The `DatagramSocket` class supports a *datagram* service, an *unreliable* mechanism for sending packets of data between two computers on the Internet. This class uses the UDP/IP protocol for communication. The Java `MulticastSocket` class supports *multicast* communication, a method of sending the data to many computers simultaneously.

Reading data from an instance of a `Socket` class in Java *blocks* the execution of the program until data is available. This can be a useful feature, since you will often want your program to wait until at least some data is available before continuing execution. In the next section you will take advantage of this feature. Later on, you will discover the limitation of this approach and learn how to write your first *multi-threaded* application to overcome this problem.

Your next task is to use the `Socket` class to connect to a server on the Internet to retrieve the data sent by the server and print it to the console. The data sent by the server starts with a welcome message, and then proceeds to send the current time, once per second.

2. Start the Ticklet 1 task by visiting the following URL to Chime: https://www.cl.cam.ac.uk/teaching/current/FJava/ticklet1.

3. Clone the git repository for Ticklet 1 onto your local machine and open your preferred editor. Please see the Chime instructions for further detail if required: https://www.cl.cam.ac.uk/teaching/current/FJava/chime.html.

4. Find the Java documentation for the `Socket` class (e.g. search in https://docs.oracle.com/en/java/javase/11/docs/api/index.html) and read the documentation for the constructors, as well as the `getInputStream` and `getOutputStream` methods.

5. Create the class `StringReceive` in the package `uk.ac.cam.your-crsid.fjava.tick1`. (Hint: in IntelliJ you can do this by right-clicking on the package where you want to add the class and select **New → Java Class**.[1]) When adding new files you may see a pop-up dialog box asking whether you want to add the file to Git. You should choose **Yes** so that your work is stored in Git. The main method should accept two arguments, the name of the server first, and the port number of the service second. Your program should connect to the server using an instance of the `Socket` class, and do the following three tasks in an infinite loop: (1) read bytes from the `Socket` object; (2) interpret the bytes returned as text; and (3) print the text to the console. You can test your program by connecting to the machine `java-1b.cl.cam.ac.uk` on port `15000`. If all is well, you should see the following output:

   ```
   Welcome to the StringReceive server!
   Mon Oct 14 10:11:16 BST 2019
   ...
   ```

   If you are running your program on the command line you can terminate it with **Ctrl+c**. If you are running it inside IntelliJ, you should click on the red square to the left of the console sub-window to terminate it.

   Some hints and tips for this part can be found below.

6. Make sure that your program checks the number of arguments provided and parses them correctly. If the number of arguments are incorrect or malformed, your program should print

   ```
   This application requires two arguments: <machine> <port>
   ```

   to the *standard error stream* (hint: use `System.err`) and call `return` from the `main` method to halt your program. If your program cannot connect to the server on the provided port number it should print

   ```
   Cannot connect to [machine] on port [port]
   ```

   to the standard error stream and terminate. Some example arguments and associated responses are shown in Table 1, “Unit tests for `StringReceive`”.

[1]Note: there's a problem with the template configuration for IntelliJ on MCS machines which means this might fail. If this happens to you, please fix as follows.

Hints 'n' tips

- Use the appropriate constructor for the `Socket` class to initiate a connection to the server.
- Create a local array of bytes of a small fixed size (e.g. `byte[] buffer = new byte[1024];`) to store the responses from the server.
- The method `getInputStream` on the `Socket` object will return an `InputStream` object which allows you to read in the data from the server into an array of bytes.
- An instance of the `InputStream` class has an associated `read` method which will pause the execution of your Java program until some data is available; when some information is available, the `read` method will copy the data into the byte array and return the number of bytes read; the number of bytes read is a piece of information you will find useful later.
- The `String` class has a number of constructors which enable you to reinterpret an array of bytes as textual data. Use the following constructor to convert data held in the byte array into a `String` object:

  ```
  public String(byte[] bytes, int offset, int length)
  ```

args	**response**
none	This application requires two arguments: <machine> <port>
java-1b.cl.cam.ac.uk twenty	This application requires two arguments: <machine> <port>
java-1b.cl.cam.ac.uk 1024	Cannot connect to java-1b.cl.cam.ac.uk on port 1024

