

LAB GUIDE. SESSION 0

GOALS:

- **Installation of the course project**
- **Preparation of a repository for lab deliveries**
- **Working with matrices**

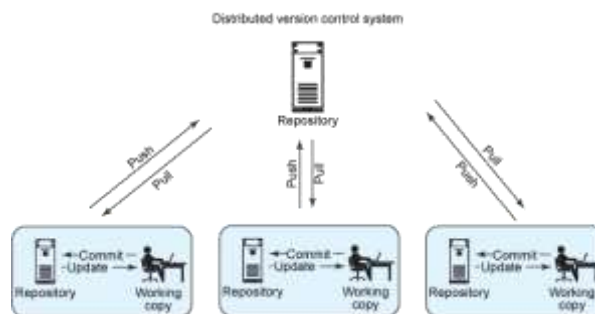
1. Installation of the course project

This section explains how to install the Java course project with codes and samples, useful to better understand the explanations of the subject and to facilitate learning.

A. Git

Git is a free and open source distributed version control system (<https://git-scm.com/>) that is used to share and keep safe the source code of some of the most important software projects in the world. There are other systems such as Subversion, CVS, Perforce, ClearCase, Mercurial but Git is nowadays the most common used version control system in both Academia and Industry.

It is designed to manage everything, from small to very large projects, with speed and efficiency. In the official website you can find books, tutorials and lots of free available resources.

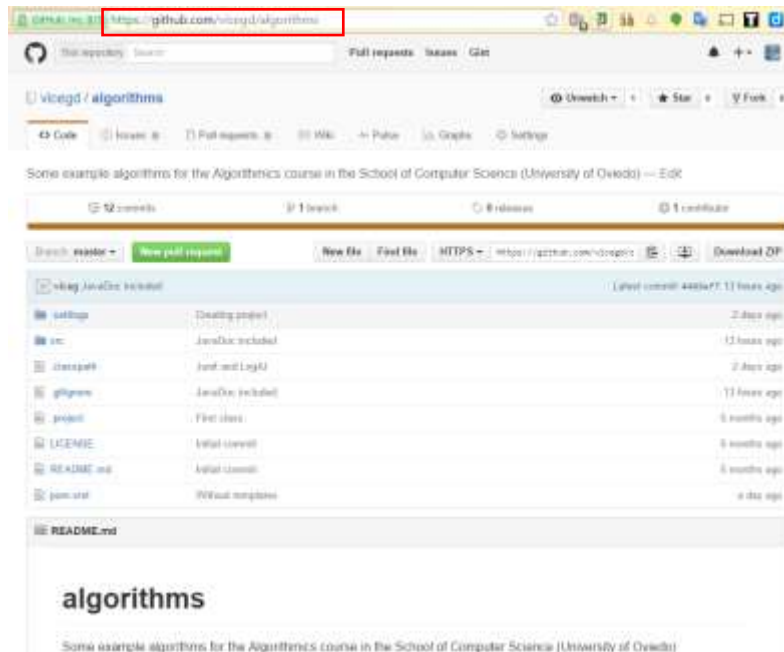


What is the idea? We have a central online repository, in which we have the last version of the code for the team. In the previous picture we have 3 developers. But at the same time all the developers can work on local copies of the code making changes to the code without the need of being connected to the central repository. In fact, all the developers have a local repository with a copy of the code. At same point, any of them can decide to place the code online for the others, then he/she needs to do a PUSH using a Git command. In the same way, other developers that want the last version of the source code should do a PULL using a Git command to get the last version of the code.

There are a lot of options such as getting a specific version of the code, doing forks, checking all the changes performed in the code during all the lifecycle of the project, or managing collisions between developers, but there are beyond the scope of this course (you will see in the future during the degree but it is out of the scope of this course).

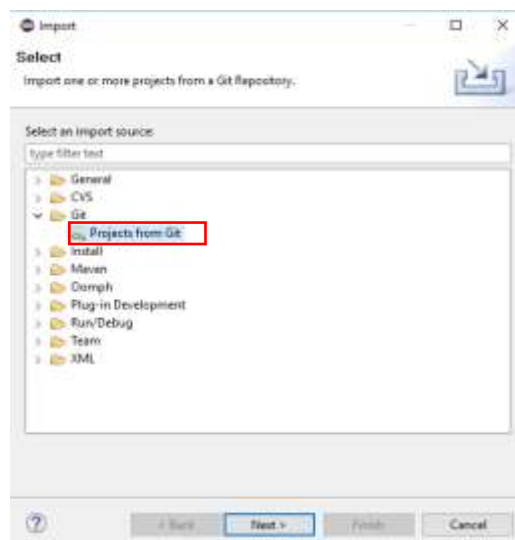
Course project with sample codes

The sample codes for this course are available on GitHub at <https://github.com/vicegd/algorithms>

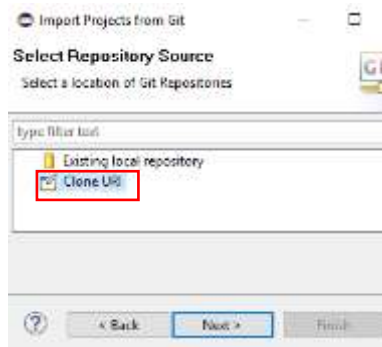


There is a button (Download ZIP) that can be used to download a ZIP with all the files of the Eclipse project. However, that is not the preferred option. It is much more convenient to use the Eclipse environment (or any other IDE) to download and synchronize the files.

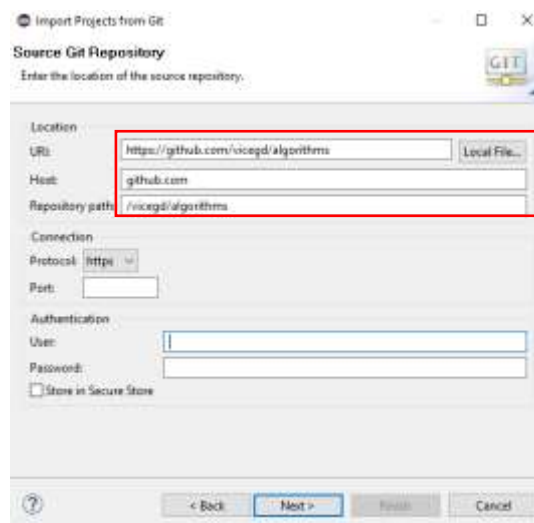
For that, **open Eclipse and go to File → Import → Git → Projects from Git**



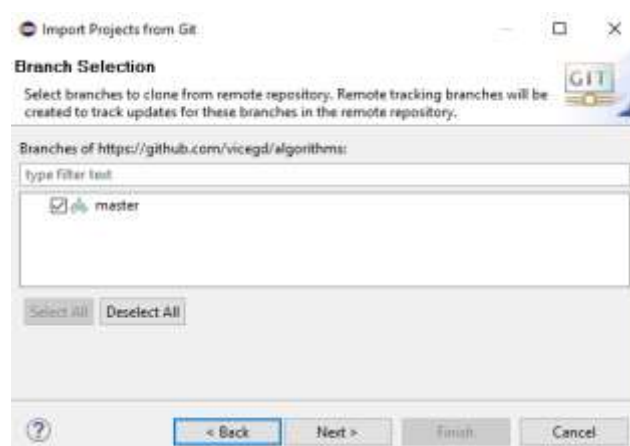
To download the code from the online repository, you should clone such a repository in your computer:



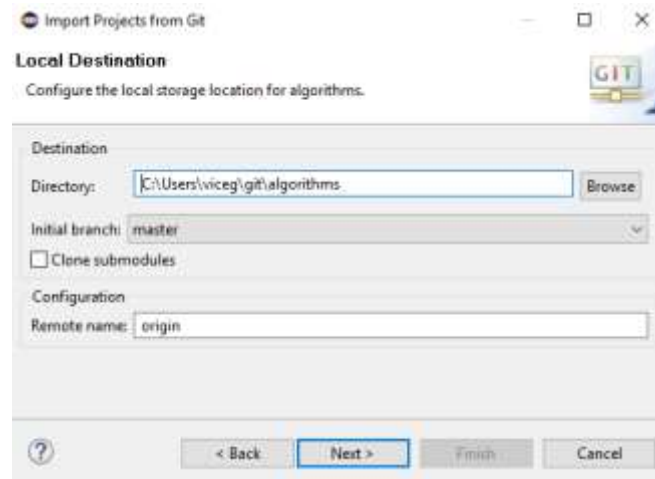
After that, it is needed to copy and paste the URI of the repository:



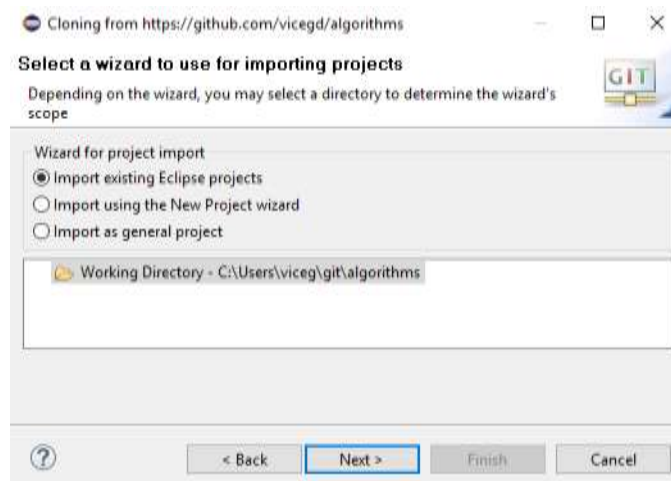
There is no need to indicate a user and a password since you will not have write permissions (only read) and reading can be done without authentication on GitHub. There is only one branch for the code (called **master**), so that is the option:



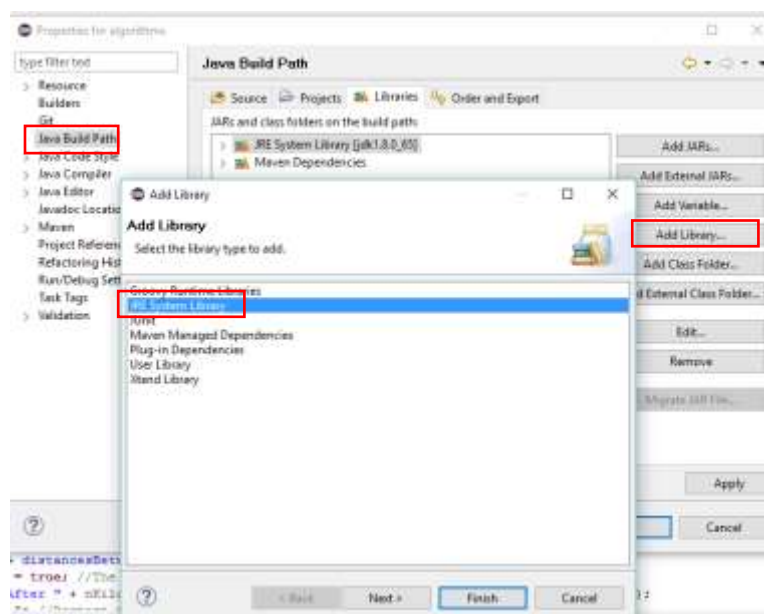
Finally, you can indicate the local folder in which you want to work with the local repository for the project (to save the code on the hard disc and to modify it if you want):



Since it contains an Eclipse project, you can import it safely:



If you get errors, it could be related to the Java Build Path (your JDK is probably in a different location than mine). So, you should go to the project properties and change the location of the JDK to correctly point to the folder in which you have it installed in your computer.



How to get the last version of the code?

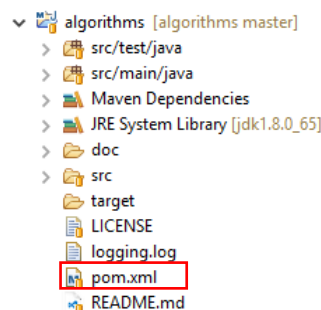
Whenever you want to check whether there is new available code, you will just need to do the following: **right click on the project → Team → Pull**

That way you will get the last version of the project synchronized with your Eclipse workspace. That's all.

B. Apache Maven

Although you may not need to use this tool in your projects, it is a great tool that is widespread used (<https://maven.apache.org/>). The course project is based on Maven.

Maven¹ is used to manage different steps/tasks in any project lifecycle such as structure, builds, reports, testing, etc. However, it is commonly used to manage project dependencies. All the configuration is done through a file called `pom.xml` that should be placed in the root folder.



There, you can indicate different tasks to be done. For example, to import the specific version of some dependencies (JUnit, SLF4J and Log4J) that are being used in the project:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  <modelVersion>4.0.0</modelVersion>
  <groupId>algorithms</groupId>
  <artifactId>algorithms</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.13</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.13</version>
    </dependency>
  </dependencies>
</project>
```

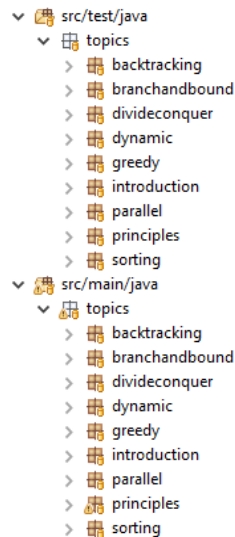
There are websites such as <http://mvnrepository.com/> in which you can check the different versions of libraries that can be automatically integrated in Maven or in other similar tools to manage dependencies (Ivy, Gradle, etc.).

¹ Maven is integrated in Eclipse by default

C. Junit

JUnit is a unit-testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of the family of unit testing frameworks, which is collectively known as xUnit. We will use this framework frequently from now on.

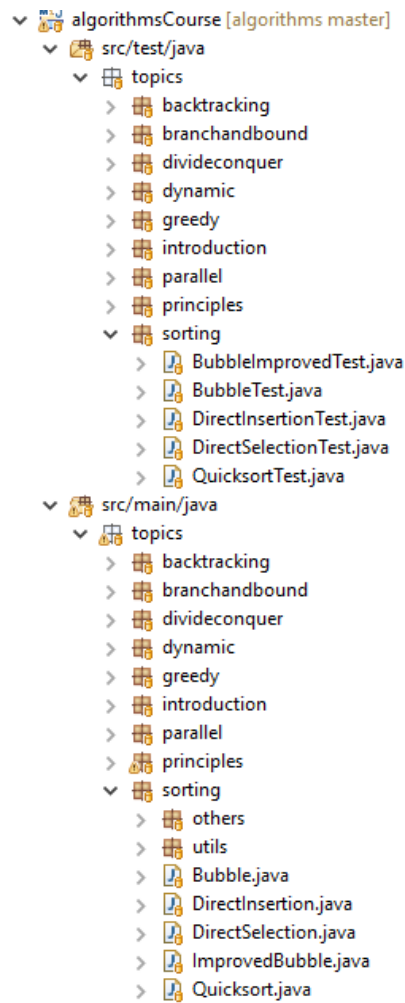
Back to the course project, it is divided by topics related to each of the lectures of the course.



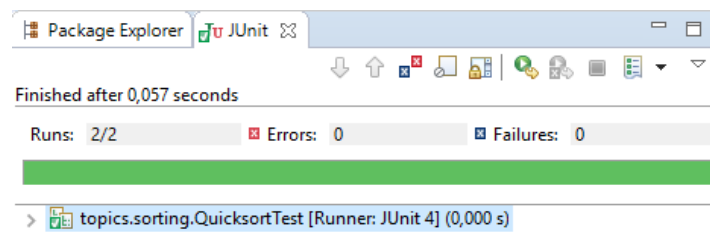
There are two main groups: **main** and **tests**. For example, the code contained in `src/main/java/topics/sorting` is related to the sorting topic we will see in class. Such code is in a package called `topics.sorting`. In the same way, the code contained in `src/test/java/topics/sorting` is related to the same topic and included in the same package. However, the latter is code that allows testing on the different sorting algorithms that are implemented in the former.

For example, if we want to test the Quicksort sorting algorithm, we just need to:

- Go to the file `QuicksortTest.java` in the `topics.sorting` package → **Click the right button** → **Run As** → **JUnit Test**



If everything is OK, the two tests that are included in the file should pass and a green mark will be showed:



In the test classes, you will find some of them with an attribute `@Ignore("Not ready yet")`. That is because the code it is supposed to test is not yet complete (**it is your homework to complete the code, remove the ignore attribute in the test, and make the test case work!**). For example:

```
/**
 * It gives the total waiting time. Optimal solution.
 * Assuming that tasks arrived in a smallest-first order {2, 3, 4, 5, 6, 7, 8}
 */
@Ignore("Not ready yet")
@Test
public void testWaitingTimeSmallestFirst() {
    tasks = new int[] {2, 3, 4, 5, 6, 7, 8};
    plumber = new Plumber(tasks);

    int result = plumber.getTotalTimeOfWait();

    assertEquals(112, result);
}
```

The code you should complete/change is always marked with an `UnsupportedOperationException` like in the following example:

```
/**
 * Calculates the total waiting time of customers
 * Return Total waiting time
 */
public int getTotalTimeOfWait() {
    throw new UnsupportedOperationException("This operation needs to be implemented");
}
```

In the README file of the project you will find links to the solutions for those missing fragments of code in case you cannot solve them by your own.

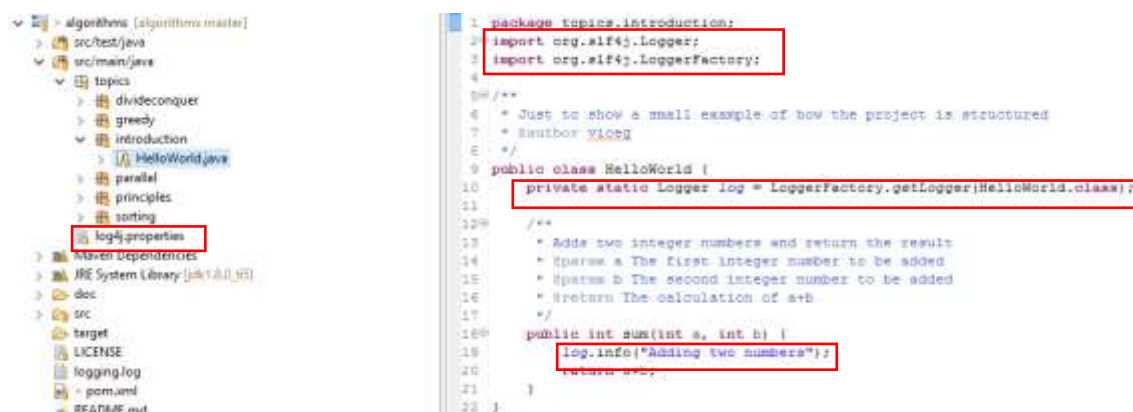
D. SLF4 and Log4J

SLF4J and Apache Log4J are two different, but integrated tools:

- SLF4J stands for Simple Logging Facade for Java (<http://www.slf4j.org/>). It is used only as a facade (abstraction layer) for various logging frameworks (java.util.logging, logback, log4J, etc.). The idea is to use the same methods regardless of the underlying logging framework we will use. But why we will need a logging framework? Logging frameworks are used instead of typical `System.out.print` instructions because there are much more powerful and you will not need to type and remove such instructions in the code depending on if you are testing, debugging or sending the code to production.
- Apache Log4J is one of the more popular logging frameworks out there.

To use it in your projects, you just need to:

1. Import the dependencies.
2. Create a `log4j.properties` file with the configuration.
3. Create a static object for every class in which you want to use it.
4. Type logging instructions!



There are different methods to be used:

Level	Description
OFF	The highest possible rank and is intended to turn off logging.
FATAL	Severe errors that cause premature termination. Expect these to be immediately visible on a status console.
ERROR	Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console.
WARN	Use of deprecated APIs, poor use of API, 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong". Expect these to be immediately visible on a status console.
INFO	Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep to a minimum.
DEBUG	Detailed information on the flow through the system. Expect these to be written to logs only.
TRACE	Most detailed information. Expect these to be written to logs only. Since version 1.2.12 ^[8]

The level of severity is important and ranges from **trace** to **fatal** (trace, debug, info, warn, error, fatal), being **trace** the lowest one and fatal the highest one.

That is why the `log4j.properties` file is important. The configuration of that file in the course project is like:

```
# initialize root logger with level ERROR for stdout and fout
log4j.rootLogger=ALL,stdout,fout
# FATAL, ERROR, WARN, INFO, DEBUG, TRACE, (ALL, OFF)

# add a ConsoleAppender to the logger stdout to write to the console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.threshold=DEBUG
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%m%n
#log4j.appender.stdout.layout.ConversionPattern=%d{HH:mm:ss,SSS} - %m%n

# add a FileAppender to the logger fout
log4j.appender.fout=org.apache.log4j.FileAppender
log4j.appender.fout.threshold=TRACE
log4j.appender.fout.layout=org.apache.log4j.PatternLayout
log4j.appender.fout.layout.ConversionPattern=%p\t%d{ISO8601}\t%c\t\t\t\t\t - %m%n
# create a log file
log4j.appender.fout.File=logging.log
log4j.appender.fout.MaxFileSize=1KB
log4j.appender.fout.MaxBackupIndex=3
```

There, I am saying that I want just two different types of output:

- **stdout** → I use the **ConsoleAppender**, so those are the messages that will be printed out by the console. I want only messages with a severity level of **DEBUG** or more (that is, all the messages except **trace** messages). In addition, the **ConversionPattern** as `%m%n` indicates that only the message (`%m`) is printed in the console (without any further information). Note that `%n` is just a line break.
- **fout** → I use the **FileAppender** to indicate that I want those messages to be saved in a text file (in the example `logging.log`). In this case, messages with a severity level of **TRACE** or more (that is, all the messages) are stored in the text file. The **ConversionPattern** in this case is much larger, indicating that we want to show the severity level (`%p`), the date time (`%d` in ISO8601 format), the class in which the message was written (`%c`), the thread (`%t`) and the message itself (`%m`). As an example:

```
INFO    2016-01-12 07:53:33,244 topics.introduction.HelloWorld [main] - Adding two numbers
```

A great point is that it is possible to change everything at any time without modifying the source code, just changing it in the configuration file.

E. Javadoc

Javadoc is a documentation generator based on comments included in the Java code. It is integrated in several IDEs such as Eclipse. In addition, it is strongly recommended that all the public elements of any project should be commented using Javadoc syntax.

The following picture shows how Eclipse is using the comments written with Javadoc syntax to show help for a method called `sum`



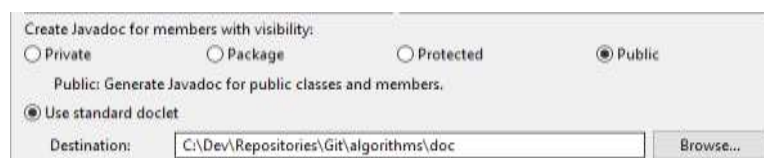
The “magic” comes from the comments. In this example:

```
/**
 * Adds two integer numbers and return the result
 * @param a The first integer number to be added
 * @param b The second integer number to be added
 * @return The calculation of a+b
 */
public int sum(int a, int b) {
    log.info("Adding two numbers");
    return a+b;
}
```

Typing `/**` in Eclipse and pressing enter in a method or in a class, will provide a template to be filled with text.

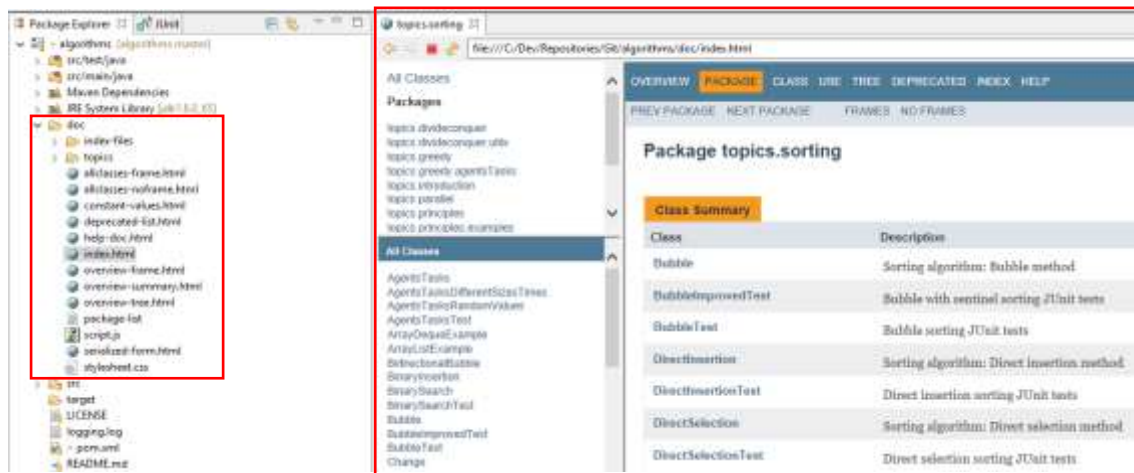
Besides, to generate the HTML files with the documentation of the course project, you only need to **select the project and go to Project → Generate Javadoc**.

You will find some configuration options, but it is enough if you select the visibility of the members from which you want to generate the documentation and the output folder, like the following:



Finally, you will find the generated documentation HTML files. You can open them with any Web browser or even from Eclipse. Of course, those HTML files are usually distributed as documentation of the APIs of several libraries. As an example, please, go to <http://twitter4j.org/javadoc/> to see the API documentation of a library that is used to get data from Twitter using the Java programming language. It is a best practice to keep in any project you

work, or you will work in the future (at least for the public members). It is also used in the course project:




2. Preparation of a repository for lab deliveries

During the development of the laboratories, students will use the Github tool to work with their code. Steps to follow to prepare the environment:


1. Go to <https://github.com/>
2. If you don't have an account, sign up and then sign in with the user account of your choice.
3. Create a new repository. Please, pay attention to the following configuration:
 - a. The name of the repository must be `algorithmics<YOUR_SURNAME><YOUR_NAME><YOUR_UO>`. Example: **algorithmicsGarciaDiazVicenteUO42478**.
 - b. The description must be **Repository for the Algorithmics course at the School of Computer Engineering of the University of Oviedo**.
 - c. The repository must be **private**.
 - d. The repository must be initialized with a **README file**.
 - e. The `.gitignore` file should be included for the **Java** programming language. This file is useful to indicate files you don't want to store in the repository (e.g., files to configure your specific Eclipse project in your system).


This would be a screenshot of the example:

Owner:  vicegdlive ▾ / Repository name: ✓

Great repository names are short and memorable. Need inspiration? How about **cuddly-robot**?

Description (optional):

☐  **Public**
Anyone can see this repository. You choose who can commit.

☒  **Private**
You choose who can see and commit to this repository.


Skip this step if you're importing an existing repository.

☒ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: ▾ | Add a license: ▾ ⓘ



[Create repository](#)

4. You can see that you have two files in your new repository. The `.gitignore` file (to indicate which files you don't want to be synchronized between your computer and Github), and the `README.md` file (to show general information about the project).

 vicegdlive Initial commit	Latest commit 7eca79 20 seconds ago
.gitignore	Initial commit 20 seconds ago
README.md	Initial commit 20 seconds ago

5. In fact, the default content of `README.md` can be already seen just below that file:



6. Let's modify the `README.md` file by clicking on the file  [README.md](#) and then on the  icon.

7. Edit the `README.md` file to include some information and then save it (**commit changes**). The information must contain your complete name, the current year and the URL, as in the following example:

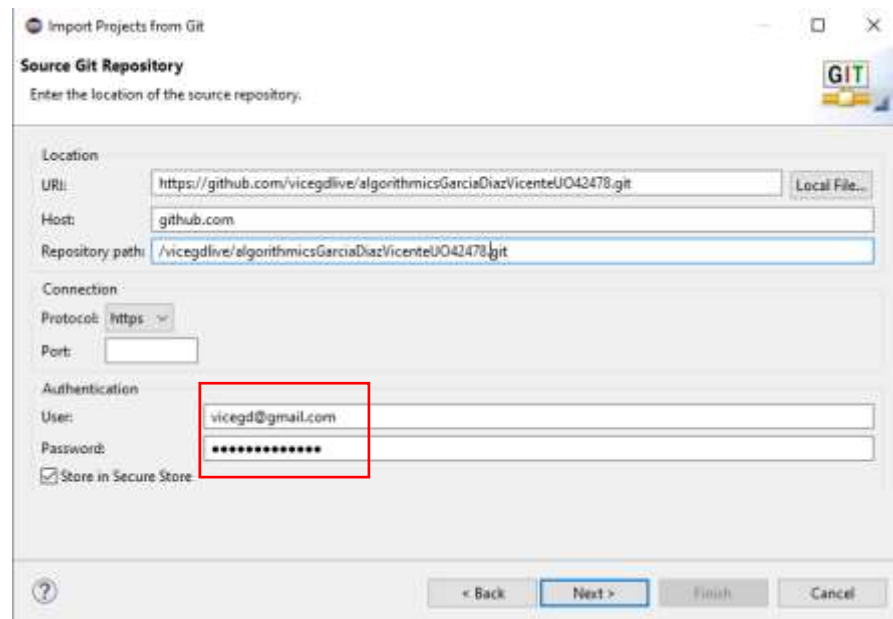


