Universidad Carlos III de Madrid

Systems Programming
Audiovisual Systems Engineering, Communications Systems Engineering, Telecommunication Technologies Engineering and Telematics Engineering Degrees

Leganés, May 20th, 2014.	Final Exam. Ordinary Call. Problems.
Duration: 120 min.	Maximum Grade: 5 points out of 10 of the exam.

Problem 1 (1.25 points)

We want to code several classes to represent the color concept within a computer program. Since there exist several models to represent a color, a generic class, called *Color*, will be coded. This class will be common to all the models. Then, there will be several classes inheriting from *Color*, each of them representing the color with a different model. Specifically, in this problem we will work with two different color models:

- Gray scale model: a color is defined by means of a unique value, called luminance. The luminance is a real number, taking values on the range between 0 and 1, both inclusive. This model can only represent black, white and diverse gray colors.
- RGB model: a color is defined as a combination of three values: red intensity, green intensity and blue intensity. Each intensity is a real number taking values on the range between 0 and 1, both inclusive.

Write the code for the *Color*, *GrayscaleColor* and *RgbColor* classes. *Color* class must be such that it is not possible to create instances of it, and that the rest of the classes inherit from it. Take into account the following requirements:

- Every color, whichever model it is based on, must have a label (of *String* type) that defines the color. This label will allow a person to recognize it (e.g., a color might be labeled as "cobalt" to indicate that it is a cobalt blue). Since assigning a label to a color will not always be possible, the label can be just an empty *String* in some cases.
- The classes must have the following constructors:
  - For the *Color* class, you must decide if it needs constructors, and if affirmative, which parameters they must have.
  - For the *GrayscaleColor* class, there must be two constructors: one that receives only the luminance value (the label value will be an empty *String*), and another constructor that receives the luminance value as well as a label.
  - For the *RgbColor*, there must be two constructors: one that receives the values for the red, green and blue intensities (the label value will be an empty *String*), and another constructor that receives those intensities and a label.
- All colors must have a `public String toString()` method. This method must return a textual representation of the color, similar to the one in the following examples:
  - In the *Color* class: "`Color cobalt`", where "cobalt" is the label of the color.
  - In the *GrayscaleColor* class: "`Color lightgray <luminance 0.8>`", where "lightgray" is the color label, and 0.8 its luminance.

- In the *RgbColor* class: "`Color cobalt <RGB 0.0, 0.278, 0.671>`", where "cobalt" is the color label, 0.0 its red intensity, 0.278 its green intensity, and 0.671 its blue intensity.

- All colors must have a `public GrayscaleColor toGrayscale()` method. This method returns a new instance of a *GrayscaleColor* color type. This new instance contains the result of translating the color to grayscale, with an empty *String* as label value. The translation is done as follows:

  - In the *Color* class: there is not enough information to provide the code of this method in this class.
  - In the *GrayscaleColor* class: a new object with the same luminance value is returned.
  - In the *RgbColor* class: a new object, which luminance value is given by the formula $0{,}299r + 0{,}587g + 0{,}114b$, is returned. r, g and b represent the red, green and blue intensities, respectively.

Besides the above requirements, your solution must comply with the following constraints:

- All attributes of the classes must be private.
- You should avoid attributes or code duplicates, when possible.
- You should foresee that, in the future, new color models might be added. Since those new models will inherit from *Color*, they will be automatically forced to implement the `GrayscaleColor toGrayscale()` method.
- Assume that the values provided to the constructors arguments will always be correct. Therefore, you do not need to check if they are valid.

Problem 2 (1.25 points)

Consider the *Stack* interface and an implementation, called *LStack*, given below.

```
public interface Stack<E> {
    boolean isEmpty();
    int size();
    void push(E info);
    E pop();
    E top();
}
```

```
public class LStack<E> implements Stack<E> {
    public LStack() {
        (...)
    }

    (...)
}
```

Assume that the `push` method in the *Stack* interface does nothing if invoked with a *null* argument. You can also assume that `pop` and `top` methods return `null` if the stack is empty.

It is required to write the code for a new class, called *StackUtils*, which definition is shown below:

```
public class StackUtils<E> {
    (...)
}
```

Write a *split* method in the *StackUtils* class. This method receives as arguments a stack $\mathcal{P}$ of generic elements, `E`, and an integer number, n. The method returns a stack $\mathcal{P}'$ of generic elements, `E`. Make sure, in the definition of the method, that it can receive objects of any implementation of the *Stack* interface. The return type must also comply with this requirement.

