Lab session 6
Interfaces & Abstract Data Types
Polytech Montpellier – IG3

To propose Abstract Data Types in Java we've seen both abstract classes and interfaces. We will apply what we learned to propose a **graph library**. You can work in teams of 2 students.

In each part below, **before implementing**, provide / extend a UML class diagram.

PART 1 - *The basics of a package for representing graphs*

(1) You've just started to see the Graph formalism (some of you for the first time), which can be used to model a very large number of engineering problems. Gather the first elements you've seen on this topic to propose a `graph` package containing **only declarations** of:

- a `Vertex` class:
  - `id` (unique to each vertex)
  - `info: Object`
  - `color:…`
  - `…  // additional stuff but no reference to the Edge class`
- an `Edge` abstract class or concrete class or interface (which is best suited here?) with:
  - `id` (unique to each edge)
  - `color: …`
  - `ends: Vertex[2]`
  - `value: double`
  - `...`
- a `UndirectedEdge` class (or is it the same as Edge?):
  - `getEnds(): Vertex []`
- a `DirectedEdge` class:
  - `source: int // 0 or 1`
  - `getSource(): Vertex`
  - `getSink(): Vertex`
- a `Graph` abstract class or interface (choose the best one!) containing the necessary methods to be used for solving a graph problem such as
  - `nbOfVertices():int`
  - `nbOfEdges():int`
  - `addVertex(Vertex)`
  - `addEdge(Vertex,Vertex,EdgeKind) // EdgeKind: 'directed' or 'undirected'`
  - `isConnected(Vertex,Vertex):boolean // whether there is a path between the two vertices (without accounting for the edge directions)`
  - `isConnected(): boolean // says whether all vertices are interconnected`
  - `getEdges(Vertex, Vertex): Edge[] // give edge(s) connecting these vertices`
  - `getEdges(): Edge [] // give all edges of the graph`
  - `getVertices(): Vertex [] // give all vertices of the graph`
  - `getNeighborEdges(Vertex): Edge [] // give edges connected to this vertex`

- Add informative `toString()` methods in the above classes.
- a `MyTest` class that just declares and initializes a `Vertex` and an `Edge`. These files should be working fine. Your files should be in `src` and `bin` folders. Classes for vertices and edges should contain constructors so that new instances can be generated easily.

(2) Generate the Javadoc documentation of your source files in a `doc` folder:

```
javadoc -sourcepath src/ -d doc/ graph
```

Use command `man javadoc` for more details about Javadoc or visit the following Webpage: https://docs.oracle.com/en/java/javase/15/javadoc/javadoc-command.html

PART 2 - *Packaging and releasing your package*

(3) Pack up the doc and the bin (only those folders) into a `graph.jar` file to release your library. In the manifest file, indicate that TestGraph is the Main class.

(4) Create a code repository named `graphJava` on GitHub hosting only the two following elements:
- your `graph.jar` library
- a `README.md` explaining in very few instances what we've got here, mentioning the name of the developers, the date, the name of the module and the school.

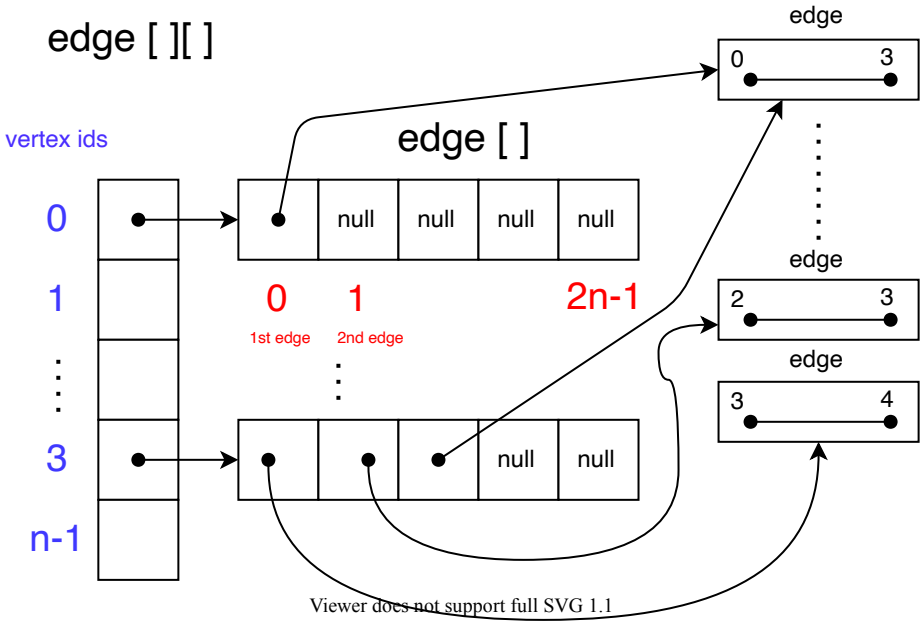
PART 3 - *Proposing an implementation of the Graph abstract class/interface*

(5) Choose an animal as team name (as this is the theme of the session on interfaces!). Add an entry in the Moodle assignment (session 6) to the name of your animal and precise the URL of the GitHub page of the package you just developed.

