

Lab exercise 0: Java and Git

24292-Object Oriented Programming

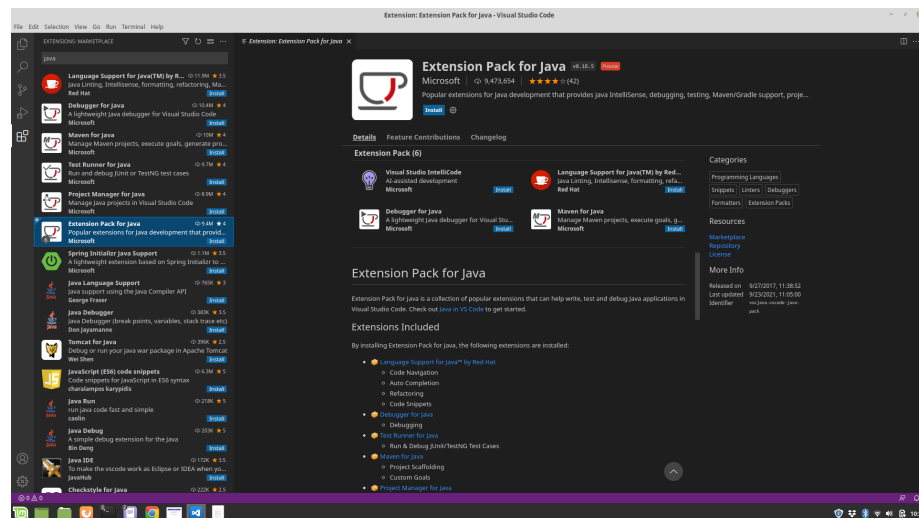
1 Introduction

The purpose of this lab is to become familiar with Java and Git. In this lab you will implement a simple Java project, including the basic components that are always present in a Java program. Git is a version control system that allows multiple users to share and maintain projects in a central repository. The lab is introductory and will not require any delivery at the end.

2 Create a Java program

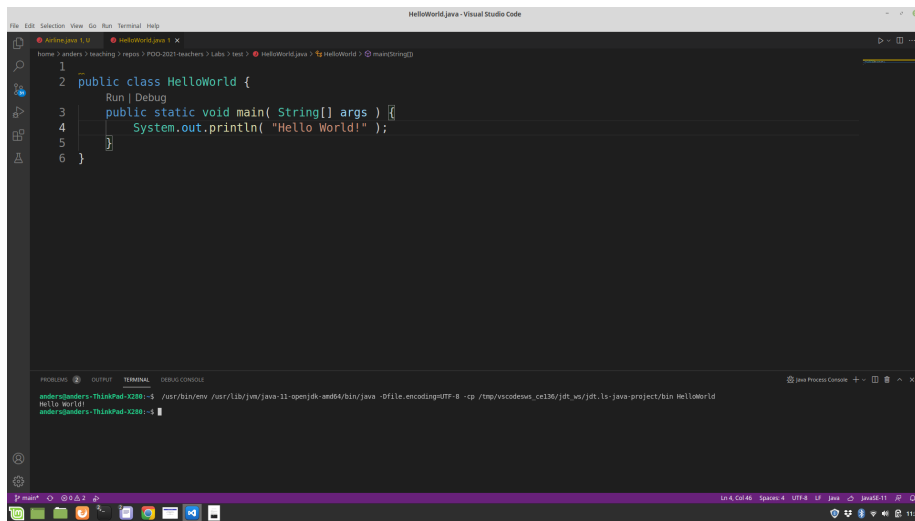
In order to implement a program in Java, it is a good idea to use an IDE (Integrated Development Environment), an editor used to develop code in various programming languages. There are many IDEs for programming in Java.

The lab handout explains how to write Java code in Visual Studio Code (VSCode), but you are free to use the IDE of your choice. To implement Java programs in VSCode, you need to 1) install the Java Development Kit (JDK); 2) install VSCode itself; and 3) install the Extension Pack for Java in VSCode:



Once you have the Java extension installed, the next step is to create a new file from the menu. When you create a file, you can manually specify the programming language, or just start typing code in that language. The Java compiler requires all public classes to be stored in a file with *the same name* as *the class* (including uppercase and lowercase letters) and extension `.java`.