The method must extract the first n elements from the $\mathcal{P}$ stack, and return a new $\mathcal{P}'$ stack, which must contain those n elements in the same order they were in the $\mathcal{P}$ stack (i.e., the top element of $\mathcal{P}'$ must be the element that was at the top of $\mathcal{P}$). When the method finishes, the $\mathcal{P}$ stack must have the same elements than in the beginning, in the same order, but without the n extracted elements.

If n is lower or equals to zero, $\mathcal{P}'$ will be an empty stack and $\mathcal{P}$ will remain unchanged. If n is greater or equals to the $\mathcal{P}$ stack size, the $\mathcal{P}'$ must contain all the elements from $\mathcal{P}$ in the same order, and $\mathcal{P}$ will become empty.

You can see an example of use of this method below:

```
Stack<String> stack = new LStack<String>();
StackUtils<String> utils = new StackUtils<String>();
stack.push("A");
stack.push("B");
stack.push("C");
stack.push("D");
Stack<String> result = utils.split(stack, 2);
System.out.println(stack.pop());              // prints "B"
System.out.println(stack.pop());              // prints "A"
System.out.println(stack.isEmpty());          // prints "true"
System.out.println(result.pop());             // prints "D"
System.out.println(result.pop());             // prints "C"
System.out.println(result.isEmpty());         // prints "true"
```

Using a third auxiliary stack might help you to achieve the required order of the elements in $\mathcal{P}'$.

Efficiency will be taken into account during the evaluation of your solution, i.e., avoid unnecessary stack operations.

Problem 3 (2.5 points)

We want to write the code for a binary decision tree. The tree will represent a questions & answers game. You can find an example below:

```
                   Is it a mammal?
             _____________|______________
            |                            |
Does it live in the water?           Can it fly?
      ______|______                ______|_____
     |             |              |            |
  dolphin  Does it eat meat?     bird        turtle
             ______|______
            |             |
           dog          rabbit
```

As you can see in the example above, the leaves represent the solutions of the game, whilst the internal nodes represent the questions. For each question, the left subtree corresponds to the "yes" answer, whereas the right subtree corresponds to the "no" answer.

The mechanics of the game is as follows:

- Step 1: a list of elements (in the example, a set of animals) is shown to the user. She should choose mentally one. In the previous example, the list shown to the user would be: *dolphin*, *dog*, *rabbit*, *bird* and *turtle*.

- Step 2: next, the program will interactively ask a set of questions to the user about the chosen animal. The answers to these question can only be affirmative ("yes") or negative ("no"). The program decides, regarding the answers given by the user, and taking into account the decision tree, which is the next question to ask. In the example, if the user answers "yes" to the "Is it a mammal?" question, the program will ask next the "Does it live in the water?" question. If the user answers "no", the next question would be "Does it eat meat?". You do not have to implement this algorithm. You only need to know that, once the questioning process finishes, the system will store all the answers given by the user in an *array* of *String*s. In the previous example, assuming the user thought of a *dog*, the array would be `{"yes", "no", "yes"}`.

- Step 3: based on the decision tree and the answer list given by the user, the program will try to guess which animal the user thought of. For instance, the answer sequence `{"yes", "no", "yes"}` would correspond to *dog*, and the answer sequence `{"yes", "no", "no"}` would correspond to *rabbit*. The `{"yes", "yes", "yes"}` or `{ }` answer sequences are wrong, since they do not correspond to any animal in the decision tree.

The binary tree code representing the decision tree is shown next.

```
public class Tree {
    public static final int LEFT = 0;
    public static final int RIGHT = 1;

    private Node root;

    public Tree() {
        root = null;
    }

    public Tree(String info) {
        root = new Node(info,
                        new Tree(),
                        new Tree());
    }

    public boolean isEmpty() {
        return root == null;
    }

    public void insert(Tree tree, int side) {
        /* Assume this method is
         * already coded
         */
    }

    public void printOptions() {
        /* ... section 1 ... */
    }

    public String
    findSolution(String[] answers) {
        return findSolution(answers, 0);
    }

    private String
    findSolution(String[] answers, int pos) {
        /* ... section 2 ... */
    }
}
```

```
public class Node {
    private String info;
    private Tree left;
    private Tree right;

    public Node(String info,
                Tree left, Tree right) {
        setInfo(info);
        setLeft(left);
        setRight(right);
    }

    public String getInfo() { return info; }
    public Tree getLeft() { return left; }
    public Tree getRight() { return right; }

    public void setInfo(String info) {
        if (info == null) {
            throw new
                  IllegalArgumentException();
        }
        this.info = info;
    }

    public void setLeft(Tree tree) {
        if (tree == null) {
            throw new
                  IllegalArgumentException();
        }
        left = tree;
    }

    public void setRight(Tree tree) {
        if (tree == null) {
            throw new
                  IllegalArgumentException();
        }
        right = tree;
    }
}
```

