Series H: In preparation for the exam

Table of contents

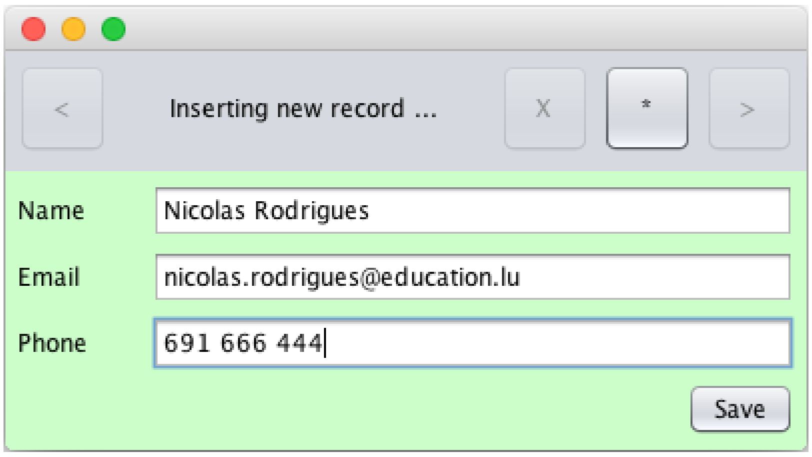
Exercise H.1: Telephone book

In this exercise, you are going to develop an application for managing a telephone directory.

It must be possible to store the following data for each person:

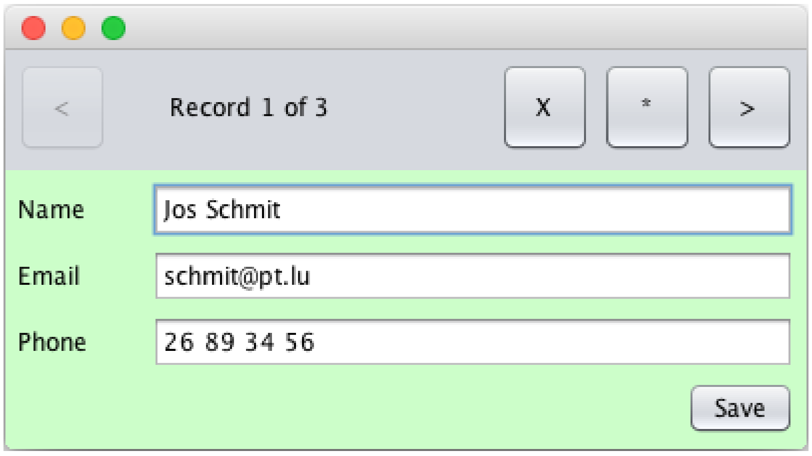
- full name
- telephone number (spaces allowed!)
- email address

All entries must be sorted alphabetically by the person's name. The user must be able to add an entry:



Name	Nicolas Rodrigues
Email	nicolas.rodrigues@education.lu
Phone	691 666 444

It also must be possible to navigate through the different entries.



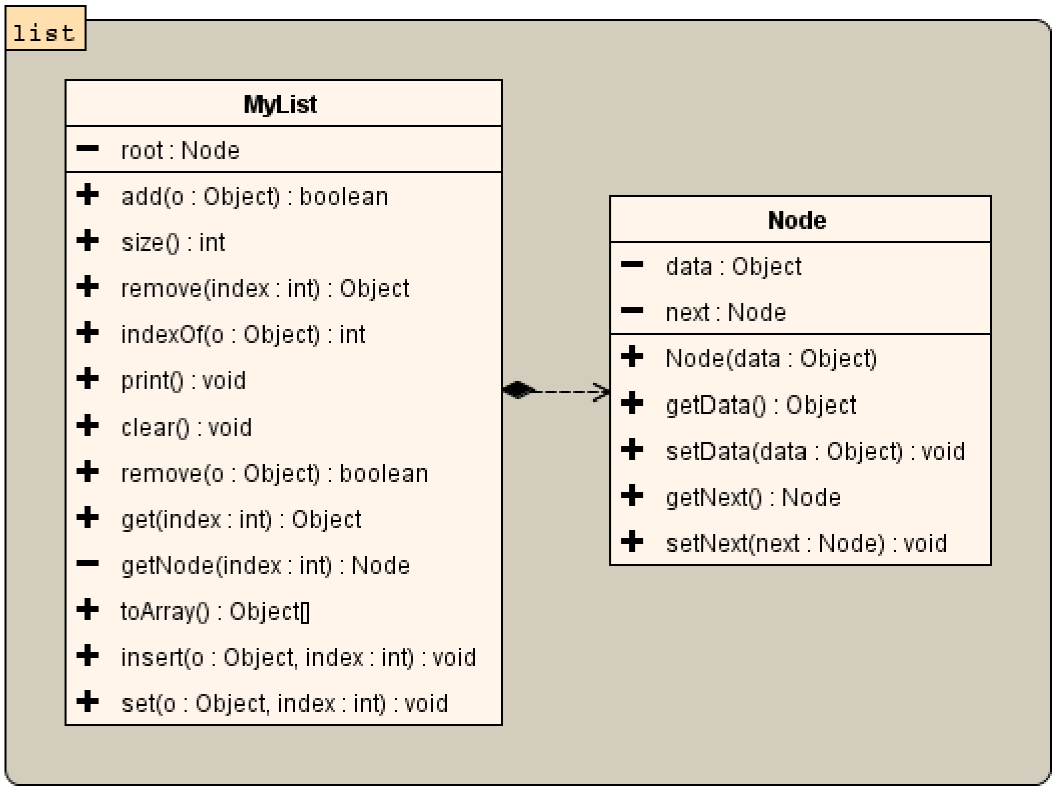
Name	Jos Schmit
Email	schmit@pt.lu
Phone	26 89 34 56

As deletion is a delicate operation, don't forget to display a message such as "Are you sure you want to delete this entry permanently? before deleting it.

When it is opened, the program automatically loads the directory from the "phonebook.json" file and saves the entries there when it is closed. The data must be saved in JSON format (a list of objects of the **Person** type).

Exercise H.2: Linked list

Develop the **`MyListe`** class using the following UML diagram:


list
MyList
− root : Node
+ add(o : Object) : boolean
+ size() : int
+ remove(index : int) : Object
+ indexOf(o : Object) : int
+ print() : void
+ clear() : void
+ remove(o : Object) : boolean
+ get(index : int) : Object
− getNode(index : int) : Node
+ toArray() : Object[]
+ insert(o : Object, index : int) : void
+ set(o : Object, index : int) : void
Node
− data : Object
− next : Node
+ Node(data : Object)
+ getData() : Object
+ setData(data : Object) : void
+ getNext() : Node
+ setNext(next : Node) : void

Explanation of methods:

- **`add(Object)`** adds an element to the end of the list.
- **`size()`** determines and returns the number of elements in the list.
- **`remove(int)`** removes the element from the list at the specified position.
- **`remove(Object)`** removes the specified element from the list.
- **`print()`** displays the contents of the list in the console.
- **`clear()`** clears the list.
- **`indexOf(Object)`** determines and returns the position of an object in the list.
- **`get(int)`** returns the element at a given position.
- **`getNode(int)`** returns the node at a given position.
- **`toArray()`** creates and returns a copy of the linked list as a static list.
- **`insert(Object,int)`** inserts an element at the specified position.
- **`set(Object,int)`** replaces the element at the indicated position with a new one.

