Introduction to Java, Maven and Git

LAB OBJECTIVE: The practical study of algorithms in this course will be based on implementing them as programs written in the **Java** programming language. Due to its ease of use and compatibility across multiple platforms, Java is a good language for writing *concise*, *elegant* and *complete* descriptions of algorithms which are easily *testable*.

Java is a high-level programming language with a syntax similar to C++, but is strictly an **object-oriented programming language**. Most Java programs contain *classes*, which are used to define *objects*, and *methods* that are assigned to individual classes. Java programs are interpreted by the Java Virtual Machine, or JVM, making all Java programs *multiplatform* and executable on different platforms, including Macintosh, Windows, and Unix computers [1].

The goals of this lab are to:

1. go over some *basic concepts* of Java, such as working with classes, creating and running a project,
2. introduce you to writing **unit tests** using JUnit
3. showcase how to create an executable JAR file

Assistant's note: This course's main focus will be on implementing data structure algorithms in Java, and **not** on the actual Java language and object-oriented programming. Some previous understanding of object-oriented concepts (and Java) is highly desirable, and most of you should have had some related courses by now. However, even if you are not familiar with OOP/Java, Java is a *C-family language*, meaning that previous experience with other C-family languages (C/C++, C#, etc.) should help you quickly get the grasp on Java concepts. The reading material for this week posted on LMS is also a helpful resource. Moreover, only a basic understanding of Java is required for most of the assignments, and all necessary advanced Java topics (such as generics, interfaces, etc. in the upcoming lectures) will be covered in detail during their respective lectures/labs. Of course, you are also **always welcome** to talk to me for any further clarifications/explanations of programming concepts.

LAB TASK: Implement a simple Java program which reads in a list of integers from a file, finds the maximum integer and writes it to a new file. Afterwards, implement a few simple JUnit tests to judge the correctness of the function(s) and export your program to a JAR file.

Setting up your environment

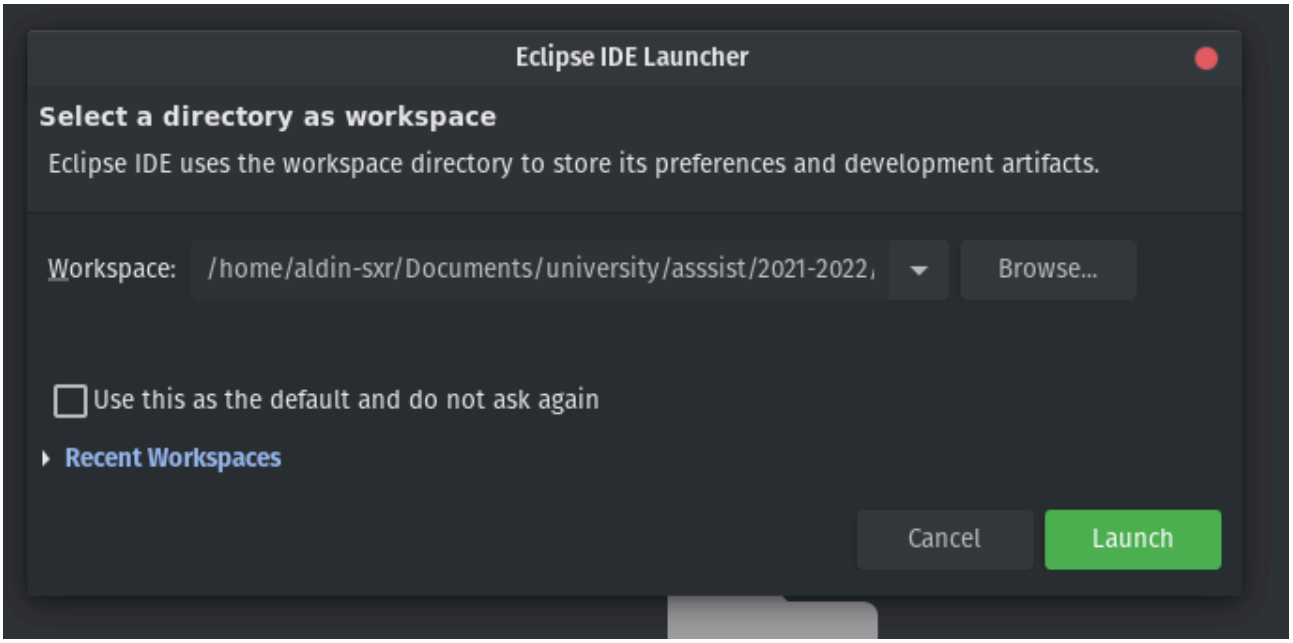
If you did not set up Java on your machine previously, you will need to install the Java Development Kit (JDK), officially named "Java Platform Standard Edition" or "Java SE". Moreover, the editor used in this course and during lab sessions will be the Eclipse IDE for Java, but you are free to use whichever editor you prefer.

The simplest way to obtain all prerequisite software is to install Eclipse IDE for Enterprise Java and Web Developers, which should set you up with everything you need. In case you find out you cannot run Java code after installation, you can follow this comprehensive guide, which details the installation steps of both JDK and Eclipse on all major platforms (Windows, MacOS, Linux). The installation should be simple enough when following the guide above, so we will not go over it in great detail. Moreover, everything should already be set up on the lab computers. However, should you have any difficulties with installation/setup, feel free to contact me.

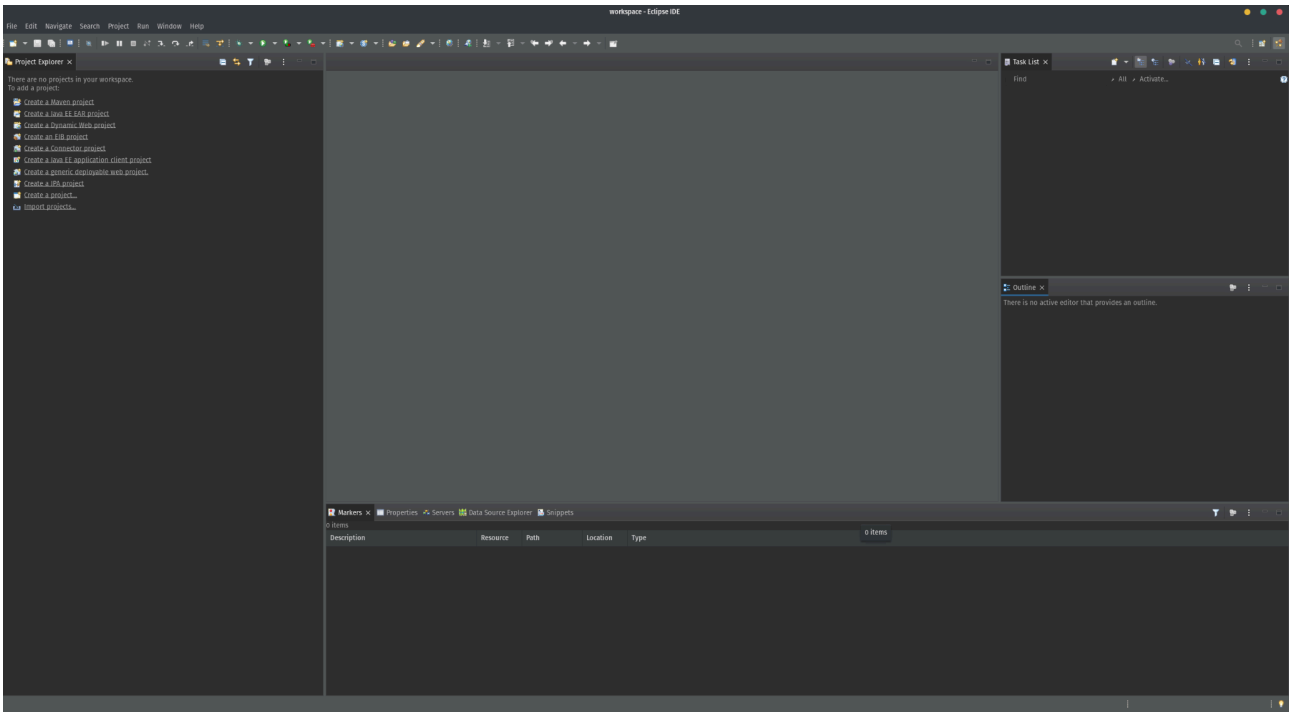
Once both JDK and Eclipse are installed, we are ready to create our project.

Creating a new project

Run your Eclipse IDE. Depending on the way you installed it, it will either be installed as a system program, or you will run it from the folder you unzipped (if you downloaded the ZIP version). On the first run, Eclipse will ask you to specify the default *workspace directory*, where all your projects will be saved. You can specify the workspace as the *default*, if you do not want to be asked this question on every run.



The IDE will now be launched. You will be greeted by a welcome screen on every run, which you can turn off (uncheck “Always show Welcome at startup”). When you close the Welcome screen, you should see a screen similar to this:



Using Maven

We can create a new Java project in a few different ways. One of the most popular approaches to modern Java development is via **Maven**. Maven is a popular open-source build tool developed by the Apache Group to build, publish, and deploy Java projects.

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal, Maven deals with several areas of concern:

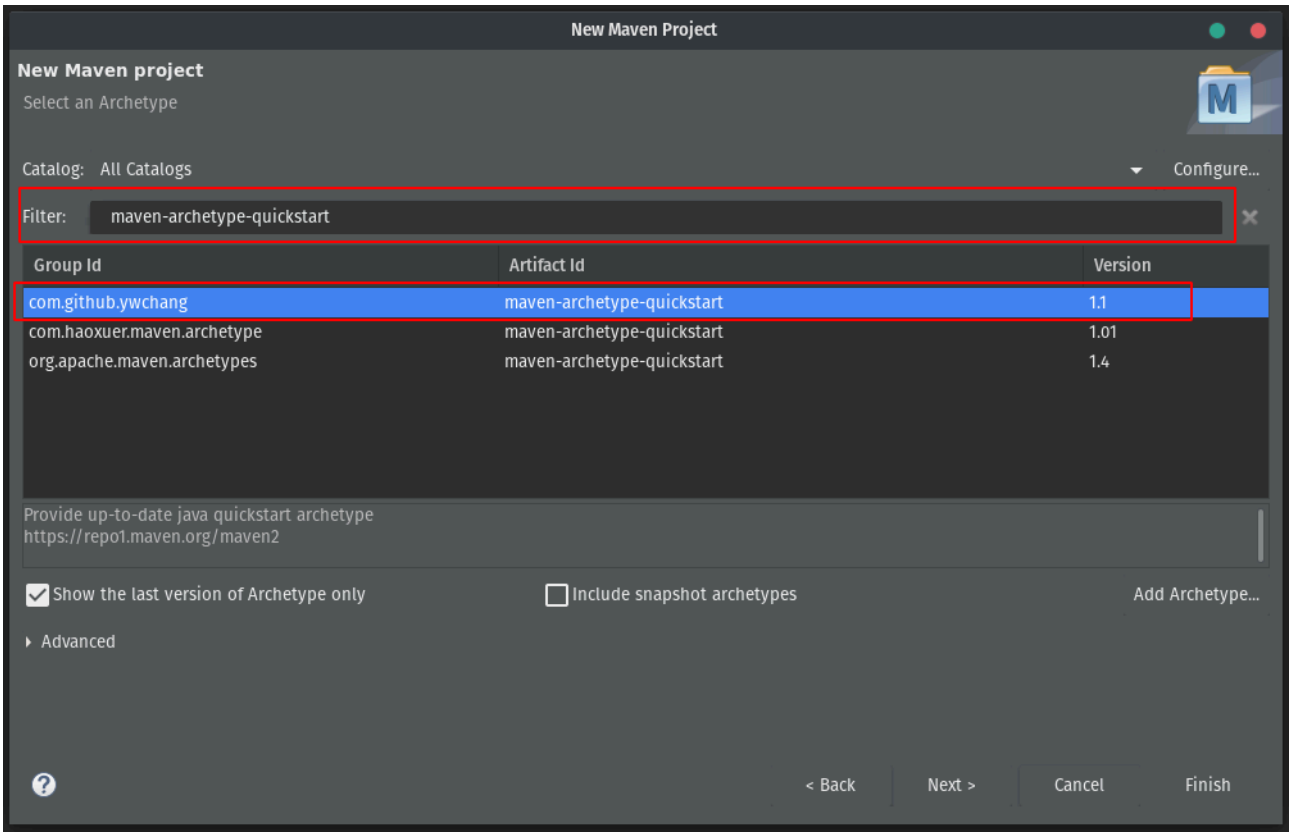
- making the build process easy
- providing a uniform build system
- providing quality project information
- encouraging better development practices

Maven is not IDE-specific; you can install it as a standalone application, and use it from the terminal (console) in a variety of operating systems. If you want to install a standalone Maven binary, you can follow the official instructions. Of course, a lot of popular Java IDEs - Eclipse included - offer built-in support for Maven.

In order to create a new Maven **project**, in the menu go to **File > New > Maven Project**. In the window that opens up, click **Next** (no need to change any values).

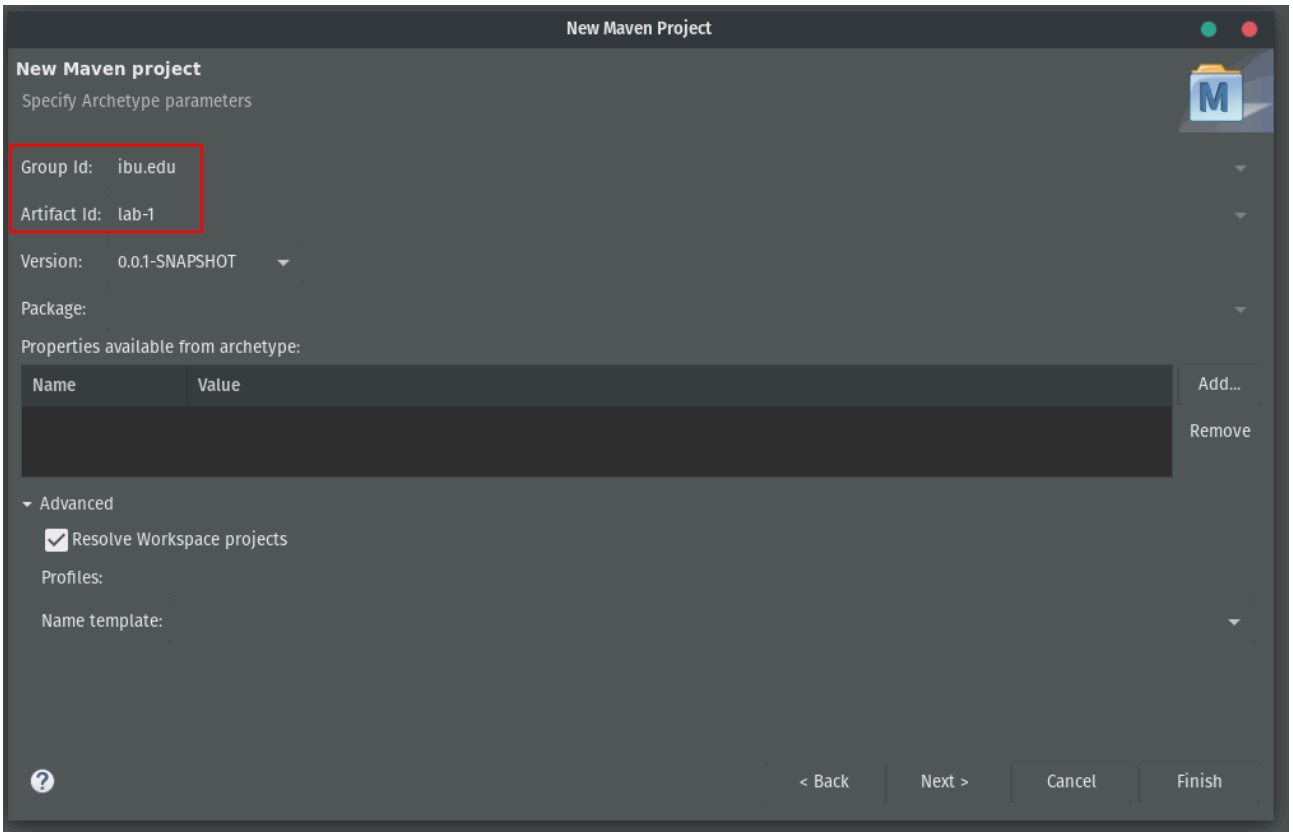
In the next window, you will need to select your **project archetype**. Simply put, an **archetype** is a Maven project *templating toolkit*. An archetype is defined as an original pattern or model from which all other things of the same kind are made. Maven archetypes provide a consistent way of

generating Maven projects. Using archetypes provides a great way to enable developers to get started with their projects quickly and in a consistent way, depending on their needs. In the **Filter** section, search for the **maven-archetype-quickstart** by com.github.ywchang, version 1.1[1], which will provide a template for a basic Maven project (note: it might take some time to load the archetype list).



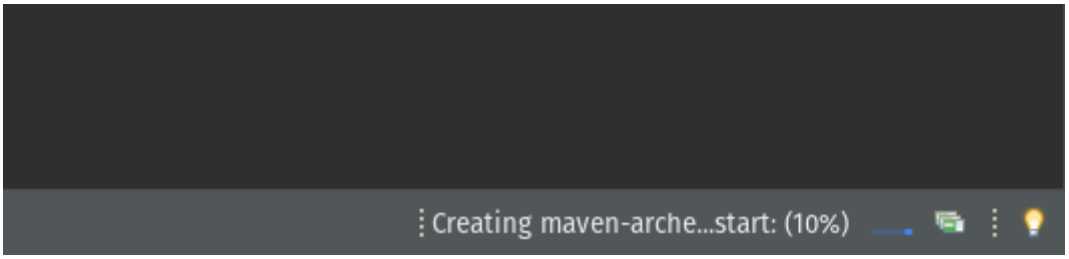
Once you select the archetype, click **Next**. In the following window, you need to define your **group ID** and **artifact ID**. A **group ID** uniquely identifies your project across all projects. You can think of it as a common "domain name" or "package" for all your projects. Group IDs are especially important if you are planning to later publish your project/library on the Maven central repository. The **artifact ID** represents the *concrete name* of your project.

[1] We're not choosing the "official" version 1.4 since it contains outdated JDK and JUnit references. Of course, you can edit the dependencies later on, but this will make it easier for us.



After specifying the group and artifact ID, confirm the creation of the project by clicking on the **Finish** button.

If you have just installed Maven, it may take a while on the first run. This is because Maven is downloading the most recent artifacts (plugin JARs and other files) into your local repository. You can track the progress in the bottom right corner of Eclipse IDE.



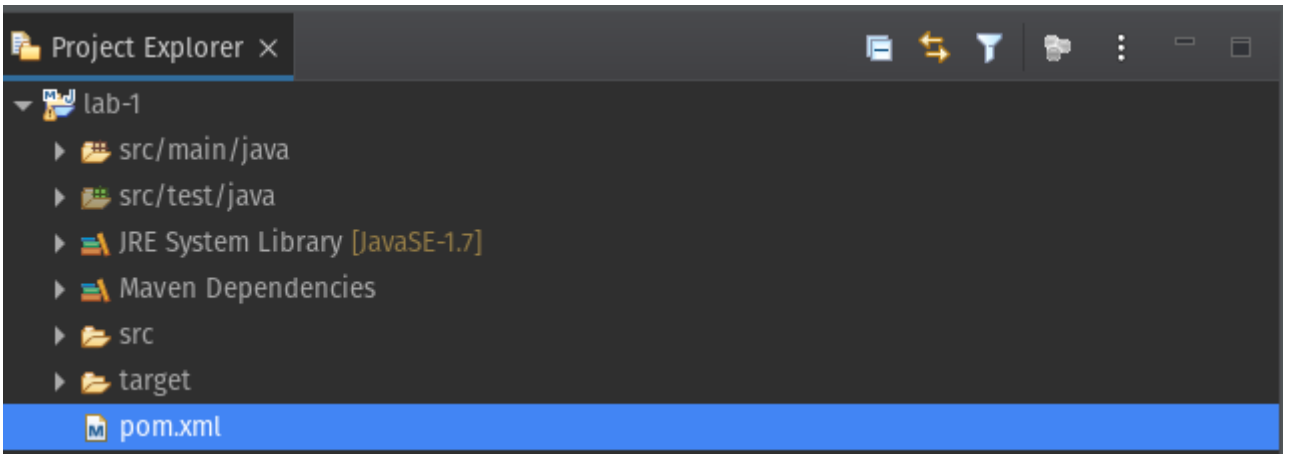
The explanation above details how you can create a Maven project from inside Eclipse. But, as mentioned earlier, Maven can also be run from the terminal, making it IDE-independent. The equivalent command for the process above would be:

```
mvn archetype:generate -DgroupId=ibu.edu -DartifactId=lab-1
-DarchetypeGroupId=com.github.ywchang
```

```
-DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.1
-DinteractiveMode=false
```



If the project was created successfully, you should see it in the left-side panel. Clicking on the downward arrow reveals the structure of the project. You will write your code inside the **src** folder.



The **src/main/java** directory contains the project source code, where you will write your code. The **src/test/java** directory contains the *test source* (more on that later on), and the **pom.xml** file is the project's **Project Object Model**, or POM.