```
1 # algorithmicsGarciaDiazVicenteUO42478
2 Repository for the Algorithmics course at the School of Computer Engineering of the University of Oviedo
3
4 ## Information
5 **Student**: Vicente García Díaz
6
7 **Year**: 2020
8
9 **Repository URL**: https://github.com/vicegdlive/algorithmicsGarciaDiazVicenteUO42478
10
```

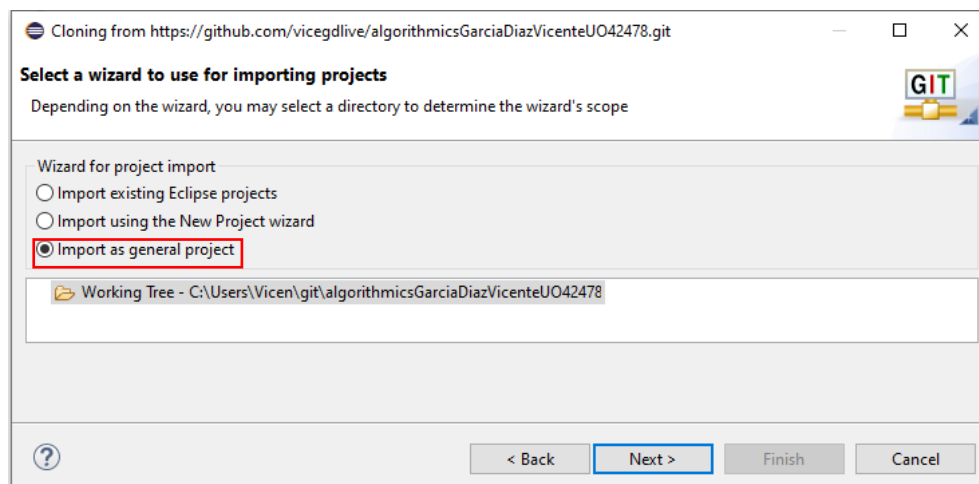
Once committed, we will see something like the following on the website:



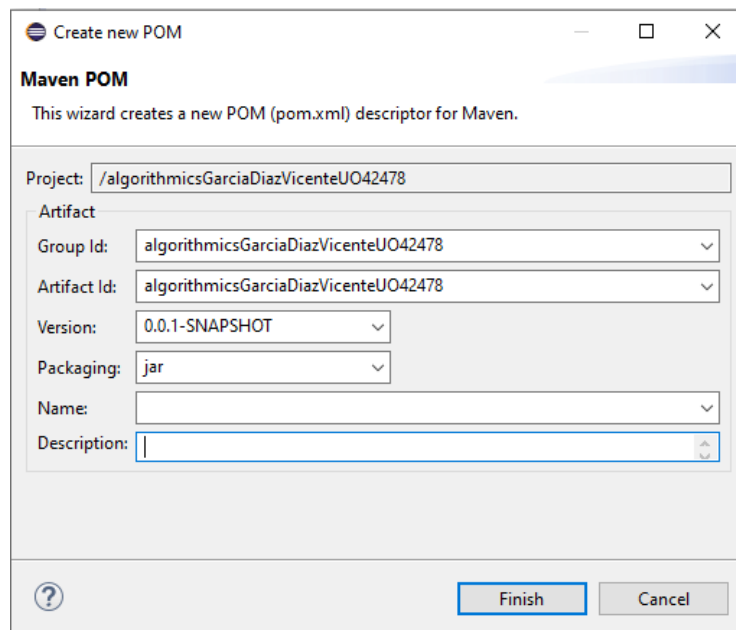
8. In the project webpage, go to **Settings** → **Collaborators** and add the **vicegd** Github account as a collaborator. You don't need to copy the invite link.
9. Clone the repository in your Eclipse workspace the same way you did it for the course project in the previous section. Your URL should be something like this: <https://github.com/vicegdlive/algorithmicsGarciaDiazVicenteUO42478.git>.
 - a. Remember that since this is a private repository, you must include your user and password during the import process. If not, you will not be able to access your own files.



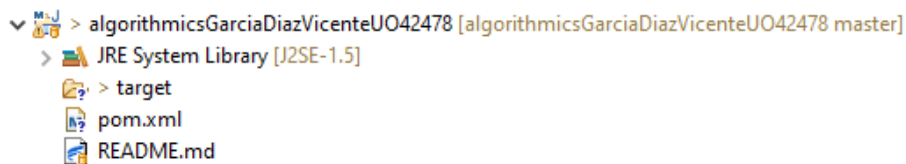
- b. Use the default values during the process. However, since we are not working with a Java project yet, we should mark the option “Import as general project”.



10. In this moment, you would be able to see just one file (README.md). The other file (.gitignore) is also in your project folder but Eclipse hides it. Now, we should create a project to work with. For that, we can create a Maven project. It is very easy: **Right click on the project → Configure → Convert to Maven Project**. The default values are fine now.



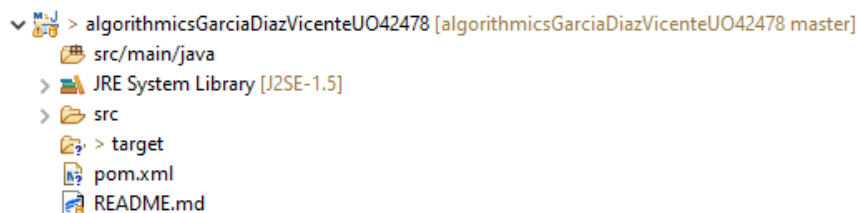
11. After that, we will have a project like the following one:



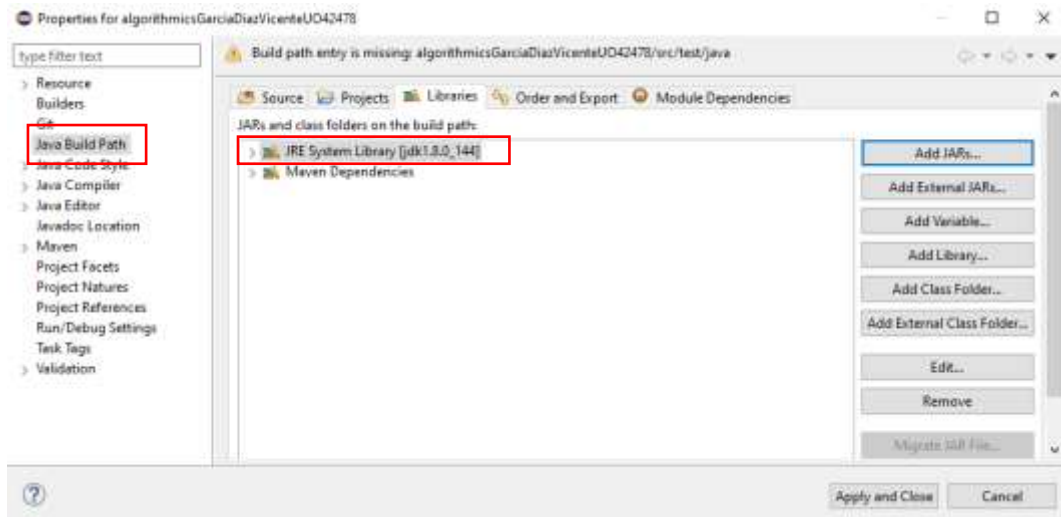
12. We should now create the following folder in the project:

- `src/main/java`

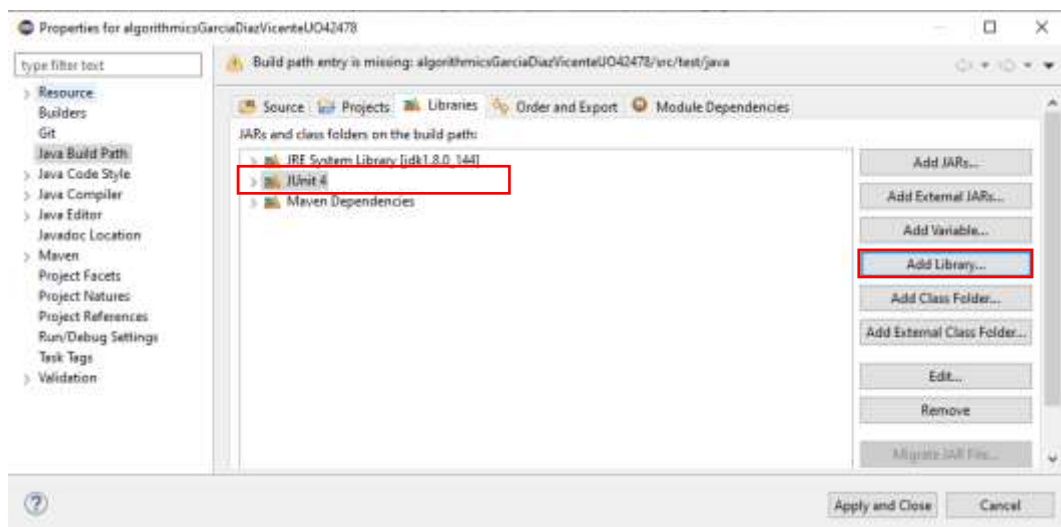
The Eclipse environment may detect them as **source folder** automatically.



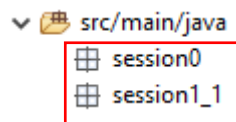
13. Regardless of the default Java environment used, it is convenient to change the version of the Java runtime environment to use new features and performance improvements: **Right click on the project → Properties → Java Build Path → Libraries → JRE System Library → Edit → Workspace default JRE** (in the lab you will have with Java 8).



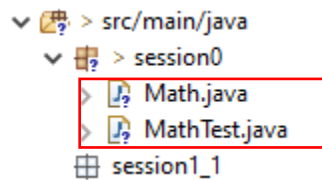
14. Since we are going to work with JUnit, let's add the library. In the previous window, you just need to click on **Add Library** → **JUnit** → **JUnit library version: JUnit 4**. Now you will see something like this:



15. As a general rule, we are going to create a package for each session. For example. Let's create a `session0` and a `session1_1` packages in the `main` folder for this and the next session.



16. Let's create now two sample files, `Math.java` and `MathTest.java` in `session0` package. The best way to work is to put the JUnits files in the **test** folder (for separation). But, but for now, we will put it everything in the **java** folder, since the number of files we will handle will be very small in each package (you are free to delete the **test** folder).



17. This may be the code for **Math.java**. Copy and paste it.