Table 1. Unit tests for `StringReceive`

A simple Java instant messaging client

Your final task is to use the `Socket` class to write your first instant messaging client. Your client will communicate with a server running on `java-1b.cl.cam.ac.uk` on port number `15001` (note this is a different port number from the one used earlier). The clients and server will communicate by sending text between them. You may have noticed that you've already written half the code! Your implementation of `StringReceive` is already capable of connecting to the server and displaying any messages sent between users.

7. Change the port number used by `StringReceive` in IntelliJ to display the instant messages sent by any client to the server. You may not see any messages since nobody else is sending any; you should receive a brief welcome message from the server however.

Your final task in this workbook is to successfully send messages to the instant messaging server. One of the difficulties which must be overcome is that reading user input from `System.in`, and reading data from the server via a `Socket` object, both block the execution of your program until data becomes

a. Go to **File** → **Settings**.

b. In the search box, type "template". Choose **File and Code templates** from the left-hand side, then choose **class** from the list of templates. You'll see the template box is empty. Put the following into it:

```
#if (${PACKAGE_NAME} && ${PACKAGE_NAME} != "")package ${PACKAGE_NAME};#end
#parse("File Header.java")
public class ${NAME} {
}
```

c. Click **OK**.

available. Consequently you must write a multi-threaded application: one thread will run in a loop, blocking until data is available on a `Socket` object (and then displaying the information to the user when it's available); a second thread will run in another loop, blocking until data has been typed in by the user (and sending the data to the server when it becomes available).

Threads in Java are created either by inheriting from the class `java.lang.Thread` or implementing the interface `java.lang.Runnable`. In either case, you place the code which runs inside the new thread inside a method with the following prototype

```
public void run();
```

You should create a new thread via inheritance today, but you will often find it useful to implement the `Runnable` interface in more complicated programs because Java only allows a class to inherit code from one parent.

A partially completed class, `StringChat` is available in your repository. Take a careful look at the use of class `Thread`. The variable `output` is a reference to an unnamed child of the class `Thread` which overrides the `run` method. The method `start` is then called on the reference. When `output.start()` is invoked, the body of method `run` is executed as the rest of the program proceeds. In other words, the body of `run` is executed concurrently with the rest of the program. As a result the above program structure provides one (new) thread of execution to print out messages from the server, whilst another (the original) thread of execution handles the user input.

8. Why is the `Socket` object `s` declared `final`? Is the keyword `final` required here, and if not, what property or properties must `s` have? Write a comment above the definition of `s` to explain why.
9. Complete the implementation of `StringChat` by completing the sections marked `TODO`. Your implementation should not rely directly your implementation of `StringReceive`, although you should take inspiration from it, and copy code snippets where appropriate.
10. Test your implementation works by connecting to `java-1b.cl.cam.ac.uk` on port `15001` and send (and hopefully receive!) messages from your classmates. If you're testing this by yourself you can always start two copies of `StringChat` and send messages between them.

Ticklet 1

You have now completed all the necessary code to gain your first ticklet. Your repository should contain the following source files:

```
src/uk/ac/cam/your-crsid/fjava/tick1/StringChat.java
src/uk/ac/cam/your-crsid/fjava/tick1/StringReceive.java
```

When you are satisfied you have completed everything, you should commit all outstanding changes and push these to the Chime server. On the Chime server, check that the latest version of your files are in the repository, and once you are happy schedule your code for testing. You can resubmit as many times as you like and there is no penalty for re-submission. If, after waiting one hour, you have not received a final response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a version of your code which successfully passes the automated checks by the deadline, so don't leave it to the last minute!