Workbook 1

In this workbook you will learn how to use Eclipse as a development environment for Java programs. Using Eclipse, you will then 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.

Using Eclipse

Last year you used the command-line tools `javac` and `java`. This year you will learn to use Eclipse to write, compile, and test your Java programs. Eclipse is a very powerful development environment, but to paraphrase Stan Lee, with great power comes great responsibility.

In this section of the workbook you will learn how to create a new Java project in Eclipse, how to create new packages and classes, and how to compile and test your code. You will also learn how to export a Java jar file in order to submit your tick. You will begin by writing a simple "Hello, world" program to learn the basic operations of Eclipse.

This course uses PWF Linux. If your PWF workstation is currently running Windows please reboot it into Linux and log in using your PWF username and password. Once you have logged in, please do the following:

1. Start the Eclipse program by using the application menu on the left-hand side of the screen and selecting the top icon (this icon is labelled **Dash Home** when the mouse is hovered over it). Inside the Dash Home panel type **Eclipse** into the search box and then click on the Eclipse icon.

2. A dialog box will appear asking for you to select your "workspace". Leave the default as is, which should be `/home/crsid/workspace`.

3. A splash screen will be visible for thirty seconds or so, after which the main screen of Eclipse will load.

4. Create a new project by using the Eclipse window menus **File** → **New** → **Java Project** to bring up the New Project dialog box. Enter "Further Java" as the project name and click on the **Finish** button.

5. At the moment you will still see an Eclipse welcome screen. Close the tab with the label "Welcome" to reveal the main developer screen. The left-hand window pane contains the Package Explorer, the centre and right-hand portions of the screen are currently mostly blank.

6. In the Package Explorer pane, click on the triangle or *arrow* next to the "Further Java" project you've just created. The arrow should now point downwards (▼) to reveal a sub-directory called "src" associated with the project. Right-click on the "src" icon to reveal a menu; on this menu select **New** → **Package** to reveal a pop-up dialog box; in the dialog box enter `uk.ac.cam.crsid.fjava.tick1` as the package name and click the **Finish** button to close the dialog box.

7. Your next task is to create a class called `HelloWorld` inside the package you have just created. This can be done from either the **File** menu (then **New** → **Class**) or by right-clicking on the "src" directory in a similar fashion to the previous step. In the dialog which appears, check the tick box beside "public static void main(String[] args)" to auto-generate a `main` method in your class. Click the **Finish** button to create the class.

8. You should see a source file `HelloWorld.java` appear in the Package Explorer, although you may need to expand the view of the package you have created by clicking on the arrow next to the package name so it points downwards (▼).

9. The central portion of the screen should display an editor window for the file `HelloWorld.java`. (In future, you can open source files by double-clicking on the icon representing the source file in the Package Explorer.) Edit the contents of the file in the central edit portion of the screen to read as follows:

```
package uk.ac.cam.crsid.fjava.tick1;

public class HelloWorld {
 public static void main(String[] args) {
  System.out.println("Hello, world");
 }
}
```

At this point you might like to explore some of the more advanced auto-complete features in Eclipse. For example, when writing the body of the `main` method, type `Sys` and then type **Ctrl+space** to see a list of possible completions; choose `System` and press **Enter**. Now press "`.`" and wait briefly for a set of possible classes, methods and static references within `System`; select `out`. Press "`.`" and wait briefly for a set of possible classes, methods and static objects within `System.out.println` and select `println`. Typing the two characters `("` will auto-generate their compliments and place the cursor inside the quoted string, enabling you to simply type `Hello, world`. Finally, press the **tab** key twice to move outside the terminating quote and round bracket and enter `;` to finish the line of code.

10. Save your work. To compile and run your new Java program, click on the "Run" button in the Eclipse menu bar, or right-click on the source file `HelloWorld.java` in the Package Explorer and then select **Run As...** → **Java Application**. If all has gone well, you should see `Hello, world` printed in the bottom Console portion of the Eclipse window.

11. Create a Java jar file using the code you have just created. To do this, go to the **File** menu and select **Export...**. A dialog window should pop up; expand the folder **Java** and select **JAR** and press the **Next** button. In the subsequent dialog window, expand the "Further Java" project and tick the box next to the package `uk.ac.cam.crsid.fjava.tick1` to indicate you wish to export the code in this package. In addition, tick the box further down the dialog box labelled "Export Java source files and resources". *You must remember to do this whenever you generate a jar file for submission as practical work since the source files are required for marking.* Enter a name for the jar file by clicking on the **Browse** button and selecting a suitable destination directory; please save your file as `crsid-HelloWorld.jar`. Once you have entered a suitable filename for the jar file, press **Finish** on the export dialog to complete the export.

