# Java Labs 2021
# Run baby, run !

Christophe Barès, Nicolas Papazoglou,
Sylvain Reynal, Antoine Tauvel, ENSEA

October 2021

## 1 Introduction

Welcome to this lab.

Here are some basic rules for this Lab :

- You're now on your fourth year of graduate study. We won't correct missing semicoloms.
- At the first session, your setup must be working, and you must be able to compile and execute a simple "Hello World" program (see next section).
- Your backup, your problem.
- Use a tool to follow your software modification. Git is the one you're looking for.

## 2 Software requirement / Homework

This series of Labs runs on the JavaFx librairies of the Java Language. This library is available for all kind of Operating Systems [1]. The screenshot made here we'll be done using IntelliJ IDE. While you're free to use any IDE you want, we strongly advise you against plain old Geany (or any other IDE without refactoring or code completion ability).

First, we'll need a basic setup :

- Download the last stable version of the Java Development Kit.
- Download the community edition of IntelliJ.
- Download and unzip in a "simple" path the last stable version of JavaFx. Copy in the clipboard the path to the JavaFx Library (it must end with //lib).
- See the tutorial beneath to create a simple "Hello world" application.

Then we'll need a working Hello world graphic application.

## 3 A few things you need to know about JavaFx 2D

The coordinates system is Y-down.

A JavaFx application as this code structure :

In this code, the "main" method is not mandatory, it's here so that you can try some code in standard plain old Java, without JavaFx.

In JavaFx the Application (a class that your primary class must extends) is the main class. This class must Override the start method. In this start method, we set up the Stage (the application) that comprise a

[1]The 3D extension of JavaFx is still not available for Unix / Linux users, but we won't be using it

```java
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{

        primaryStage.setTitle("Hello world");
        Group root = new Group();
        Pane pane = new Pane(root);
        Scene theScene = new Scene(pane, 600, 400,true);
        primaryStage.setScene(theScene);

        primaryStage.show();
    }


    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 1: JavaFx Basic ”Hello world”

Scene, the main container of graphical contents. Here, we just define a pane, (a graphical container of lesser level) and add this pane to a Scene of 600x400 pixel.

The following figures 1 and 2 illustrates the relation between Stage, Scene, and Pane.

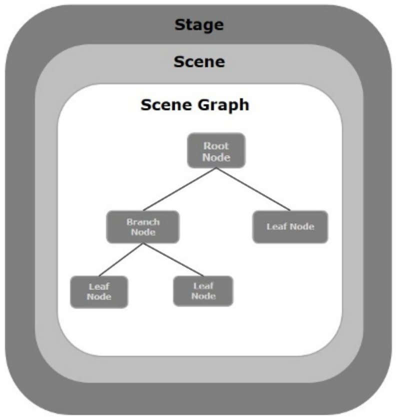

Figure 1: Java Fx Application structure

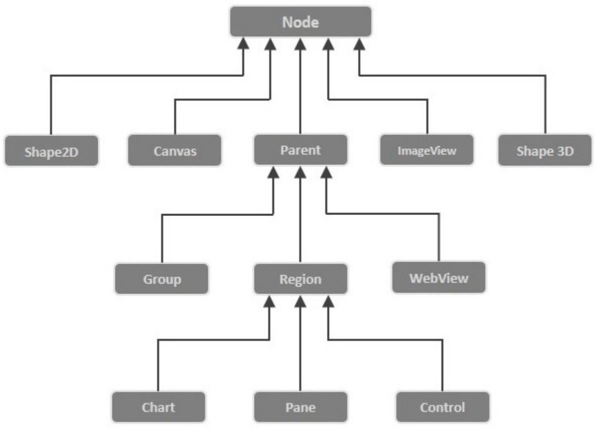

Figure 2: Nodes relation in JavaFx

Tutorial on how to use IntelliJ with JavaFx.

- Download the right version of JavaFx from the web. Unzip the files in a folder you can easily retrive. Locate the /lib folder and Copy his location in the clipboard.
- Create a new project in IntelliJ, in plain Java (do not use the JavaFx project type).
- Add the "Hello world" above to your code. All the "javafx" related instruction should appear in red.
- Open the "Project Structure" Menu. Add the "/lib" path from previous to the library path. Your Hello World should now compile without error. Correct any error you find before the next step.
- Launch the Hello World to create a running configuration.
- In the "Edit configuration" menu, click on Modify Option / Add VM Option.
- The VM option is `--module-path "Your /lib library" --add-modules javafx.controls`.
- Your program should now run normally.

## 4 First sessions - Display image

Two classes are important to display an image : Image helps you open the image file, and ImageView is the class that can be displayed on the Group related to the Scene. In the ImageView, three methods will prove usefull :

- `setViewPort(Retangle2D)` sets a rectangle in the image. Only what's in this rectangle will be displayed.
- `setX(double)` sets on the Group attached to the Scene where the Image will be displayed on X.
- `setY(double)` sets on the Group attached to the Scene where the Image will be displayed on Y.

As always, don't forget to read the javadoc on these methods and on the ImageView class.

Here's a sample code that loads the sprite sheets, crop it on the first image (coordinates (20,0), size(65,100), and displayed it in the center of the screen (200,300).

