AP-23 – Programming Assignment #1- v1

November 15, 2023

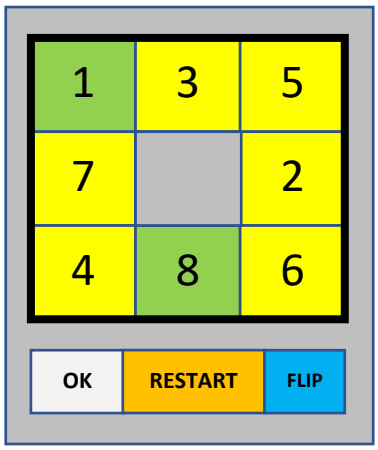
This programming assignment consists of three exercises, on Java Beans, Java Reflection and Annotations, and on Haskell, respectively.

Exercise 1 (Java Beans) – The 8 puzzle

The 8 puzzle (picture on top left) is a reduced version of the more famous 15 puzzle (on the top right). Starting from a random configuration of the tiles, the puzzle consists of reaching the final configuration (the one shown above) with a sequence of moves. Each move consists of sliding one tile on the hole, thus exchanging the positions of that tile and the hole.

We want to implement the 8 puzzle using Java Beans. The project will only provide a usable board to try to solve the puzzle, it will not address its automatic resolution.

The system is made of a graphical dashboard, **`EightBoard`**, containing the board, an **`EightController`** label, and two buttons: **`RESTART`** and **`FLIP`**, as shown in the figure to the right. The board is made of 9 buttons, which must be instances of a Java bean called **`EightTile`**, and that we will call *tiles* in the following. Each tile implements therefore the same logic. Conceptually, the **`EightBoard`** dashboard displays the board, while the **`EightController`** label monitors that only legal moves are triggered by clicking the tiles.



Requirements for the `EightTile` bean

Tiles must inherit from **`JButton`**. A tile has (at least) two **private** properties: **`Position`** and **`Label`**, which hold integer values in the range [1, 9]. **`Position`** is a constant: it is set at startup by the constructor, and it identifies a specific position on the board in the order shown in the top left picture (where the hole is in position 9). **`Label`** is, as property, **both bound and constrained** (in the JavaBean's sense).

The background color of a tile must be grey if the **`Label`** is 9 (we call this tile *the current hole*), green if **`Position`** = **`Label`** (and they are different from 9), and yellow otherwise. The text of the tile must be equal to the value of **`Label`**, with the exception of the current hole, whose text is empty. Background color and text of each tile must change if **`Label`** is changed, in order to preserve these constraints. Note that the in this way the final configuration is the only one where all tiles except the hole are green.

When the player clicks on a tile ("I want to slide that tile to the hole"), the **`Label`** property is changed to 9 (i.e., the cell becomes itself the current hole), ***if this change is not vetoed*** [(1)]. Only if the change is not vetoed, the tile must pass, in some way, its previous label value to the current hole, which must change its label to that value[(2)]. If instead the change is vetoed, the tile "flashes", for example by changing the color to red for half a second.

Tiles must also provide support for a "restart" action, with a method that takes as argument a permutation of [0,9] and sets the label to an initial value. This action will be triggered by the **`Restart`** event of the board, thus tiles have to register as listeners for this event.

(1) By the logic of the game, the change of the **Label** property to 9 should be vetoed if the tile is the current hole, or if the tile is not adjacent to the current hole.
(2) You should decide how to pass the label value from the clicked tile to the old hole. This can be implemented in any way: public methods are accepted, but event-based communication is more appreciated.

Requirements for the `EightController` bean.

The **EightController** is a bean that has the graphical appearance of a label (e.g., it can extend **JLabel**). It has to check that only legal moves are performed. To this aim, it must be registered as **VetoableChangeListener** to all the tiles.
At the beginning the controller displays "**START**". Every time a tile is clicked, it must veto the change if the tile is the hole or it is not adjacent to the hole, displaying "**KO**". Otherwise, it must display "**OK**".
The controller must also provide support for a "restart" action that restores its internal information about the configuration when the game is restarted.
Warning: The controller cannot read the labels of the tiles using methods. You should decide which kind of information the controller has to keep about the configuration of the board to implement correctly the above veto policy. Several choices are possible: the simpler the better.

Requirements for the `EightBoard` dashboard

The main class of the application is the **EightBoard** dashboard, that must be defined as extending **JFrame**. It dispays a grid of 3x3 tiles, an **EightController** label, a **RESTART** button, and a **FLIP** button, as shown in the figure above.
When the **RESTART** button is clicked, a random configuration (i.e., a random permutation of [0, 9]) is passed to all beans registered as listener to the **Restart** event.
At startup the board initializes the grid with the nine **EightBoard** beans, passing to them their position as initial value of **Position**. All tiles and the controller are registered as listeners to the **Restart** event, and such an event is fired to complete the initialization of the board with a random configuration. Also, the controller is registered as **VetoableChangeListener** to all the tiles.

