# Assignment 3

Web Architectures 2018-2019

Nicolò Pomini

October 6, 2018

## 1 Introduction

The goal of this assignment is to analyze a jar file using the technique of the reflection, or to examine the behaviour and the structure of a program at run-time. In particular, the problem asks to discover which are the name of the two classes included in a given jar file and to say from which class they inherit from. Furthermore, for the class called *Mistery*, the list of the methods exposed and the functionality of the latter are expected to be discovered.

## 2 Explanation

The problem is solved writing a Java program that does the inspection using reflection. It performs several steps in order to fulfill what is requested by the problem statement.

First of all, the jar file is read, to find out the name of the two classes: all the jar entries are scanned, and for those whose name ends with the string ".class" the latter is saved into a list.

Secondly, the jar file is loaded to start to perform the reflection. For each name discovered before, an instance of the class `Class` is created, and from the latter the name of its superclass is found and printed.

Thirdly, the class *Mistery* is analyzed. In particular, the list of its declared method is extrapolated and printed, and finally each method is tested.

## 3 Implementation

Each part listed in the Explaination Section is implemented in a file called `Inspector.java`. This program takes as argument the absolute path of the jar file (in case no arguments are passed the program aborts).

To read the jar file to discover the names of the two classes, two types of objects are used: a `JarInputStream`, to open and read the file; and a `JarEntry`, to analyze each part of the latter. Once a class is found, its name is saved into a list of strings. This is the piece of code that performs this operation:

```
ArrayList<String> classes = new ArrayList<>();
JarInputStream crunchifyJarFile = new JarInputStream(
                        new FileInputStream(jarpath));
```

```
JarEntry crunchifyJar = crunchifyJarFile.getNextJarEntry();
while(crunchifyJar != null) {
    if ((crunchifyJar.getName().endsWith(".class"))) {
        //for every class found
        //parse the class name and add it to the list
        String className = crunchifyJar.getName().replaceAll("/", "\\.");
        String myClass = className.substring(0, className.lastIndexOf('.'));
        classes.add(myClass);
    }
    crunchifyJar = crunchifyJarFile.getNextJarEntry();
}
```

Every jar entry whose name ends with the string ".class" is a class, thus its name is parsed and added to the list of names.

Now the jar is loaded in order to perform the reflection. The following code is able to load jars that are not in the `classpath`, using an `URLClassLoader` object. At this point, for each name previously found, an instance of the class `Class` is created, to discover its superclass. Finally, the class name and the superclass name are printed out.

```
for(String c: classes) {
    try {
        //for each class, print its name and its superclass
        Class classe = Class.forName(c, true, loader);
        String superclass = classe.getSuperclass().getName();
        System.out.println("Class name: " + c + "\t-> Inherits from " + superclass);
    } catch (ClassNotFoundException ex) {
        System.err.println("Class not found");
    }
}
```

As regards the `Mistery` class, all its methods are requested, using the following instruction:

```
Method m[] = classe.getDeclaredMethods();
```

and printing the signature of each entry of this array.

Finally, every method is tested. The first one is a static method (called `create`) that returns an instance of the `Mistery` class, on which all the other instance methods are then tested. To try the first one, the following code is used:

```
Method method = classe.getMethod("create", null);
Object o = method.invoke(null, null);
```

while for the other methods the following code is used:

```
method = classe.getMethod("setColor", int.class);
method.invoke(o, 1);
```

where the parameters of `getMethod` are the name of the method and the type of its parameters respectively, and the parameters of `invoke` are the object upon which the method is called (in the first case `null` is passed because it is a static method), and the actual value of the parameter.

