TP3 Labyrinthe en mode graphique : semaines 19/3 et 2/4

Table des matières

1) Présentation du sujet

Un labyrinthe est enregistré dans un fichier texte, en respect du format ci-dessous :

5 5 0 0 4 4 (taille du labyrinthe en X et Y, point de départ en X et Y, point d'arrivée en X et Y)
_X___ (dessin du labyrinthe avec **X** un mur et **_** une case à trou)
_XX__
_ XX __
_ XX __
__X__

L'objectif est de charger le fichier dans des objets (en mémoire) du labyrinthe, de l'afficher et de se déplacer du point de départ au point d'arrivée en passant par les cases à trou **_** et en évitant les murs **X**.

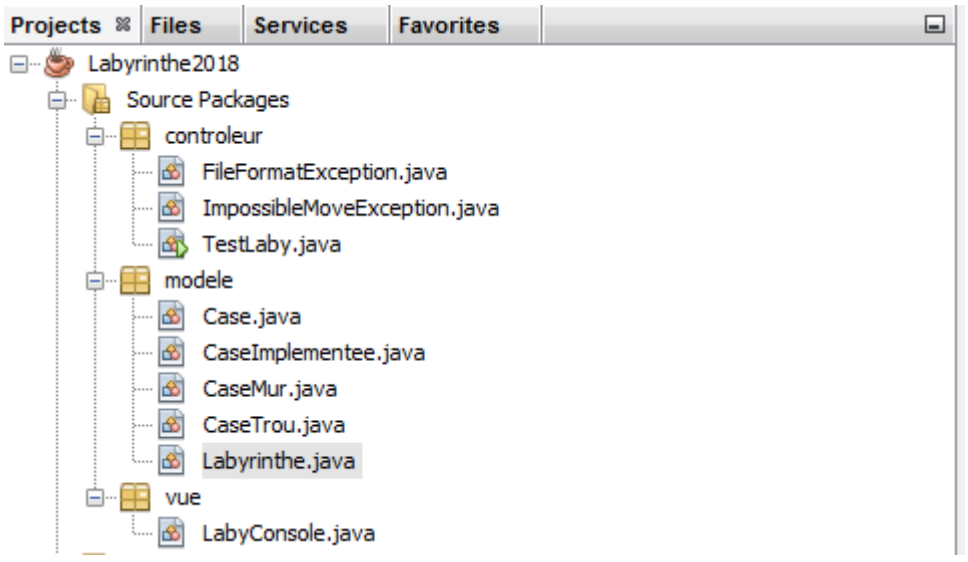
2) Code en mode console à télécharger de campus

Le fichier **Labyrinthe2018.zip** contient le code en mode console de ce TP3, à télécharger et à dézipper de la page campus de « POO Java » http://campus.ece.fr/course/view.php?id=124. Le fichier labyrinthe.txt se trouve dans ce fichier zip. Vous pouvez le modifier, en respect du format décrit plus haut.

La **javadoc** (documentation en ligne) de ce code se trouve dans le dossier **dist** de ce fichier zip. Pour récupérer le code de ce fichier sur NetBeans, consulter le tutoriel Mes premiers pas sur NetBeans.

3) Le pattern Modèle Vue Contrôleur du code

Comme le montre la copie d'écran ci-dessous, ce code est structuré en 3 packages selon le pattern MVC *modele*, *vue* et *controleur*.



Pour plus d'informations sur le pattern MVC, consulter les sources suivantes :

https://openclassrooms.com/courses/apprenez-a-programmer-en-java/mieux-structurer-son-code-le-pattern-mvc

https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur

3.1) Le package *modele*

Le package *modele* contient les classes suivantes où se trouvent les données :

- Une interface *Case* représente une case du labyrinthe avec les prototypes des méthodes (getters et setters) suivantes :

```
public interface Case {
    public int getPositionX() ; // retourne la position en X (colonne) de la case
    public int getPositionY() ; // retourne la position en Y (ligne) de la case
    public boolean getVisited(); // retourne un booléen indiquant si la case est visitée ou non
    public void setVisited(); // modifie la case pour qu'elle soit visitée
    public Case getVoisin(int i); // retourne le voisin numero i de la case
    public int getNbVoisins(); // retourne le nombre de voisins de la case
}
```

