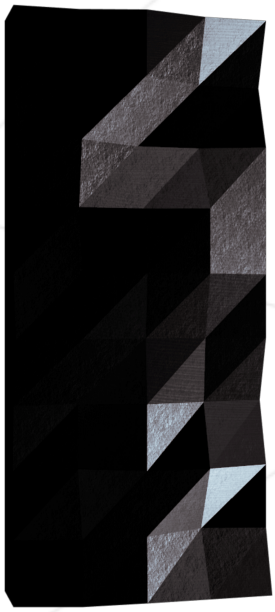

Avaj Launcher

Enter the Java path

Alex alex@academyplus.ro
42 Staff pedago@42.fr

Summary: This project is the introduction to the **Java** *world at 42.*

Contents

Chapter I

Forewords

Maverick: "This is what I call a target rich environment."
Goose: "You live your life between your legs Mav."
Maverick: "Goose, even you could get laid in a place like this."
Goose:" Hell, I'd be happy to just find a girl that would talk dirty to me."

Viper: "Good morning, gentlemen, the temperature is 110 degrees."
Wolfman: "Holy shit, it's Viper!"
Goose: "Viper's up here, great... oh shit..."
Maverick: "Great, he's probably saying, "Holy shit, it's Maverick and Goose." "
Goose: "Yeah, I'm sure he's saying that."

Chapter II

Introduction

This is the first project in a series of 4 projects with a focus on `Java` and the first project created by Academy+Plus. We are the counterpart of 42 located in Cluj-Napoca, Romania, and the first implementation of the 42 program outside of France. We started in 2014, and each year with a growing number of students that apply and also a growing number of students we can accomodate. Feel free to pay us a visit when you feel like traveling (Ping me if you need info about cheap flights).

ACADEMY+PLUS

In order to create great software, one doesn't only write code, one needs to design it first. This project will introduce to you the concept of UML class diagrams and object oriented design patterns, all implemented in the `Java` language. Writing `Java` code is easy since it's very similar to `C`, but the real challenge is to write good OO code in `Java`.

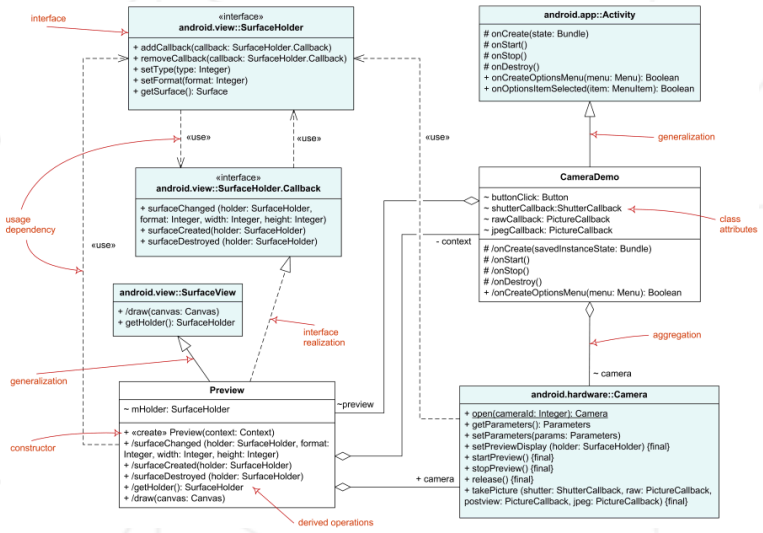

Figure II.1: A sample UML diagram. It can contain many more symbols.

You will have to implement a minimal aircraft simulation program based on a given UML class diagram. The `Unified Modeling Language` is used in software engineering for visualizing the design of an application. It offers programmers a standardized way of thinking about Object Oriented software, before writing any line of code.

`Thinking in Java`

Chapter III

Goals

Frankfurt airport recently discovered that due to frequent weather changes they have a bottleneck on some of the landing tracks. In order to find a solution, they first need to know which scenarios create the worst bottlenecks. So they decided to use a simulator where they configure and analyze multiple scenarios and hope that this will highlight them were the real problem is.

So they reach out to their local top software shop and assign them this task. Here the chief designer starts working on the concept and after analysing all the facets of the software, he makes some design decisions which, he then passes on to you in order to create the simulator.

Since the software will run on a multitude of operating systems in a very strict enterprise environement, he decides to use a classic Object-Oriented language: Java.

He will provide you:

- the UML class diagram
- the must-have Object Oriented design patterns

What you need to know in order to be on the team that develops the simulator is:

- Interpreting class diagrams - this is the way the architect uses to communicate with you
- Observer, Singleton and Factory design patterns - he knows that this will not be the final version of the simulator and he aims to extend it in order to address other needs that the airport may have
- The basic syntax of `Java` and some of the core features of the language. - this is obvious, since this is the language agreed upon

Object Oriented Design and design patterns are topics that cover by themselves thousands of pages, so feel free to explore this domain and you will discover a new way of thinking about software engineering. And who knows? Maybe you will take that architects place one day.

Only a good implementation will be accepted, since this is the top software shop in the city. For this to happen, it will have a clean design, will be easy to read and understand by your peers and will be easy to change in case the requirements are modified.