Exercise H.3: Linked list (continued)

Complete the class in the following exercise by adding the following methods:

- **`count(Object)`** counts the number of occurrences of an element in the list.
- **`sort()`** sorts the list by ascending ochre.
- **`verifySort()`** tests whether the list is sorted in ascending order.
- **`reverse()`** reverses the order of the elements in the list.
- **`isEmpty()`** determines and returns whether the list is empty.
- **`toJson()`** creates and returns a JSON text representing the list and its elements.
- **`toXML()`** creates and returns an XML text representing the list and its elements.

Exercise H.4: Linked list (improvements)

To reduce the complexity of certain methods in the **`MyList`** class, make the following changes:

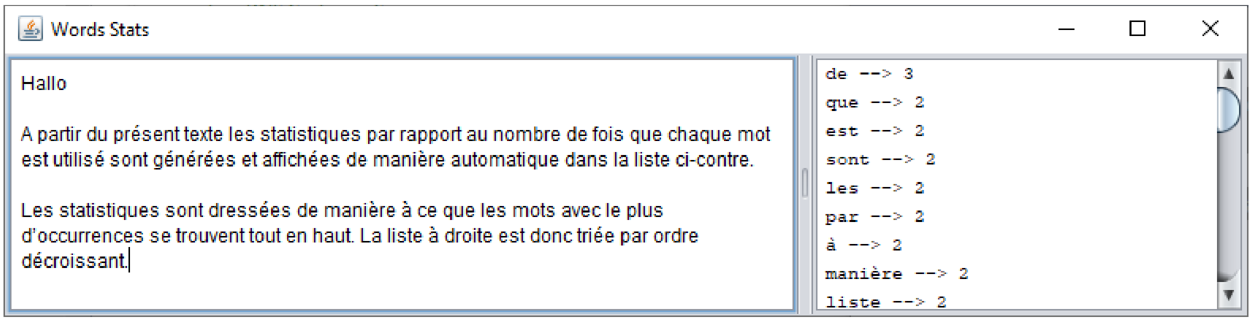
- Adds the **`last`** attribute, which points to the last element in the list at any time.
- Adds the **`size`** attribute, which always contains the size of the list.

What other improvements could be made?

Exercise H.5: Words Stats

Write a program to automatically create statistics such as those shown in the screenshot below.

The statistics are updated automatically, even while the text is being entered. So, it's sort of "live".



Hallo

A partir du présent texte les statistiques par rapport au nombre de fois que chaque mot est utilisé sont générées et affichées de manière automatique dans la liste ci-contre.

Les statistiques sont dressées de manière à ce que les mots avec le plus d'occurrences se trouvent tout en haut. La liste à droite est donc triée par ordre décroissant.

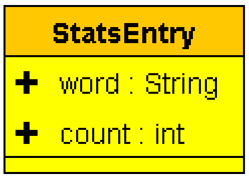
```
de --> 3
que --> 2
est --> 2
sont --> 2
les --> 2
par --> 2
à --> 2
manière --> 2
liste --> 2
```

Version 1

Use an associative table (**`HashMap`**) to create the statistics.

Version 2

Try to find an alternative implementation without an associative table that uses a list made up of instances of the following class:



Question

Analyse the complexity and performance of the two versions!

Exercise H.6: Vector drawing application

Create an application for creating vector drawings, like Illustrator, but in a 'light' version.

- Start by thinking about the structure needed to represent a design and the different elements that can be added.
- In the first phase, only the "rectangle", "ellipse" and "line" elements need to be implemented, but the structure of the program should be such that it remains very simple to extend this palette.
- All elements must be drawn with the mouse in real time on the canvas and it must be possible to select them to move them.
- It must also be possible to change the colour of the elements, both the outline and the internal surface.
- Don't forget that you also need to be able to delete an element from the design.
- **Challenge:** Implement something to allow the user to change the size of elements using the mouse!
- All drawings must be saved in JSON format. Use the JSON-P API to implement this.
- Add a function that allows you to modify the position of an item in the list and thus change the display. In effect, an item at the front of the list is drawn before an item further back in the list, so that the second item is superimposed on the first.

<u>Exercise H.7:</u> <u>Performance tests</u>

1. Create a static list of 1,000,000 integers and initialise it with random values in the range [0,999];
2. Create an **`ArrayList`** and copy all the values in your list into the **`ArrayList`**. See how long it takes!
3. Create a **`HashSet`** and copy all the values in your `list` into the **`HashSet`**. Measure the time needed to complete the operation!
4. Create a **`HashMap`** and copy all the values in your `list` into the **`HashMap`** using the index as a key (type = `Interger`). Just think how long it will take!
5. Now compare the three beats! What can you see?
6. Perform a linear search in the three structures and measure the time it takes the programme to determine whether a value is contained therein. Use both the predefined methods and your own paths! Compare the following cases:
   - Search for a value at the beginning.
   - Search for a value in the middle.
   - Search for a value that is not in the list (e.g., 1000). This is, of course, the "worst case".

Exercise H.8: Spell-Checker

Create a spellchecker for German texts using the file "*DE_Wortliste_UTF8.txt*". Enter the texts to be checked in a text field (TextArea). The words are checked one by one, and the programme displays any incorrect words (which are not in the file) in a text field so that they can be corrected.

Add an **`analyseText`** method which provides a list of all the words not found in each text. The list also contains the positions at which these words are in the original text.

Use different data structures: **`ArrayList`** (predefined) and **`TreeSet`** (predefined). Test and compare the performance of the different structures, e.g., by determining how long it takes your algorithm to analyse a long German text.

First, develop an appropriate class structure to avoid duplication of code and to make it easier to test the different data structures.

Exercise H.9: Dictionary

Develop a German French dictionary (uni-directional) using.

(a) the predefined **`HashMap`** structure,

(b) the predefined **`TreeMap`** structure,

(c) the predefined **`TreeSet`** structure,

The file "*DE-FR_dictionary_UTF8.txt*" is provided.

What difficulties do you encounter (for each of the structures) if you want to add several translations for the same word?

For the fastest:

Copy the project and modify it so that it can be used in both directions. How do you do it?

1. if research must be highly effective in both directions?
2. if research must be very effective in one direction but can be less effective in the other?

Exercise H.10: Play Grid

Create a `PlayGrid` class that can represent a square playing surface with $n \, x \, n$ squares. Each square can contain a piece (DE: Spielstein). There are different kinds of pieces, each with a different representation, e.g., different colours or shapes (blue crosses, red squares, green discs, etc.).