Let us define a class `HelloWorld` that prints “Hello World!” to the screen. The code is given below. When you run the code, a console window opens at the bottom part of the screen, and this is where the written output of the program appears. The class definition should be stored in a file “`HelloWorld.java`”.



The screenshot shows the Visual Studio Code editor with a file named `HelloWorld.java` open. The code in the editor is as follows:

```
1 public class HelloWorld {
2     Run | Debug
3     public static void main( String[] args ) {
4         System.out.println( "Hello World!" );
5     }
6 }
```

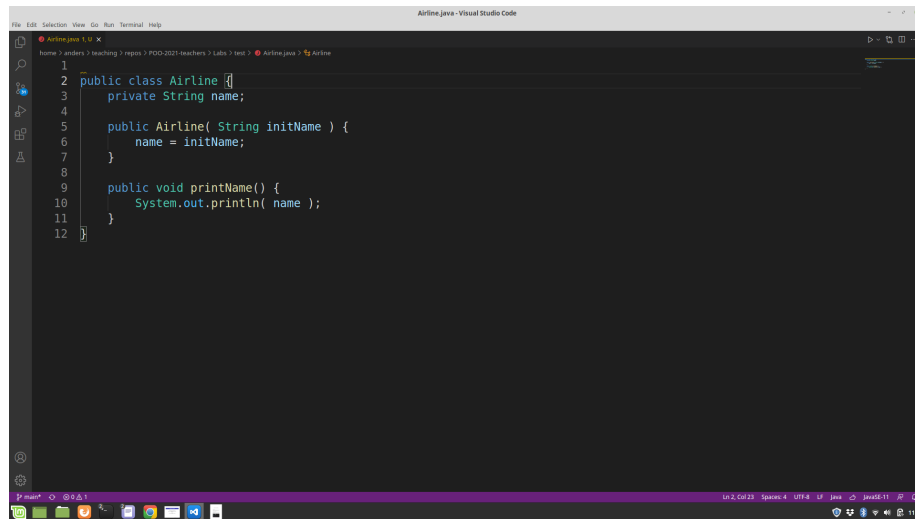
Below the editor, the TERMINAL window is open, showing the command used to compile and run the program:

```
anders@anders-ThinkPad-X280:~$ /usr/bin/env /usr/lib/jvm/java-11-openjdk-amd64/bin/java -Dfile.encoding=UTF-8 -cp /tmp/vscodeuser_cx38/jdt_wt/jdt.ls-java-project/bin HelloWorld
hello world!
```

From the console window it is also possible to manually compile individual Java programs and execute them from the command line. To execute Java code, we need two separate components: a *compiler* (`javac`) and an *interpreter* (`java`). The compiler takes as input Java source code and outputs an intermediate representation called byte code, stored in a file `X.class`, where `X` is the class name. The interpreter takes as input the byte code and interprets it to produce a program execution.

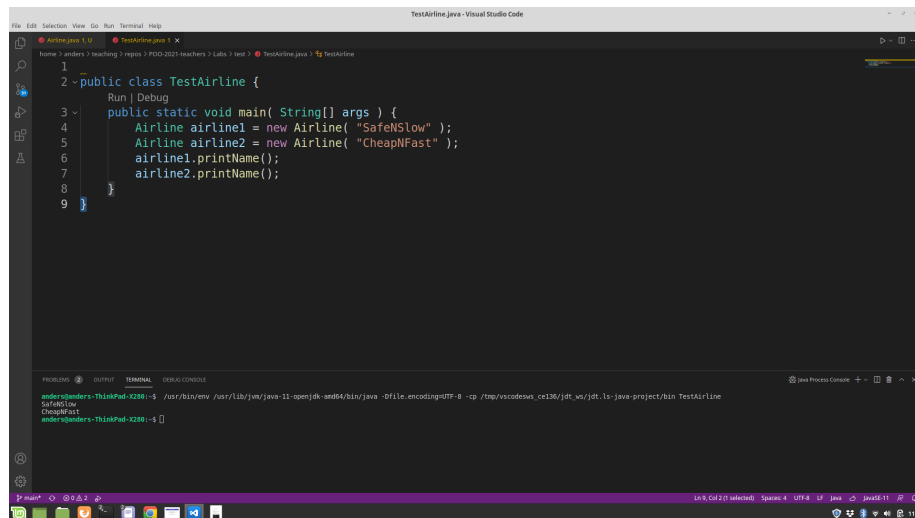
VSCoDe does not explicitly organize Java files in projects; rather, a Java project is implicitly defined by a given directory. Hence if you want multiple Java files to interact, you should always save these files in the same directory (or in subdirectories of that directory).

