
Workbook 1

Introduction

In this course you will work through a series of practical exercises which will teach you the skills necessary to program in Java. There are no lectures associated with this course, but there is a question based on the material in this course in the final exam. This course relies heavily on the material taught concurrently in Software Design and Programming Methods.

In contrast to the practical assessment associated with Foundations of Computer Science last term, this course requires you to attend a two hour practical lesson once a week for the eight weeks in Lent Term. During the practical lesson you will work through this workbook. You may take the workbook home at the end of the lesson to complete any unfinished sections. You will also need to complete the associated assessed exercise, or *tick*, described at the end each workbook.

Important

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

<http://www.cl.cam.ac.uk/teaching/0809/ProgJava>

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.

Getting help

In each practical class you will find three course instructors who have the following duties:

- | | |
|---------------------|---|
| <i>Lecturer</i> | The lecturer has written the material for this course and is able to offer advice on its content and correctness as well as more general help with programming. |
| <i>Assessor</i> | The assessor or <i>ticker</i> marks submitted exercises and discusses them with students. |
| <i>Demonstrator</i> | The demonstrator is responsible for providing practical help on the content of the workbook and associated Java programming exercises. |

Aim of this course

The course is designed to accommodate students with diverse backgrounds in programming ability; consequently Java is taught from first principles in a practical class setting where you can work at your own pace. Students with prior experience at Java should complete this workbook as normal and then seek an extension exercise from the course lecturer.

Each practical class introduces a new programming topic and provides pointers to further background reading. Each exercise is designed to provide experience and insight into the practical effort of designing, building and testing software. The work for each week is cumulative, building on work undertaken for previous weeks, so that by the end of the course you will have written a piece of software capable of playing Conway's Game of Life.¹ By way of incentive, a screenshot of the graphical user interface of the final piece of software you will write is shown in Figure 1, "Screenshot: Conway's Game of Life".

¹http://en.wikipedia.org/wiki/Conway's_Game_of_Life

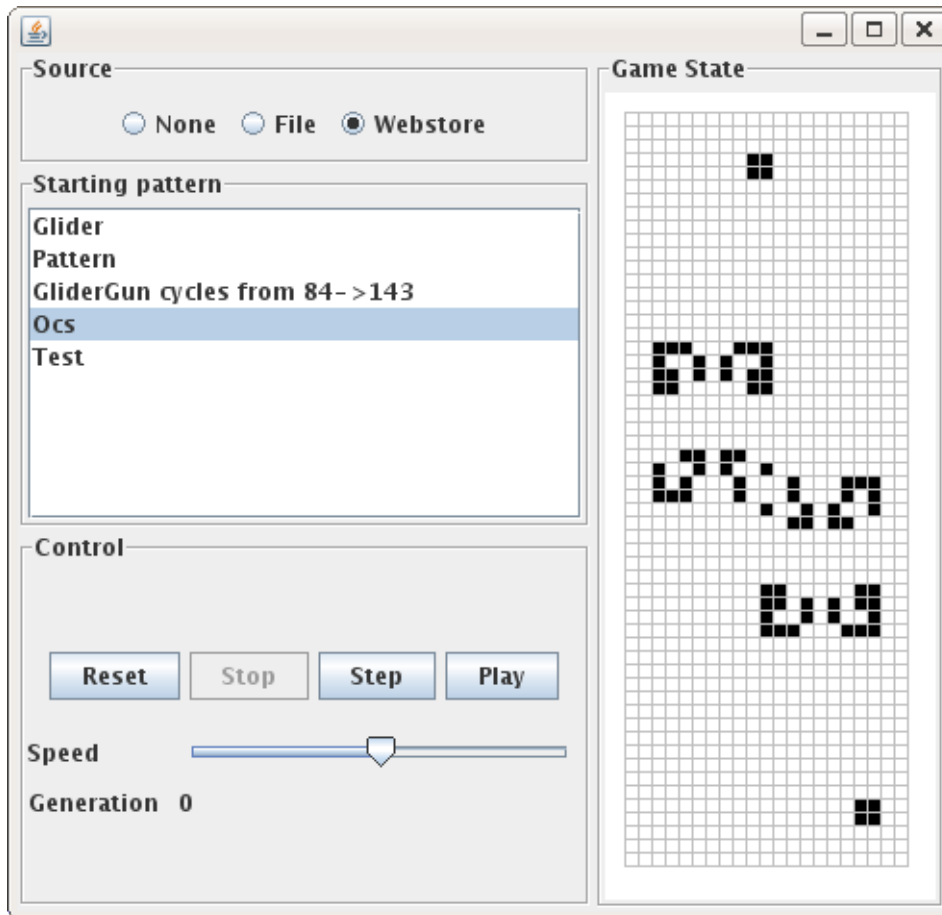


Figure 1. Screenshot: Conway's Game of Life

What to do each week

The tasks you need to perform on week n of the practical class can be expressed as follows:

1. Arrive at the practical class on week n at the start of your allocated time period (1000-1200, 1200-1400, 1400-1600, or 1600-1800).
2. Find your computer as allocated on the seating plan.
3. Collect workbook for week n .
4. When the ticker comes to your desk, be prepared to discuss the practical work you completed for week $n-1$.
5. Leave your computer at the end of the session, taking your workbook(s) with you; you may continue to complete any exercises in your own time.
6. Submit your solution to the exercises for week n (which must pass the automated tests) before Wednesday at 5pm following the practical class on week n .

Note: when $n=1$ (i.e. the first week), the ticker will mark your submission for ML tick 6; the last tick of the Java course should be submitted (and pass the automated tests) by 5pm on the first Wednesday of Easter Term; you will be assessed on this tick on the first Thursday of Easter Term.

Bear in mind that this term the ticker will come to you, not the other way around; therefore you must sit in the correct place!

Using PWF Linux

This course will use the Computing Service Public Workstation Facility (PWF), for which you will need a username and password. Computing Service Reception can reset forgotten passwords. The PWF computers are capable of running either Microsoft Windows, or Linux. PWF computers will be running one of these two operating systems at all times. In order to save electricity, if there are no local or remote users, the computer will enter a low-power sleep state; this feature may mean that you need to wait 8 to 10 seconds after moving the mouse before the machine appears to be awake.

Once your PWF machine has woken up, you will see one of three screens:

1. A PWF Windows Login window dominated by the word Novell
2. A large message window which refers to the Computer Misuse Act of 1990. If you see this click OK and the screen will change to the third possibility...
3. A PWF Linux login window.

Starting PWF Linux

This course uses Linux, so if your computer is currently in state (1) above then you will need to reboot it. If your computer is already running Linux, you may skip to the section called "Logging into PWF Linux". To reboot your computer you should:

1. Press **Ctrl+Alt+Delete** to reveal a Microsoft Windows version of the Computer Misuse Act 1990 message.
2. Click OK to display a new window with the title "Novell Client" at the top.
3. Do not login! Instead, select Shutdown (one of the three buttons at the bottom of the window) and select Shutdown and Restart and then click OK.
4. Wait one minute while the computer reboots, at which point a menu will appear. Use the down arrow (↓) to select "PWF Linux" from the menu and hit **Enter** to boot the computer.
5. Linux may take up to four minutes to boot, during which time you may see a message which begins "ATTENTION NO SIGNAL"; take no action and wait until you see the Linux variant of the Computer Misuse Act 1990 message. Click OK to get to the login prompt.

Logging into PWF Linux

You should now see a PWF Linux login screen. To log on to the computer:

1. Type in your username in the text box next to the label "Username" and press **Enter**.
2. When prompted, type in your password and press **Enter**.

If you've typed your password in correctly your Linux desktop should now appear with various icons on it, and a task bar at the bottom. In the centre of the screen is the "PWF Message of the Day" window; you may close this window down after you've glanced at its contents.

At the end of your session you should log off. To do so:

1. Choose Log Out from the System menu at the bottom of the screen.
2. Click on Log Out in the message window.

This will clear all your windows and return you to the Linux variant of the Computer Misuse Act 1990 message. At this point it is safe to walk away from the machine. Do not switch off the computer. Leaving

the computers on allows Computing Service personnel to maintain them; it also allows other users to run jobs on them remotely via the network.

Answering questions

As you progress through the workbook each week, you will be presented with questions in a *question box* just like the one shown below. Make sure you follow the instructions and answer any questions in the question boxes; you will need to complete these instructions in order to gain your tick.

Providing answers

Write answers to all of the questions found in this workbook into a plain text file called `answers.txt`. You will need to include your `answers.txt` file in your submission for your first Java Tick. Further information about the submission of the tick can be found at the end of this workbook.