Gang of Four

Even though the mirage of a powerful IDE can be tempting, I strongly advise you to work with a nice text editor. It is important in the begging to understand the internal workings of Java, and an IDE will hide them from you.

Chapter IV

General instructions

- You are allowed to use language features up to Java 7 included.
- You are not allowed to use any external libraries, build tools or code generators.
- Do not use the default package.
- Create your own relevant packages following the `Java` package naming conventions.
- Java is compiled into an intermediate language. This will generate some .class files. Do not commit them on your repository!
- Make sure you have javac and java available as commands in your terminal.
- Compile the project running the commands bellow in the root of your project folder.

```
$find -name *.java > sources.txt
$javac -sourcepath @sources.txt
```

Chapter V

Mandatory part

You need to implement an aircraft simulation program based on the class diagram provided to you. All classes are required to be implemented respecting every detail provided in the diagram. Feel free to add more classes or include additional attributes if you think it is necessary, but do not change access modifiers or the class hierarchy for the classes provided in the diagram.

V.1 Program behaviour

Your program will take one and only one argument from the command line. This argument represents the name of a text file that will contain the scenario that needs to be simulated. You can find an example file provided with the subject.
Executing the program will generate a file `simulation.txt` that describes the outcome of the simulation.

Example:

```
$java ro.academyplus.avaj.simulator.Simulator scenario.txt
$cat -e simulation.txt
Tower says: Baloon#B1(1) registered to weather tower.
Tower says: JetPlane#J1(2) registered to weather tower.
Tower says: Helicopter#H1(3) registered to weather tower.
Tower says: Helicopter#H4(4) registered to weather tower.
Baloon#B1(1): Let's enjoy the good weather and take some pics.
JetPlane#J1(2): It's raining. Better watch out for lightings.
Helicopter#H1(3): This is hot.
Helicopter#H4(4): My rotor is going to freeze!
Baloon#B1(1): Damn you rain! You messed up my baloon.
JetPlane#J1(2): OMG! Winter is coming!
Helicopter#H1(3): This is hot.
Helicopter#H4(4): My rotor is going to freeze!
Baloon#B1(1): It's snowing. We're  gonna crash.
JetPlane#J1(2): It's raining. Better watch out for lightings.
Helicopter#H1(3): This is hot.
Helicopter#H4(4): My rotor is going to freeze!
Baloon#B1(1): Damn you rain! You messed up my baloon.
Baloon#B1(1) landing.
Tower says: Baloon#B1(1) unregistered from weather tower.
JetPlane#J1(2): OMG! Winter is coming!
Helicopter#H1(3): This is hot.
Helicopter#H4(4): My rotor is going to freeze!
```

V.2 Scenario file

The first line of the file contains a positive integer number. This number represents the number of times the simulation is run. In our case, this will be the number of times a weather change is triggered.
Each following line describes an aircraft that will be part of the simulation, with this format: `TYPE NAME LONGITUDE LATITUDE HEIGHT`.

V.3 Weather generation

There are 4 types of weather:

- RAIN
- FOG
- SUN
- SNOW

Each 3 dimensional point has its own weather. Feel free to use whatever generation algorithm you want, as long as it takes into account the point's coordinates.

V.4 Aircrafts

- JetPlane:
  - `SUN` - Latitude increases with 10, Height increases with 2
  - `RAIN` - Latitude increases with 5
  - `FOG` - Latitude increases with 1
  - `SNOW` - Height decreases with 7
- Helicopter:
  - `SUN` - Longitude increases with 10, Height increases with 2
  - `RAIN` - Longitude increases with 5
  - `FOG` - Longitude increases with 1
  - `SNOW` - Height decreases with 12
- Baloon:
  - `SUN` - Longitude increases with 2, Height increases with 4
  - `RAIN` - Height decreases with 5
  - `FOG` - Height decreases with 3
  - `SNOW` - Height decreases with 15

V.5 Simulation

- Coordinates are positive numbers.
- The height is in the 0-100 range.
- If an aircraft needs to pass the upper limit height it remains at 100.
- Each time an aircraft is created, it receives a unique ID. There can't be 2 aircrafts with the same ID.
- If an aircraft reaches height 0 or needs to go below it, the aircraft lands, unregisters from the weather tower and logs its current coordinates.
- When a weather change occurs, each aircraft type needs to log a message, as seen in the example. The message format is: `TYPE#NAME(UNIQUE_ID): SPECIFIC_MESSAGE`. A funny message will be appreciated during the correction.
- Each time an aircraft registers or unregisters to/from the weather tower, a message will be logged.

V.6 Validation

The input file needs to be validated. Any abnormal behaviour due to invalid input data is not acceptable. If the input file data is not correct the program stops execution. Any error messages will be printed to the standard output.

Figure V.1: On Coruscant they used a simillar simulator.

Chapter VI

Bonus part

Bonus points will be given if:

- You create your own custom exceptions for treating abnormal behaviour.
- Your program can read the input file contents when they are encrypted in MD5.

Chapter VII

Turn-in and peer-evaluation

Turn your work in using your `GiT` repository, as usual. Only work present on your repository will be graded in defense.