Assignment 1: How to draw a tree?

summary

You have to develop Java code to work with formal grammar used in the modeling of biological structures (or self-similar structures in general). This TP aims firstly of skills of software engineering:

- manipulation of aggregated types: input / output, dictionaries, lists, geometric objects
- development of code according to a complex specification, use of opportunity libraries
- advanced programming in Java: parametric types, iterators, recursion, encapsulation, internal classes.

Estimated work: 200-400 lines of Java code (spacious source, with comments).

modalities

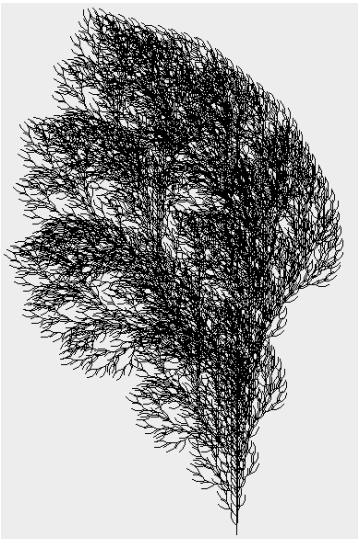
Work in teams of 2.

You need to submit only one JAR package, called `lindenmayer.jar` which contains all your compiled classes (`lindenmayer.*`) and source code (.java files).

If you have not taken the IFT1025 course (Programming II), you *may* have the right to use a language other than Java with my special permission: ask for an appointment to discuss the adjustment of the specification.

Deadline for submission: 1 October 2017, 20:15.

Lindenmayer System



We want to implement a program that draws random graphs using a **system of Lindenmayer** or **L-system** . In particular, one wants to produce drawings which resemble plants. The L-system was invented for this purpose: it allows to model the development of plant structures. This is a formal grammar that defines the generation of string symbols on an alphabet. The symbols correspond to structural units of the plant (branches), and the resulting string defines the structure to be drawn.

We specify the system for an alphabet fixed by the **starting string** s (an *axiom*), and a set **of rewrite rules** in the form $\langle \text{symbole} \rangle \rightarrow \langle \text{chaine de symboles} \rangle$, specifying that an application of this rule replaces the symbol to the left by the string to the right. Example on the alphabet { `F` , `-` } with only one rule:

```
s → F
F → FF-F
```

In such a system, one generates strings $S_0, S_1, S_2, \ldots$ by applying the replacement rules to all the characters of S_i in parallel to arrive at S_{i+1}. If there are no rules to apply, then we keep the symbol as it is (these are *terminal* symbols). We start with $S_0 = s$. With the system below,

$$\begin{aligned} s &\Rightarrow \texttt{F} && \{= S_0\} \\ &\Rightarrow \texttt{FF-F} && \{= S_1\} \\ &\Rightarrow \underbrace{\texttt{FF-F}}_{\text{1re F}}\underbrace{\texttt{FF-F}}_{\text{2me F}}\texttt{-}\underbrace{\texttt{FF-F}}_{\text{3me F}} && \{= S_2\} \end{aligned}.$$

If there are several rules with the same left side, one of the rules applicable to chance (to the uniform) is chosen. This allows to generate a multitude of different structures with a small set of rules.

```
s → F
F → FF
F → + F
```

$$\texttt{F} \underbrace{\Rightarrow}_{\text{avec proba } 1/2} \texttt{F-F} \underbrace{\Rightarrow}_{\text{avec proba } 1/4=1/2\times 1/2} \texttt{+F-F-F} \Rightarrow \cdots .$$

Turtle Graphics

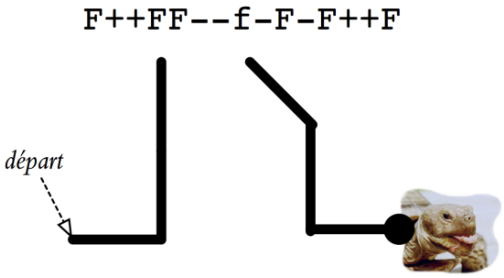
The derived string is interpreted by applying rules as an instruction sequence for turtle graphics. The turtle, working in a Cartesian coordinate system (with axis Y pointing upwards and X to the right), has a pencil to draw following a series of simple instructions. The **state of the turtle** is triple (x, y, θ) with its position (x, y) and the angle θ of its nose with respect to the horizontal line ($0°$ to the right, $90°$ to the top). The turtle also has a memory organized in pile which allows to return to a visited state. The instruction set includes the following actions:

- **draw** : move forward by a unit distance (d) by drawing a line. The state of the turtle changes (x, y, θ) to $(x + d\cos\theta, y + d\sin\theta, \theta)$.
- **move** : move forward by a unit distance (d) without tracing. The state of the turtle changes (x, y, θ) to $(x + d\cos\theta, y + d\sin\theta, \theta)$.
- **turnL** : turn against the needle direction by a single angle (δ). The state of the turtle changes (x, y, θ) to $(x, y, \theta + \delta)$.
- **turnR** : turn to the needle direction by a unit angle (δ). The state of the turtle changes (x, y, θ) to $(x, y, \theta - \delta)$
- **push** : saves (stacks) the current state (position + angle) of the turtle. The state does not change.
- **pop** : unpacks the most recently saved state. The state changes to (x', y', θ') what is the last **save state** .
- **stay** : the state does not change.

A drawing is specified by a chain of symbols of the L-system, according to an association of the alphabet with actions of the turtle. Standard interpretation of symbols:

- `[`= **push**
- `]`= **pop**
- `+`= **turnL**
- `-`= **turnR**

- `F` and any other uppercase letter = **draw**
- `f` and any other lower case = **move**



The current drawing is realized by specifying the state at the beginning (by default, the turtle starts at $(0, 0, 90°)$ with the nose up), and the units (distance d and angle δ).

Implementation

We want a Java implementation with the following features:

- representation of an L-system as an abstract type with operations to manipulate string symbols, apply rules, and follow the turtle.
- reading a file specifying an L-system and settings for the turtle.

Specifications

Tortoise

The abstract type of the turtle comprises drawing operations (`draw`, `move`, `turnL`, `turnR`, `push`, `pop`, `stay`), the initialization and status quests (`init`, `getPosition`, `getAngle`), and the allocation of the scaling parameters (`setUnits`). We can imagine different implementations (eg, PostScript drawing, or on the screen), so we declare the turtle through an interface.

```
1  package lindenmayer;
2  import java.awt.geom.Point2D;
3  public interface Turtle
4  {
5      public void draw();
6      public void move();
7      public void turnR();
8      public void turnL();
9      public void push();
10     public void pop();
11     public void stay();
12     /**
13      * initializes the turtle state
14      * @param pos turtle position
15      * @param angle_deg angle in de
16      */
17     public void init(Point2D posit:
18     /**
19      * position of the turtle
20      * @return position as a 2D po:
21      */
22     public Point2D getPosition();
23     /**
24      * angle of the turtle's nose
25      * @return angle in degrees
26      */
27     public double getAngle();
28     /**
29      * sets the unit step and turn
30      * @param step length of an adv
```

```
        * @param delta unit angle char
        */
       public void setUnits(double ste
}
```

Lindenmayer System

The TA of the L-system includes the representation of symbols, strings, and rules, as well as the operations to derive strings, and to interpret them as chelonian instructions:

- representation of alphabet symbols (a simple class `Symbol`encapsulating a character)
- representation of symbol strings: character iterator `Iterator<Symbol>`
- representation of the system:
  - initialization: `Symbol addSymbol(char sym)`adds a symbol to the alphabet and returns the associated symbol, `addRule(Symbol sym, String expansion)`adds the rule $\texttt{sym} \rightarrow \texttt{expansion}$, `setAction(Symbol sym, String action)`defines the action of the turtle for the given symbol, `setAxiom(String str)`and `getAxiom` stores or retrieves the axiom.
  - access rules and actions: `Iterator<Symbol> rewrite(Symbol sym)`returns the substitution according to a randomly chosen rule among those with `sym` to the left or `null`if no rule applies, `tell(Turtle turtle, Symbol sym)`requests the turtle to execute the associated statement with `sym` (the one specified by `setAction`).
  - advanced operations (description below): `readJSONFile`to read a file, `tell`and `applyRules`to perform several rewrite cycles, and `getBoundingBox`to determine the size of the drawing.

Signing methods to implement:

```
package lindenmayer;
import java.util.Iterator;
import java.awt.geom.Rectangle2D;
public class LSystem {
    /**
     * constructeur vide monte un s
     */
    public LSystem(){ ... }
    /* méthodes d'initialisation de
    public Symbol addSymbol(char sy
    public void addRule(Symbol sym,
    public void setAction(Symbol sy
    public void setAxiom(String sti

    /* initialisation par fichier *
    public static void readJSONFile

    /* accès aux règles et exécutio
    public Iterator<Symbol> getAxio
    public Iterator<Symbol> rewrite
    public void tell(Turtle turtle,

    /* opérations avancées */
    public Iterator<Symbol> applyRu
    public void tell(Turtle turtle,
    public Rectangle2D getBoundingB
    ...
}
```