(6) Ask for the name of your neighboring team in the room, then get the latest working version of their `graph` package (git clone or simply download from their GitHub page).

(7) Run their library with the `-jar` flag (it should run the `TestGraph` class). In case of a problem detected in the `graph` library you fetched, inform the concerned team (create an `issue` in their GitHub repository) so that they patch their library, update their code repository, then download the new version to go on with your tests. This procedure should be repeated later when needed.

(8) As a physical representation for a graph, propose an `IncidenceArrayGraph` class implementing their `Graph` interface through:
- two arrays for `vertices` and `edges`
- an incidence array (`array` with one entry per `Vertex` and each entry is itself an `array` of `Edges`) that relies only on primitive data types and `arrays` of these types (no Java Collection allowed yet).



Each `Edge[i]` stores references to all edges to which the vertex with id `i` is connected (whatever the direction of the edge in the case of directed edges).

Provide the necessary constructor(s) (accepting the max number `n` of vertices as parameter) and the methods implementing those listed in `Graph`.

(9) Write a `TestGraphImpl` class that declares a graph, initializes it with a call to the constructor of your implementing class, then add 3 vertices and 2 edges between them. The code should then ask for the list of vertices (obtaining an array), or be able to check whether there is an edge between this and that vertex.

(10) Once you're happy with your work, package the implementation and `TestGraphImpl` inside a jar file such as `graphImpl.jar`, then test that it is runnable (you should need to indicate `-cp graph.jar`).

(11) Now you will propose your implementation as an addition to the repository where you found the `Graph` interface:

(1) Fork their repository (see https://help.github.com/articles/fork-a-repo/ for this), this should create a new repository for you (attached to your GitHub account)

(2) Clone this new repository in a new folder on your computer, then add your files in this folder and its subfolders (source and bytecode separately), test that the integration is working fine (java execution) and push online.

(3) Issue a pull request on the repository of the other team (see https://help.github.com/articles/using-pull-requests/) so that they consider the modifications of their repo that you propose in your forked repo.

PART 4 - *Representing a maze by a graph*

A maze can be represented as a 2D array (cf previous sessions) but also as a `Graph` with an edge between `cells` which are communicating (that is adjacent, see wood and rope paths between rock platforms in the picture). Because we're using graphs, we're not restricted anymore to the case of regular mazes where each cell has the same dimensions, and where cells are organized in rows (of the same size), as in the picture.

(12) In a new project/package (called something like `MazeGame`) create a `CellDescription` class with

- `width` and `length` fields of `int` type.
- A `containsMonster` field (being a `boolean`) indicating whether a monster is initially present in the cell.
- An `nbPerson` field indicating how many person can be in the cell simultaneously

Then create a `Maze` class, referring to the `Graph` class (should we inherit from it or just use it by composition?). The maze has a `cells` and a `links` field. This fields will be initialized thanks to `getVertices()` and `getEdges()` methods of the `IncidenceArrayGraph` class (what? You don't have such methods? Then add them quickly: there just simple getters).
Add to the `Maze` class `departure` and `arrival` fields of `cell` type (see 'D' and 'A' in the picture). It has a `Maze(size,density,dangerousness)` constructor that:

- initializes the maze with `size` cells, each having a probability of `dangerousness` to host a monster.
- add links between randomly chosen cells (the `density` indicates which percentage of `cells x cells` possible links should be made available).
- choose a `departure` and `arrival` cells.
- if `departure` and `arrival` cells are not already `isConnected(…,…)`, then it creates a path between them by traversing `size/2` intermediary cells (chosen at random).

(13) Does modifying `cells` and `links` changes the `vertices` and `edges` fields of the Graph from which `cells` and `links` were obtained? Why?

(14) Implement `toString()` methods in necessary classes and a `main` method in `Maze` to test your maze generation program. You should be able to see that when increasing `density`, then the two chosen departure and arrival cells are more and more likely to be connected without the need to build a path between them.

(15) Propose a `Game` class declaring and printing a `Maze` with 10 cells.

PART 5 - *Exploring the maze*

(16) When exploring the maze we want to know whether we already `visited` a cell that we encounter (so as not to turn around). Add what is necessary to your cells to know whether they have been explored or not.

(17) Implement an `findPath` algorithm recursively exploring the maze from the `departure` cell until the `arrival` cell is reached. Don't forget to mark the cells when you visit them, and to unmark them later if you come back from a dead end to try an alternative path. If you find a solution path in the given maze, print the vertices `id`s in the order they are visited to reach the other entrance of the maze.

(18) Improve your path finding algorithm to propose a path with the smallest number of monsters encountered.