- Une classe *CaseImplementee* implémente l'interface Case et toutes ses méthodes. Ses attributs protégés sont :

```
    protected int positionX, positionY; // position courante dans la case
    protected boolean visited, moved; // booléens de visite et de d'accès à la case
    protected ArrayList<Case> voisins; // Liste des cases voisines
    protected int nb_voisins; // nombre de cases voisines
```

- Les classes *CaseMur* et *CaseTrou* héritent de leur classe mère *CaseImplementee* et leur constructeur initialise le booléen *moved* de la classe mère.

- Une classe *Labyrinthe* dispose des attributs privés suivants :

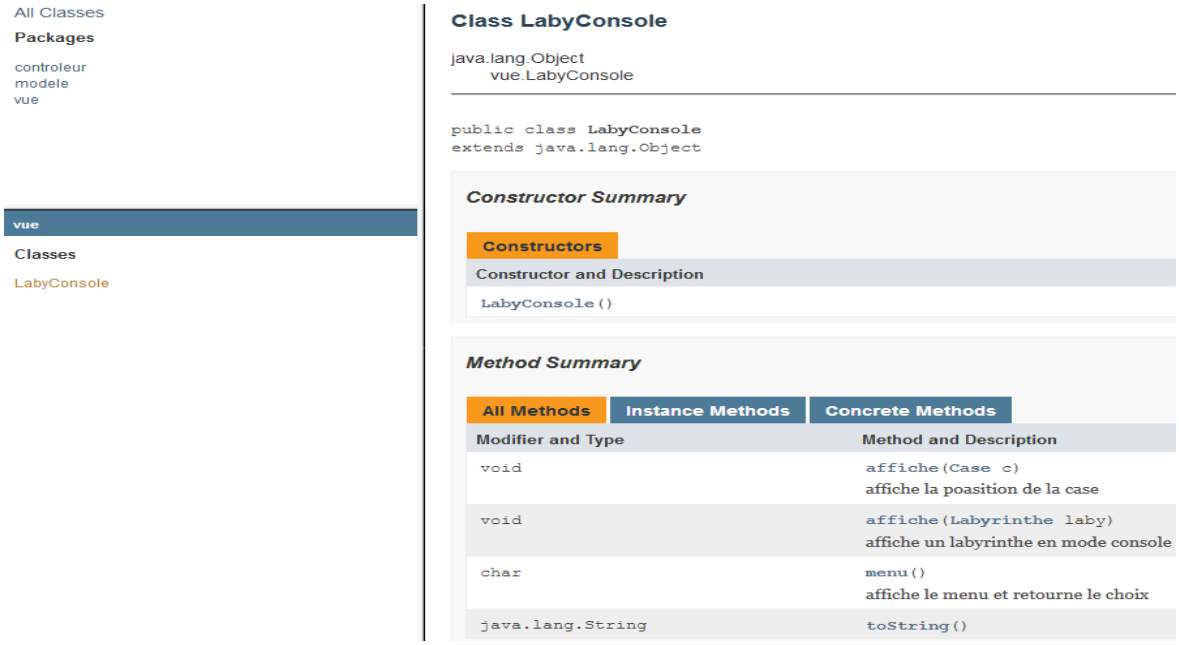
```
private int tailleX, tailleY; // largeur et hauteur
private int departX, departY; // coordonnées de départ
private int arriveeX, arriveeY; // coordonnées d'arrivée
private int posX, posY; // coordonnées courantes
private ArrayList<Case> grille; // // L'ensemble des cases du labyrinthe
```

Elle contient entre autres les méthodes suivantes :

```
// Constructeur qui initialise le labyrinthe à partir du fichier en  paramètre
public Labyrinthe(File fic) throws FileFormatException { ... }

// Tente de se déplacer dans la case ligne, colonne et de la visiter
public void move(int ligne, int colonne) throws ImpossibleMoveException { ... }

// deplacement automatique sans deborder et si pas déjà visité
public void autoMove() throws ImpossibleMoveException { ... }

// retourne une case du labyrinthe
public Case getCase(int lig, int col) { ... }

// setter pour visiter une case du labyrinthe
public void setVisited(int lig, int col) { ... }

// Les autres getters pour les autres attributs privés
...
```

