

## COMP1102/8702 – Practical Class 1

### Syntax, Semantics and Program Development Tools

#### Aims and Objectives

This part of the laboratory has been designed to help you to

- become familiar with the program development environment *IntelliJ*,
- use *IntelliJ* 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 50% 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. 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 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. The last task is for those students who wish to extend their programming. 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). 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, *IntelliJ*, through the *Run Project* button (see below).

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

1. Once you have logged in, create a directory (folder) called cp1 (or similar)

2. Create a subdirectory called Practicals within the cp1 directory
3. Download **Practical01.zip** from FLO (in *Practical 1 Files* under the *Assessment module*) and extract it into your Practicals directory. You should now have a Practical01 directory that contains an *IntelliJ* Java project

### Checkpoint 1

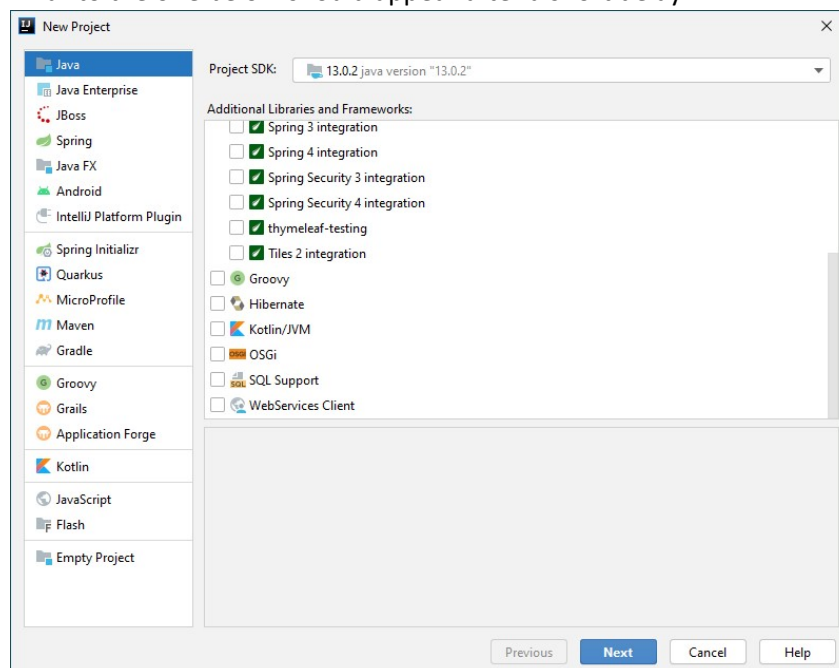
Congratulations, you have just completed your first CP1 checkpoint! Show a demonstrator your file listing and press on...

### Using IntelliJ

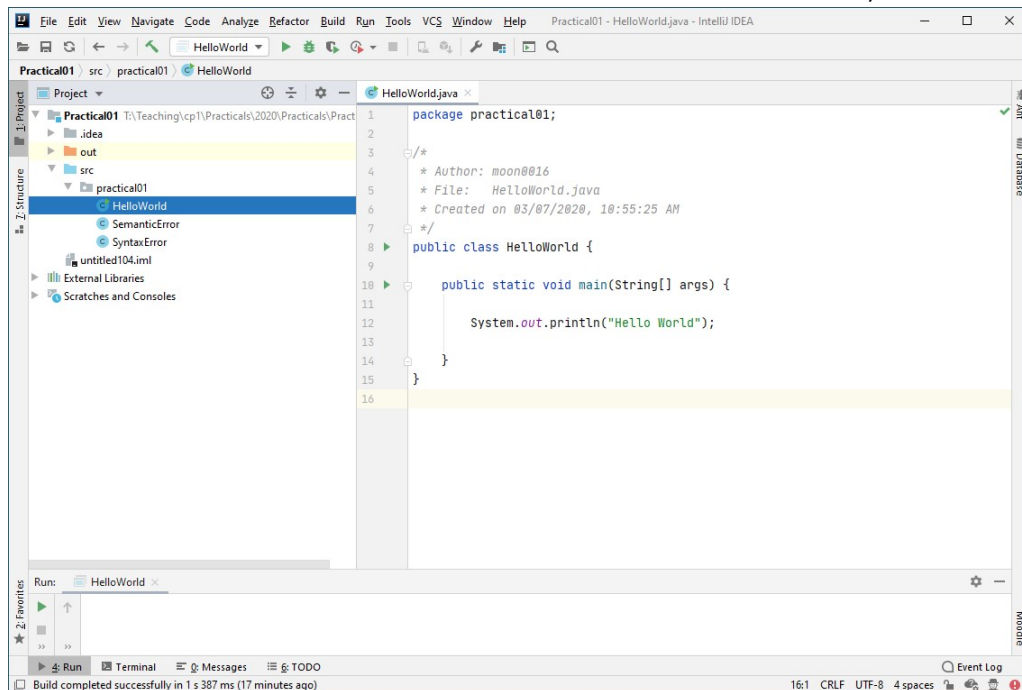
*IntelliJ* is a tool which supports the development of programs. It has features which help the programmer with the tasks of writing program code, Java in our case, detecting and correcting syntax and semantic errors, viewing the structure of a program, compilation and execution.