# 4 Deployment

To deploy the project, firstly is necessary to compile the java program that performs the jar analysis. To do so, once the current directory is the main directory of the assignment, the following commands have to be exectued:

```
cd src
javac jarinspection/Inspector.java
```

After that, the program can be executed with the command

```
java jarinspection.Inspector jar_path
```

where `jar_path` is the absolute path of the jar to be analyzed. For example, on my computer I launch the program writing:

```
java jarinspection.Inspector ~/Desktop/mistery.jar
```

Regarding class names and superclass names, this is what the program gives as output:

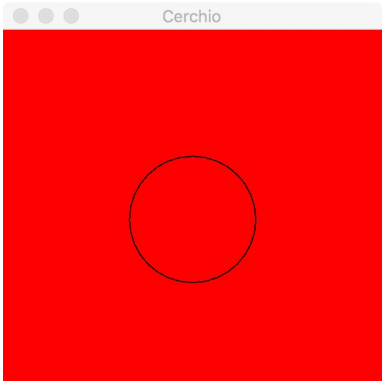
```
Class name: p1.Mistery  -> Inherits from java.lang.Object
Class name: p1.MyPane   -> Inherits from javax.swing.JPanel
```

The class `Mistery` exposes the following methods (taken from the output of the program):

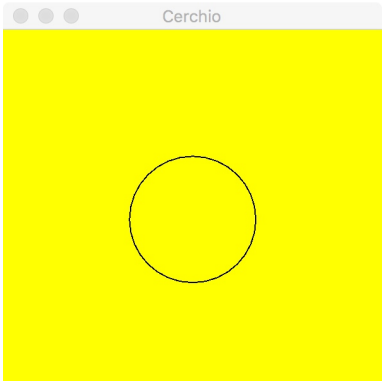
```
Methods of the class Mistery:
public static p1.Mistery p1.Mistery.create()
private void p1.Mistery.resize()
public void p1.Mistery.setColor(int)
public void p1.Mistery.setC(int)
public void p1.Mistery.setW(int)
public void p1.Mistery.setH(int)
public void p1.Mistery.setCW(int)
public void p1.Mistery.setCH(int)
```

Finally, this is what the various methods do:

- `create` returns an instance of a `Mistery` object, which is a canvas with a white background and a circle placed in the middle, the latter with a black edge.
- `setColor` change the background color of the object. In particular, when the value 0 is passed, the background color becomes red, while with value 1 it becomes yellow. These results can be seen in Figure 1.
- `setW` changes the width of the window (an example can be seen in Figure 2.a).
- `setH` changes the height of the window (an example can be seen in Figure 2.b).
- Regarding the other methods, no changes in the graphical aspect or in the behaviour of the object were seen. Several tests were tried, with the input in the order of units, tens, hundreds and thousands, both positives and negatives, but nothing changed.

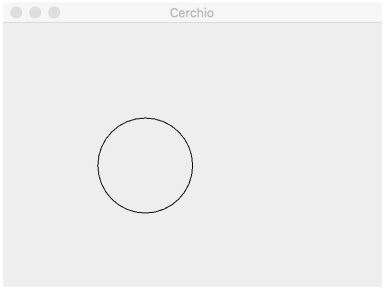



(a) Red background with `setColor(0)`

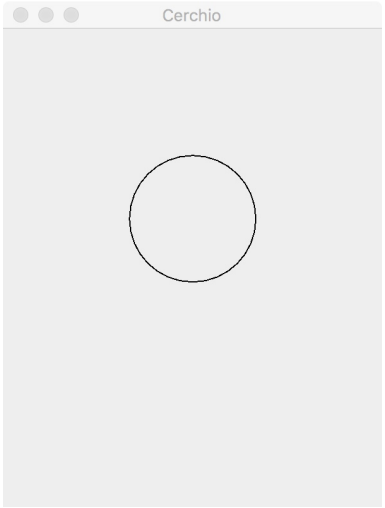



(b) Yellow background with `setColor(1)`

Figure 1: The results after the call to the method `setColor`.




(a) The width of the window changed after the call `setW(400)`




(b) The height of the window changed after the call `setH(400)`

Figure 2: The results after the call to the methods `setW` and `setH`.

# 5 Comments and notes

This program is able to load jar files that are not include in the `classpath` variable. In this way is not necessary to include manually the jar in the `classpath`, but on the other hand the absolute path of the jar has to be given as argument. For this trade-off I chose the way that seemed for me the easiest for the final user of the program: in fact, in my opinion is more immediate to specify an absolute path, instead of include a file into the `classpath`.