The `Flip` button

In the 8 (or 15) puzzle, from half of the initial configurations the final one cannot be reached with any sequence of moves (see the Wikipedia page for more infos). This can be frustrating for the player.
The **Flip** button of the dashboard, when clicked, switches the labels of tiles in position 1 and 2, but only if the hole is in position 9, otherwise it has no effect. Switching the position of two non-hole tiles guarantees that if the previous configuration was not solvable, the new one is.
You must implement the effect of clicking the **Flip** button as described. Any correct solution is accepted. Solutions which follow the JavaBean patterns are more appreciated (e.g., letting the Controller to check if the flipping is permitted, or using event-based communications between beans).

Solution format

Adequately commented source files for the beans and three **jar** archives, one for each bean, plus a brief document **of at most two pages** (in PDF) reporting the main design decisions. In particular, describe how did you implement the effect of clicking the **Flip** button.

Exercise 2 (Java Reflection and Annotations) – XML serialization

XML is a meta-language which can be used to describe structured documents in a machine-readable way. Information is packaged in elements. For a simple introduction to XML elements look at the following page:

[https://www.w3schools.com/xml/xml_elements.asp]

An XML document contains one or more (possibly nested) elements. For instance, the following is a valid XML document (the first line is not mandatory, but it is better to include it at the beginning of the file):

```
<?xml version="1.0" encoding="UTF-8"?>
<Student>
     <firstName type="String">Jane</firstName>
     <surname type="String">Doe</surname>
     <age type="int">42</age>
</Student>
```

As XML is widely used to export data, you must implement a Java serializer to export an array of objects in XML format. (Note that there are several Java APIs for XML serialization. Here we ask something different and much simpler).

Write a Java class **`XMLSerializer`** which offers a static method with signature

`void serialize(Object [ ] arr, String fileName)`

For each object in the array **`arr`**, the method should introspect its class searching for information (provided using annotations) to serialize the object. The output must be an XML file called **`fileName.xml`** containing one element for each object in **`arr`**. Each element must have as main tag the name of the class of the corresponding object.

Annotations are as follows:

- **`@XMLable`** provides information about the class. The presence of this annotation says that the objects of this class should be serialized. In this case the corresponding element will contain other elements for the instance variables, if any. If instead the annotation is absent, the element corresponding to the object must contain only the empty element **`<notXMLable />`**.
- **`@XMLfield`** identifies serializable fields (i.e., instance variables, only of primitive types or strings). The presence of this annotation states that the field must be serialized. The annotation has a mandatory argument **`type`**, which is the type of the field (a **`String`**, for example **`"int"`**, **`"String"`**,…), and an optional argument **`name`**, also of type **`String`**, which is the XML tag to be used for the field. If the argument is not provided, the variable's name is used as a tag.

Once all the information about the class is collected and analyzed, the method serializes the object producing the corresponding XML element, and then proceeds with the next element of the array.

Note: If the array contains several objects of the same class the introspection of that class has to be made only once.

As an example, consider the XML element above: it should be the result of serializing an object of the following Java class, created by calling the constructor with parameters **"Jane"**, **"Doe"**, and **42**:

```
@XMLable
public class Student {
     @XMLfield(type = "String")
     public String firstName;
     @XMLfield(type = "String", name = "surname")
     public String lastName;
     @XMLfield(type = "int")
     private int age;
     public Student(){}
     public Student(String fn, String ln, int age) {
          this.firstName = fn;
          this.lastName = ln;
          this.age = age;
     }
}
```

Solution format

- The Java files defining the annotations **@XMLable** and **@XMLfield**
- The class **XMLSerializer.java**

Additionally, for testing:

- The class **Student** above, and at least one additional class annotated as described above. The **@XMLfield** annotation must be used, with fields of at least two different data types, and both using the optional argument and not using it.
- A main class building an array of objects and invoking **XMLSerializer.serialize()** on it. The array must contain **at least**: (1) one object of the above class Student generating the element in the previous page; (2) one object of a non-XMLable class; (3) one or more objects of the class of the previous point.

Exercise 3 (Haskell) – Multisets in Haskell

This assignment requires you to implement a type constructor providing the functionalities of *multisets* (also known as *bags*), that is, collections of elements where the order does not count, but each element can occur several times. Your implementation must be based on the following concrete Haskell definition of the `MSet` type constructor:

```
data MSet a = MS [(a, Int)]
            deriving (Show)
```

Therefore, an `MSet` contains a list of pairs whose first component is an element of the multiset, and the second component is its *multiplicity*, that is the number of occurrences of such element in the multiset. An `MSet` is **well-formed** if for each of its pairs `(v,n)` it holds `n > 0`, and if it does not contain two pairs `(v,n)` and `(v',n')` such that `v = v'`.