Open a copy of Gnome Editor, or your favourite Linux text editor if you have one, and create a file called `answers.txt` (Hint: try looking in the applications menu at the bottom left of the screen.) Whenever you see a numbered question or task in a workbook, follow the instructions or provide a *brief* answer (usually one or two sentences will suffice) prefixed with the question number. Do this now for the following task:

1. Write your given name followed by your family name and your username in brackets in the first line of the file. For example, "Alastair Beresford (arb33)".

Using Bash

Under Unix or Linux, a *shell* or *terminal* is often used to control the creation, execution and termination of applications. In the first part of the course we will be using a common shell, *Bash*, to control the Java tools. There are many other Integrated Development Environments (IDEs) available for Java, but we will avoid them in this course in order to provide you with a sound understanding of the fundamentals. We recommend you use the Gnome Editor to write your Java applications—it's simple and has all the features you require—however if you are familiar with Linux or Unix you may use an alternative text editor of your choice.

Bash is a programming language in its own right. Unfortunately we won't have time to explore many of its features in this course, however improving your Bash skills in your own time will pay dividends in later years. If you are interested in finding out more you might like to take a look at the reference manual, but only if you have time.²

Bash commands and programs

To start a Bash shell or terminal, right-click on the desktop and select Open Terminal. You should now see a new window with a white background. The text displayed in the Bash shell is sometimes called a *prompt* or *command line* and should look something like the following

```
crsid@machine:~>
```

where `crsid` should be your own login name (e.g. `arb33` or `acr31`), `machine` is the name of the computer you are using and the text between `:` and `>` represents the *current working directory*, in this instance your home directory, which is represented by the special symbol `~`.

You can execute Bash commands or start programs by typing the command or program name at the prompt. For example to print the current date and time you can use the Unix program `date` :

²<http://www.gnu.org/software/bash/manual/bashref.html>

```
crsid@machine:~> date
Wed Oct 29 19:45:35 GMT 2008
crsid@machine:~>
```

Try typing this command into your terminal now. Notice that the program `date` prints out the current date and time to the shell. After the date and time has been printed by the program `date`, the program terminates and the shell prompt is displayed again. We can pass arguments to the program which will modify its operation. Arguments are written on the same line as the program and are separated by whitespace. For example, we can modify the output of the `date` program by specifying an alternative output format for the date and time:

```
crsid@machine:~> date +"%Y-%m-%d %H:%M:%S"
2008-10-29 19:49:35
crsid@machine:~>
```

In this case the additional argument after `date` instructs the program to print the date and time formatted according to the ISO 8601 standard.³ Many programs in Unix have an electronic manual associated with them which describe the additional arguments the program will accept. You can view the manual associated with a program by using the `man` program. The `man` program accepts the name of a program and displays the relevant manual (if any). To list the options associated with the `date` program, type the following at your prompt now:

```
crsid@machine:~> man date
```

Your shell should now display the manual page for `date`. You can scroll through the options for `date` by using the up arrow (↑) and down arrow (↓) keys on your keyboard. You can quit the `man` program by typing `q`.

Providing options to commands

2. What is the format string required to ask `date` to display time and date information in the following format: 10:47 AM on Thursday 15 January 2009? (Hint: you might like to use the command `man date` to help you.)

Update: On some Linux machines the locale setting may prevent the display of AM or PM; consequently an answer which does not display AM or PM is acceptable.

Working with files and directories

Many Bash commands or Unix programs operate on files contained in the current working directory. When you first created your Bash shell, your working directory was configured to be your home directory. You can print out your current working directory in full by using the program `pwd`:

```
crsid@machine:~> pwd
/home/crsid
crsid@machine:~>
```

You can also list the files and directories inside your current working directory by using the `ls` command:

```
crsid@machine:~> ls
```

The number and names of the files displayed will depend on whether you have any files saved in your PWF home directory yet or not. Not all the files will be shown by default; any files starting with a full stop

³<http://www.cl.cam.ac.uk/~mgk25/iso-time.html>

(.) will be excluded since these are files are, by convention, hidden by default on Unix systems. You can list all the files, including the hidden ones by specifying the appropriate option to `ls`.

Providing options to commands

3. What is the correct option to pass to `ls` to display hidden files?

The Bash shell supports a variety of *wildcard* options which pattern match against filenames. A useful canonical example is `*` which matches zero or more characters in a filename. This can be used to list all the files which end in a particular string (e.g. `ls *.txt` matches all files which end with `.txt`) or all files which start with a particular prefix (e.g. `tick*` matches all files which start with `tick`, such as `tick3-report.txt`). The shell expands the list of files which match the pattern and passes them on to the program, so this feature works with all programs which accept command line arguments, not just `ls`.