Integrate the model into a programme in which you can place counters on free squares and move the counters with the mouse.

The playing surface must be resizable, but it must always be square.

Examples of applications: Draughts, TicTacToe, ...

Exercise H.11: Power

Create the recursive method **power** which calculates x^n for a real x and a positive or zero integer n. There is no need to define special processing for cases where x^n is not defined.

Complete the power recursive method to calculate x^n for a real x and a positive, negative or zero integer n.

Exercise H.12: GCD

Create the recursive method **gcd** which calculates the greatest common divisor of two integers $\boldsymbol{x}$ and $\boldsymbol{y}$ $(\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{N})$ according to the following definition:

$$\operatorname{gcd}(x,y) = \begin{cases} \operatorname{gcd}(x,y) = x & y = 0 \\ \operatorname{gcd}(x,y) = \operatorname{gcd}(\,y, x \,\%\, y) & y > 0 \end{cases}$$

Exercise H.13: Machin & Truc

Do this exercise on paper, without programming it!

Or the following method:

```
public boolean machin(int x)
{
   if (x==0) return true;
   else return !machin(x-1);
}
```

a) Indicate the sequence of recursive calls and the result of the **machin(3)** call.

b) What's the point of the **machin** method?

c) For what values of x does the **machin** method give a result?

Or the following method:

```
public double truc(double x, int y)
{
    if (y==0) return 1;
    else if (machin(y)) return truc(x*x,y/2);
    else return truc(x,y-1) * x;
}
```

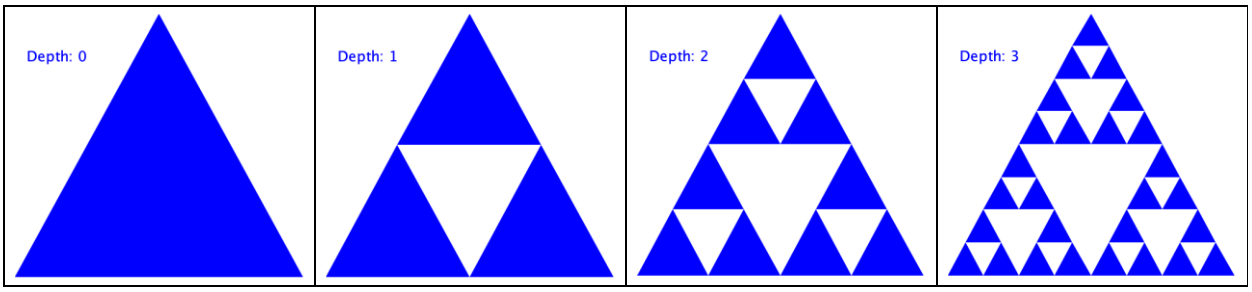
d) Give the sequence of recursive calls to **truc** and the result of the call **truc (2,5)**?

e) What is the purpose of the **truc** method?

"This sentence is meaningless because it refers to itself".

<u>Exercise H.14: Sierpinski triangle</u>

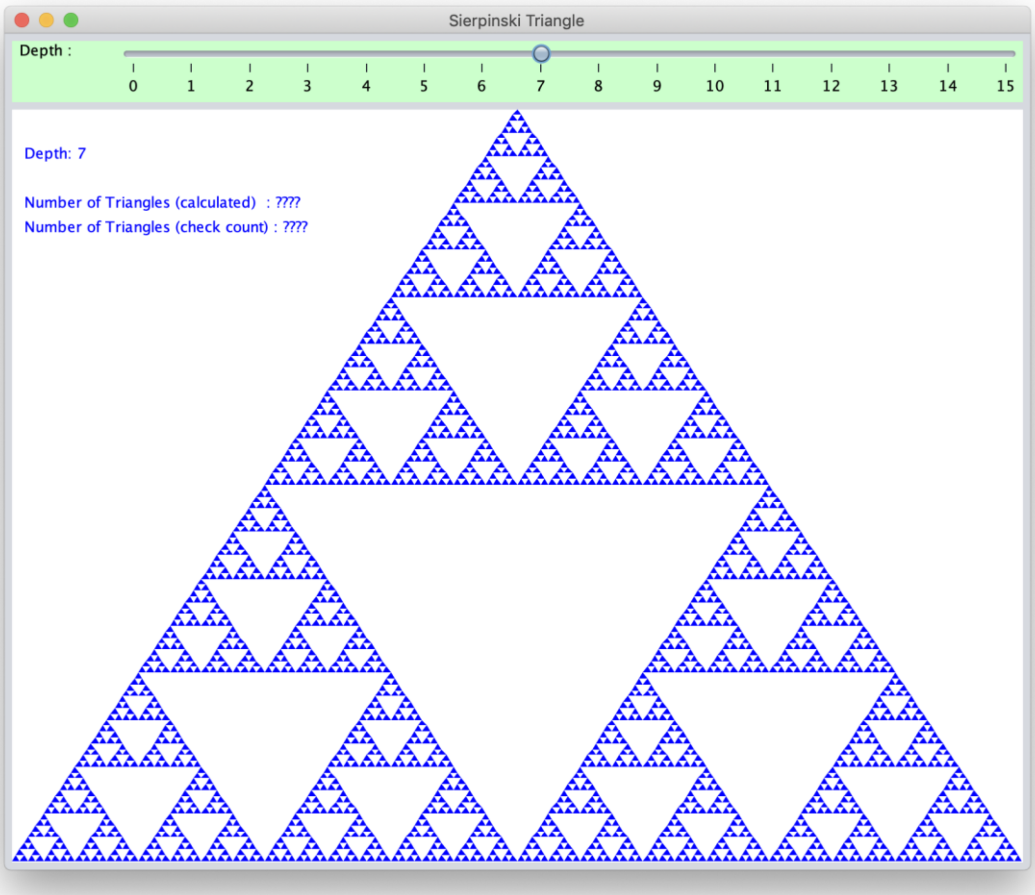
In a ***SierpinskiTriangle*** class, develop a ***draw*** method that draws the fractal known as the "Sierpinski Triangle". For recursion depths of zero to 3, the Sierpinski triangle looks like this:



Use the **fillPolygon** method to draw the triangles.

Deduce, calculate, and display the number of triangles drawn for the different recursion depths. Check by counting the triangles drawn.

What modification needs to be made so that the program draws the triangles up to the degree of recursion where the triangles are still just visible?

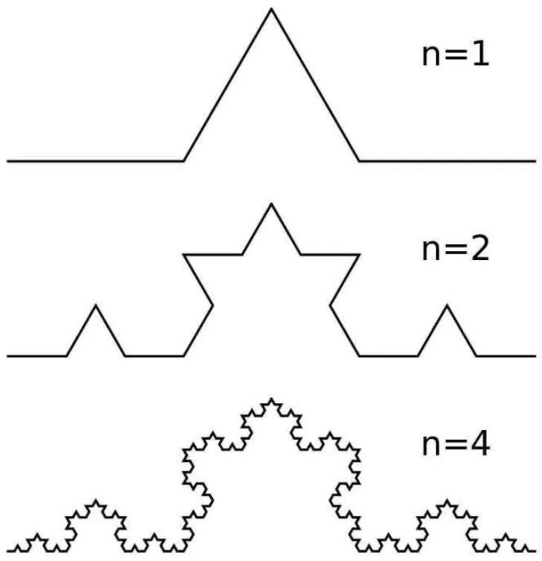


Depth: 7

Number of Triangles (calculated) : ????
Number of Triangles (check count) : ????