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. Start *IntelliJ*

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



2. Select *Open* and then select Practical01 directory you extracted earlier from the dialog box that appears. The project you downloaded from FLO will be loaded into *IntelliJ*, giving a window similar to the following: (with HelloWorld.java shown open) If you do not see the file listing on the left, click the vertical *Project* button.



Compile the program by first selecting the HelloWorld file in the project listing on the left (if it is not already selected) and then selecting *Recompile 'HelloWorld.java'* from the *Build* menu. To execute the program, select *Run 'HelloWorld'* from the *Run* menu<sup>2</sup>. Check that the following output appears in the populated Run window

Hello World



## Checkpoint 2

Show the execution window to a demonstrator

2.

<sup>2</sup>You can also execute the program using either of the green 'play' buttons (located in the margin next to the class and main method declarations) and selecting *Run 'HelloWorld.main()'*.

## 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. This time select the file `SyntaxError.java`.
2. You will notice in IntelliJ that the code will (by default) be displayed in grey text. This is because all the code for `SyntaxError` has been “commented out”. This means that there is a line at the top of the file containing the symbols `/*` , and a line at the bottom containing the symbols `*/` (note the order of the characters). In terms of the Java language, this means that everything in this file has been turned into a comment, and so the content of the file between the two lines `/*` and `*/` will be ignored by the Java compiler. “Commenting out” is a technique that programmers sometimes use to hide code temporarily (for example, to see what the effect is of removing a line from a program, without actually deleting it and having to type it in again later).
3. This style of indicating a comment is for comments that run over several lines. Java has another style of commenting that indicates only a single line as a comment; we will use that style in Checkpoint 4 below.
4. Now “comment the code back in again” – in other words, delete the line `/*` at the top, and the line `*/` at the bottom. This restores all of the code that was inside the comment block, and the compiler will now read the file and try to understand it as a legitimate Java program. You will probably see the colour coding of the text changing in IntelliJ.
5. The program will not run as it is, because it 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.

---

### 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 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. This time select the file SemanticError.java.
2. In SemanticError, there are three lines inside the main() method block that have been commented out. This time we have used the other style of commenting in Java, where we use the symbols // to indicate that everything occurring to the right of these symbols, until the end of the line, should be treated as a comment. In other words, these three lines, even though they are valid Java, are not treated as Java by the compiler, but are simply ignored.
3. Now put the lines back into the program, by simply removing the // symbols at the beginning of each line. Now you have a program which is syntactically correct, but contains semantic errors.

4. 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.

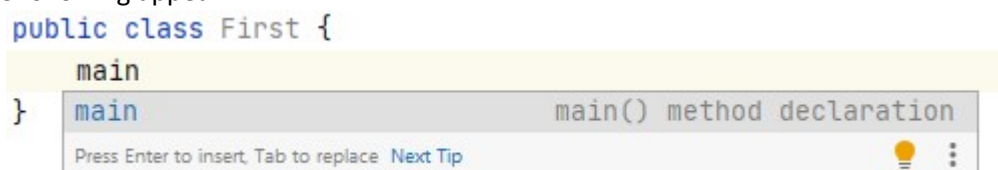
**Hint:** Page 89 of the text book explains how to include quotes in a string.

Proceed as follows:

1. Create a new file by selecting New from the File menu and then selecting Java Class.
2. In the Name prompt that appears, type First, and press Enter/Return on your keyboard. You should see package practical01;

```
public class First {
}
```

3. Place your cursor between the { } of the First class and type main. You should see a panel similar to the following appear:



```
public class First {
    main
}
```

The screenshot shows a code completion panel with the text 'main' entered. The panel displays 'main' as a suggestion, with a tooltip showing 'main() method declaration'. Below the tooltip, it says 'Press Enter to insert, Tab to replace' and 'Next Tip'.

Press Enter/Return on your keyboard to generate the main method. Your editor window should now contain the following: package practical01;

```
public class First { public static void main(String[] args) {  
  
    }  
}
```

4. Within the main method write the application logic according to the specifications above 5.

Identify and correct any errors and then execute (run) the program.

---

#### **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.