

Computer/Engineering Programming 1 – Practical Class 2

Syntax, Semantics and Program Development Tools

Aims and Objectives

This laboratory has been designed to help you to

- become familiar with the program development environment *jGrasp* ,
- use *jGrasp* to compile and execute Java applications (programs),
- recognise and correct syntax, semantic and logical errors in simple Java applications and
- write a complete, small Java application.

The Purpose of Checkpoints, Monitoring Progress and Optimising Performance

As the assessment policy details, checkpoints account for 30% of the total assessment. The tasks leading to checkpoints have been designed to improve learning outcomes by guiding you to solutions which require the application of techniques and concepts from course material. They are **not** designed to be done in isolation.

We expect students to use lecture notes and the text book as a basis for constructing solutions. You should consult reference material **before** requesting help from a demonstrator. For example, if you are unsure about what form a method definition takes, try and find an example in the notes and use it as a template or make use of *jGrasp*'s template facility. In this way you will learn more and will require less help from demonstrators.

The other purpose of the tasks is to allow us to assess your level of understanding and your ability to apply the knowledge you have acquired. For this reason, checkpoints are not of equal difficulty. The first 2 or 3 have been designed to test basic competence, that is, to measure performance from just below a pass to a high P. The last 2 checkpoints usually require a greater depth of understanding and correspond to the range CR to HD. For some students it may be possible to complete some of these checkpoints by spending a large amount of time, however, you should take care not to allow completing later checkpoints to dominate your study time (or other aspects of your life).

A good strategy is to attempt checkpoints from the current practical before attempting incomplete checkpoints from previous practicals. It is also wise to allocate a fixed amount of time per week for practical work.

Finally, we encourage discussion of the tasks with your peers but plagiarism is dishonest and detrimental to learning, and can result in severe consequences (see [Academic integrity](#)). To discourage the poor learning outcomes which result from plagiarism, demonstrators will ask you to explain your solutions unless they have observed their step by step development and are convinced they are your own work. You will also find that the ideas which lead to the solution will become more clear through their enunciation.

Getting Started

Although Java programs will need to be compiled and executed (run) during the laboratory, neither the compiler (javac) nor the interpreter (java) will be used directly. Instead, they will be used by the program development environment, *jGrasp*, through the *Build* menu (see below).

The first part of this laboratory requires some basic computer knowledge.

1. Once you have logged in, create a directory called **cp1**
2. Create a directory called **prac01** (in your **cp1** directory) and make it your current working directory.
3. Copy all the Java files contained in *Week 1 on FLO* to your current working directory

Checkpoint 1

Congratulations, you have just completed your first CP1 checkpoint! Show a demonstrator your file listing and press on...

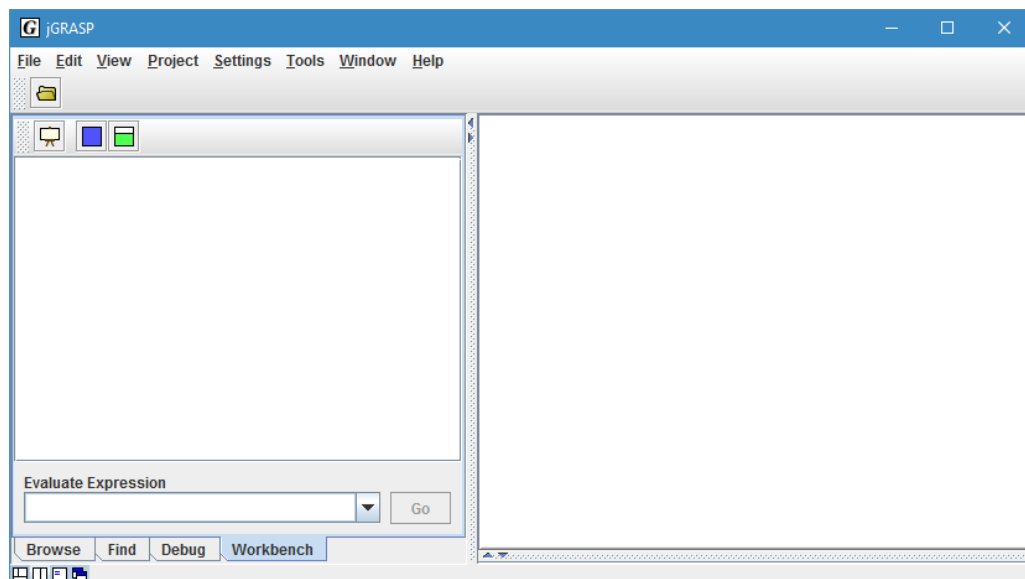
Using *jGrasp*

jGrasp is a tool which supports the development of programs. It has features which help the programmer with the tasks of writing program code, Java in our case, detecting and correcting syntax and semantic errors, viewing the structure of a program, compilation and execution.