The **pom.xml** file is the core of a project's configuration in Maven. It is a single configuration file that contains the majority of information required to build a project in just the way you want. The POM is huge and can be daunting in its complexity, but it is not necessary to understand all of the intricacies just yet to use it effectively. The most important (and interesting) part for us is the **<dependencies>** structure, which contains all the dependencies (additional libraries) used by our project - in our example, the JUnit testing framework.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

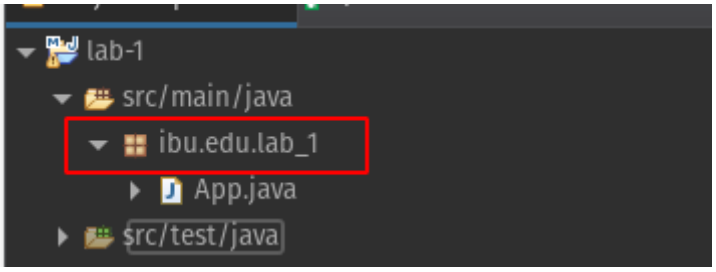
  <groupId>ibu.edu</groupId>
  <artifactId>lab-1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>lab-1</name>
  <url>http://www.example.com</url>

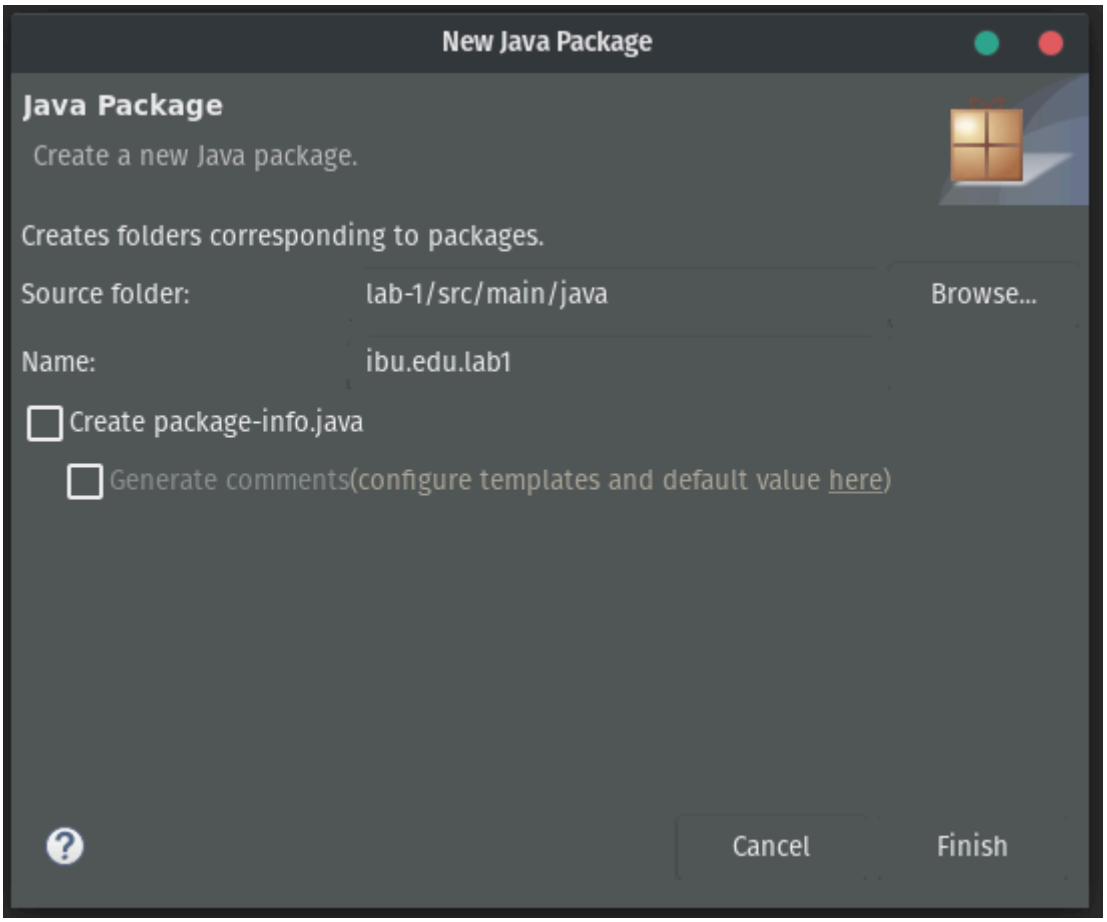
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.7.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <version>5.7.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

If you expand the src/main/java directory, you'll notice a **package** structure.



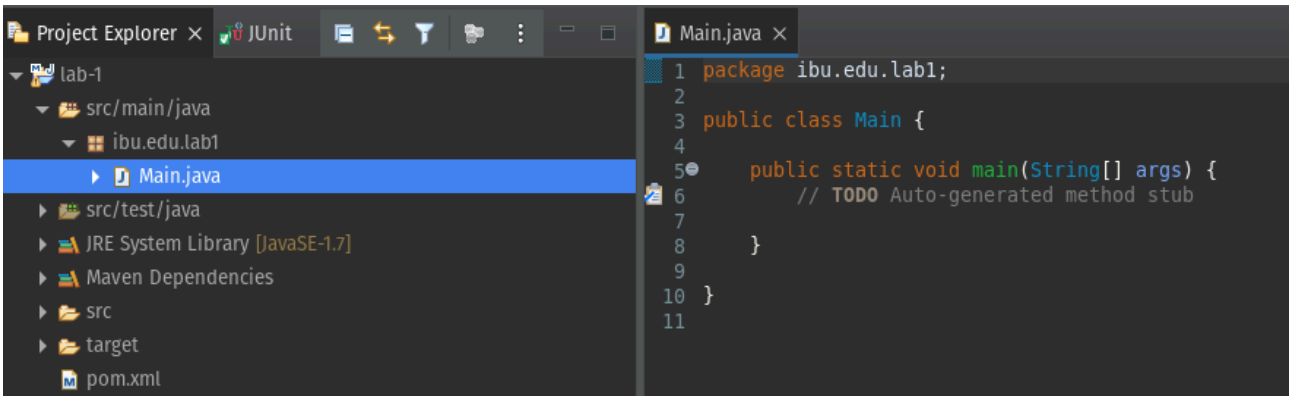
A package in Java is equivalent to a folder in a file management system, and it is used to group related classes, avoid name conflicts, and to write a better maintainable code [2]. The Maven archetype has already created a basic package structure for you, based on your group and artifact ID. Of course, you can also create your own packages and modify existing ones. To create a package, *right click* on the starting location of the package (either src/main/java or an already existing package to create a sub-package) and choose **New > Package**. In the resulting window, specify the package name. The package name can be anything, but it is usually named as something descriptive, and individual words are separated with dots.



In the underlying file hierarchy (which we can see by right clicking the package, and then choosing Show In > System Explorer), this means that we are creating a folder **ibu**, which contains the folder **edu**, which contains **lab1**. This is where we will write our *classes*.

With the initial package created, we can now write a **main**, or **driver class.** Since Java is an object-oriented language, we need to write an initial class which will execute the main program. In object-oriented programming, a **class** is a blueprint for creating **objects** (particular instances of classes), providing initial values for *state* (member variables or attributes) and implementations of *behavior* (member functions or methods) [3]. We will deal with classes and objects in more detail during the next lectures (especially once we delve into creating data structures), but for now, it is enough that you know that classes *group related parts of code*.

To create a new class, *right click* on the package, and select **New > Class**. In the new window, specify the class name (Main, in our example). By default, this process will create a new empty class. If you also want a stub **main method** to be created as well, check the option *public static void main (String[] args)*. You are now ready to write your code.



Writing the program code

In the newly created class, you can see the **main** method. The main method represents a function that serves as the entry point of your program, from which all other code will be executed (note: in OOP, function = method). The method is prefixed by *public static void*, which means:

- *public*: this method is publicly accessible (OOP concept)
- *static*: the method is shared across all instances of an object (OOP concept)
- *void*: return type of the method (as in C/C++, ...)

All methods written inside the main class (the class with your main method) *need to be* public and static. The return type depends on what the function does, and what you expect to get back from it.

To implement the solution to our lab task, we will define *three methods* inside the Main class, which we will utilize inside the main method:

1. *readNumbers*: read the number list from a textual file
2. *findMax*: find the maximum integer from an array of integers
3. *writeToFile*: write the maximum integer (obtained by the previous method) to a new file

We will be working with the *numbers.txt* file which contains 100 randomly generated numbers from 0 to 1000. The file is available on LMS.

We will begin by writing the **readNumbers** method. This method returns *an array of integers*, and expects the *integer file path* as its input parameter.

```java
/* Read the integer list from a file in the given path */
public static int[] readNumbers(String path) throws FileNotFoundException {
    int[] numbers = new int[100];
    Scanner scanner = new Scanner(new File(path));
    int i = 0; // counter
    while (scanner.hasNextLine()) {
        numbers[i] = Integer.parseInt(scanner.nextLine());
        i++;
    }
    scanner.close();
    return numbers;
}
```

If you have previous experience with C-based languages, you can easily notice the similarities in Java. Much of the code follows the same patterns and logic, with Java having some more complex built-in functions and data types. Moreover, Java shares most of the data types with other C-family languages (int, string, double, float, char, ...).

In the function definition, you can notice the "*throws FileNotFoundException*" part: since the method will be working with files, it needs to be able to handle missing file exceptions. In the first line of the function, we define *an array* that will hold the numbers from the file. In Java, arrays are declared using the following syntax: **type[] name = new type[100]**, where *type* represents the data type (int, char, ...) and *[100]* the number of positions we want to reserve in the array. Afterwards, we define a *Scanner* object, used to read input. Scanner is just one of the many possible ways of reading a file in Java (Scanner, FileReader, BufferedReader, ...). Do not be afraid to look up new methods and experiment with them; you will often need to.

The next few lines are mostly self-explanatory. Using the scanner, we iterate over the file as long as there are lines remaining. We convert every line (as the line is read as a *String* object) to an integer and add them to our array (note: it is possible to immediately get the integer values, by using *hasNextInt()* and *nextInt()* scanner methods; however, I wanted to showcase the method of reading in an entire line, as you will be working with more complex file formats in the future). Lastly, we close the scanner and *return* the newly filled numbers array[2].

Note: While writing the code, you may notice that certain parts of it are underlined with a red squiggly line. This (most probably) happens because many of the methods/classes Java uses (Scanner, File, ...) are packages which need to be **imported** into your class. In order to import a package/class in Eclipse IDE, start writing the package/class name and press <CTRL + Space>. This will open an autocomplete list. If there is only one module that matches what you have typed in, it will automatically be imported. If there are more, you will be presented with a shortlist from which you can select the desired module. Once selected, the module will be imported and ready to use.

