The Java Compiler – An Introduction

COMP40009 - Computing Practical 1

12th – 13th January 2023

Aims

- To gain experience using the Java Programming Language Compiler
- To practise using Java from the command line
- To master a series of increasingly complex compilation exercises
- To practise familiarising yourself independently with available programming tools and command line options

The Problem

Knowing the programming tools available to you and being able to familiarise yourself with new ones quickly is the bread and butter of a programmer. This exercise will guide you through a series of compilation exercises, by the end of which you will not only have familiarised yourself with the Java Compiler, but will also have gained exposure to doing so largely independently. Chances are, you may need to do that many times with many different tools throughout your degree and professional life. This exercise will introduce you to some of the fundamental mechanisms of the Java Compiler, while your task will be to conduct your own research on how to actually use the tool, and what command line options it provides to help you complete the exercises.

What to do:

Clone your exercise repository (remembering to replace *username* with your username) using:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab2223_spring/javacompilerintro_username.git
```

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files[1]. On the lab machines, this has already been installed for you, so you can extract the archive by typing:

```
7z x skel.7z -p61F75A54
```

(*n.b., 61F75A54 is the password*). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named "skel", because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.

[1] `https://www.7-zip.org/download.html`

The exercise is divided into five parts, each of which has some code that is ready to be compiled and executed – you do not need to change this source code (but you are encouraged to read it!). Each exercise subdirectory contains a shell script called **compileAndRun.sh** which you will need to complete. All exercises can be completed with a single call to **javac**, followed by a single call to **java**.

In order to complete this exercise, remember that most programs come with help flags (typically `<program name>--help`), manual pages (`man <program name>`), online documentation, and much more – you are encouraged to make use of anything you find helpful to complete the exercise, so long as by the end of it you understand how it works.

You will need a Java Development Kit (JDK) installed in order to run the Java compiler and execute Java programs. Having already worked in Kotlin, you should already have a JDK installed. If not, see the Installing Kotlin video for how to install Open JDK – we recommend using version 14 or 15. If you are using a lab machine, the JDK will already be installed.

Exercise 0 - Hello World!

This exercise prints **Hello World!** to the command line, a common first program that people write to ensure their development environment is working as intended.

First, you will need to invoke the Java Compiler, **javac**. This will generate a **.class** file containing Java Bytecode, a low-level representation of Java code that can be executed on the **Java Virtual Machine** (JVM).

Now you can invoke **java** to execute your code in the JVM. You will need to provide it with the class name (not the file name) of the class you wish to be run, and the class loader will scan the current working directory, looking to find where that class is defined, and executing its main method.

Hint: You will find plenty of detailed material available online – or in textbooks – on Java Bytecode, JVM, the class loader, etc.

Exercise 1 - Packages

A Java program can be split over separate **.java** files, and these files can in turn be split over separate folders called **packages**. Java classes that are defined in the same package do not need to be explicitly imported to be able to 'see each other' and interact with each other. However, classes from a different package need to be imported explicitly. You will see such import statements at the top of the **Main.java** file. The package name is separated by dots and maps directly to a corresponding directory structure – i.e. package **uk.ac.ic** maps to folders **uk/ac/ic** and so on. If all classes from a particular package should be imported, a wildcard can be used (marked as *****).

The **Main** class sits at the top of the package hierarchy, which is called the default package. All classes outside of the default package need to explicitly state which package they belong to, as you can see in the first line of the source code. Question: what happens if you **cd** into a package and invoke **javac** from there? Will it go with the package declaration, or assume the class to be in the default package? You will see that the **Main** class imports two other classes. Complete the **compileAndRun.sh** script to compile and run the program.

Exercise 2 - Build Directory

You will have noted that your **.class** files have been appearing next to your source files when **javac** is called. However, we would like to be able to separate out the **.java** files from the produced **.class** files, thus keeping our source directory clean and tidy by having a separate build directory. This exercise comes with one directory, and you will need to create one more:

- **src/** – containing the source code
- **out/** – an empty directory to hold **.class** files (you need to create this)

As briefly mentioned previously, **javac** and **java** will scan the current working directory for relevant files – however, neither the source code nor the compiled Java bytecode are in the current directory.

As such, you will need to do three things:

- Tell **javac** where to find source files
- Tell **javac** where to output the compiled files
- Tell **java** where to find compiled files

As before, fill in the **compileAndRun.sh** script. You will need no more than one call each to javac/java. Do not commit the **out** directory, or the generated .class files, to version control.

Exercise 3 - Libraries

Many programs use external libraries, which contain code that is packaged up in a **jar** file. This exercise is as before, but with an added **lib/** directory. You will notice that it makes *absolutely no difference at all* in the code whether it is using a class from a source package, or from a library - the import and usage is exactly the same. See whether you can complete the exercise without any further help.

Exercise 4 - Unit Tests

Unit testing is important to ensure correctness of code, especially in large projects where different people are working on the same code base, making changes that potentially interfere with each other. Having unit tests in place ensures that for every code change performed, the developer can have confidence that the code is behaving correctly.

This exercise has the following directory structure:

- **src/** – containing source code to be tested
- **test/** – containing test code
- **lib/** – containing libraries used for testing
- **out/** – an empty directory to hold compiled classes (you need to create this)

Out of the four directories, you will notice that essentially two of them contain source files (namely, src and test). If you look at the unit test, you will see that it imports classes and methods required for testing as before, but that it does not import the class that is being tested. The reason is, that although the two files are in different directory structures, both are in the same package, the default package, and thus they can interact with each other without requiring explicit import statements. Although coming from different locations on the file system, all that matters to Java is what package a class is in. The two directory structures are thus effectively merged internally, while we benefit from having our source and test files organised in separate directories.

After the tests have been compiled, they can be run from the command line via:

```
> java [...] org.junit.runner.JUnitCore SquareTest
```

... and you only need to fill details on where to find the code and where to find any additional libraries needed for testing.

As a result, you should see something like:

```
JUnit version 4.11
...
Time: 0.003

OK (3 tests)
```

Submission

This exercise exists to help you get accustomed to the Java compiler. There are **no tests** for this exercise. **You only need to: a) complete each of the .sh files and then b) answer the question in cmdLineOptions.txt**. Push and submit the final version of your code, as per usual, via LabTS, **without using the "Request test" button**.

Assessment

```
This is an unassessed exercise.
```