You can make a new subdirectory inside your current directory using the `mkdir` command, and change your current working directory to it using the `cd` command:

```
crsid@machine:~> mkdir java
crsid@machine:~> cd java
crsid@machine:java> pwd
/home/crsid/java
crsid@machine:java>
```

You can move back out of your newly created subdirectory by using the `cd` command together with the special directory `..` which represents the parent directory of the current directory:

```
crsid@machine:java> cd ..
crsid@machine:~>
```

Background tasks

The programs we have used so far don't take very long to run and consequently we don't mind waiting for the program to finish executing before the prompt returns and we are able to issue another command. However some programs, such as a text editor, may continue running for hours. In such cases we don't want to wait for such programs to finish before we can issue more commands; instead we would like to run several commands simultaneously. For programs which are likely to take a long amount of time, we can append an ampersand (`&`) to the end of the command to start the program as a *background task*. Start the Gnome Editor, which has the program name `gedit`, using this technique:

```
crsid@machine:~> gedit&
```

The Gnome Editor has now started, perhaps over the top of your Bash shell; if the shell is obscured, you can view the bash prompt again by selecting it on the task bar at the bottom of the screen. View the Bash shell now. You will notice that you are able to issue further commands, despite the fact that the Gnome Editor has not yet finished executing. You can see this is the case by using the `ps` program to list the running processes associated with the shell:

```
crsid@machine:~> ps
  PID TTY          TIME CMD
 18718 pts/2    00:00:00 bash
 26050 pts/2    00:00:00 gedit
 26083 pts/2    00:00:00 ps
crsid@machine:~>
```

Notice that, in addition to the `gedit` program, you can see that the `bash` program and the `ps` program were also running at the point when the `ps` program recorded the programs associated with the shell. The `ps` program can accept lots of options on the command line just like `date`. These options control which running programs are listed and how to format the output. Type `man ps` and briefly explore a few of these options.

Redirecting output

Sometimes it is useful to redirect the output from one program and use it as input to a second program. For example, if we only want to see the line of output associated with `gedit` in `ps` above we can use the pipe operator (`|`) to redirect the output of `ps` into the input of another program. One suitable program is `grep`, which can be used to select a subset of lines from the input to reflect in its own output. For example, the following combination of `ps` and `grep` can be used to list only those lines of output from `ps` which contain the string `'gedit'`:

```
crsid@machine:~> ps | grep gedit
26050 pts/2    00:00:00 gedit
crsid@machine:~>
```

We might wish to redirect the output of a program to a file. For example, to keep a copy of the output of `ps` we can use the redirect operator (`>`) to save it into a file:

```
crsid@machine:~> ps > ps-output.txt
```

Execute this command and open the file `ps-output.txt` in `gedit` by typing:

```
crsid@machine:~> gedit ps-output.txt &
```

Once you have checked that the output has been saved correctly you can close the Gnome Editor and delete the file by making use of the `rm`. (Hint: type `man rm` for help.)

Further questions on Bash

4. Use the electronic manual to describe each of the following Bash commands in one sentence each:

```
ls pwd cd cp mv mkdir rm rmdir ps kill man grep gedit
```

5. How would you find out what arguments can be passed to the `man` program?

6. What changes to the filesystem occur when the following commands are executed in turn, and what does each command produce as output to the terminal?

```
i. echo Java1A > file.txt
ii. cat file.txt | grep Java
iii. rm -i file.txt
```

(Hint: It is safe to type the three commands above to see what they do provided you don't have a file called `file.txt` in your current working directory.)

Java tools

Java is a *compiled* language, which means that the Java programs you write must be transformed into another format before execution. To perform this transformation you need a *compiler* and in this course we will use the Java compiler called `javac` to convert Java *source code* into Java *bytecode*. Java bytecode can then be run using the Java runtime program `java`. Compilers for some languages, such as C, produce *machine code* as output which can then be run directly on the CPU. The Java designers

chose to use the bytecode format because it is portable and can be moved between computers and operating systems without the need for recompilation; the price paid for this portability is the need for a runtime program to assist the computer in the execution of the bytecode.

Create a new file in `gedit` by selecting the New button on the graphical interface or typing **Ctrl+n**. In this new file, type in the following Java source code:

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

Save this Java source code to disk using the filename `Example.java`. (In Java the filename must be the same as the name following the keyword `class` with the extension `.java` appended to it.)