[2] *readNumbers()* code: https://pastebin.com/9hX9uLby

The **findMax** methods return the *maximum integer* from the input *number array*.

```java
/* Find the maximum integer */
public static int findMax(int[] numbers) {
    int max = numbers[0];
    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max;
}
```

This is a pretty basic method of finding a maximum number in an array. We begin with the assumption that the first array element is the maximum. Next, we iterate over the remaining array elements (until we reach the length of the array) and if the current element is greater than the current maximum, the element becomes the new maximum. In the end, we return the maximum integer. As you can see, the loop structure(s) in Java are mostly identical to those in C/C++[3].

In order to persist the maximum integer to the disk, we will define the **writeToFile** method, which *does not return* any data (void), and takes in the *new file path* and *maximum integer* as its input parameters.

```java
/* Write the maximum number to a file */
public static void writeToFile(String path, int maxNumber) throws IOException {
    File file = new File(path);
    FileWriter fw = new FileWriter(file);
    fw.write(Integer.toString(maxNumber));
    fw.close();
}
```

Similarly to the first method, this method throws an IOException, as it works with files. Here, we are using the *FileWriter* object for file creation, but there are many other alternatives (FileWrite, BufferedWriter, FileOutputStream, ...), so feel free to experiment. In order to append an integer to a textual file, we first need to *convert* it into a string. Once the writing process is finished, we close the FileWriter object[4].

With all three crucial methods defined, we can write our **main** method and test the program.

[3] *findMax()* code: https://pastebin.com/yYvYB0tz
[4] *writeToFile()* code: https://pastebin.com/KpCTLVAY

```
public static void main(String[] args) throws IOException {
    System.out.println("Reading numbers...");
    int[] numbers = readNumbers("/home/aldin-sxr/numbers.txt");
    int maxNumber = findMax(numbers);
    System.out.println("Maximum number is: " + maxNumber);
    System.out.println("Writing to file...");
    writeToFile("/home/aldin-sxr/maxNumber.txt", maxNumber);
}
```

The main method begins with printing some introductory text to the console. **System.out.println()** is Java's equivalent of a "print to console" (with a new line) function (cout, printf, print, … in other languages).

- **Hint**: In Eclipse, you can type *sysout*, hit <CTRL + Space> and the autocomplete will fill in the function.

Afterwards, we call the readNumbers method with the the *full file path* to our numbers.txt file (**note**: if you are on Windows, you will need to use \\ as folder/file separators, since a single \ is an escape character) and get an array of integers back. Afterwards, we find the maximum number and print it to the console. When using System.out.println(), you can mix and concatenate strings and data types using the **+ operator**. Lastly, we save the maximum integer to a new file, with our specified path[5].

If you have properly followed all the steps, the full Main.java class should look similar to this[6]:

[5] *main()* code: https://pastebin.com/LkQkMuwc
[6] Full Main.java class code: https://pastebin.com/ZZajSUCL

Main.java

```java
package ibu.edu.lab1;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Main {

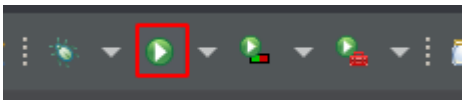
    public static void main(String[] args) throws IOException {
        System.out.println("Reading numbers...");
        int[] numbers = readNumbers("/home/aldin-sxr/numbers.txt");
        int maxNumber = findMax(numbers);
        System.out.println("Maximum number is: " + maxNumber);
        System.out.println("Writing to file...");
        writeToFile("/home/aldin-sxr/maxNumber.txt", maxNumber);
    }

    /* Read the integer list from a file in the given path */
    public static int[] readNumbers(String path) throws FileNotFoundException {
        int[] numbers = new int[100];
        Scanner scanner = new Scanner(new File(path));
        int i = 0; // counter
        while (scanner.hasNextLine()) {
            numbers[i] = Integer.parseInt(scanner.nextLine());
            i++;
        }
        scanner.close();
        return numbers;
    }

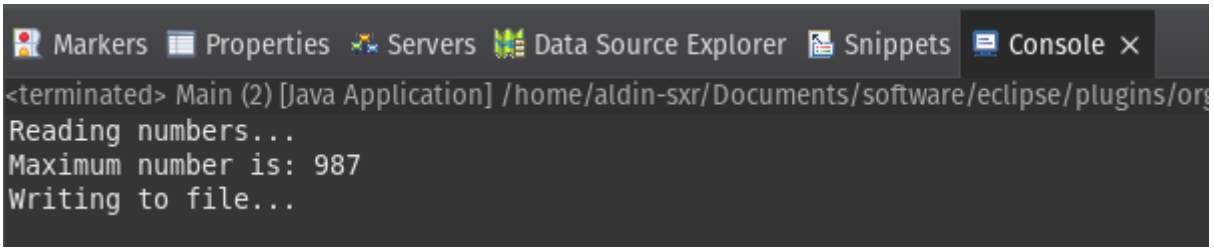
    /* Find the maximum integer */
    public static int findMax(int[] numbers) {
        int max = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] > max) {
                max = numbers[i];
            }
        }
        return max;
    }

    /* Write the maximum number to a file */
    public static void writeToFile(String path, int maxNumber) throws IOException {
        File file = new File(path);
        FileWriter fw = new FileWriter(file);
        fw.write(Integer.toString(maxNumber));
        fw.close();
    }
}
```

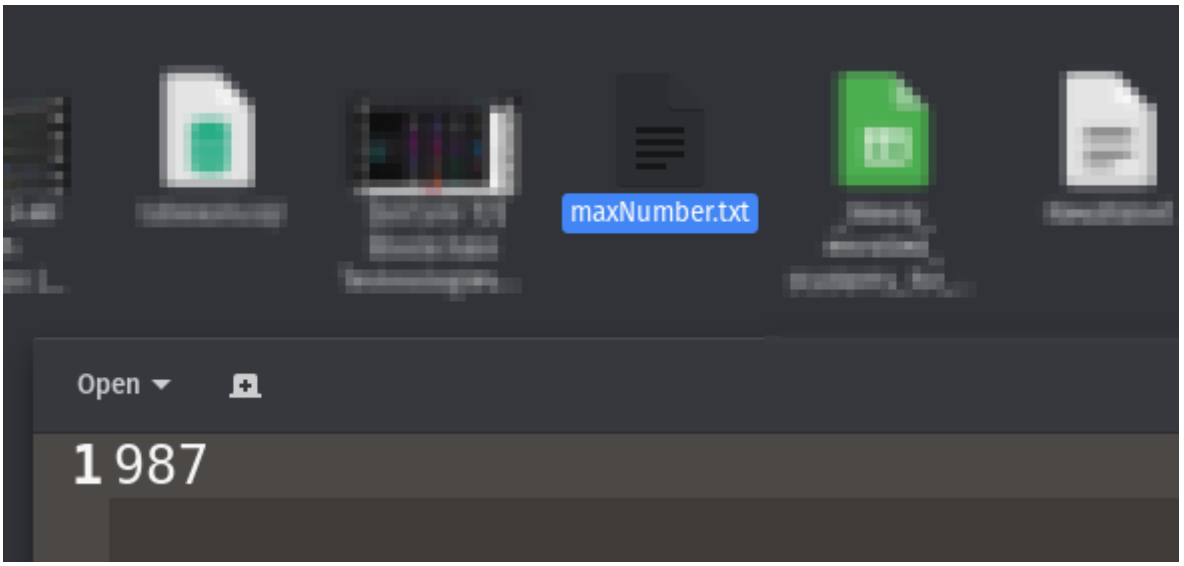
In order to run the program, save the file and click on the **Run** icon in the toolbar below the menu.



Alternatively, you can use the <CTRL + F11> shortcut. The program will run, opening the “Console” tab in the Eclipse IDE, where you can see the console output.



Indeed, the maximum integer in the file was 987 (I checked). If we check the location we specified for the output file, we should also see the number 987 written in it.

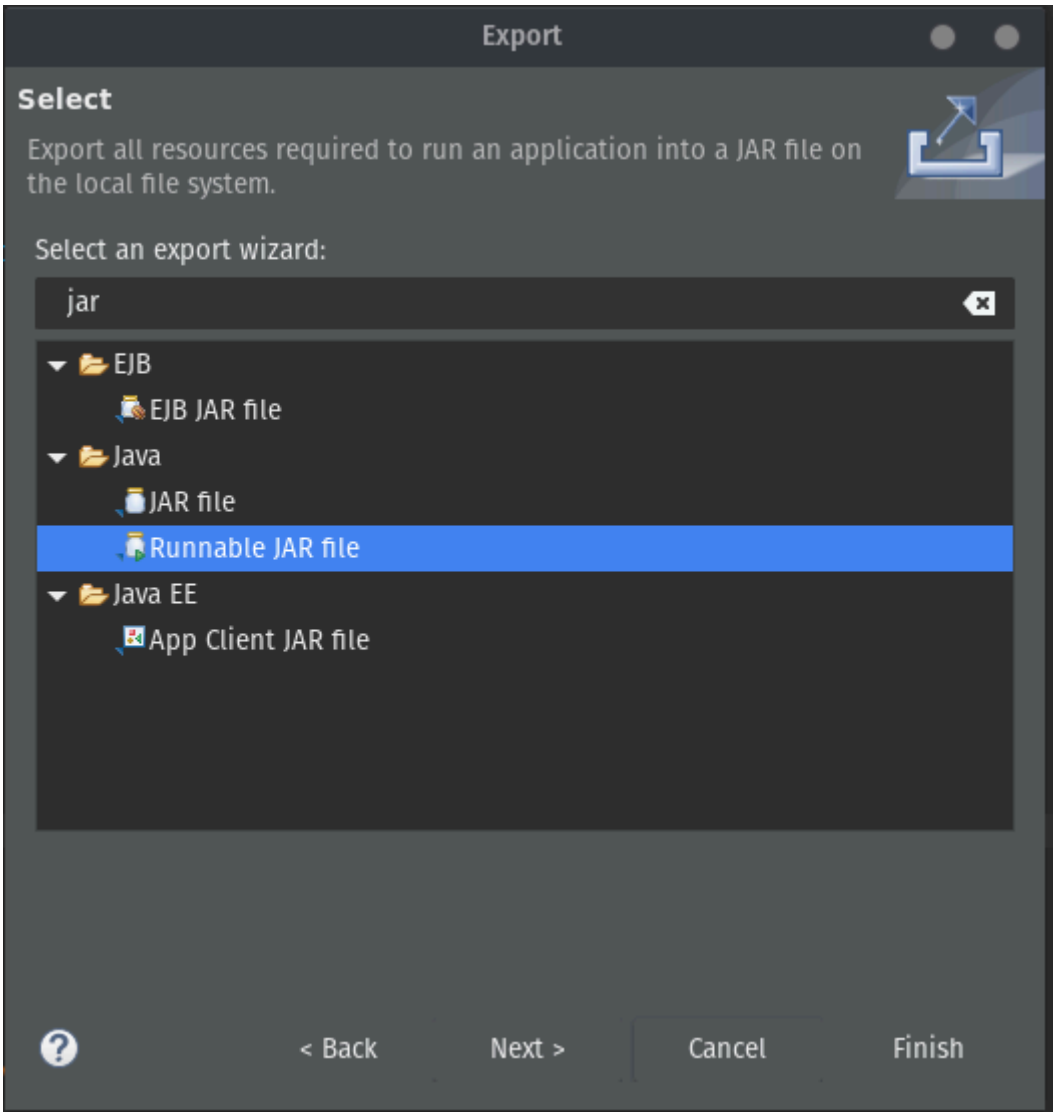


Congratulations on writing a simple Java program.