Section 1 (1.25 points)

Write the code for the *public void printOptions()* recursive method in the *Tree* class. This method must print to the standard output just the information of the leaf nodes. For instance, in the previous example, the output might be:

```
dolphin
dog
rabbit
bird
turtle
```

Section 2 (1.25 points)

Write the code for the *private String findSolution(String[] answers, int pos)* recursive method in the *Tree* class. This method receives as arguments an *array* of *String*s that contain the answers given by the user with the format previously described, and an index that indicates the *array* position where you should start to process it.

The method must return the system answer that complies with the sequence of answers given by the user to the system questions, or *null* if the answers sequence does not match with any solution in the decision tree.

For instance, taking the tree in the initial example. If the input argument received by *public String findSolution(String[] answers)* is the *array* `{"yes", "no", "yes"}`, then the method must return `"dog"`.

Take into account the following assumptions:

- Assume that the tree will always be correctly built, i.e., every question will always have two children subtrees (two answers, or two questions, or a question and an answer), and that the answers will always have two empty children subtrees.
- Assume that the *array* containing the user answers will never be *null*, and it will never contain any other *String* different from "yes" or "no".
- Before processing the *array*, you must check the *pos* value. If it is not within the correct range to traverse the *array*, the method must return *null*.
- If a leaf is reached when there are still user answers to process, the situation will be understood as if the answers sequence is wrong, and therefore the method must return *null*.

Problem 1

```
public abstract class Color {
    private String name;

    public Color(String name) {
        this.name = name;
    }

    public String toString() {
        return "Color " + name;
    }

    public abstract GrayscaleColor toGrayscale();
}

public class GrayscaleColor extends Color {
    private double luminance;

    public GrayscaleColor(double luminance, String name) {
        super(name);
        this.luminance = luminance;
    }

    public GrayscaleColor(double luminance) {
        this(luminance, "");
    }

    public String toString() {
        return super.toString() + " <luminance " + luminance + ">";
    }

    public GrayscaleColor toGrayscale() {
        return new GrayscaleColor(luminance);
    }
}

public class RgbColor extends Color {
    private double red;
    private double green;
    private double blue;

    public RgbColor(double red, double green, double blue, String name) {
        super(name);
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public RgbColor(double red, double green, double blue) {
        this(red, green, blue, "");
    }

    public String toString() {
        return super.toString() + " <RGB " + red + ", " + green
            + ", " + blue + ">";
    }

    public GrayscaleColor toGrayscale() {
        return new GrayscaleColor(0.299 * red + 0.587 * green + 0.114 * blue);
    }
}
```

Problem 2

```
public Stack<E> split(Stack<E> stack, int n) {
    Stack<E> output = new LStack<E>();
    Stack<E> aux = new LStack<E>();
    for (int i = 0; i < n && !stack.isEmpty(); i++) {
        aux.push(stack.pop());
    }
    while (!aux.isEmpty()) {
        output.push(aux.pop());
    }
    return output;
}
```

Problem 3

Section 1

```
public void printOptions() {
    if (isEmpty()) {
        return;
    } else {
        if (root.getLeft().isEmpty()
            && root.getRight().isEmpty()) { // not really needed on well formed trees
            System.out.println(root.getInfo());
        } else {
            root.getLeft().printOptions();
            root.getRight().printOptions();
        }
    }
}
```

Section 2

```
private String findSolution(String[] answers, int pos) {
    // error checks
    if (isEmpty()
        || answers.length == 0
        || pos < 0
        || pos > answers.length) {
        return null;
    }

    // base case
    if (root.getLeft().isEmpty()
        && root.getRight().isEmpty()) { // not really needed on well formed trees
        if (pos == answers.length) {
            return root.getInfo();
        } else { // more answers than needed
            return null;
        }
    // recursive case
    } else {
        if (answers[pos].equals("yes")) {
            return root.getLeft().findSolution(answers, ++pos);
        } else { // answers[pos] equals "no"
            return root.getRight().findSolution(answers, ++pos);
        }
    }
}
```