Next, let us define a class `Airline` with an attribute called “name”. Let us also define a constructor that takes a string argument and sets the name to the value of the argument. Finally, let us define a method called “`printName`” which will just print the name to the screen. Then save the class definition in a file called “`Airline.java`”. The code should look like this:



```
1 public class Airline {
2     private String name;
3
4     public Airline( String initName ) {
5         name = initName;
6     }
7
8     public void printName() {
9         System.out.println( name );
10    }
11 }
12 }
```

To execute Java code, at least one of the classes need a main method, which always has the same syntax. As a rule of thumb, it is a good idea to implement the main method in a *separate class*. In the following example, the main method is implemented in a class TestAirline, which tests the Airline class by creating instances of Airline and printing their names. Recall that TestAirline and Airline have to be stored in the same directory.



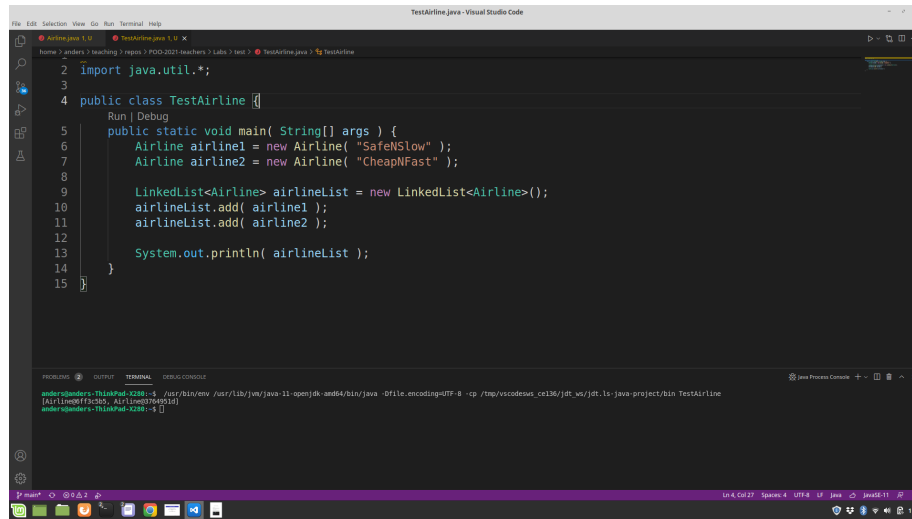
```
1
2 public class TestAirline {
3     public static void main( String[] args ) {
4         Airline airline1 = new Airline( "SafeNSlow" );
5         Airline airline2 = new Airline( "CheapNFast" );
6         airline1.printName();
7         airline2.printName();
8     }
9 }
```

```
ander@spandevs-ThinkPad-X208:~/java$ java -Dfile.encoding=UTF-8 -cp /tmp/vscodeuser_cel36/jet_vs/jdt.ls-java-project/bin TestAirline
SafeNSlow
CheapNFast
ander@spandevs-ThinkPad-X208:~/java$
```

2.1 Trying available Java classes

Java has available a set of libraries containing lots of pre-programmed classes. It is often a good idea to reuse existing classes rather than implementing a new class from scratch. Here we include a small example of how to include a *list* of

instances. More precisely, we will use a class called `LinkedList` that allows us to store any type of instance in a list data structure. To access the `LinkedList` class, one must include it from the package `java.util` at the beginning of the source code, as in the following program:



```

1  import java.util.*;
2
3
4  public class TestAirline {
5      public static void main( String[] args ) {
6          Airline airline1 = new Airline( "SafeNSlow" );
7          Airline airline2 = new Airline( "CheapNFast" );
8
9          LinkedList<Airline> airlineList = new LinkedList<Airline>();
10         airlineList.add( airline1 );
11         airlineList.add( airline2 );
12
13         System.out.println( airlineList );
14     }
15 }

```

```

andres@andres-ThinkPad-X280:~$ ./usr/bin/env /usr/lib/jvm/java-11-openjdk-amd64/bin/java -Dfile.encoding=UTF-8 -cp /tmp/vscodesuo_cx136/jdt_ws/jdt.ls-java-project/bin TestAirline
[Airline@ff3c05, Airline@706051d]
andres@andres-ThinkPad-X280:~$

```

We see that a list of instances is printed with its associated *pointers*. To be able to call the print method of each instance we need to access each element in the list individually. For this purpose substitute the print line by the following loop:

```

for( Airline airline : airlineList ) {
    airline.printName();
}

```

Execute it and now you can see the the result correctly prints the names of each airline.