This example contains a lot of syntax and keywords. Java has quite a lot essential syntax, even for the most basic program, which you do not need to fully understand this week. You will hear more about it in the Software Design course, and you will gain a thorough understanding of it in future practical classes. Even if it looks complex now, it won't do by the end of the term, and the main message at the moment is *don't panic*, you're not supposed to understand it! This week you are learning how to control the Java tools.

Having said all this, you might find an overview of the example helpful, so here it is. The phrase `class Example { ... }` provides a definition of a new data structure, in this case called `Example`. This data structure, or *class*, can store values (such as integers, strings, and so on) which in object-oriented programming parlance are called *fields*; the example class above has a single field called `message`, which stores an item of type `String`. A class can also contain program code written as one or more *methods*; the example above has a single method called `main` which takes an array of strings (`String[] args`) and returns nothing (`void`). This is conceptually similar to an ML function which takes a string list and returns `unit`. A method which is of the form `public static void main(String[] args)` is special in Java and indicates the point at which program execution begins. Next week we will explore fields, methods and arrays of values in much more detail.

To compile the source code to bytecode make sure that your working directory for your Bash shell is the same as the directory containing your source code. For example, if you saved `Example.java` into the directory `java` then you may need to change into that directory before trying to compile the code. (Hint: you might need to issue a command which looks something like `cd java`).

To compile your Java source code you should type in the following:

```
crsid@machine:~> javac Example.java
```

The prompt should return without any textual output. If the compiler does print some output it will describe one or more warnings or errors in the program because you have not typed in the source code correctly. In this case take another look at your file and correct any mistakes you find; the output from the compiler may help you to identify where these are.

If your program has compiled correctly, the compiler will have generated a file called `Example.class`. You can check this exists by using the program `ls`. The file `Example.class` is the bytecode version of your Java source code which you can execute with the `java` program. Run your Java bytecode now as follows:

```
crsid@machine:~> java Example
Hello, world
crsid@machine:~>
```

Notice that we do not include the `.class` file extension when calling the `java` program; one of the strategies used by the `java` runtime program to find the class `Example` is to look for a file called `Example.class` in the current directory.

Creating a jar file

In all but the most trivial of Java projects you will produce multiple source files, and because the Java compiler produces one bytecode file for each source file, Java provides a convenient mechanism for encapsulating multiple bytecode files into a single file on disk. The encapsulating file is called a *jar* file and is similar in concept to zip files (Windows), DMG files (MacOS X) or tar files (Unix systems).

To create a jar file we will use the command line tool called `jar`. The `jar` program accepts a variety of options on the command line which you can read about by typing `man jar`. To create a suitable jar file for the example above you should type the following, making sure that your current directory includes the `Example.java` and `Example.class` files before doing so:

```
crsid@machine:~> jar cfe Example.jar Example Example.*
```

The command line arguments to the execution of program `jar` above do the following things:

<code>cfe</code>	Instructs the <code>jar</code> program to create a file with the specified entry point (see below).
<code>Example.jar</code>	The name of the jar file to be created.
<code>Example</code>	The <i>entry point</i> , specified as the name of the class which contains the special main function, where program execution should begin.
<code>Example.*</code>	The list of additional files to include in the jar file. In this case the <code>*</code> wildcard has been used and is expanded by Bash into all files in the current directory which match the pattern <code>Example.*</code> . This will include both <code>Example.class</code> and <code>Example.java</code> ; these two file names will then be passed separately to the <code>jar</code> program. It is equivalent to writing: <code>jar cfe Example.jar Example Example.class Example.java</code> .

After executing the above program on the shell you should find a jar file called `Example.jar` in your current directory which contains both `Example.java` and `Example.class` and creates an internal *manifest* file which states that the `Example` class contains the special 'main' function where execution begins. You can check the contents of the jar file by using the `test` feature of the `jar` tool:

```
crsid@machine:~> jar tf Example.jar
META-INF/
META-INF/MANIFEST.MF
Example.class
Example.java
crsid@machine:~>
```

You can see that the `jar` tool has created an additional directory containing a file called `MANIFEST.MF` as well as including both `Example.java` and `Example.class`. In general, the manifest file can contain a multitude of configuration parameters and options to control the execution of the Java application. You can write a manifest file by hand to specify such options, however this will not be necessary in this course.

Once you have built a jar file, you can load the classes contained within it and execute the class specified by the entry point using the `java` program with the `-jar` option:

```
crsid@machine:~> java -jar Example.jar
Hello, world
crsid@machine:~>
```

If you do not see the output 'Hello, world', then you have not packaged your jar file correctly; in this case you will need to review the steps covered in this section and if you're still stuck seek help from the demonstrator or lecturer.