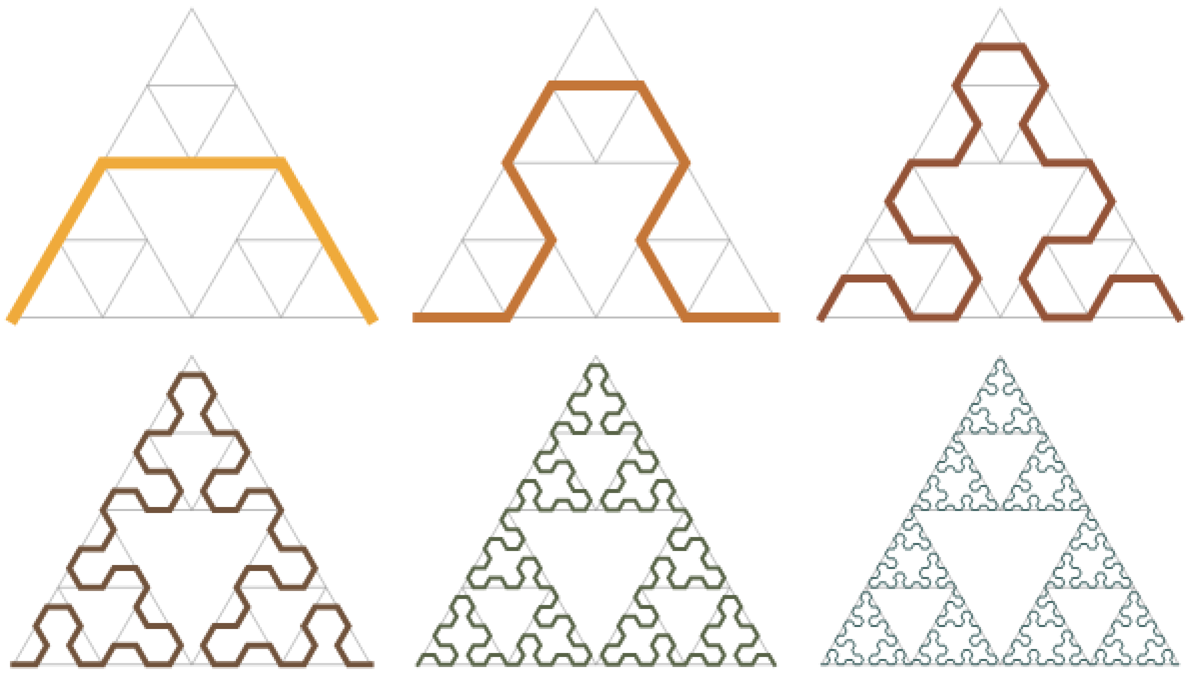
Exercise H.15: Koch curve

Like the previous exercise, draw the Koch curve.



Exercise H.16: Sierpinski curve

There is also the Sierpinski curve, whose high iteration version begins to resemble its triangle.



Exercise H.17: GameZone

The GameZone.lu website has asked you to create the template for their new application for recording their video game news articles. To do this, they have given you the following information:

An article is published on a specific date and consists of a title, a subtitle, and a body. Some articles may also have a summary. In the case of GameZone.lu, all articles are always written by a single author, who in turn may write several articles. If the article is a review of a new game, it also includes a score (decimal number from 0.00 to 10.00) which reflects the author's appreciation. An article can be linked to other articles (related articles).

Draw up the UML diagram for the model described above!

Exercise H.18: SQL Telephone Directory

Take exercise H.2 "Exercise H.18: Telephone directory" and modify it so that the data is no longer stored in memory but comes directly from a database.

Exercise H.19: File Search

Write an application that allows the user to search for all the files in each directory containing a certain character string.

- The application allows the user to specify the name of a directory.
- The user can choose whether this directory is searched recursively or not.
- The user can specify a search string.

Hint: https://javaconceptoftheday.com/list-all-files-in-directory-in-java/

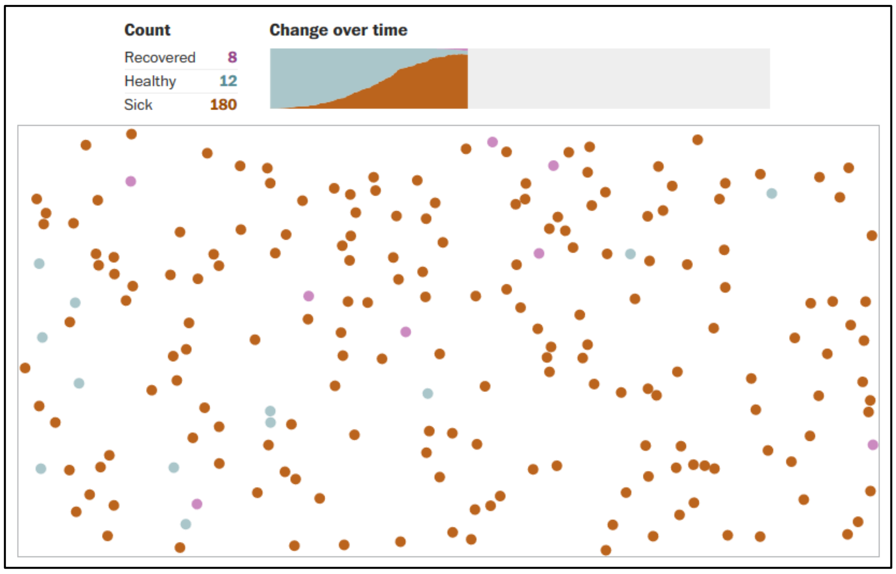
Example: The user searches for all files containing the word "clean" in their name.

Exercise H.20: Simulitis

The aim is to create a small simulation of a virus called "Simulitis", which simulates the spread of the Corona virus as a function of the population immobilisation rate. The simulation follows the following rules:

- Each person is represented by a disc travelling at constant speed through the simulation zone.
  - During an iteration, each person makes a single movement (at their constant speed).
  - By touching the limit, any disc is pushed back in accordance with the laws of physics!
  - If one person touches another, the two will also repel each other in accordance with the laws of physics.
  - If two people are touching:
    - if one of them is ill, the other also becomes ill,
    - otherwise, their status does not change.
  - After X iterations (X has yet to be determined), a sick person recovers with a 98% probability. In the remaining 2% of cases, the person dies and disappears.
  - Colours:
    - blue = person is not ill
    - brown = sick person
    - pink = person healed

Start small!