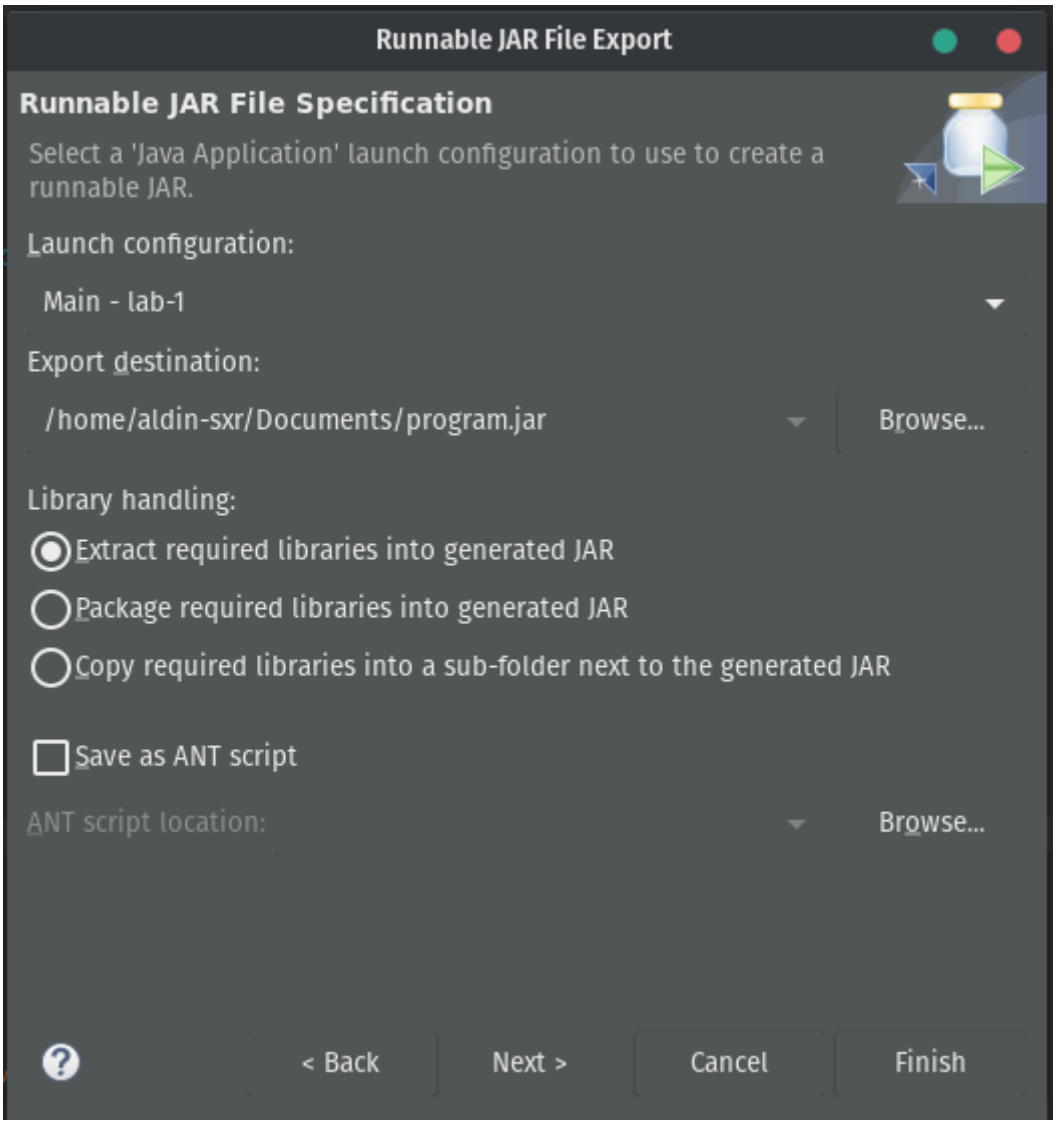
Creating an executable JAR file

Right now, your program can be executed from the Eclipse IDE. Alternatively, you can compile its bytecode using *javac* and then run it. However, we want to create an easily portable and executable file which can be run from any machine or platform that supports Java. To do that, we will generate a **JAR** (**J**ava **AR**chive) file for our project. A JAR is a package file format typically used to aggregate many Java class files and associated resources (text, images, etc.) into a single file suitable for distribution [4].

In the menu, go to **File > Export**. In the resulting window, type in “JAR” and select “Runnable JAR file”.



In the next window, specify the “Launch configuration” by selecting the main class, and “Export destination” by specifying the location and name of the JAR file. After you click **Finish**, the JAR file will be generated at your chosen location.



In order to execute a JAR file, open the system console (cmd/Powershell in Windows, terminal in Unix-based systems) from the folder where the executable file is located and run the command: **java -jar program.jar**, where “program.jar” is the name you gave to the file.

```
aldin-sxr ~ Documents 1 java -jar program.jar
Reading numbers...
Maximum number is: 987
Writing to file...
aldin-sxr ~ Documents
```

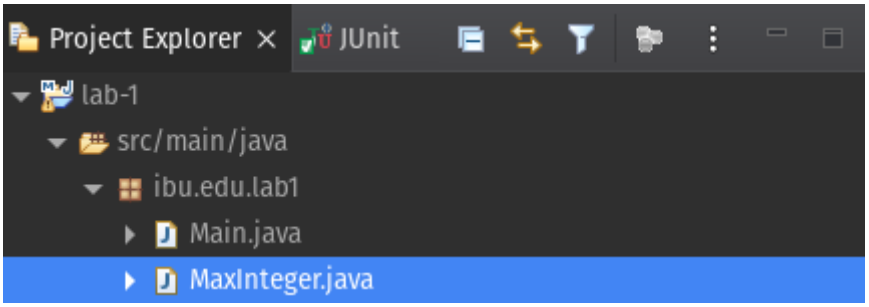
Congratulations on writing a simple and portable Java executable application.

Using a new class

Since we are dealing with a relatively simple program in our example, we could keep all required methods inside the same Main class. However, once our programs grow more complex and we start dealing with advanced data structures, we will need to create additional *classes* to better group and organize our code and related functionalities. Moreover, having related parts of code in separate classes makes it easier to **test** and debug our applications.

In order to get comfortable with creating new classes, we will transfer the three methods (readNumbers, findMax, writeToFile) to a *separate* class, called MaxInteger. Afterwards, we will invoke the MaxInteger class, and its methods, inside the main method of the Main class. You can create the new class in the same way you did the Main class (right click on the package, **New > Class**) and name it appropriately. However, you do not need to create a main method this time around, as there can be *only one* main method per a project.

Once you have created the new class, your project structure should look similar to this:



Now, you can simply copy/cut the methods from the Main class, paste them into the new class, and save the file. Upon pasting, Eclipse IDE should automatically import the required packages, as well. The final MaxInteger class should look like this:

Main.java | MaxInteger.java

```java
package ibu.edu.lab1;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class MaxInteger {
    /* Read the integer list from a file in the given path */
    public static int[] readNumbers(String path) throws FileNotFoundException {
        int[] numbers = new int[100];
        Scanner scanner = new Scanner(new File(path));
        int i = 0; // counter
        while (scanner.hasNextLine()) {
            numbers[i] = Integer.parseInt(scanner.nextLine());
            i++;
        }
        scanner.close();
        return numbers;
    }

    /* Find the maximum integer */
    public static int findMax(int[] numbers) {
        int max = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] > max) {
                max = numbers[i];
            }
        }
        return max;
    }

    /* Write the maximum number to a file */
    public static void writeToFile(String path, int maxNumber) throws IOException {
        File file = new File(path);
        FileWriter fw = new FileWriter(file);
        fw.write(Integer.toString(maxNumber));
        fw.close();
    }
}
```

In order to utilize the new class, simply replace all the references to the original methods in the Main class with the methods from the MaxInteger class. You can do that by prefixing method names with *MaxInteger.* This means that the methods "belong" to the said class[7].

[7] MaxInteger.java full code: https://pastebin.com/vC04YCyL

```
public static void main(String[] args) throws IOException {
    System.out.println("Reading numbers...");
    int[] numbers = MaxInteger.readNumbers("/home/aldin-sxr/numbers.txt");
    int maxNumber = MaxInteger.findMax(numbers);
    System.out.println("Maximum number is: " + maxNumber);
    System.out.println("Writing to file...");
    MaxInteger.writeToFile("/home/aldin-sxr/maxNumber.txt", maxNumber);
}
```

If you re-run the program, you should achieve the same output as before, which means that you successfully implemented the methods inside the new class.

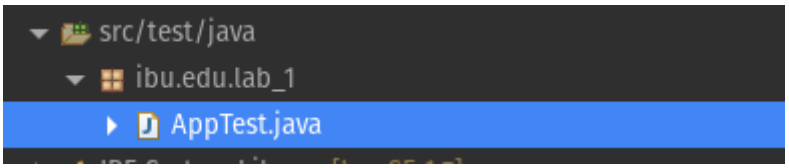
Testing your code with JUnit

One of the reasons we moved our code to a separate class is to make it **testable**. Even though our current program is relatively simple, it is still generally easy to make mistakes while coding. With larger and more complex programs, the probability of accidentally sneaking in a “game-breaking” bug is much higher. Therefore, careful software testing is more of a necessity, rather than a “nice to have” workflow.

There exist various levels of software testing, based on the scope and complexity of testing. One of the most immediate (and useful) testing flows a developer can implement is **unit testing**. Unit testing is a level of software testing where individual units/components (smallest testable parts) of a software are tested, with the purpose of validating that each unit of the software performs as designed [5]. In object-oriented programming, those smallest units are the *classes*.