```
package session0;

public class Math {
    public int sum(int a, int b) {
        return a + b;
    }
}
```

18. This may be the code for **MathTest.java**. Copy and paste it.

```
package session0;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class MathTest {
    Math math;

    @Before
    public void initialize() {
        math = new Math();
    }

    @Test
    public void test2Plus3Equals5() {
        assertEquals(5, math.sum(2, 3));
    }
}
```

19. There are some files that we don't want to upload to Github everytime we do some changes because it is a waste of time and the resources (they are files that depend on the specific versions of Eclipse and Java that are being used). To indicate which ones we don't want to upload, we need to edit the `.gitignore` file. The "problem" is that such a file cannot be opened from the project explorer directly. However, if you go to the project system folder you may see the file.

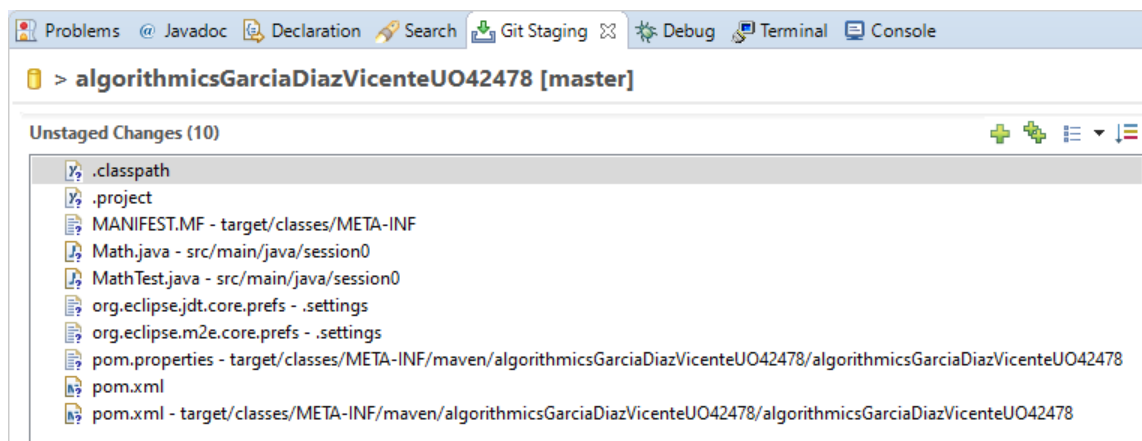
equipo > Disco local (C:) > Usuarios > Vicen > git > algorithmicsGarciaDiazVicenteUO42478

Nombre	Fecha de modificación	Tipo	Tamaño
.git	24/01/2020 10:16	Carpeta de archivos	
.settings	24/01/2020 10:07	Carpeta de archivos	
src	24/01/2020 10:15	Carpeta de archivos	
target	24/01/2020 10:07	Carpeta de archivos	
.classpath	24/01/2020 10:15	Archivo CLASSPATH	2 KB
.gitignore	24/01/2020 10:04	Documento de te...	1 KB
.project	24/01/2020 10:07	Archivo PROJECT	1 KB
pom.xml	24/01/2020 10:07	Documento XML	1 KB
README.md	24/01/2020 10:04	Archivo MD	1 KB

20. Open the file and add the following to the end of the file. Then, save the file:

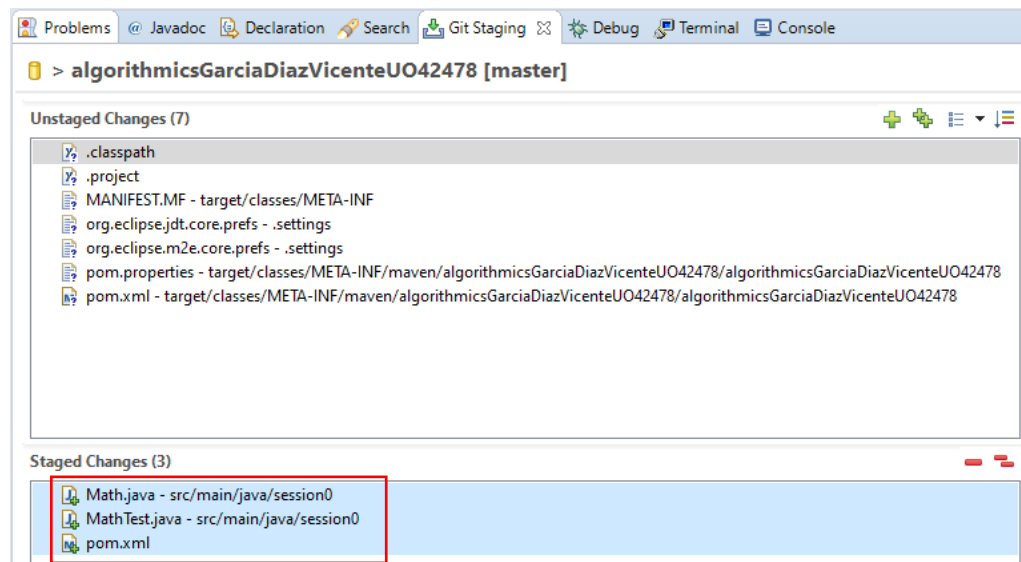
```
target/
.settings/
.classpath
.project
```

21. In Eclipse, a **Git Staging view** will be opened with a list of files that you **can upload to Github**. Since, it is the first time we do the whole process, we can see some files that we should not see there, taking into account what we have just added them to the `.gitignore` file.

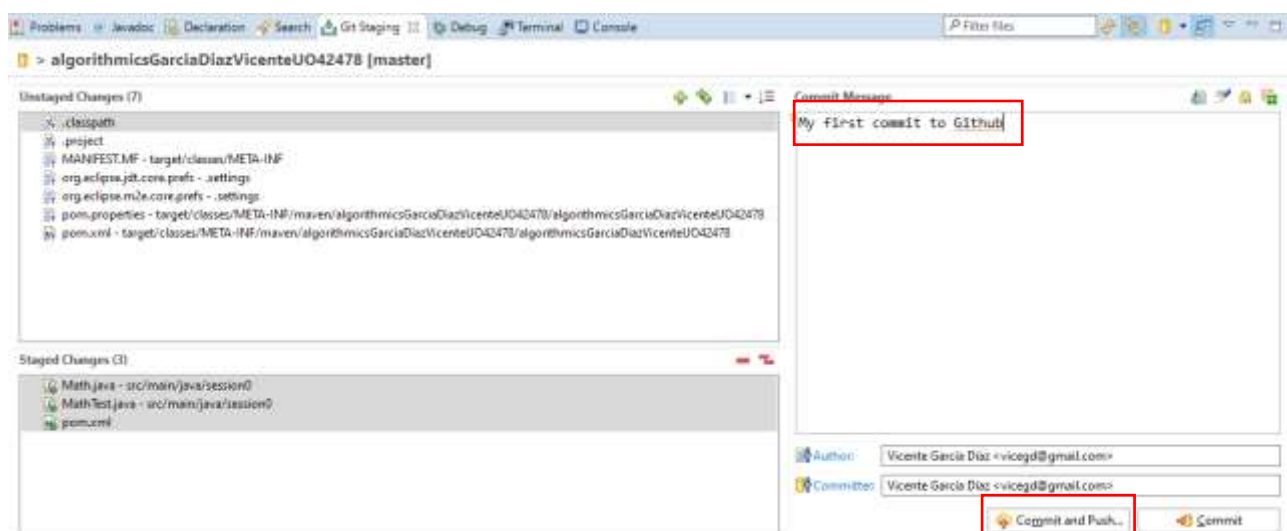


NOTE: If you don't see the Git Staging view, you can start the process with: **Right Click on the project → Team → Commit**

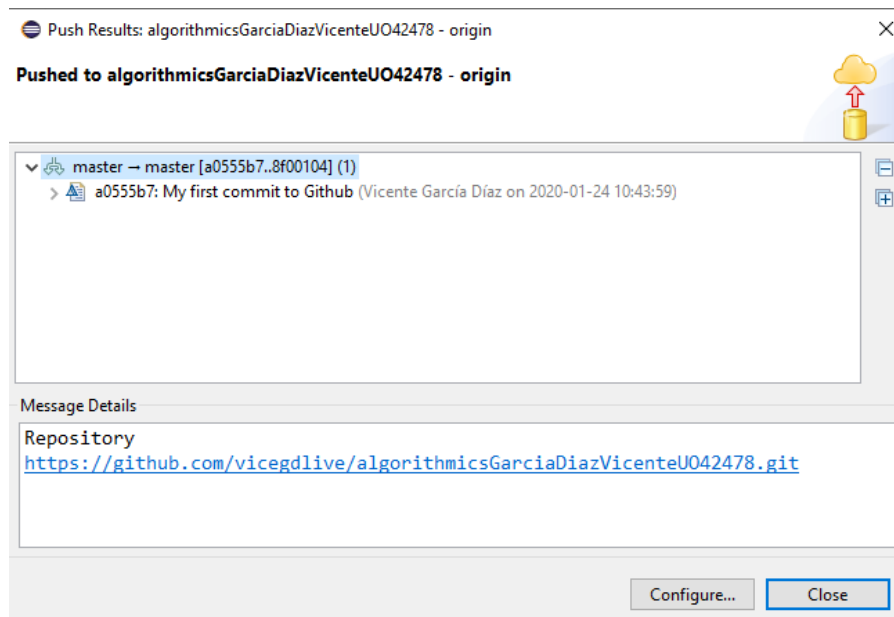
22. However, before uploadind the files to Github, we need to click on the **green crosses**. You will see that only 3 files were moved to the next step (**staged files**). The others are ignored as we indicated in the `.gitignore` file. Take into accout that it is possible to see 4 files instead of 3 if the environment includes the `.gitignore` file that we modified manually in the previous step; however, that is not important because if we don't include the file in this iteration it will be included the next time we do another commit).



