# FIT5171 Applied Session 1
# Hands on with Maven & JUnit

Week 1, 2024

| Please do complete the preparation tasks before coming to the tutorial. Your active participation is the most important! |
|---|

In this tutorial, you will work individually to get familiarised with some of the frameworks and tools that we will be using throughout the course assignments: Maven and JUnit.

Each student would preferably have a laptop/desktop computer. Please let your tutor know if you don't have one so that we can try to make alternate arrangements for you.

As all other tutorials, this will be a hurdle. Have fun!

## 1 Preparation

Before coming to the tutorial, you will need to:

- Install Java JDK (1.8) on your computer.[1]
- Install your IDE (integrated development environment) of choice: Eclipse,[2] or IntelliJ IDEA Ultimate Edition.[3]

  Note that Eclipse and NetBeans are open-source software while the Ultimate Edition of IntelliJ IDEA is commercial software. However JetBrains, the developer of IntelliJ, offers free licenses to students for a large number of their products, including IntelliJ. Details can be found on their web site.[4]
- Install Apache Maven (3.x).[5]
- Make sure that your IDE works with Maven. Depending on your IDE of choice, you may need to install plugins. For Eclipse, you may need to install the plugin `m2eclipse`.

## 2 Testing with Java, Maven & JUnit

In this tutorial, you will be developing and testing a very simple application without any graphical user interface (GUI). During the tutorial, you will need to

- Set up a Maven project for the application.
- Adopt the Test-driven Development (TDD) process to develop and test the main *functionality* of the application.

You are free to design the structure of the application.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
[2] `http://eclipse.org/downloads/`
[3] `http://www.jetbrains.com/idea/`
[4] `https://www.jetbrains.com/student/`
[5] `http://maven.apache.org/download.cgi`

## 2.1 Application: anagram detector

An anagram of a word is another word produced by rearranging the letters of the original word, using all the original letters exactly once.[6] For example, "`listen`" is an anagram of "`silent`".

This application will need to include the following functionality.

1. Given two words as input, return whether they are anagrams of each other. For example, for "`listen`" and "`silent`", the detector should return `true`. For "`pizza`" and "`donut`", the detector should return `false`.

2. Given a list of words separated by *whitespace characters*, return *all* the groups of words that are anagrams of each other.

   For example, assume that the detector is given the following text as input.

   ```
   tews tis lives tamed elvis ream evils comics stew wets markers dashed
   west veils rat mace sit mated cosmic mare remarks shaded
   ```

   Given the above input, the detector will find the following groups of anagrams as output:

   - `tis sit`
   - `tamed mated`
   - `comics cosmic`
   - `mare ream`
   - `markers remarks`
   - `dashed shaded`
   - `west wets tews stew`
   - `lives veils elvis evils`
   - `rat`
   - `mace`

   Note that the last two groups in the output represent the two words that do not have anagrams in the input.

   You can decide exactly how the detector should output the groups of anagrams.

Making use of the TDD process, write some tests for the above functionality and use them to guide you in writing the actual code that actually implements the functionality. You should do this iteratively. If you have difficulty completing the implementation, write some pseudo-code in comments.

Have a discussion during the tutorial with the class on what tests need to be written, and how many tests will need to be written.

## 2.2 Application: $k$-th largest element of an array

Given an array of `int`s and an `int k`, find the $k$-th largest element in this array. For example, given an array {`-1, 0, 2, 6, 4`} and an `int 3`, you are required to find the 3rd largest element of the array, which is 2.

Making use of the TDD process, write some tests for the above functionality and use them to guide you in writing the actual code that actually implements the functionality. You should do this iteratively. If you have difficulty completing the implementation, write some pseudo-code in comments.

Have a discussion during the tutorial with the class on what tests need to be written, and how many tests will need to be written.

[6] `http://en.wikipedia.org/wiki/Anagram`

# 3 Assessment

You will be assessed on the following:

- Has the Maven project been successfully set up?
- Have unit tests been written with the correct `@Test` JUnit annotation?
- Has JUnit been integrated with Maven so that unit tests can be executed successfully?
- Do the tests make sense?