12. Finally, to check that your export was successful start up a Bash terminal (hint: look in the Applications menu at the very bottom of the screen). Change your working directory using the command `cd` so that your working directory is the same as the location of your jar file. Run your program as follows, making sure you get the same output as shown below:

```
crsid@machine:~> java -cp crsid-HelloWorld.jar \
uk.ac.cam.crsid.fjava.tick1.HelloWorld
Hello, world
crsid@machine:~>
```

1. Modify the `HelloWorld` program so that if the program is given a single argument on the command line the program will say hello to the argument provided. For example:

   ```
   crsid@machine:~> java -cp crisd-HelloWorld.jar \
   uk.ac.cam.crsid.fjava.tick1.HelloWorld crsid
   Hello, crsid
   crsid@machine:~>
   ```

   If the program receives no arguments on the command line it should print `Hello, world` as before.

2. It is often useful to test programs with command line arguments inside Eclipse. Test your program in this way now by editing the settings needed to run your `HelloWorld` program. To do so, go to the main Eclipse menu select **Run → Run configurations...** (in some versions of Ecplise this is **Run → Run ...**). This opens a new dialog window which allows you to create a custom mechanism for running the `HelloWorld` program. Check that the name of the run configuration is something sensible (like "HelloWorld") and that the Main class is `uk.ac.cam.crsid.fjava.tick1.HelloWorld`. Click on the "Arguments" tab and add your crsid as an argument to the "Program arguments" text box; finally click the "Run" button. You should see `Hello, crsid` printed to the console in Eclipse.

Connecting to the Internet

The second half of today's class uses the Eclipse environment to write a simple chat client.

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 is 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 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 first 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.

3. Find the Java documentation for the `Socket` class (http://download.oracle.com/javase/6/docs/api/) and read the documentation for the constructors, as well as the `getInputStream` and `getOutputStream` methods.

4. Create the class `StringReceive` in the package `uk.ac.cam.crsid.fjava.tick1`. 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!
   Mon Nov 2 14:23:18 GMT 2009
   Mon Nov 2 14:23:19 GMT 2009
   Mon Nov 2 14:23:20 GMT 2009
   ...
   ```

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

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

5. 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`”.

Hints 'n' tips

- Use **Ctrl+Shft+o** to organise your `import` statements and fill in any missing ones.
- 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 than 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.

6. Change the port number used by `StringReceive` in Eclipse to display the instant messages sent by any client to the server. (If you complete this task early in the session, you may not see many 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 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 is shown below:

```
//TODO: import appropriate classes here

public class StringChat {
 public static void main(String[] args) {

  String server = null;
  int port = 0;

  //TODO: parse and decode server and port numbers from "args"
  //      if the server and port number are malformed print the same
  //      error messages as you did for your implementation of StringReceive
  //      and call return in order to halt execution of the program.

  //TODO: why is "s" declared final? Explain in a comment here.
  final Socket s = //TODO: connect to "server" on "port"
  Thread output = new Thread() {
   @Override
   public void run() {
    //TODO: read bytes from the socket, interpret them as string data and
    //      print the resulting string data to the screen.
   }
  };
  output.setDaemon(true); //TODO: Check documentation to see what this does.
  output.start();

  BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
  while( /* TODO: ... */ ) {
   //TODO: read data from the user, blocking until ready. Convert the
   //      string data from the user into an array of bytes and write
   //      the array of bytes to "socket".
   //
   //Hint: call "r.readLine()" to read a new line of input from the user.
   //      this call blocks until a user has written a complete line of text
   //      and presses the enter key.
  }
}
```

Take a careful look at the use of class `Thread` in the above code. 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 and *simultaneously* execution of 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 the another (original) thread of execution handles the user input.

7. Why is the `Socket` object `s` declared `final`? Write a comment above the definition of `s` to explain why.
8. Complete the implementation of `StringChat` by completing the sections marked `TODO`.
9. 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.

Ticklet 1

You have now completed all the necessary code to gain your first ticklet. Please generate a jar file which contains all the code you have written for package `uk.ac.cam.crsid.fjava.tick1`. Please use Eclipse to export both the class files *and the source files* into a jar file called `crsid-tick1.jar`. Once you have generated your jar file, check that it contains at least the following classes:

```
crsid@machine~:> jar tf crsid-tick1.jar
META-INF/MANIFEST.MF
uk/ac/cam/crsid/fjava/tick1/StringChat$1.class
uk/ac/cam/crsid/fjava/tick1/StringChat.class
uk/ac/cam/crsid/fjava/tick1/StringChat.java
uk/ac/cam/crsid/fjava/tick1/StringReceive.class
uk/ac/cam/crsid/fjava/tick1/StringReceive.java
uk/ac/cam/crsid/fjava/tick1/HelloWorld.class
uk/ac/cam/crsid/fjava/tick1/HelloWorld.java
crsid@machine~:>
```

When you are satisfied you have built the jar correctly, you should submit your jar file as an email attachment to `ticks1b-java@cl.cam.ac.uk`.

You should receive an email in response to your submission. The contents of the email will contain the output from a program (written in Java!) which checks whether your jar file contains all the relevant files, and whether your program has run successfully or not. If your jar file does not pass the automated checks, then the response email will tell you what has gone wrong; in this case you should correct any errors in your work and resubmit your jar file. 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 any response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a jar file which successfully passes the automated checks by the deadline, so don't leave it to the last minute!