<u>**Exercise H.21:**</u> <u>**Commission sales**</u>

You have been asked to create an application that will make it easier to manage a commission sale (DE: Kommissionsverkauf). The principle is as follows:

- Each vendor places a certain number of items they wish to sell on commission. To do this, they must stick a label with the price and the seller code they have received from the event organiser on their items. They must also submit a signed form containing the following information:
    - Full name,
    - address,
    - telephone number,
    - email address,
    - IBAN account number with corresponding BIC number.

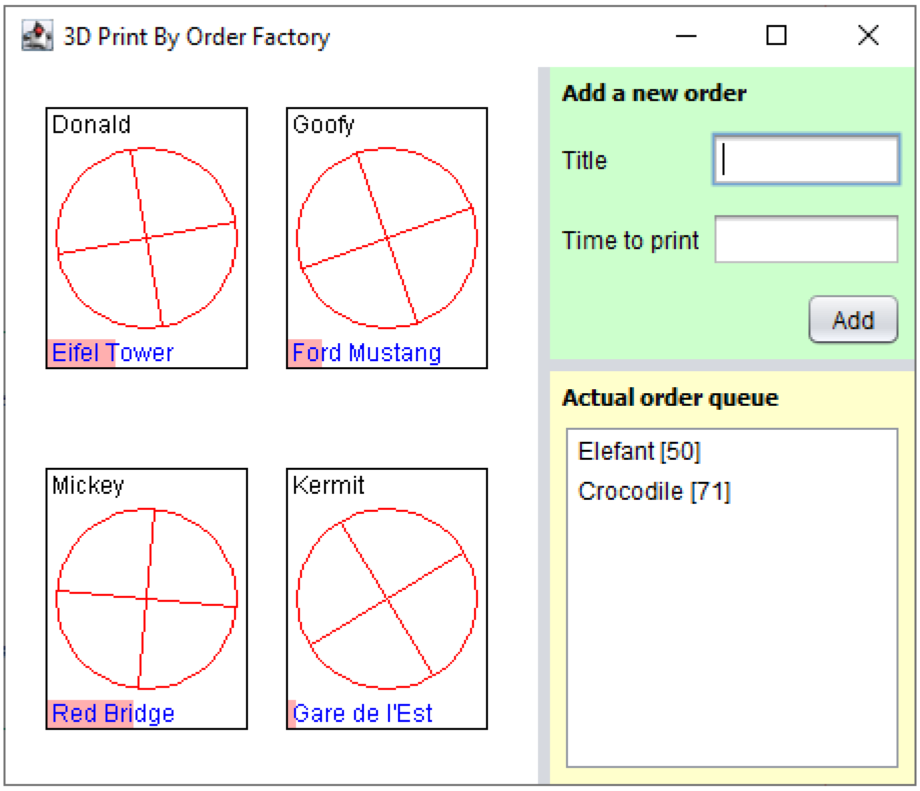
  Each seller receives a certain percentage of the items they sell. This percentage is not necessarily the same for each seller.
- During the sale, cashiers enter the seller's code and the price of each item sold. It should therefore be possible to determine:
    - The total to be paid for each customer,
    - the total transferred to each seller's bank account,
    - the organiser's total winnings.
- The system must also be such that it is possible to print a receipt for each customer served.
- To be able to trace and correct any errors at a later date, all sales actions must be logged. This information includes:
    - Number of the till that entered an item,
    - date and time an item were entered,
    - date and time of a customer's payment,
    - type of customer payment (cash, VPay or PayConiq).

Tasks to be carried out:

- Develop the UML model of the application for a system operating with a single till.
- The system needs to work with several checkouts. How should you adapt your software?
- Don't forget to think about data protection too!

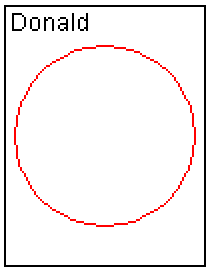
Exercise H.22: 3D Print by order factory

You need to implement a small 3D printing factory like this one:



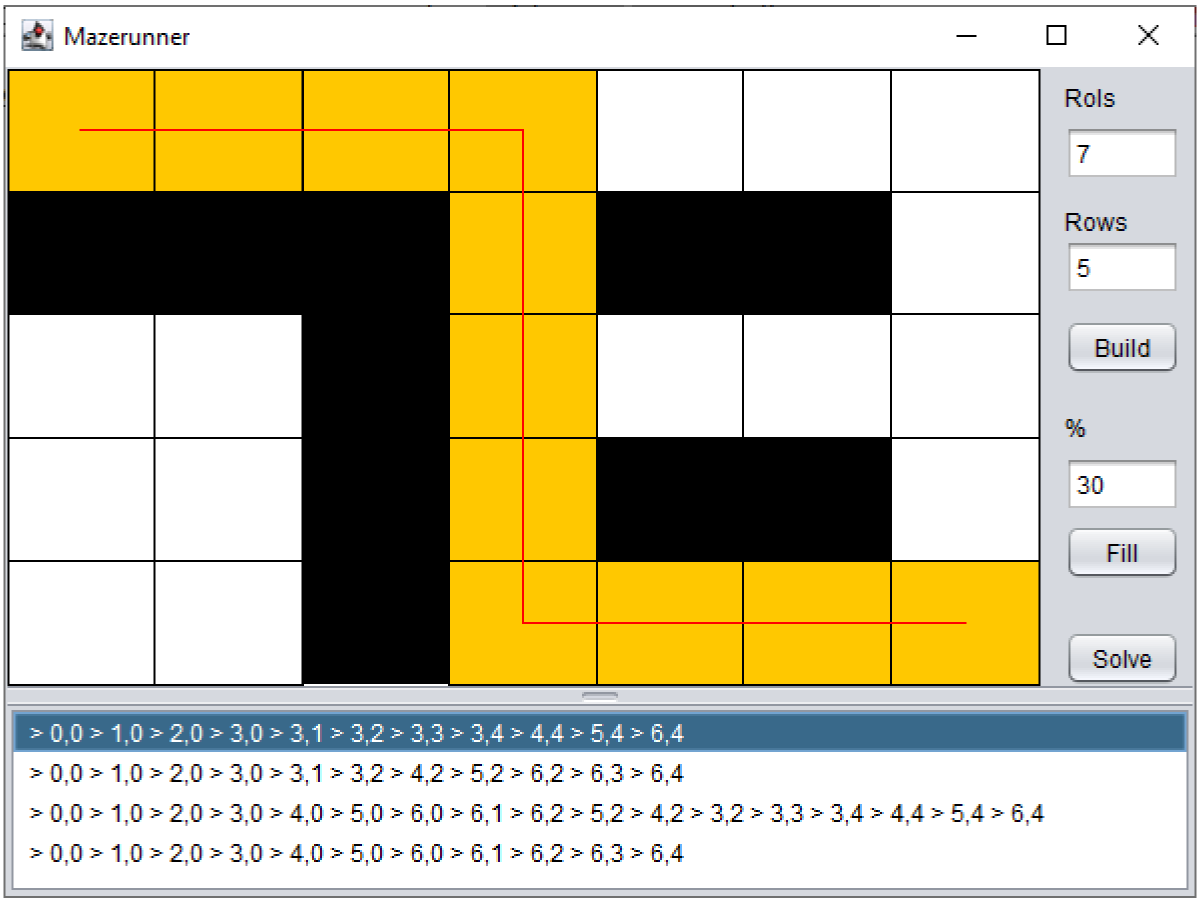
Elefant [50]
Crocodile [71]