Java Packages

In order to manage large projects and prevent two (or more!) programmers from accidentally giving the same name to two different implementations of a class, Java classes can be placed in a *package*. By convention, Java package names use an Internet Domain Name Service (DNS) name as a prefix for a package name. Consequently in this course you should package all of your code inside `uk.ac.cam.crsid` where `crsid` should be replaced by your own PWF username. In other words, if you log in to PWF machines with the username `arb33` all your java code should have the package prefix `uk.ac.cam.arb33`.

You must declare which package a class is associated with at the very top of the source file. For example, to place the 'hello world' example above into the package `uk.ac.cam.arb33.examples` you write the following:

```
package uk.ac.cam.arb33.examples;

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

The source file should still be saved into a file called `Example.java` but should be placed inside the directory hierarchy which mirrors the package name. Therefore in the above example, `Example.java` should be saved inside a directory heirarchy of `uk/ac/cam/arb33/examples`. Such a directory path can be created with the following Bash commands:

```
crsid@machine:~> mkdir uk
crsid@machine:~> mkdir uk/ac
crsid@machine:~> mkdir uk/ac/cam
crsid@machine:~> mkdir uk/ac/cam/arb33/
crsid@machine:~> mkdir uk/ac/cam/arb33/examples
```

although if you had read the electronic manual for `mkdir` you might have noticed you can actually do this in one step:

```
crsid@machine:~> mkdir -p uk/ac/cam/arb33/examples
```

You should then move `Example.java` into this location using the `mv` command:

```
crsid@machine:~> mv Example.java uk/ac/cam/arb33/examples
```

To compile a program which is inside a package, the current working directory of the shell must contain the outermost directory of the package (in other words, in the case above, the directory `uk` should be directly inside the current working directory). The example can then be compiled as follows:

```
crsid@machine:~> javac uk/ac/cam/arb33/examples/Example.java
```

the compiler will put the `Example.class` file in the same directory as `Example.java`.

The associated jar file can be built in a similar fashion, except that the name of the class must now include the name of the package it is within:

```
crsid@machine:~> jar cfe Example.jar \  
uk.ac.cam.arb33.examples.Example \  
uk/ac/cam/arb33/examples/Example.*  
crsid@machine:~>
```

(Note: the backslash character (\) is used to continue a single command onto an extra line; you do not have to type this character in if you write all the text on a single line.)

The jar file can then be executed using the `java` program as before:

```
crsid@machine:~> java -jar Example.jar  
Hello, world  
crsid@machine:~>
```

When you become a more proficient Java programmer, you may wish to keep your source files in a separate location from your bytecode files. This is possible, but for now we will forego this complexity and keep all the files in one directory hierarchy.

Java Tick 1

Your task is to modify the example program you have created in earlier parts of this workbook. The `Example` class should be renamed `Tick1` and placed inside the package `uk.ac.cam.crsid.tick1`, where `crsid` is your own username (e.g. `arb33`). When executed, your new class should print out your name followed by your username in brackets, for example, for student "Alastair Beresford" with username "arb33", the program should execute as follows:

```
crsid@machine:~> java uk.ac.cam.arb33.tick1.Tick1
Alastair Beresford (arb33)
crsid@machine:~>
```

Finally, create a jar file containing the source code (the `.java` file), the bytecode (the `.class` file) and your plain text file `answers.txt` which by now should contain the answers to all the questions found in this workbook. You should save the jar file as `crsid-tick1.jar`, where `crsid` is your username. A correct submission by Alastair Beresford (`arb33`) would display the following information when tested:

```
crsid@machine:~> jar tf arb33-tick1.jar
META-INF/
META-INF/MANIFEST.MF
uk/ac/cam/arb33/tick1/Tick1.class
uk/ac/cam/arb33/tick1/Tick1.java
uk/ac/cam/arb33/tick1/answers.txt
crsid@machine:~>
```

Executing the main method inside the entry point class of the jar file should produce the following output:

```
crsid@machine:~> java -jar arb33-tick1.jar
Alastair Beresford (arb33)
crsid@machine:~>
```

When you are satisfied you have written your source file, compiled it, and built a jar correctly, you should submit your jar file as an email attachment to `ticks1a-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 passes the automated checks then you need to take no further action. However, 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 `ticks1a-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!