3 Create a Project with a Graphical Interface

Java makes it relatively easy to create graphical user interfaces, or GUIs. To fully understand how GUIs are created, it is necessary to first study several concepts that will be covered later during the course. However, for today's class we will give you a template for creating GUIs that you can modify according to your needs. The template is provided as two Java files in the supplied compressed file `P0Code.zip`. The class `MyGUI` defines the graphical components, and the class `TestGUI` is in charge of testing the graphical interface by creating an instance of `MyGUI`.

The example in `MyGUI` shows how to create a graphical interface with four components: a button (`JButton`) that you can press, a text field (`JTextField`)

which allows users to write text, a combo box (JComboBox) for selecting between predefined options, and a label (JLabel) for simply showing text. To change the graphical interface, one would simply change the number and types of components, and change the method “initComponents” accordingly, which is where the components are initialized and added to the GUI.

Graphical components are organized using *panels* and *layouts*; the example uses a JPanel on top of which the other graphical components appear, and a GridLayout which organizes components in a grid. Other forms of layouts are also possible if you want to experiment. It is also possible to draw geometric figures on a panel, something we will exploit in future lab sessions.

To implement events, Java uses a concept called *action listeners*. In the MyGUI example, the button has an associated action listener, whose effect is implemented in the method “actionPerformed”. In this particular example, pressing the button has the effect of changing the label to say “Pressed!”, using the method “setText” of JLabel.

3.1 Creating a GUI for the Airline class

To create a GUI for the Airline class, we will modify the class MyGUI. It is a good idea to make a copy of the file first. To interact with the Airline class, remember that it has to be stored in the same directory as the GUI.

Create a GUI with three components: a text field, a button, and a label. The GUI should allow a user to create an airline with the name specified in the text field, and print the name of the airline to a console. To access the text of a text field, you can use the method “getText” of JTextField:

```
public void actionPerformed( ActionEvent evt ) {  
    Airline airline = new Airline( field.getText() );  
    airline.printName();  
}
```

Optionally, if you have time and you want to practice more:

- Change the code of “actionPerformed” such that the name of the airline is written to the label rather than to the console. Note that you may have to make changes to the class Airline.
- Add an attribute to the Airline class that represents the number of airplanes that the airline operates.
- Add a method printAirline that prints both the name and the number of airplanes.
- Add the corresponding fields in the graphical interface.
- Design a flight search engine (that allows a user to search for flights) by adding graphical objects to a JPanel.

4 Git

A version control system automizes the tasks of saving, loading, registering, identifying, and mixing different versions of the files created throughout a project. In software projects, files are frequently updated, making this type of system highly useful. Files are maintained in a central repository which makes it easy to access the project remotely from different computers.

In this course we use a version control system called **Git**. Each pair of students will create and maintain their own **Git** repository, using a webpage called **GitHub** which allows free private repositories of up to 5 users. Note that it is somewhat easier to manage **Git** repositories in Linux or macOS, but it should be possible to make it work in Windows. In Aula Global there is a tutorial for setting up **Git** to work with VSCode. In this session, the aim is to create our first repository and test some basic **Git** commands from the command line.

4.1 Create an account on GitHub

1. Go to <https://github.com/> and create an account (one per student). It is best to use your university email since university students have free access to several premium features.
2. Log in to your account.

4.2 Create a repository

1. Choose “New repository” in the “+” menu at the top right of the page to create a new repository (**only one per group**).
2. Name the repository “UPF-POO21-GX-YY” and **replace** X with a value in {101,102,201,202,203} (i.e. your lab session group) and YY with a number that will be assigned to you by your lab session teacher.
3. Set the access to “Private”.
4. Click “Create repository”.

4.3 Cloning a repository

The next step is to clone the repository, i.e. to make a local copy of the repository on your machine. Under “Quick setup” are the instructions for cloning a repository onto your local machine. These steps are necessary each time you want to download a local copy of the repository on a new machine. After cloning the repository, create a new directory called **doc** in the repository root. Write down the answers to the questions below and save the answers to a file **answers.txt** in the **doc** directory.