- In the green panel, the user can add an order, indicating the name and the printing time required.
- The order is then added to the order queue. The implementation of the order queue must be carried out using a linked list.
- When a printer is free, the next command in the order queue is assigned to it and disappears from the list.
- In the picture on the right, you can see a printer that does not print.
- When a command is being printed, a cross is drawn inside the disc, which rotates at a speed of one degree per second. The name of the job is displayed at the bottom, along with a progress bar.



Exercise H.23: Mazerunner

The aim of this program is to create a program that searches for and displays all the possible paths between two cells in each field. The starting cell is always at the top left, while the destination cell is always at the bottom right.



```
> 0,0 > 1,0 > 2,0 > 3,0 > 3,1 > 3,2 > 3,3 > 3,4 > 4,4 > 5,4 > 6,4
> 0,0 > 1,0 > 2,0 > 3,0 > 3,1 > 3,2 > 4,2 > 5,2 > 6,2 > 6,3 > 6,4
> 0,0 > 1,0 > 2,0 > 3,0 > 4,0 > 5,0 > 6,0 > 6,1 > 6,2 > 5,2 > 4,2 > 3,2 > 3,3 > 3,4 > 4,4 > 5,4 > 6,4
> 0,0 > 1,0 > 2,0 > 3,0 > 4,0 > 5,0 > 6,0 > 6,1 > 6,2 > 6,3 > 6,4
```

Develop a program to do this:

1. Use the "Build" button to generate a field with $m \; x \; n$ cells. Each cell is either "free" or "occupied". The left mouse button can be used to mark a cell as occupied, while the right mouse button can be used to mark it as free.
2. The "Fill" button automatically and randomly fills certain cells as occupied, according to the percentage indicated.
3. The 'Solve' button searches for all possible paths between the start and finish points and displays them in the list at the bottom. By clicking on one of the paths, the relevant cells are coloured orange, and their centres are connected by a red line.

Exercise H.24: Code interpreter

You are going to develop an application for reading, interpreting, and executing a very simplistic programming language. You will be provided with the graphical interface and the UML diagram (see last page of the brief).

The programming language consists of 5 instructions:

`move` x	moves the pencil x pixels, where x is a natural number.
`rotate` x	rotates the pencil by x degrees, where x is a natural number.
`up`	lifts the tip of the pencil from the drawing surface.
`down`	place the tip of the pencil on the drawing surface.
`loop X {...}`	repeats the instructions contained in the { ... } block x times, where x is a natural number

Internally, the various instructions are stored in a sort of *simple linked list*.

The "Line" class

This class represents a line between the start point (fx, fy) and the finish point (tx, ty).

- The `draw(...)` method draws the line in the current canvas colour.

The "Lines" class

This class manages a list of lines.

- The delegated `add(...)` method adds a line.
- The `clear()` method deletes all rows.
- The `draw(...)` method draws all the lines in the class on the canvas passed as a parameter.

The 'Pen' class

This class represents the virtual pencil that draws on the canvas. It has several attributes:

`linesis`	a list of lines
`colorrindicates`	the colour of lines, the default value being yellow.
`x,yindicate`	the current position of the pencil
`angleis`	the direction - in degrees - in which the pencil is turned.
`downindicates`	whether the pencil is placed on the canvas or not.

The following methods must be implemented:

- The `draw(...)` method draws all the lines on the canvas.
- The `execute(Instruction)` method executes the instruction passed as a parameter **and** starts execution of the next instruction in the current instruction chain.

o `down`	place the pencil on the canvas.
o `Up`	lift the pencil from the canvas.
o `move`	moves the pencil by the distance of the parameter in the in the current direction of the pen. A line is drawn if the pen is placed on the canvas.
o `rotate`	changes the direction of the pencil by as many degrees as indicated by the instruction parameter.
o `loop`	repeats the sub-instructions of the instruction as many times as indicated by the instruction parameter.

- The `execute(Program, int, int, int)` method executes, if possible, the instruction passed as a parameter while defining the *initial* position and direction of the pencil.

The "Instruction" class

This class represents a single instruction. As the instructions in a program are stored internally in a sort of *simple linked list*, an instruction points to another instruction if necessary. A `loop` instruction also points to the first instruction in the block of instructions to be repeated.

The class has the following attributes:

• `cmd`	represents the name of the instruction (`move`, `rotate`, `up`, `down` or `loop`)
• `param`	represents the instruction parameter with a default value of 0.
• `next`	the attribute which points to the next instruction.
• `subp`	points, in the case of a loop instruction, to the first instruction in the instruction block to be repeated (default value null)

To be able to present an instruction graphically, it also has the following attributes:

• `x,y`	represents the coordinates of the top left point of the
• `width`	the width.
• `height`	the height

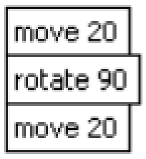
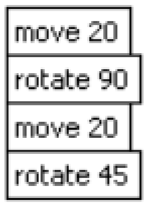
Some attributes have modifiers and/or accessors. Please refer to the UML diagram.

The class has a single constructor which, if possible, converts the second parameter.

The following methods are available:

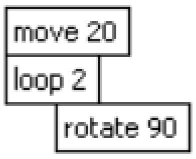
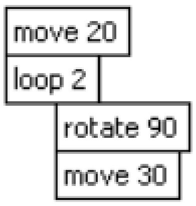
- The `append(...)` method adds the instruction passed as a parameter to the very end of the instruction chain to which the current instruction belongs.

Example:

Initial state	Modification	Final state
	Adding the instruction "rotate 45" to one of the three instructions (it doesn't matter which one) adds this instruction to the end of the chain.	

- The `addSub(...)` method adds the instruction passed as a parameter to the very end of the instruction chain of sub-instructions of the current instruction. This method will only be used for `loop` instructions.

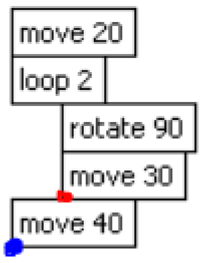
Example:

Initial state	Modification	Final state
	Adding the "move 30" instruction to the "loop 2" instruction adds this instruction to the end of the chain of "loop 2" sub-instructions.	