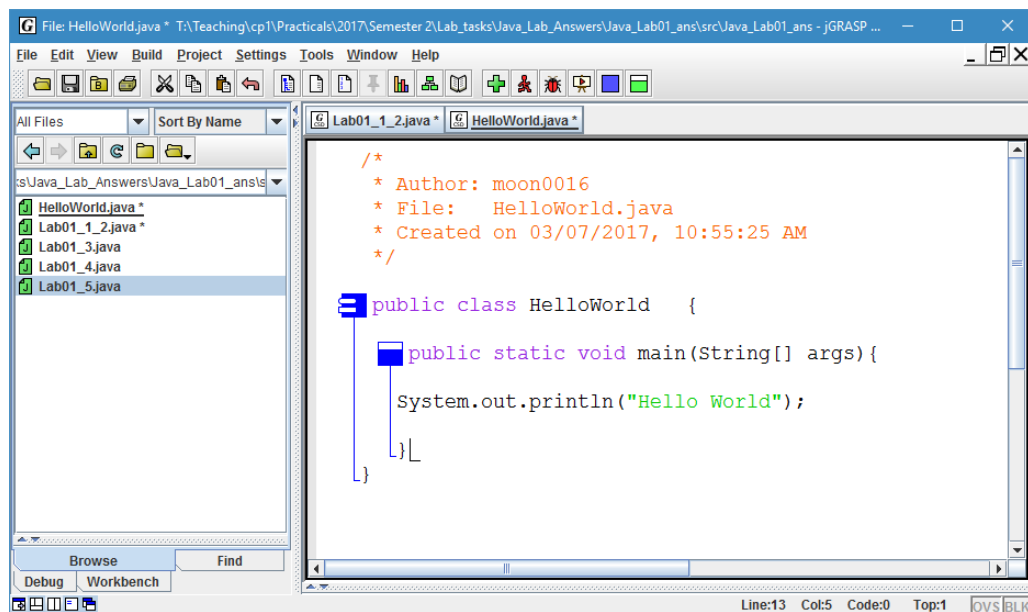
The file `HelloWorld.java` (see lecture 2) has no syntax or semantic errors. For the next checkpoint you will need to compile and execute (run) the program. Here is how to do it:


1. Invoke *jGrasp*


A window like the one below should appear after a short delay.

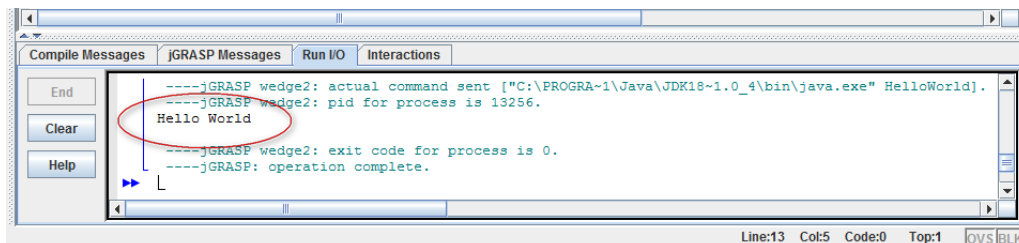


2. Select *Open* from the *File* menu and then select `HelloWorld.java` from the “select a directory/file” window which appears. The file `HelloWorld.java` will be loaded into *jGrasp*, giving:



3. Compile the program by selecting *Compile* from the *Build* menu (or by clicking the Compile icon ).

4. From the *Build* menu, select *Run* (or click the Run icon ) to execute the program. Check that the output Hello World appears in the newly created “run” window.



Checkpoint 2

Show the execution window to a demonstrator

Syntax Errors

“Syntax” refers to the sequence in which the basic elements of a program (identifiers, reserved words, literals and other symbols such as “+”) appear. Java has a set of rules which define exactly how these symbols can be put together to form programs.