```
Image spriteSheet = new Image("..\\img\\heros.png");
sprite = new ImageView(spriteSheet);
sprite.setViewport(new Rectangle2D(20,0,65,100));
sprite.setX(200);
sprite.setY(300);
```

On the 3, you'll see how are interconnected the game, the camera, and the background images.

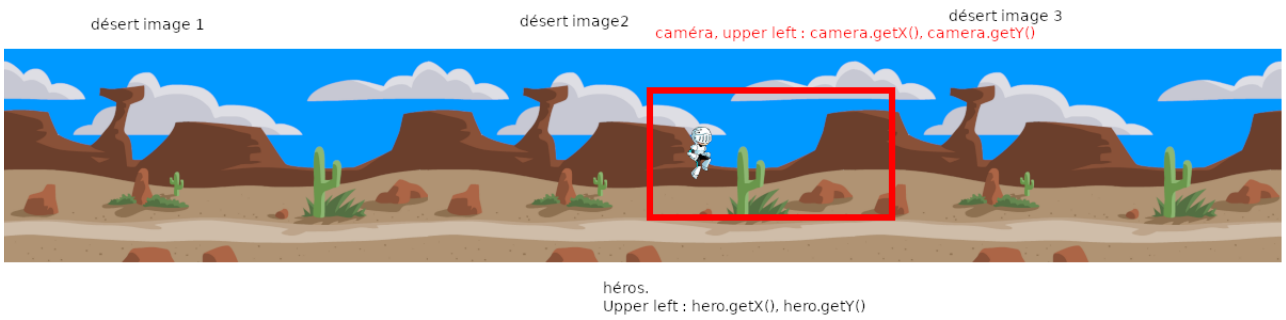
désert image 1
désert image2
désert image 3
caméra, upper left : camera.getX(), camera.getY()
héros.
Upper left : hero.getX(), hero.getY()

Figure 3: Relation between camera and display coordinates

- Create "Camera" class.
- This very basic class (at first) is just two sets of coordinates (x and y), with getter methods.
- The constructor of this class takes two Integer arguments.
- Overload the "toString" method to display x and y comma separated.

- Create a class "GameScene" that extends the Scene class.
- Add to this class one Camera.
- Modify the Main code.
- Test your code.

- Create a class "StaticThing" that we'll use to display static element (background or number of life). This class is defined by two double (sizeX,sizeY), one ImageView.
- Create the constructor of this class. The constructor has many parameters, one of which is a String fileName to point to the Background.
- Create a getter for the ImageView member of the class.
- Add to the gameScene class two StaticThings for the BackGround, one left and one right.

- Inside the GameScene class, instanciate two StaticThing that displays the background (left and right) on the Scene. Create a *render* method in the GameScene that modify every position on the Scene according to the camera.
- Test your code with different value for the camera's position.

Bonus manipulation :

- Inside the GameScene class, instanciate *numberOfLives* small hearts to indicate the remaining lives. *numberOfLives* is set to 3 by the Constructor.
- Test your code.

- Create an abstract class "AnimatedThing" that we'll use as a mold for the heros and its oponent. This class is defined by two double (x,y), one ImageView on the spriteSheet, an Integer that defined his attitude (still, running, jumping up, jumping down).
- Add to this class every number you'll need to display an animation (at least : an index, a duration between two frames, a maximum index, the size of the window, the offset between each frame).
- Create the constructor of this class. Inside this constructor, the view Port of the ImageView is set to the first frame. The constructor has many parameters, one of which is a String fileName to point to the Sheet.
- Create a getter for the ImageView member of the class.

- Create a "Hero" class that extends the AnimatedThing class. His constructor is for now a simple call to the super constructor with the right parameter.
- Inside the GameScene class, instanciate a Hero. Display the Hero on its starting position.
- Test your code with different starting position and different value for the camera's position.

## 5 Animation of the Hero

We now want to Animate the Hero by modifying its index on the SpriteSheet. The AnimationTimer class will help us doing that. Don't forget to read it's documentation on the javadoc.

Add this code to your gameScene class :

This create a new AnimationTimer, a self-called function, whose method (*handle*) will be called every few milliseconds, with a parameter that is the number of nanoseconds ellapsed since last call.

Of course, this timer has to be started, so inside the constructor of the GameScene class, you'll have to had `timer.start();`

The hero is a 75 x 100 pixels box, sets via setViewPort on the *AnimatedThing* class we're working on.

