Java Programs and the Command Line Interface

0. Overview

Some brief notes on compiling and running Java programs from the windows command line:

(1) Setting up.
(2) A brief introduction to the windows command line.
(3) Compiling and running Java programs, and command line arguments.

1. Setting up

Instead of using JGrasp, you can compile and run your programs by typing commands using the windows 'command prompt'.

To be able to do this, however, you need to check two settings on your computer.

1.1 The Path variable

Firstly, we need to make sure that Windows can find the '*javac*' and '*java*' applications; the applications we need to compile and run Java programs.

Windows keeps a list of the folders where it searches for applications. This list is in the PATH environment variable – we have to make sure that it contains the folder where the Java applications are stored.

On your desktop, right-click '*My Computer*' and select '*Properties*'. Click on '*Advanced*' then '*Environment Variables*'. Under '*System variables*', click on '*Path*' then '*Edit*'.

Make sure that the '*Variable value*' starts with the text corresponding to the location of the *javac* and *java* applications:

```
C:\Program Files\java\jdk1.7.0_45\bin;
```

(This is the location of the Java applications on SciLab A computers – it could be different on yours!)

1.2 The Classpath variable

Secondly, we need to make sure that Java can in fact find the programs we want to execute. Java maintains a list of all places where it looks for compiled files in the system variable CLASSPATH. We need to tell Java that it should first look in the current folder.

Under '*System variables*', click on '*Classpath*' then '*Edit*'. Make sure that the '*Variable value*' starts with the text:

```
.;
```

(A single decimal point and a semicolon.)

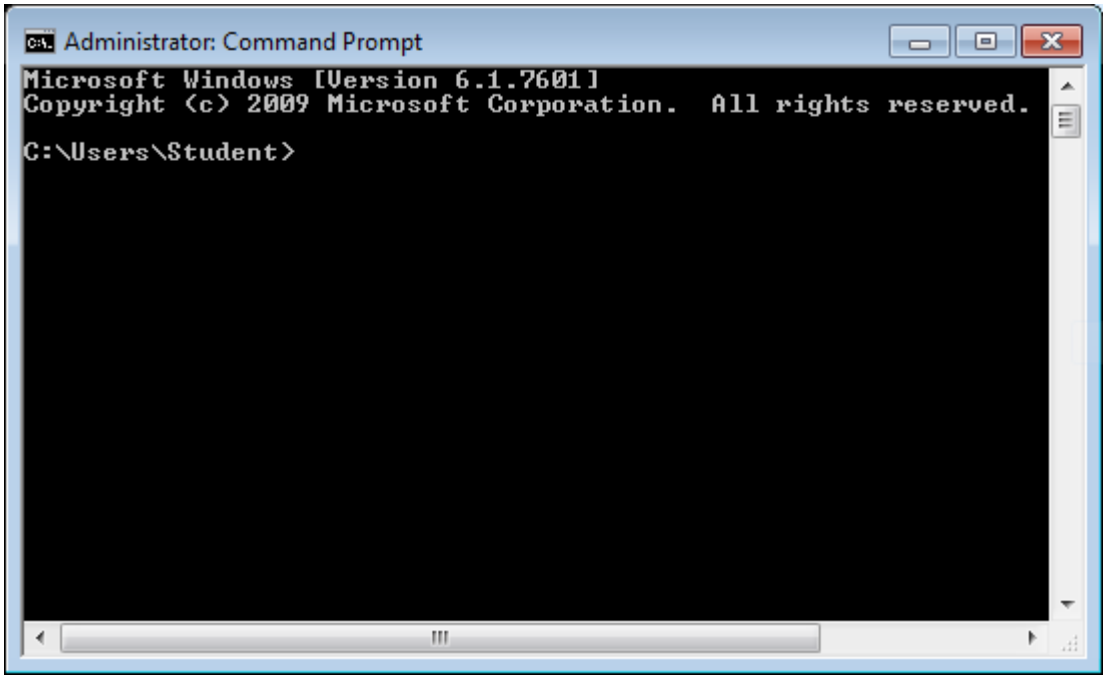
The single decimal point refers to the current folder. The semicolon separates multiple entries in the list. If there is no '*Classpath*' listed, click on '*New*' under '*System variables*' and enter the '*Variable name*' as '*CLASSPATH*' and the '*Variable value*' as indicated above.

CONTINUED

2. The Windows Command Prompt

The Windows command prompt is obtained by clicking the "*start*" button, selecting "*All Programs*", then "*Accessories*", and then "*Command Prompt*".