The compiler will not translate a program unless it is free of syntax errors. Examples of syntax errors are:

- a missing semicolon (a semicolon indicates the end of a statement)
- misspelling a reserved word
- using a reserved word where an identifier is expected
- not terminating a string with a quotation mark
- incorrectly formed numbers, for example, two decimal points in the one number
- not enclosing the body of a class within braces ({})

Another related error is the pragmatic one of not giving a file the same name as the class it contains or not including a main method.

1. Again, select Open from the file menu and this time select the file `SyntaxError.java`.
2. The program contains a number of errors. Try compiling the program to locate the errors. The first line in the program which contains an error will be highlighted. Determine what is wrong with that line and correct it. Fix as many of the other errors as you can and then try to compile the program again. Repeat the above steps until all the syntax errors have been corrected.
3. Once the program compiles successfully, click on Generate CSD and note the way the program code is augmented with a graphical indication of its structure.

Checkpoint 3

Execute (run) the program and show the output to a demonstrator.

Semantic Errors

Semantic errors result from programs which although syntactically correct, do not make sense according to the way the language is defined. For example, the following expression is valid.

`"one" + "two"`

The plus sign (+) acts to concatenate two strings and not as arithmetic addition (like the use of the word “hack” in English which can mean *a type of horse*, *a pick* or *a quick program fix* depending on its context).

The following expression is invalid since minus can only be applied to numbers (there is no interpretation of minus applied to strings).

`"one" - "two"`

Another common form of semantic error is mistyping the name of a method or class. In this case the compiler will not recognise the name as valid method or class name. For example, in the following statement the name of the `println` method has been mistyped.

```
System.out.pritln("Hello");
```

Logical Errors

Logical errors result from writing program code which does not perform the desired actions. That is, there is a mismatch between what the programmer thought the code would compute and what it does actually compute.

A simple example of this is misplaced or missing bracketing in an expression. For example, the value of the the following expression

```
3 + 4 + "5"
```

is the string "75" since Java interprets the expression from the left to give $7 + "5"$ and finally "75". If the programmer expected the value to be "345" then brackets should have been introduced to force Java to interpret the expression from the right, as in:

```
3 + (4 + "5")
```

Note that both expressions are semantically and syntactically correct but the first does not calculate the desired result. Although the example is not realistic (the programmer could simply write "345"), once we start using variables, similar, but realistic, problems will arise.

Your next task is to correct semantic and logical errors in an existing program.

1. Select Open from the file menu and this time select the file `SemanticError.java`.
2. Compile the program and use the same process as above to correct semantic errors. Your corrections should alter as little of the file as possible (add, remove or change as few characters as possible). When the corrected program is executed it should produce the following output:

```
6 plus 4 equals 10,  
4 - 3 equals 1.  
Well done!
```

Checkpoint 4

Show the corrected program and its output to a demonstrator.

Creating a Program

So far we have worked with existing programs. We will now create a Java program and go through the same process of correcting all the syntax and semantic errors to produce a program that will execute. Your program may still have run-time errors which are semantic errors which cannot be detected by the compiler. Run-time errors are reported by the interpreter and should be located and corrected in a manner similar to the other types of errors.

The task is to write a Java program which produces the output *exactly* as it appears below:

```
42 = 7 * 6 (answer 1)
```

```
3 + 5 = 8 (answer 2)
```

```
The ideas of "state" and "sequence"  
are fundamental to most programming.
```

In each case, the answer must be calculated by your program rather than appearing literally in a string. All integers must appear as integer literals (not strings) in your program. For example, the 42 in the first line of output should be generated by multiplying 7 by 6 and the 7 and 6

should be literal integers and not strings.

Hint: Page 75 of the text book explains how to include quotes in a string.

Proceed as follows:

1. Create a new file by selecting **New** → **Java** from the **File** menu.
2. Type in your program (class definition), starting with the line

```
public class First {  
  
    public static void main(String[] args) {  
        ... // Your code goes here  
    }  
}
```

3. Save the file as **First.java**
4. Identify and correct any errors and then execute (run) the program. Remember to re-save the source file before compiling it.

Checkpoint 5

Once you are satisfied that your program is error free, outputs the correct result and satisfies the requirements of the exercise, show the source code and output to a demonstrator and explain how the program works.