Work to do

1. **Imaginary Pencil Turtle**
   Implement a fake class with the turtle interface that follows states correctly but does not draw anything in truth.
   Hint : I recommend an internal class `State`to encapsulate the state, and a stack`java.util.Stack<State>`to stack / unstack when push / pop.
2. **L-system**
   Implement the main operations of the TA (addSymbol, addRule, setAction, rewrite, tell, setAxiom, getAxiom) with a class `Symbol`and a class `LSystem`.
   Hint : Always use the same instance of `Symbol` - a table `Map<Character,Symbol>`is useful for associations `char` → `Symbol`. One can store the rules in one`Map<Symbol,List<Iterator<Symbol>>>` - this allows to retrieve all the rules of a symbol in a list and to choose one randomly. It is possible to implement `tell`by accessing turtle methods directly (via `getClass().getDeclaredMethod` + `Method.invoke` if such a solution pleases you) but a series of if-else (`if ("draw".equals(str) ...)` is easier to code and is perfectly enough for us.
3. **Initialization by file**
   The system and the scale of the graph are initialized into a file of JSON format which follows the syntax of Javascript with the keys:

   - `alphabet`: table of symbols (strings)
   - `rules`: rules as an object of associations symbol → array of strings
   - `axiom`: start string (string)
   - `actions`: associations symbol → turtle instruction
   - `parameters`: object with `step`(d), `angle`(δ), `start`(array with 3 numeric elements x, y, θ)

```
{
    "alphabet": ["F", "[", "]", '
    "rules": {"F" : ["F[+F]F[-F]]
    "axiom": "F",
    "actions": {"F":"draw", "[":'
    "parameters" : {"step": 2, "a
}
```

   Implement a method `readJSONFile`in the class `LSystem`that boots the system from the given JSON file. Note that it is extremely simple to read such a file: install the JSON-java package , and consult its documentation .

4.

```
package lindenmayer;
import org.json.*;
public class LSystem {
...
    public static void readJSON
        JSONObject input = new
        JSONArray alphabet = inp
        String axiom = input.get
        system.setAxiom(axiom);
        for (int i=0; i&amp;lt;a
            String letter = alph
            Symbol sym = system
            ...
```

5. **Expanding strings**
   Implement a method `Iterator<Symbol> applyRules(Iterator<Symbol> seq, int n)`that

calculates $S_n$from the string $S_0 = \text{seq}$.

Hint : Use a list (`java.util.List<Character>`with any class to initiate) into a `1 .. n : List`.iterator () loophere.

6. **Shuffle the Turtle**
   Implement a recursive method `tell(Turtle turtle, Symbol sym, int rounds)`that performs `rounds` rewriting iterations on s `ym` , and executes the resulting string with the specified turtle.

   Hint : We want to avoid the explicit calculation of the complete string (so do not do just `applyRules(..,n)`), exploiting the recursion. If `rounds = 0`, the string (with`tell(turtle, sym)`) is executed directly , otherwise recursive calls are made. Develop the algorithm on paper before coding.
7. **Vital Domain of Testudines**
   Implement a method `getBoundingBox(Turtle turtle, Iterator<Symbol> seq, int n)`that calculates the extreme coordinates of the turtle by running the string S_n(one begins with $S_0 = \text{seq}$). The return value is of type `java.awt.geom.Rectangle2D`.

   Hint : If the turtle goes through states $(x_1, y_1\theta_1), (x_2, y_2, \theta_2), \ldots, (x_m, y_m, \theta_m)$, then we need $x_{\min} = \min\{x_1, \ldots, x_n\}, x_{\max} = \max\{x_1, \ldots, x_n\}$and $y_{\min} = \min\{y_1, \ldots, y_n\}, y_{\max} = \max\{y_1, \ldots, y_m\}$ (from where the width and height are calculated by $w = x_{\max} - x_{\min}$and $h = y_{\max} - y_{\min}$). Follow the recursive logic of `tell(.., n)` to calculate the rectangle that covers all the points visited by the turtle. The bogus implementation helps to test the code. It is advisable to work with rectangles (to pass and return to recursive calls) by calculating their union as dictated by the geometry (study the documentation of`Rectangle2D`).

References

- L-system: Wikipedia article ; the understanding drunk on algorithmic botany P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants* . Springer-Verlag 2004.
- JSON: json.org , Wikipedia article , JSON-java on github
- Java API
  documentation: List , Map , Stack , Rectangle2D , Point2D , Iterator ; Tutorials: collections , parametrized types (generics) .