```
    AnimationTimer timer = new AnimationTimer()
                {public void handle(long time){
                        hero.update(time);
                    camera.update(time);
                    gameScene.update(time);
                }
        }
```

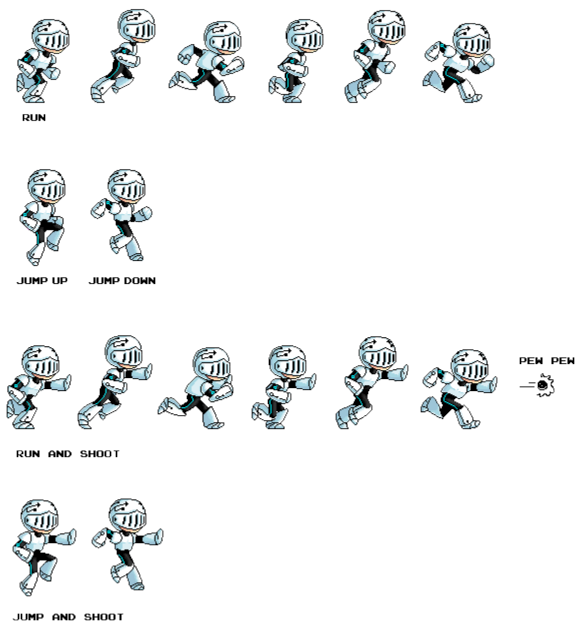

Figure 4: The hero spritesheet

- Instanciate the timer, and test it in the console.
- Reflect on how timer are handled in Microcontroler and in OOP...
- Create the update method on the animatedThings class.
- Create empty update method for the Camera class.
- Create the update method on the gameScene class.
- Visualise your hero running. Isn't he pretty ?

## 6 Add Some physics to the Camera

To spice things up, the Camera is bound to the Hero via a spring-mass like equation system.

So the camera has to memorise the reference to the Hero in order to calculate the return Force.

In physics game simulation, we'll never implement the solved equation. It's far less time-consuming to implements the equation itself. Here's the set of equation we wan't to implements in one dimension :

$$a_x = \frac{k}{m}.(x_{hero} - x) + \frac{f}{m}v_x$$

$$\delta v_x = a_x.\delta t$$

$$\delta x = v_x.\delta t$$

On figure 5 on page 7, you'll see an example of this equation implemented. Each sample being separated by 16 millisecond, you'll see that in this configuration ($\frac{k}{m} = 1$; $\frac{f}{m} = 1.2$), it takes almost 3 seconds to see the hero. This is too slow and you should find the right value.

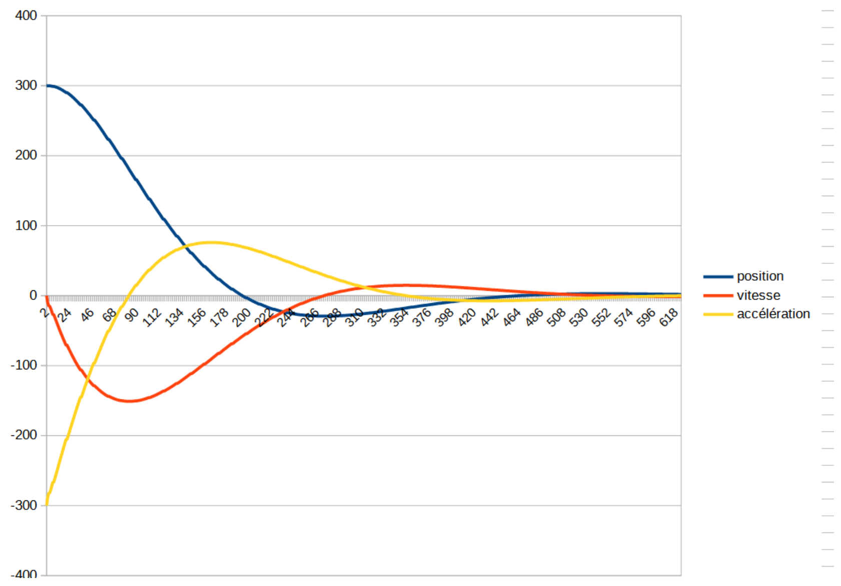

Figure 5: Response from the Camera for a static Runner

- Implements these 6 equations in the update method inside the Camera class.
- Modify the camera at the creation of the game to be slightly of the Hero.
- Test your spring effect.

- Inspired by this example, modify the Hero so that he really runs, first with a constant speed.

## 7 Response to action

Now we have a running hero... He must jump ! First, we need to capture the event that controls the Hero : space bar, tactile screen or click anywhere or click somewhere...

Here's a sample code to do that, that exploits a very usefull feature of Java : a lambda expression.