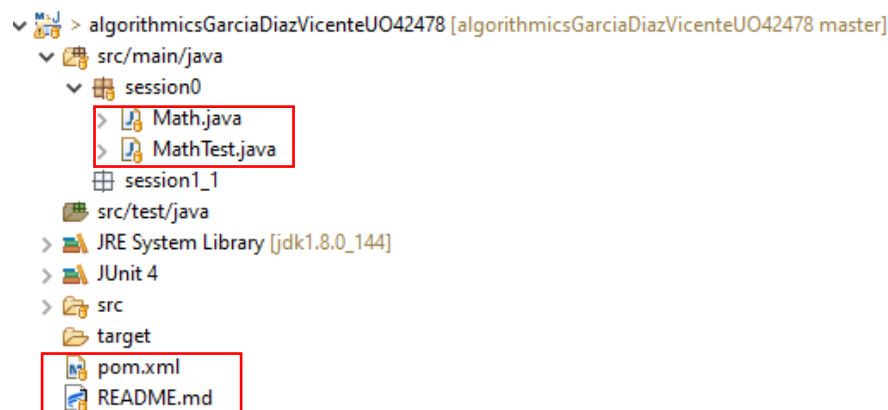
23. Before we upload files to Github, we need to indicate a message (which will be uploaded with the files). That message should be clear to help developers to know what new things they are going to find in the files. Finally, we should click on **Commit and Push**. That should update both the Github repository and our local repository. If we just click on **Commit** we would just upload our local repository but Github would not be uploaded with the changes.



24. When you finish, you will see something like the following, indicating that everything was fine.



25. In Eclipse, uploaded files will be marked with yellow marks



26. And updated files can be seen on Github, together with the corresponding messages.



Keep your Github Repository always up to date

We should repeat steps 21, 22 and 23 every time we want to upload changes to Github. In the **Git Staging view** we will always see the files that may be uploaded (because they are new or because they have changes since the last version). However, from time to time we can see files that we don't want to upload (e.g., a logging or a private file). In that case, we should modify the

.gitignore file or ignore the file directly in the Git Staging view by clicking with the right bottom on it and selecting “Ignore”.

Remember to synchronize everything always

It is important to work with the last version of the code always regardless of the computer you are working with. That is important because that way you can avoid conflicts between files (i.e., you modify one file in a computer and you modify the same file in another place, so Github does not know which one is the “good” one). To avoid that, do always the following:

- **When you start working:** Download the last version of your code from Github by doing this: **Right click on the project → Team → Pull**
- **When you finish working:** Upload the **last version of your code to Github by doing this: Right click on the project → Team → Commit** (as explained from step 21).

Let's have a conflict between our files

We are going to force a possible conflict with our files to help you understand how they can appear and how to solve them.

1. Let's go to the Github website and change something in the README.md file (e.g., adding your email just between the year and the repository URL).