We can generally discern between two main types of tests in unit testing: **positive** and **negative tests**. *Positive tests* assert that a functionality is performing and behaving correctly, as intended by the developer. For example, a sum(2, 2) function with the return value of 4 passes the positive test. On the other hand, a *negative test* asserts that the application can gracefully handle invalid input or unpredicted user performance. The fundamental purpose of executing negative testing is to ensure the application's stability against the effects of different variations of improperly validated dataset.
A question might arise here: why do we need negative tests if we showcased that a positive test passes? It is entirely possible that we might have a bug in our code which makes it seem like our methods pass the tests, when they, in fact, do not. Imagine that you have a textbox that should in theory only accept numbers, but you did add the proper validation. Performing positive tests will show that the field is properly working with number data. However, if a user tries inputting text data, the application might behave in unexpected ways. The negative tests are written to “catch” these kinds of bugs. With a proper negative test that attempts to insert text data into the numeric field, you can catch this issue before it reaches production and actual users. Therefore, it is a good idea to always write both positive and negative tests for your functionalities.

Java offers **JUnit** as its unit testing framework. Moreover, JUnit support comes bundled with the Eclipse IDE, making it easy to set up in the project. In Maven projects, test code is stored in the **src/test/java** folder.

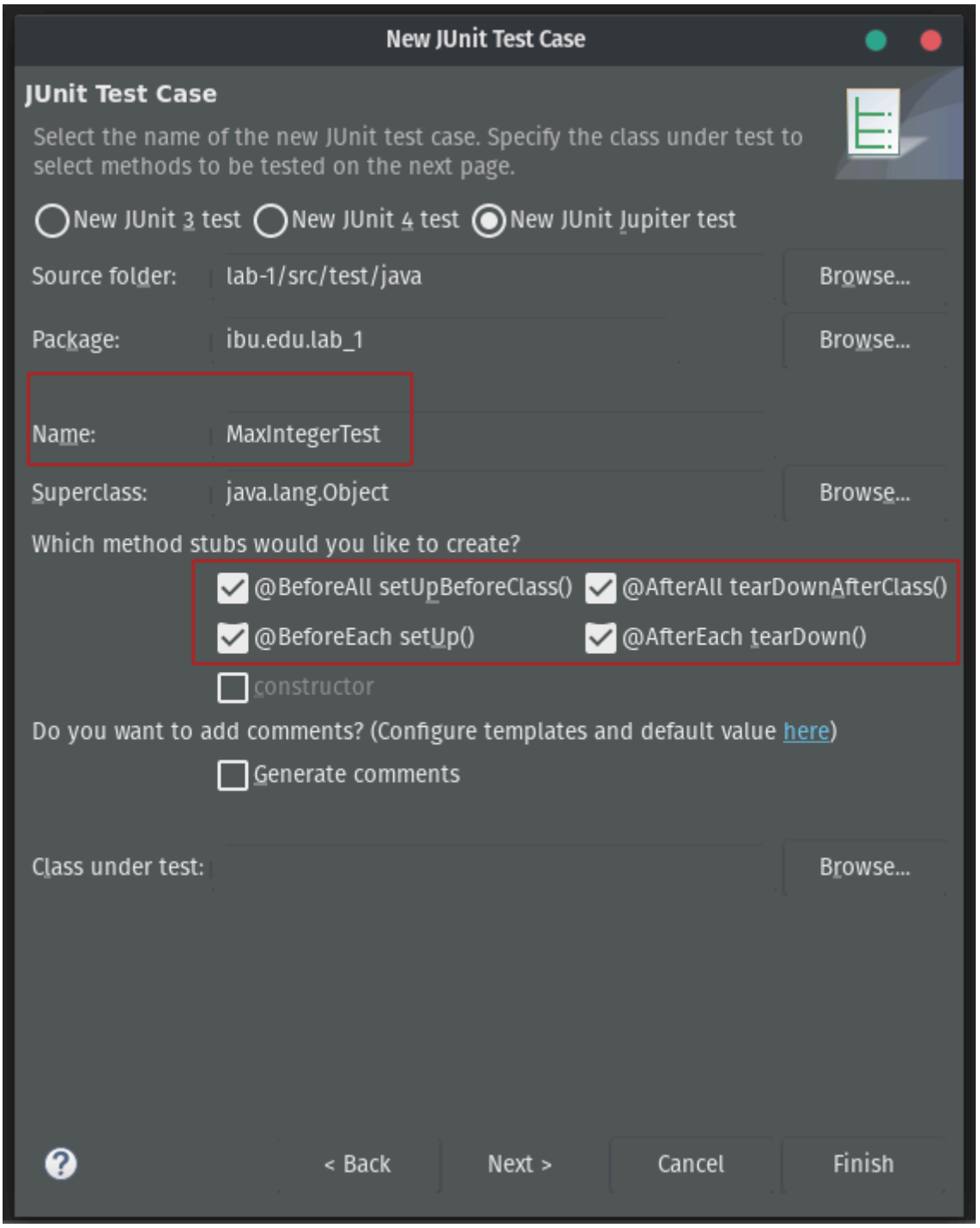


In order to create a **test case** (a collection of tests) for the project, right click on the package, and select **New > JUnit Test Case** (you can delete the auto-generated test file). If you do not see the "JUnit Test Case" option, click on **Other**, and in the window that appears search for "junit".

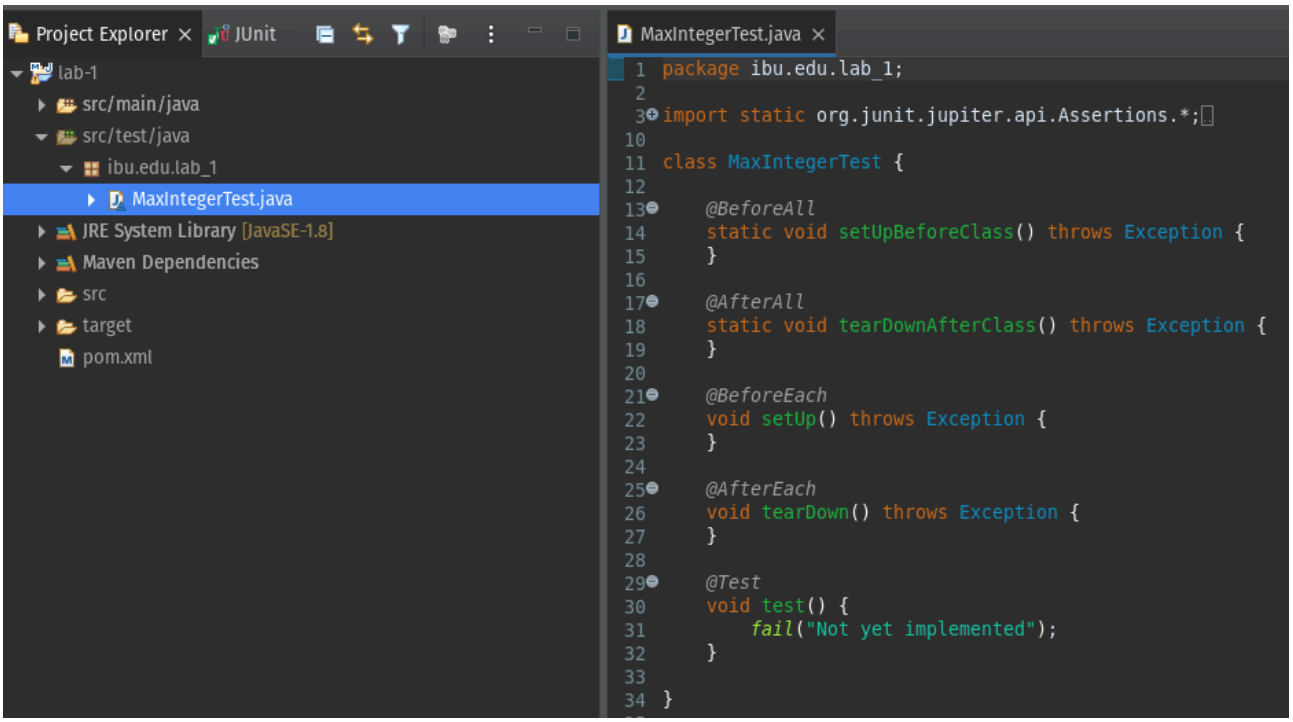
In the new window, select **New JUnit Jupiter Test** (if not already selected) and choose a test case name (it should be descriptive and based on the functionality you are planning to test; MaxIntegerTest, in our example). Below the naming field, you have an option to select several method stubs, which can be useful when setting up tests. The available methods are as follows:

- *setUpBeforeClass()*: define actions to be taken/executed before testing is started
- *setUp()*: define actions to be taken/executed before each individual test
- *tearDown()*: define actions to be taken/executed after each individual test
- *tearDownAfterClass()*: define actions to be taken/executed after all individual tests have finished; clean up the testing resources

Even though we will not be using all of these advanced methods in our simple tests, we will select all of them, so you can see an example of how they appear in code.



With this process completed, a test case file will now be added to our project structure. Upon opening the file, we can see the methods we specified previously, as well as a placeholder *test()* method, which can be used to write our first test.



At first glance, the test case file looks like a regular Java class, with its methods and definitions. However, we can notice special words (@Test, @BeforeEach, @AfterEach, ...), called **annotations** above each method. Annotations are used to configure test methods, and specify what kind of a function the method performs within the JUnit testing framework [6]. We are mostly interested in the **@Test** annotation, which signifies that its corresponding method contains a test. Whenever we add new test methods, we need to preface them with the @Test annotation.

Assertions are another crucial aspect of the JUnit framework. Assertions are in-built methods which are used to "assert" that a value provided by the tested functionality *matches*, or *does not match*, the value expected by the developer. There are numerous assertion methods available in JUnit, making it possible to test and compare equalities, inequalities, whether a value is true or false, the number of returned elements, whether a method will fail with an exception, etc. The default generated test() method contains a placeholder *fail()* method, which automatically fails the test when executed. In this lab session, we will showcase a few simple assertions. However, a full list of available assertions (and annotations) is available via the official JUnit documentation. Feel free to explore it and experiment with the possibilities.

We will create two simple test methods for our project; a positive test which showcases that the application returns the correct maximum integer, and a negative test to prove it does not return an incorrect result. Let us start with the testing setup by creating *a class variable* to hold the number array. Afterwards, in the *setUpBeforeClass()* method we will load the data into the number array. In the *tearDownAfterClass()* method, we set the number array to a null value.

```
class MaxIntegerTest {
    static int[] numbers;

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        numbers = MaxInteger.readNumbers("/home/aldin-sxr/numbers.txt");
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
        numbers = null;
    }
```

Using these two setup methods, we ensure that our testing data is loaded only once, before testing begins, and that used resources are released once all tests are completed.