```
this.setOnMouseClicked( (event)->{
    System.out.println("Jump");
            hero.jump();
            });
```

Read the Javadoc on the setOnMouseClicked method. This method is part of the Scene object. Read some documentation on the internet or ask the teacher on how to use lambda expression in Java.

- Add the jump feature on your Hero. Firstly, add gravity inside the update Method.
- Test gravity by starting the Hero up.
- The road is on pixel 50, so the Hero cannot go lower than pixel 150.
- A jump is a short burst of vertical acceleration.
- Adjust the jump to the size of your enemy.

## 8 Finally the game in itself

We now has to defined hitboxes.

- Create a new Foe class that extends the movingThingClass.
- Add an `ArrayList<Foe>` of Foe to the GameScene class.
- Add a single Foe in the ArrayList, on pixel 250 for example.
- Check its appearance.

- On each update check for each foe in the ArrayList if it intesects with the Hero. For that, add a `Rectangle2D getHitBox()` method to your AnimatedThing class.
- Detect collision in the console.

- Add a double `invincibility` to your Hero. Each time a collision is detected, invincibility is set to 25000000000 (invicibility time in nanoseconds).
- Each time the Hero is updated we substract the time to this value.
- Once it reaches a negative value, it stops.
- Add the isInvincible() method to the Hero class. If the hero isInvincible, don't check for the collision in the GameScene class.

- Finally, create in the GameScene constructor a list with a random number of Foes. To ensure there are no collision between the Foes, you can just calculate a random number with a minimum value between two Foes.

## 9 Game interface

If you're there, then congratulations are in order ! All you'll need to do now is to add a welcome scene, and a victory.

Hope this little experiment pleased you and that you've learned many programmation technics !