When you run the windows command line interface on an ICTS machine you should see something like this:



2.1 Command prompt

The short piece of text beginning "C:\Users..." is the "command prompt". Its primary purpose is to indicate that the interface is ready to receive a command, however the text that forms the prompt also provides useful information.

2.2 Current working directory

The commands that a user issues via the command line interface are understood to take place within the context of a particular directory, or file folder, to use a more recent terminology. This is known as the current working directory. The text of the command prompt indicates where it is.

The "*C:*" indicates that the directory is found on the disk drive labelled "*C*" and what follows indicates where. The first "\" denotes the top-most level (or the root) of the "*C*" drive, the "*Users*" denotes a folder that may be found there, and the "*Student*" denotes a folder that may be found within that folder.

It's unlikely that a user will have much interest in the "*C:*" drive on an ICTS machine. To move the current working directory over to the '*F:*' drive, where your files are stored, typing "F:" followed by return/enter.

2.3 The "dir" command

Typing in "*dir*" and pressing return, will produce a list of the files and folders at the top level of the "*F*" drive.

CONTINUED

Using the Windows graphical user interface, double-clicking on the "*My Computer*" icon followed by the one for the "*F:*" network drive should produce a view of the same files and folders as were listed by "dir".

2.4 The "cd" command

We can change the current working directory by using the "cd" (change directory) command. The command must be followed by an expression that indicates the new working directory.

For example, assuming that the current working directory is the top-most level of the "*F*" drive and at that level there's a folder called "*projects*", typing "*cd projects*" changes the current working directory to "*F:\projects*". Using the "*dir*" command, we can see the files in "projects".

Let's say that there's a folder called "*chapter01*" within the "*projects*" folder. By typing "*cd chapter01*" the current working directory becomes "*F:\projects\chapter01*". If we wished we could have got here in one step instead of two by using "*cd projects\chapter01*" to start with.

If the current working directory is "*F:\projects\chapter01*", we can move it back up to the "projects" directory by using "*cd ..*". The ".." symbol means "the directory that contains the current directory".

3. Using the command line interface to compile and run Java programs

To compile and run a Java program the first step is to move the current working directory to the one in which the files that make up the program are stored.

3.1 The "javac" command

The name of the Java compiler is "*javac*". If we wish to compile a Java class declaration we enter the command "*javac*" followed by the name of the file containing the declaration. So for example, say we have a declaration for a Circle class in the file "*Circle.java*", we use the command "*javac Circle.java*".

If there are any errors found during compilation then details are displayed on the screen - what they are and where they are.

If compilation is successful then there is no screen output but a quick check of directory contents with "*dir*" should show the existence of a bytecode file for the class. In the case of our example this would be "*Circle.class*".

If we ask for a declaration to be compiled that depends on other declarations in other files then in the course of affairs these are automatically compiled too.

3.2 The "java" command

The name of the Java virtual machine is simply "*java*". If we wish to run a Java program we enter the command "*java*" followed by the name of the class that contains the main method.

Let's say the Circle class contains a main method, we would enter "*java Circle*". If any runtime errors occur then program execution fails and details of the failure are output to the screen.

Note the difference between the "*javac*" and "*java*" commands. In the case of the former the argument is the name of the file containing the text of a class declaration, while in the case of the latter it's the name of a class.

CONTINUED

3.3 Command Line Arguments

In the context of command line execution we can provide some more illumination on the features of a Java main method.

Consider the following (very simple) Java program.

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

So we have a class that contains a main method that just prints "Hello". We can compile and run this at the command line like this:

```
F:\>javac Hello.java
F:\>java Hello
Hello
F:\>
```

When we run the "*javac*" command/program we supply the name of the file we want compiled. This name is a "command line" argument. Similarly, the "*java*" command/program has a command line argument. It's the name of the class containing the main method that is to be executed.

It's possible to design our Java programs so that they ALSO accept command line arguments. This is the purpose of the main method parameter "`String[] args`".

We'll modify our simple program so that it accepts a command line argument; the name of a person to greet.

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello "+args[0]);
    }
}
```

We can compile and run this at the command line like this:

```
Z:\>javac Hello.java
Z:\>java Hello Stephan
Hello Stephan
Z:\>
```

Note the difference when we run it. We use "*java Hello Stephan*". There are two command line arguments. The "*java*" command uses/consumes the first one; the name of the class containing the main method. It passes the next as an actual parameter to that method. It does this by first creating a String array of length one and putting "Stephan" in location zero.

In general, given a "java" command of the form

java <class name> <argument0> ... <argumentn>

A String array of length n-1 is created, the arguments put in it, and the array passed as a parameter to "main".

END

CONTINUED