```
# algorithmicsGarciaDiazVicenteU042478
Repository for the Algorithmics course at the School of Computer Engineering of the University of Oviedo

## Information
**Student**: Vicente García Díaz

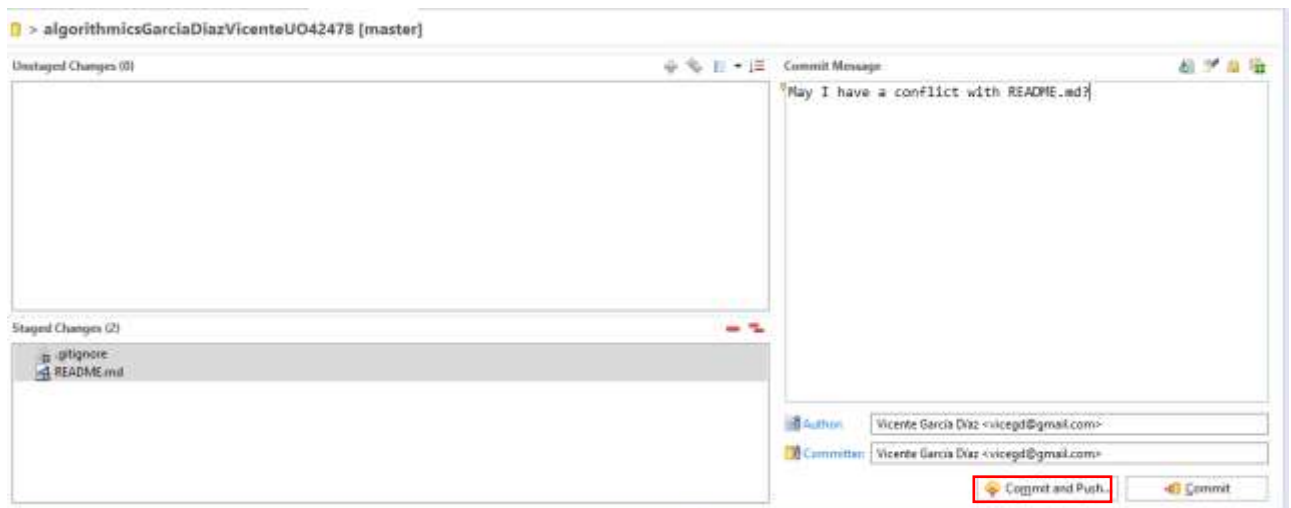
**Year**: 2020

**Email**: U042478@uniovi.es
```

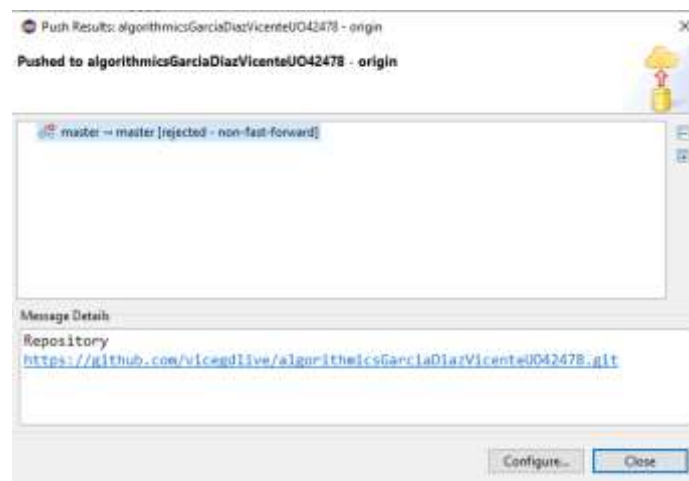
2. If we also want to modify this file from Eclipse, we should do a Pull (**Right click on the project → Team → Pull – don't do that now**) to synchronize the files before we modify it. If not, we will have problems. Let's modify the file from Eclipse too (**without doing the Pull**) with a different text:

```
*README.md
1 # algorithmicsGarciaDiazVicenteU042478
2 Repository for the Algorithmics course at the School of Computer Engineering of the University of Oviedo
3
4 ## Information
5 **Student**: Vicente García Díaz
6
7 **Year**: 2020
8
9 My email is U042478@uniovi.es
```

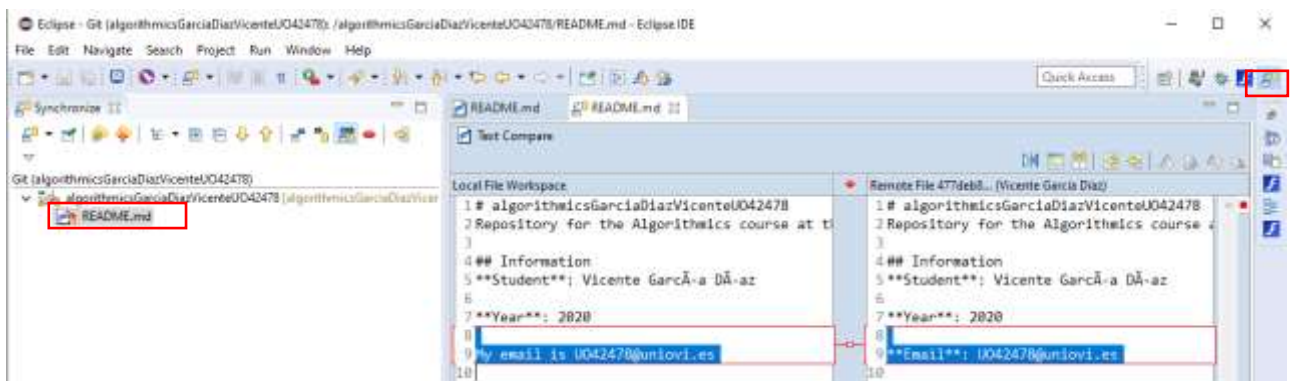
3. So, we have two conflicting versions of the same file. If we try to Commit the file (**Right click on the project → Team → Commit**), as in the following view:



Changes will not be performed and we will get an error message:



- The easiest way to solve that, is to delete your local copy (the one in Eclipse) and synchronize your content with the content that is already in Github. So, be careful and make a copy of the things you don't want to lose. To do that: **Right click on the project → Team → Synchronize workspace**. After that, the **Synchronization perspective** opens. If we click on the conflicting file, we will see the two versions (the local one on the left and the Github one on the right).



- Now we will discard the local changes: **Right click on README.md → Overwrite**.

6. It is not yet finished. We should go back to the **Java perspective** and do a Pull with the files from Github (**Right click on the project → Team → Pull**) to make both copies synchronized. Now everything will be correct.
7. From that point, we can freely work on our local copy, and when we finish we can send everything to Github (**Right click on the project → Team → Commit**) with no problems.

3. Working with matrices

In the future sessions we will usually work with multi-dimensional vectors and matrices, with integers or other data types. Since these data structures are simple and easy to understand, we will use them to operate with a huge number of elements. However, in order to test our algorithms and implementations we should start using smaller problem sizes.

In this section you should create a Java class called `MatrixOperations.java` in the `session1_1` package of your project. This represents a two-dimension square matrix. Thus, you must implement the following public methods:

- `MatrixOperations(int n)`. Creates a new matrix of size $n \times n$ and fills it with random values.
- `MatrixOperations(String fileName)`. Creates a matrix using data of the file provided as parameter. This file must have 1 integer number as the first line. Following lines contain n values to represent every element of the matrix row. Each of the values will be separated by a tabulator. Example:

10

1	1	4	1	2	2	2	2	3	4
2	4	1	1	3	4	2	1	4	3
1	4	4	4	3	2	2	3	4	3
3	2	2	3	2	1	1	2	3	3
2	3	3	4	1	1	4	3	2	3
1	2	2	3	1	4	3	2	1	2
1	2	3	3	3	1	4	3	4	3
4	3	3	4	4	2	1	2	4	2
4	2	2	3	2	1	3	1	1	1
2	4	3	2	1	3	3	2	1	1

- `getSize()`. Returns the matrix size (n).
- `write()`. Prints through the console all the matrix elements.

- `sumDiagonal1()`. Computes the summation of all the elements of the matrix diagonal. This implementation must iterate over all the matrix elements, but only sums appropriate elements. So, the complexity is quadratic.
- `sumDiagonal2()`. Computes the summation of all the elements of the matrix diagonal. This second version should only consider the elements of the main diagonal. So, the complexity is linear.
- `travelPath(int i, int j)`. Given a matrix with integer numbers between 1 and 4, this method iterates through the matrix starting at position `(i, j)` according to the following number meanings: **1 – move up; 2 – move right; 3 – move down; 4 – move left**. Traversed elements would be set to -1 value. The process will finish if it goes beyond the limits of the matrix or an already traversed position is reached. To make sure your code works, create a text file with the previous example indicated in `MatrixOperations(String fileName)` and test it. For that file, the final output for a call `travelPath(3, 0)` should be something like the following:

1	1	4	1	2	2	2	2	3	4
2	4	1	1	3	4	2	1	4	3
1	4	4	4	3	-1	-1	-1	4	3
-1	2	2	3	-1	-1	1	-1	-1	3
-1	-1	3	4	-1	1	4	3	-1	-1
1	-1	-1	-1	-1	-1	3	2	1	-1
1	2	3	-1	3	-1	-1	3	4	3
4	3	-1	-1	4	-1	-1	2	4	2
4	2	-1	-1	-1	-1	3	1	1	1
2	4	3	-1	-1	3	3	2	1	1

Number of movements = 32