- The `toString()` method returns a textual representation of the current instruction, which is the name of the current instruction followed by a space and the value of its parameter. If the parameter value is zero, it is omitted completely.
- The `draw(...)` method draws the current instruction and any sub-instructions, on the canvas and at the position passed as parameters according to the constraints below. The position is saved in the class attributes.
  - An instruction is represented by a rectangle with a white background and a black border in which its textual representation is written in black.
  - The internal margin (distance from the border to the text) must be +/- 4 pixels.
  - Its dimensions are determined and saved in the corresponding attributes of the class.
  - The method returns the abscissa (position along the Y axis) of the bottom edge of its lowest sub-instruction, or of the lowest instruction of which it is itself a part.

(Examples on next page)

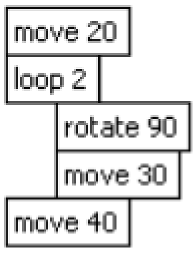
Examples:

Programme	Explanations
	The `draw` method of the "move 20", "loop 2" and "move 40" instructions returns the coordinate of the **bottom left point** of the "move 40" instruction (= the lowest sub-instruction in the string of which they are themselves a part). The `draw` method of the "rotate 90" and "move 30" instructions returns the coordinate of the **bottom left point** of the "move 30" instruction (= the lowest sub-instruction in the string of which they are themselves a part).

- The `toCode(...)` method generates the source code corresponding to an instruction **and** any sub-instructions and lower-level instructions in the same chain as the instruction itself.
    - The number of spaces by which the current instruction is to be prefixed is passed as a parameter.
    - `Loop` instructions contain a block of instructions to be repeated. These instructions are enclosed in braces and indented with 3 extra spaces compared to their parent instruction.

Examples:

Programme	Explanations	Code returned

	The `toCode(...)` method of the "move 20" instruction returns the entire source code of the program (because "move 20" is the first instruction in the program).	`move 20`
		`loop 2`
		`{`
		`   rotate 90`
		`   move 30`
		`}`
		`move 40`

	The `toCode(...)` method in the "loop 2" instruction returns the source code for itself and all the elements below it.	`loop 2`
		`{`
		`   rotate 90`
		`   move 30`
		`}`
		`move 40`

	The `toCode(...)` method of the "rotate 90" instruction returns the source code of itself, of all its sub-instructions (there are none here) and of all the instructions in its own string.	`   rotate 90`
		`   move 30`

The "Program" class

This class represents and manages a program, i.e. a logical sequence of instructions. It has a `start` attribute which points to the first instruction in the block. This attribute has an accessor.

The following methods are also available:

- The `draw(...)` method draws - if possible - the program, i.e., the instructions at a given position on a canvas.
- The `toCode(...)` method returns - if possible - the source code for the current program.
- The `saveToFile(...)` method saves the source code corresponding to the program in the text file whose name is passed as a parameter.
- The `parseLine(...)` method analyses ***a*** line of text and returns the corresponding instruction. Spaces at the beginning and end of any line are ignored. If a line is empty, the method returns `null`. Any instruction with one of the two forms:

  ```
  <instruction name> <parameter value>

  <instruction name>
  ```

  Examples:

Line	**Instruction to be created**
`move 40`	An instruction whose `cmd` attribute is "move" and `param` is 40.
`up`	An instruction whose `cmd` attribute is "up" and `param` is 0.

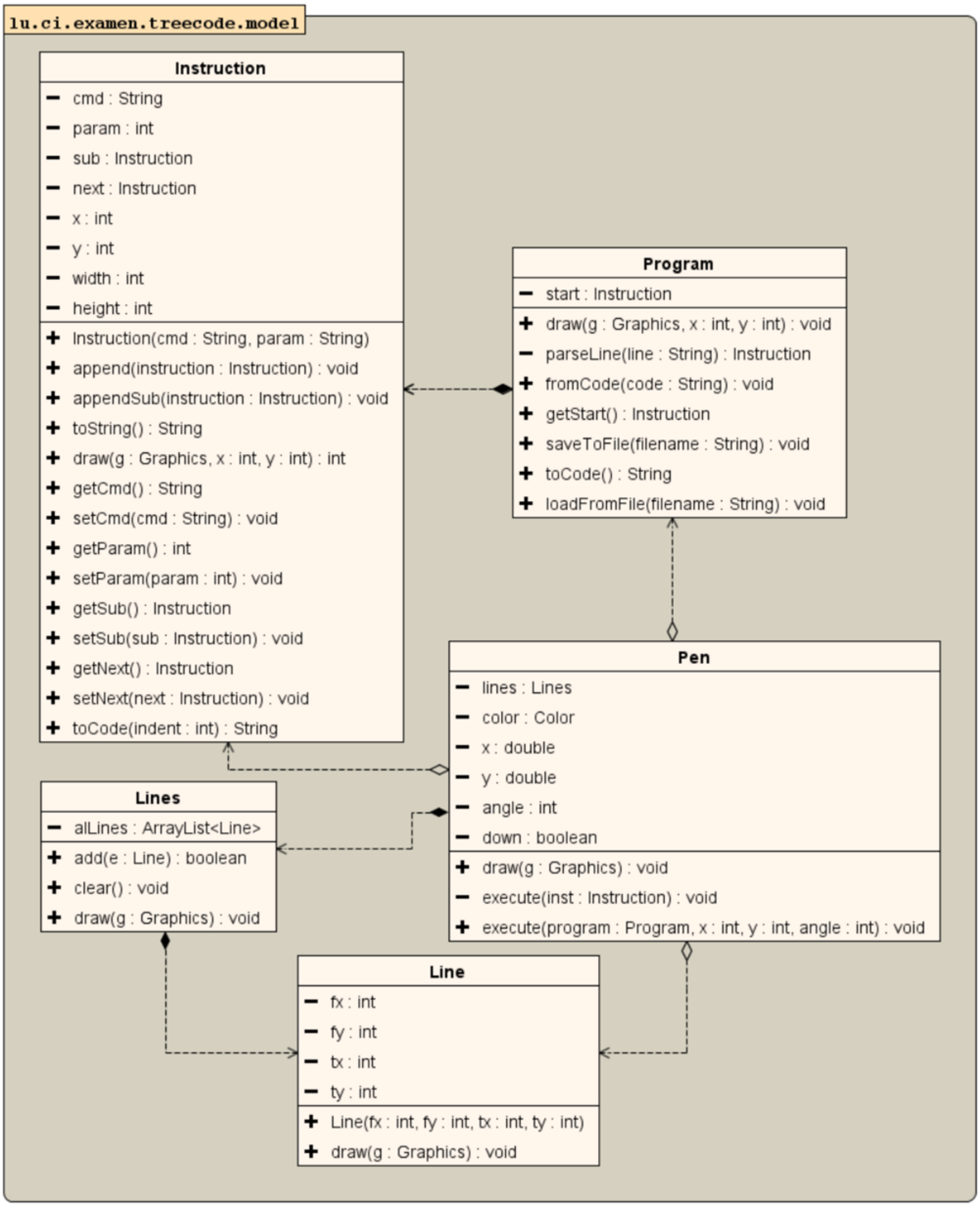
- The `fromCode(...)` method reads an entire source code and creates its internal structure.
  - The method assumes that the source code passed as a parameter is formally correct, i.e., that each line contains either a statement or an opening or closing brace.
  - The method must ignore indentation spaces in front of a line, as well as any spaces at the end of a line.
  - The case of the code should also be ignored.
  - The method reads one line at a time and reacts according to the content read.
  - It always maintains a stack of instructions. The instruction at the top of the stack always corresponds to the last instruction in the current instruction string read by the method and will be replaced when the method finds the next instruction in the same instruction string.