Now, we can move on to writing actual tests. Let us start with the positive test.

```
@Test
void testFindsCorrectMaximum() {
    int maxNumber = MaxInteger.findMax(numbers);
    assertEquals(987, maxNumber);
}
```

This is a simple test which uses the **assertEquals** assertion methods to test whether the expected value (987) matches the value actually returned by the findMax() method. In all assertion methods, the format is: **assert<Operation>(un/expectedValue, actualValue)**, where <Operation> is the type of assertion (equal, not equal, count, etc.)

The second test is the negative test, which aims to show that the method does not produce incorrect results.

```
@Test
void testDoesNotFindIncorrectMaximum() {
    int maxNumber = MaxInteger.findMax(numbers);
    assertNotEquals(42, maxNumber);
}
```

The negative test looks very similar to the positive test, with the difference of using the **assertNotEquals** assertion method and a wrong maximum value. With all of the previous test pieces implemented, the test case class should look like this[8]:

[8] Full MaxIntegerTest.java code: https://pastebin.com/vf5jhuNS

Main.java | MaxInteger.java | MaxIntegerTest.java

```java
package ibu.edu.lab_1;

import static org.junit.jupiter.api.Assertions.*;

class MaxIntegerTest {
    static int[] numbers;

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        numbers = MaxInteger.readNumbers("/home/aldin-sxr/numbers.txt");
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
        numbers = null;
    }

    @BeforeEach
    void setUp() throws Exception {
    }

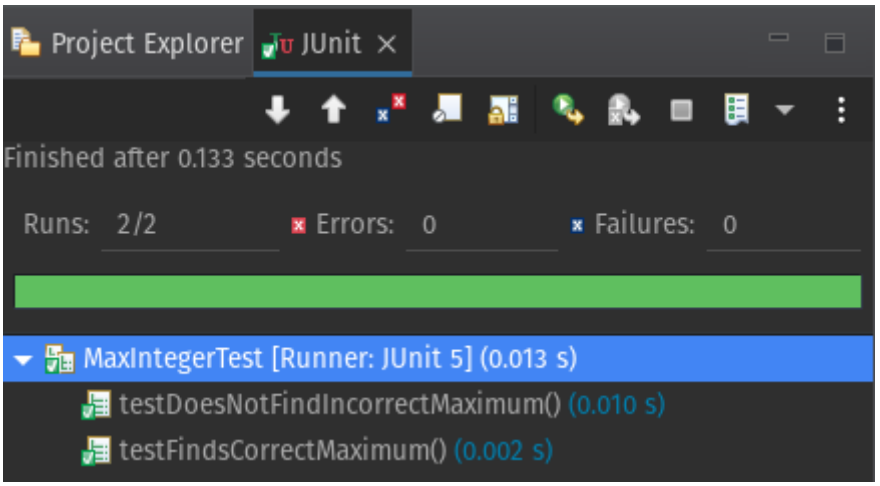
    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testFindsCorrectMaximum() {
        int maxNumber = MaxInteger.findMax(numbers);
        assertEquals(987, maxNumber);
    }

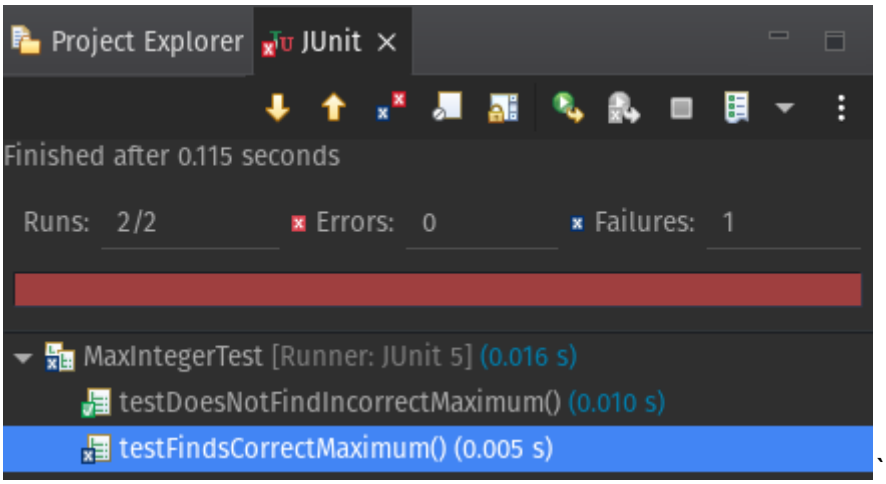
    @Test
    void testDoesNotFindIncorrectMaximum() {
        int maxNumber = MaxInteger.findMax(numbers);
        assertNotEquals(42, maxNumber);
    }

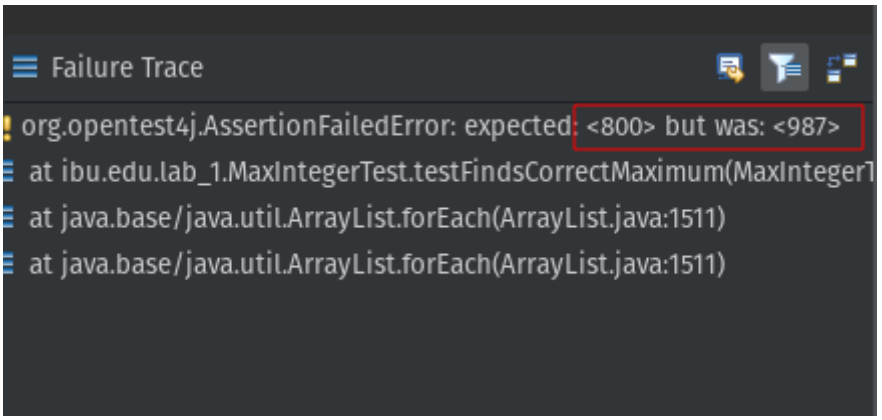
}
```

In order to see if our tests pass and our application works correctly, we need to run the test suite by either clicking on the **Run** (green play) button in the toolbar, or using the shortcut <CTRL + F11>. Once the tests are executed, you will see the results in the left Eclipse IDE panel.



As evidenced by the test results, both of our tests passed without issues. Now, if we deliberately change the expected value in the positive test to e.g. 800, the test should fail and we should see the reason why (the difference between the expected and actual values) in the Failure Trace.





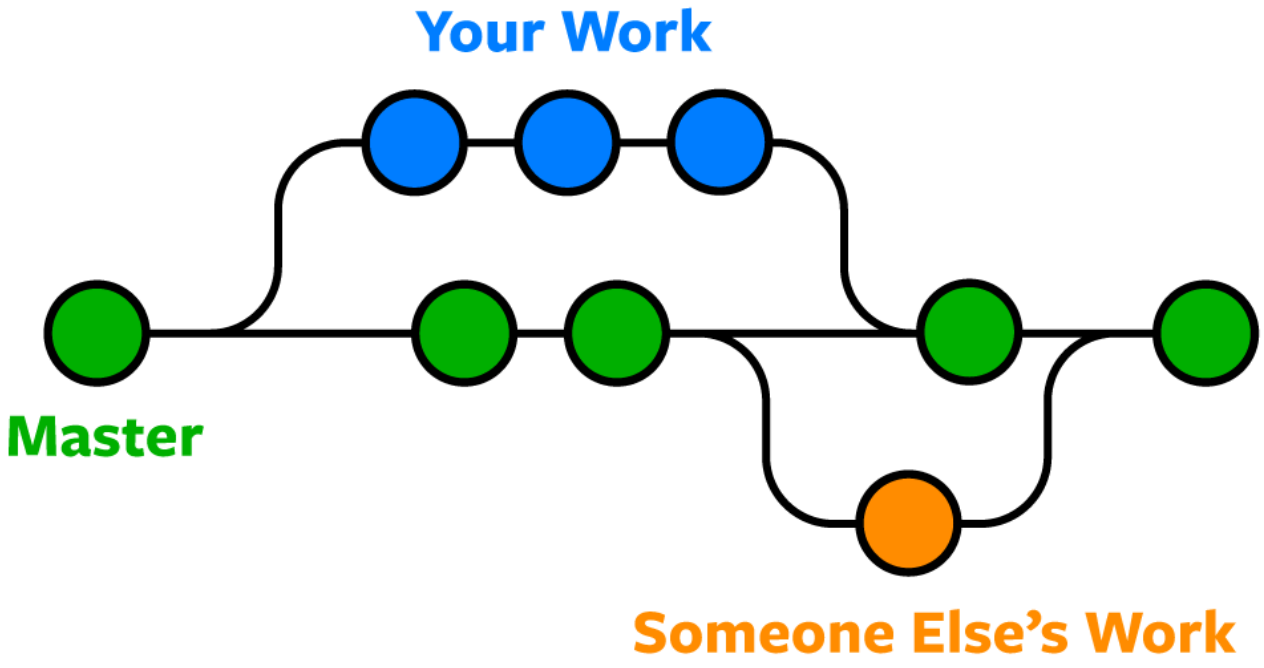
Congratulations on writing your first JUnit test suite.

Keeping up with the changes

As your projects and applications get more complex, you might end up adding more developers to your workflow, to work on separate features in parallel. Or, you might get faced with a critical error and need to quickly revert to a working version. Rather than having to keep countless copies of your project at different stages, or sharing files with other developers via USB or cloud storage, you can use a **version control system**.

What is Git

Version control systems are software tools that help software teams manage changes to source code over time. As development environments have accelerated, version control systems help software teams work faster and smarter. **Git** is the most commonly used version control system. Git *tracks the changes* you make to files, so you have a record of what has been done, and you *can revert* to specific versions should you ever need to. Git also makes collaboration easier, allowing *changes by multiple people* to all be merged into one source. Regardless of whether you write code that only you will see, or work as part of a team, Git will be useful for you.



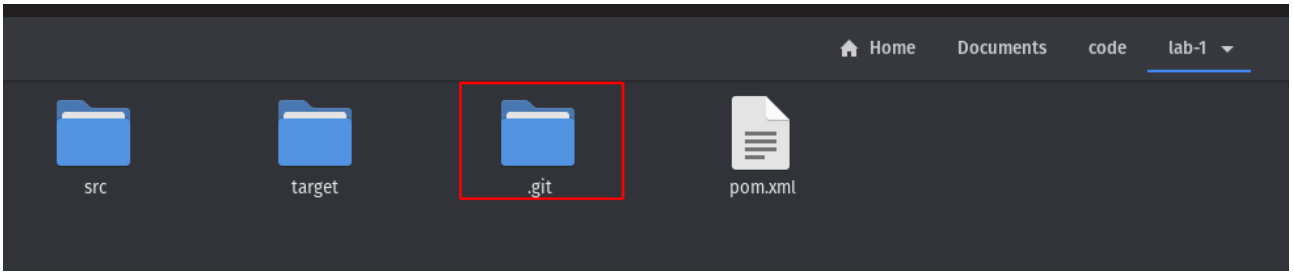
Git is software that runs locally. Your files and their history are stored on your computer. You can also use online hosts (such as GitHub, Gitlab or Bitbucket) to store a copy of the files and their revision history. Having a centrally located place where you can upload your changes and download changes from others enables you to collaborate more easily with other developers. Git can automatically merge the changes, so two people can even work on different parts of the same file and later merge those changes without losing each other's work.