Part 1: Constructors and operations

The goal of this exercise is to write an implementation of multisets represented concretely as elements of the type constructor `MSet`.

- Implement the following constructors:
  - `empty`, that returns an empty `MSet`
- Implement the following operations:
  - `add mset v`, returning a multiset obtained by adding the element `v` to `mset`. Clearly, if `v` is already present its multiplicity has to be increased by one, otherwise it has to be inserted with multiplicity 1.
  - `occs mset v`, returning the number of occurrences of `v` in `mset` (an `Int`).
  - `elems mset`, returning a list containing all the elements of `mset`.
  - `subeq mset1 mset2`, returning `True` if each element of `mset1` is also an element of `mset2` with the same multiplicity at least.
  - `union mset1 mset2`, returning an `MSet` having all the elements of `mset1` and of `mset2`, each with the sum of the corresponding multiplicites.
- Class Constructor Instances
  - Define `MSet` to be an instance of the class constructor `Eq`, implementing equality as follows: two multisets are equal if they contain the same elements with the same multiplicity, regardless of the order.
  - Define `MSet` to be an instance of the constructor class `Foldable`. To this aim, choose a minimal set of functions to be implemented, as described in the documentation of `Foldable`. Intuitively, folding a multiset with a binary function should apply the function to the elements of the multiset, ignoring the multiplicities.
  - Define a function `mapMSet` that takes a function `f :: a -> b` and an `MSet` of type `a` as arguments, and returns the `MSet` of type `b` obtained by applying `f` to all the elements of its second argument. Explain (in a comment in the same file) why it is not

possible to define an instance of `Functor` for `MSet` by providing `mapMSet` as the implementation of `fmap`.

Important: All the operations of the present exercise that return an `MSet` must ensure that the result is *well-formed*, as defined above. Your code should not use the Haskell module Data.MultiSet or other similar modules, but it can use the functions of the Prelude.

Solution format: A Haskell source file called `MultiSet.hs` containing a Module (see Section "Making our own modules") called `MultiSet`, defining the data type `MSet` (copy it from above) and *at least* all the functions described above. The module can include other functions as well, if convenient.

Note: The file has to be adequately commented, and each function definition must be preceded by its type, as inferred by the Haskell compiler.

Part 2: Testing multisets

[For this exercise you have to download and unzip the archive **`aux_files.zip`**]

The goal of the exercise is testing the implemented functionalities. In a file named `TestMSet.hs`, import `MultiSet.hs` and

1. Define a function `readMSet` that reads a text file whose name is passed as argument (as a string), and returns a new `MSet` containing the ***ciao*** of all the words of the file, each with the corresponding mutiplicity. [Given a string `str`, we define its ***ciao*** (*characters in alphabetical order*) as the string having the same length of `str` and containing all the characters of `str` in lower case and alphabetical order. As an example, the *ciao* of "Hello" is "ehllo". A *ciao string* is a string that is equal to its *ciao*. Clearly, two strings have the same *ciao* if and only if each one is an anagram of the other.]
2. Define a function `writeMSet` that given a multiset and a file name, writes in the file, one per line, each element of the multiset with its multiplicity in the format "`<elem> - <multiplicity>`".
3. Define a function `main :: IO()` which does the following:
    a. Using `readMSet`, from directory `aux_files` it loads files `anagram.txt`, `anagram_s1.txt`, `anagram_s2.txt` and `margana2.txt` in corresponding multisets, that we call `m1`, `m2`, `m3` and `m4`, respectively;
    b. Exploiting also the functions imported from `MultiSet.hs`, it checks the following facts and prints a corresponding comment:
        i. Multisets `m1` and `m4` are not equal, but they have the same elements;
        ii. Multiset `m1` is equal to the union of multisets `m2` and `m3`;
    c. Finally, using `writeMSet` it writes multisets `m1` and `m4` to files `anag-out.txt` and `gana-out.txt`, respectively.

For reading and writing files you can use the functions `readFile` and `writeFile` of the Haskell Prelude (https://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html).

Solution format: A Haskell source file `TestMSet.hs` with the functions described above, which can be executed using `runghc`

(see https://downloads.haskell.org/~ghc/9.0.1/docs/html/users_guide/runghc.html) **Note:** The file has to be adequately commented, and each function definition has to be preceded by its type, as inferred by the Haskell compiler.