Example:

N°	**Code**	**Adds instruction to …**	**Heap**
1	`loop 4`	Add the instruction to the program.	`loop 4`
2	`{`	Add the next instruction as a sub-instruction of "loop 4".	`loop 4, move 10`
3	`  move 10`		
4	`  rotate 90`	Add the instruction to "move 10".	`loop 4, rotate 90`
5	`}`	/	`loop 4`
6	`up`	Add the instruction to "loop 4".	`up`

- The `loadFromFile(...)` method loads source code from the file whose name is passed as a parameter.

UML diagram of the model to be implemented.


lu.ci.examen.treecode.model
Instruction
− cmd : String
− param : int
− sub : Instruction
− next : Instruction
− x : int
− y : int
− width : int
− height : int
+ Instruction(cmd : String, param : String)
+ append(instruction : Instruction) : void
+ appendSub(instruction : Instruction) : void
+ toString() : String
+ draw(g : Graphics, x : int, y : int) : int
+ getCmd() : String
+ setCmd(cmd : String) : void
+ getParam() : int
+ setParam(param : int) : void
+ getSub() : Instruction
+ setSub(sub : Instruction) : void
+ getNext() : Instruction
+ setNext(next : Instruction) : void
+ toCode(indent : int) : String
Program
− start : Instruction
+ draw(g : Graphics, x : int, y : int) : void
− parseLine(line : String) : Instruction
+ fromCode(code : String) : void
+ getStart() : Instruction
+ saveToFile(filename : String) : void
+ toCode() : String
+ loadFromFile(filename : String) : void
Pen
− lines : Lines
− color : Color
− x : double
− y : double
− angle : int
− down : boolean
+ draw(g : Graphics) : void
− execute(inst : Instruction) : void
+ execute(program : Program, x : int, y : int, angle : int) : void
Lines
− alLines : ArrayList<Line>
+ add(e : Line) : boolean
+ clear() : void
+ draw(g : Graphics) : void
Line
− fx : int
− fy : int
− tx : int
− ty : int
+ Line(fx : int, fy : int, tx : int, ty : int)
+ draw(g : Graphics) : void

Exercise H.25: Organization chart

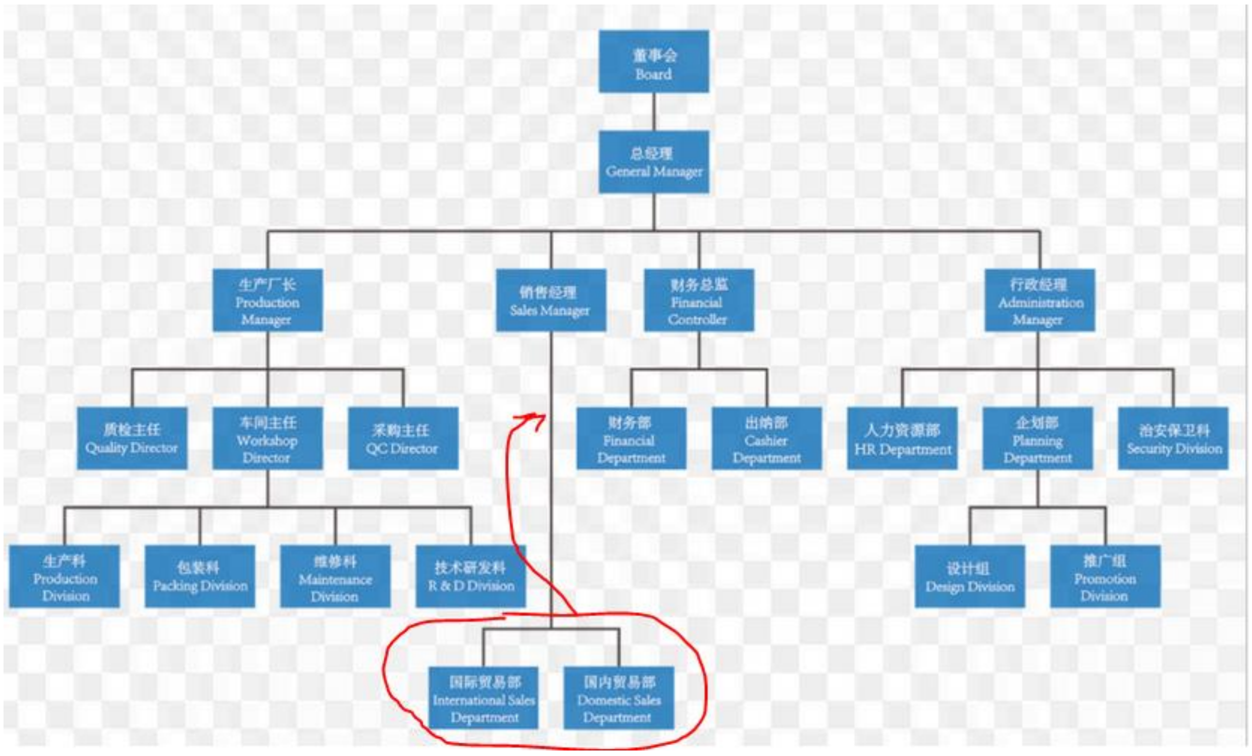
You have been asked to implement an application for creating and viewing flowcharts. Of course, you also need to be able to save them and load them from files.

The organisation chart is structured as follows:

1. A person can have one and only one boss.
2. Any boss may have one or more people working for him. The order of these people is maintained in the lexicographical order of their names.
3. A person has a surname, first name and title.

The following constraints and simplifications apply to the visuals:

1. People at the same level are all drawn at the same height. For the example below: the boss is at the very top (= level 0), the deputy boss below him (= level 1), the 4 administrators are also at the same height (= level 2), anyone working directly for one of them is drawn lower down (= level 3), ...
2. The rectangles representing a person are all the same size.


董事会 Board
总经理 General Manager
生产厂长 Production Manager
销售经理 Sales Manager
财务总监 Financial Controller
行政经理 Administration Manager
质检主任 Quality Director
车间主任 Workshop Director
采购主任 QC Director
财务部 Financial Department
出纳部 Cashier Department
人力资源部 HR Department
企划部 Planning Department
治安保卫科 Security Division
生产科 Production Division
包装科 Packing Division
维修科 Maintenance Division
技术研发科 R & D Division
设计组 Design Division
推广组 Promotion Division
国际贸易部 International Sales Department
国内贸易部 Domestic Sales Department