To download and install Git, visit the following link for instructions on different operating systems.

Git repositories

A **Git repository** (or repo for short) contains all of the project files and the entire revision history. You can take an ordinary folder of files, and tell Git to make it a repository. This creates a **.git** subfolder, which contains all of the Git metadata for tracking changes.

On Unix-based operating systems such as Linux and macOS, files and folders that start with a period (.) are hidden, so you will not see the .git folder unless you show hidden files. You might be able to see it in some code editors.



From the terminal, a new repository is initialized inside a folder using the command `git init`.

```
aldin-sxr … > ds > workspace > lab-1 git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/aldin-sxr/Documents/university/asssist/2021-2022/ds/workspace/lab-1/.git/
aldin-sxr  master … > ds > workspace > lab-1
```

Staging & committing files

Git keeps a list of changes made to files. Each recorded change to a file or set of files is called a **commit**. Before we make a commit, we must tell Git what files we want to commit. This is called **staging** and uses the `add` command.

A common question is - why can't we just commit the file directly - why is staging needed? Imagine you are working on two files, but only one of them is ready to be committed to the "master" version. You do not want to be forced to commit both files, just the one that is ready. Therefore, we first *add* the files which are ready to the *staging area*, and then we *commit the files* that have been staged. Each commit operation is also accompanied by a short message stating what change had been made. From the terminal, this would be the underlying commands:

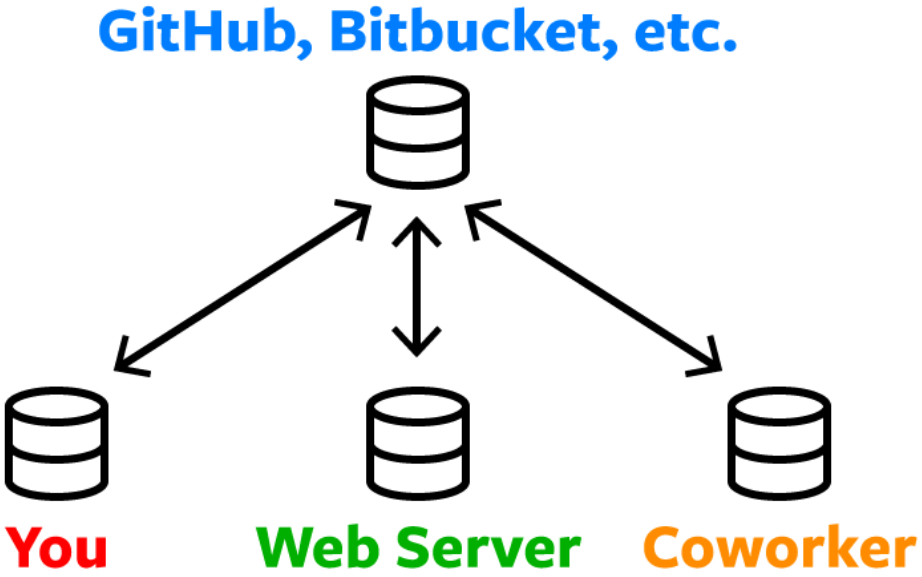
`git add .` (the dot **.** stands for "all files"; you can also specify a single file, folder or combination)

`git commit -m "Your message"`

```
aldin-sxr  master  …  ds  workspace  lab-1  git add .
aldin-sxr  master  …  ds  workspace  lab-1  git commit -m "Initial commit."
[master (root-commit) 41b4098] Initial commit.
 15 files changed, 260 insertions(+)
 create mode 100644 .classpath
 create mode 100644 .project
 create mode 100644 .settings/org.eclipse.core.resources.prefs
 create mode 100644 .settings/org.eclipse.jdt.core.prefs
 create mode 100644 .settings/org.eclipse.m2e.core.prefs
 create mode 100644 pom.xml
 create mode 100644 src/main/java/ibu/edu/lab1/Main.java
 create mode 100644 src/main/java/ibu/edu/lab1/MaxInteger.java
 create mode 100644 src/test/java/ibu/edu/lab_1/MaxIntegerTest.java
 create mode 100644 target/classes/META-INF/MANIFEST.MF
 create mode 100644 target/classes/META-INF/maven/ibu.edu/lab-1/pom.properties
 create mode 100644 target/classes/META-INF/maven/ibu.edu/lab-1/pom.xml
 create mode 100644 target/classes/ibu/edu/lab1/Main.class
 create mode 100644 target/classes/ibu/edu/lab1/MaxInteger.class
 create mode 100644 target/test-classes/ibu/edu/lab_1/MaxIntegerTest.class
aldin-sxr  master  …  ds  workspace  lab-1
```

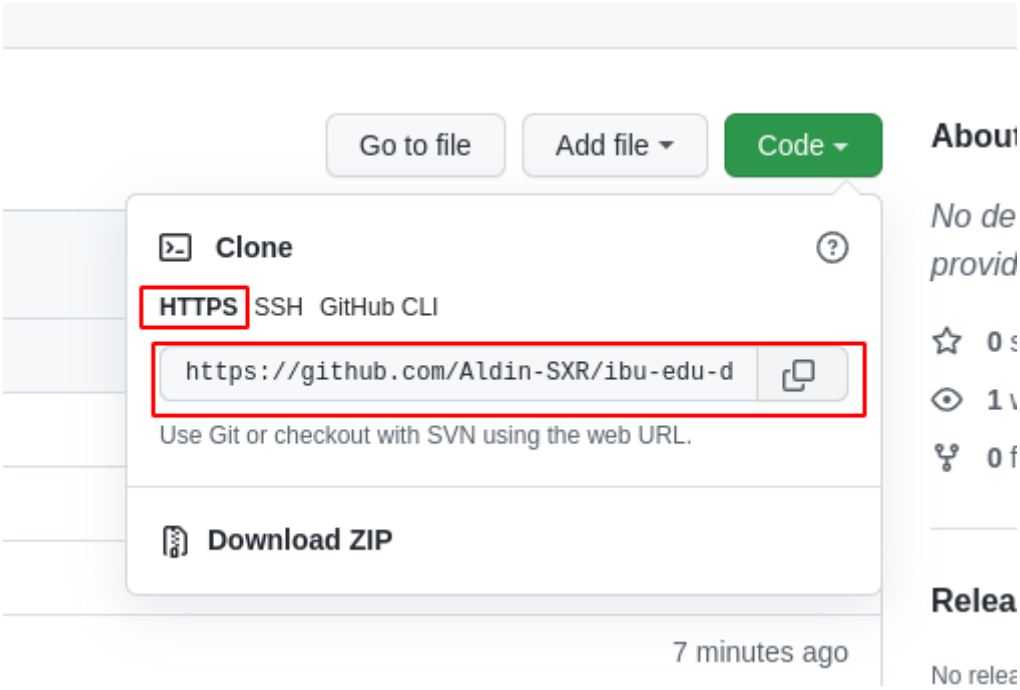
Remote repositories (GitHub, Gitlab, Bitbucket)

Storing a copy of your Git repo with an online host (such as GitHub, Gitlab or Bitbucket) gives you a *centrally located place* where you can upload your changes and download changes from others, letting you collaborate more easily with other developers. After you have a remote repository set up, you **upload (push)** your files and revision history to it. After someone else makes changes to a remote repo, you can **download (pull)** their changes into your local repo.



Downloading a repository

Downloading a remote repository for the first time is known as **cloning**. To clone a remote repository, first locate it on the online host (GitHub, Gitlab, etc). Next, find the **clone URL**. On GitHub, it is located under the **Code** dropdown inside the repository page (note: make sure to select HTTPS, if it is not already selected).



Once you copy the URL, open the terminal in the folder where you want to download (clone) the repo. The command is: `git clone repoUrl`

```
aldin-sxr ~ > Documents > code git clone https://github.com/Aldin-SXR/ibu-edu-ds-2021-2022.git
Cloning into 'ibu-edu-ds-2021-2022'...
remote: Enumerating objects: 41, done.
remote: Counting objects: 100% (41/41), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 41 (delta 0), reused 41 (delta 0), pack-reused 0
Receiving objects: 100% (41/41), 5.98 KiB | 5.98 MiB/s, done.
aldin-sxr ~ > Documents > code
```

You now have a local copy of the repository. Over time, other contributors might add new code to the repository or edit existing files. To “keep up” with all the latest changes, you can use the command: `git pull origin master`

```
aldin-sxr  master  ~  Documents  code  ibu-edu-ds-2021-2022  git pull origin master
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 9 (delta 1), reused 9 (delta 1), pack-reused 0
Unpacking objects: 100% (9/9), 633 bytes | 633.00 KiB/s, done.
From https://github.com/Aldin-SXR/ibu-edu-ds-2021-2022
 * branch            master     -> FETCH_HEAD
   d821497..08726fc  master     -> origin/master
Updating d821497..08726fc
Fast-forward
 src/main/java/ibu/edu/lab1/Main.java | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
aldin-sxr  master  ~  Documents  code  ibu-edu-ds-2021-2022
```

If there have been any changes to the repository since your last pull, they will be synchronized and you will have access to the latest version of the files.

Git is a very complex topic, and this chapter serves as a basic primer into its main functionalities. In this course, Git will be used to make all the latest lecture and lab code available to you, so make sure you are familiar with its basic operations. For more information and tutorials regarding Git, you can check out Learn Git and Git Cheat Sheet.

REFERENCES

[1] "Java," *Java Definition*. [Online]. Available: https://techterms.com/definition/java. [Accessed: 27-Feb-2020].
[2] *Java Packages*. [Online]. Available: https://www.w3schools.com/java/java_packages.asp. [Accessed: 27-Feb-2020].
[3] "Classes (OOP)," *Brilliant Math & Science Wiki*. [Online]. Available: https://brilliant.org/wiki/classes-oop/. [Accessed: 27-Feb-2020].
[4] "JAR (file format)," *Wikipedia*, 30-Jan-2020. [Online]. Available: https://en.wikipedia.org/wiki/JAR_(file_format). [Accessed: 27-Feb-2020].
[5] "Unit Testing," *Software Testing Fundamentals*, 03-Mar-2018. [Online]. Available: http://softwaretestingfundamentals.com/unit-testing/. [Accessed: 27-Feb-2020].
[6] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, J. de Rancourt, and C. Stein, *JUnit 5 User Guide*. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/#writing-tests. [Accessed: 27-Feb-2020].