4.4 Share a repository

You have to share the repository with your fellow student and all your teachers (seminar, lab and theory, up to three in total). The email addresses of your teachers are as follows:

Lorenzo Steccanella	lorenzo.steccanella@upf.edu
Bernat Gaston	bernat.gaston@upf.edu
Patricia Carbajo	patricia.carbajo@upf.edu
Federico Heras	federico.heras@upf.edu
Anders Jonsson	anders.jonsson@upf.edu

To share the repository, click “Settings” \Rightarrow “Manage access” \Rightarrow “Invite a collaborator” and enter the emails of the people that you want to share the repository with.

4.5 The structure of a Git project

There are 3 parts that communicate with each other to save files from the local machine to the repository:

4.5.1 Working Directory

The directory in your local machine where you organize, add, delete and modify different files. Also, this is where you update the project from the repository.

4.5.2 Staging Area

The purpose of this area is to list the changes you make in your working directory, which is used to group the modifications we want to submit to the repository in a package.

4.5.3 Repository

The repository is the remote location where `Git` stores the changes from the staging area forever. Every time a change is uploaded to the repository, it is saved as a new version of the project.

4.6 Checking the status

Create a file called `README.md` and use the command `git status`. This file is commonly used to document the progress of the project.

Q1. What do you see on the command screen?

4.7 Adding a file

Now use the command `git add <file>`, and add the created file `README.md`.

Q2. Use the command `git status` again and explain what has changed?

4.8 Comparing files

Edit the README.md file adding a sentence. Now there is a difference between the README.md of your working directory, and README.md in the staging area. To see what these differences are, run the command `git diff <filename>`.

Q3. What is the output of the command? What is the current status?

Now add the README.md to the staging area.

4.9 First commit

All the changes must be in the staging area at this point, so the next step is to upload them to the repository, but first we need to pack everything in a new version using the command `git commit -m <message text>`. The message is used to describe the purpose of the commit. Thus, this commit will wait to be pushed to the repository as a packet, together with a version number that identifies all the previous changes.

Q4. What can be a good message for the first commit? What is the current status?

4.10 The log information

Every time a commit is done, the log is updated, so to access to the previous commits information, we can run the command `git log`. In this log we save the identifier for version control, the author of the commit, the date and its message.

Q5. How many commits there are in the log? What are the first seven digits (letters and numbers) that identify the first README.md commit?

4.11 First push

After committed, we have to run the command `git push` that by default will upload the packet to the “origin” repository.

5 Undo Git Changes

Sometimes we make mistakes and there are two ways to recover:

1. To modify the changes and commit again.
2. To recover an earlier version of the repository.

This section will show how to erase some changes that are saved in the repository.

5.1 Head commit

The first thing we need is to know in which commit we are working on. The command used in this case is `git show HEAD` (HEAD always refers to the working copy).

Q6. Modify one line of the README.md, add to the staging area and commit. Then use the command `git show HEAD`, and explain the output.

5.2 Checkout

Remember the line you modified in the last question. You realize this line is not appropriate for the README.md file, and you want to restore the file to the one from the last commit. To do so, we use the command `git checkout HEAD <filename>`.

5.3 Reset

The command `git reset HEAD <filename>` is used to restore a file in the staging area and leave it equal to the file in the specified commit (in this case HEAD). This command doesn't affect to the working directory.

Q7. Rewrite the sentence you restored with the previous command. After that, add the README.md to the staging area. Then, run a reset in the README.md and explain what happens.

We can use resets in full commits by specifying the SHA (7 digits) code of the commit we want to move back of the commit history. The command is pretty much the same: `git reset <SHA>` (maybe you will need it for your project). To find the SHA of different commits, run the command `git log`.

Tutorials

For a longer explanation of the commands, visit the **Git** handbook on GitHub:

<https://guides.github.com/introduction/git-handbook/>

By default, your machine uses HTTPS to communicate with the repository, which requires you to type your password each time you push changes. Optionally, you can instead use SSH, which makes communication easier but is more work to set up. If you want to use SSH, read the following page:

<https://docs.github.com/en/github/authenticating-to-github/adding-a-new-ssh-key-to-your-github-account>

Codecademy

Practice with the free Git course at <https://www.codecademy.com/learn/learn-git>