3.2) Le package *vue*

Le package *vue* contient la classe *LabyConsole* associée à l'IHM, comme le montre la javadoc ci-dessous :



All Classes

Packages

controleur
modele
vue

vue

Classes

LabyConsole

Class LabyConsole

java.lang.Object
vue.LabyConsole

```
public class LabyConsole
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description
LabyConsole()

Method Summary

All Methods | Instance Methods | Concrete Methods

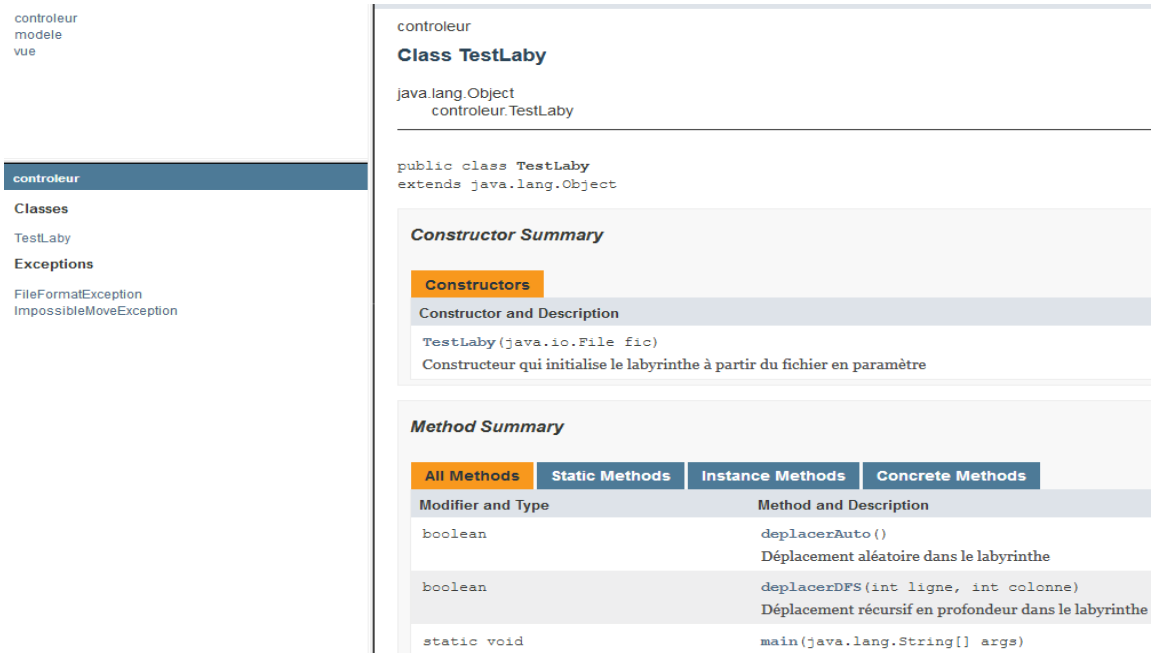
Modifier and Type	Method and Description
void	affiche(Case c) affiche la poasition de la case
void	affiche(Labyrinthe laby) affiche un labyrinthe en mode console
char	menu() affiche le menu et retourne le choix
java.lang.String	toString()

3.2) Le package *controleur*

Le *controleur* « permet de faire le lien entre la vue et le modèle lorsqu'une action utilisateur est intervenue sur la vue. C'est cet objet qui aura pour rôle de contrôler les données. » **1**

1 https://openclassrooms.com/courses/apprenez-a-programmer-en-java/mieux-structurer-son-code-le-pattern-mvc

Ce package contient la classe *TestLaby* avec le *main* et les méthodes de déplacement, comme montré dans la javadoc ci-dessous, ainsi que Les classes d'exception *FileFormatException* et *ImpossibleMoveException*



controleur
modele
vue

controleur

Classes

TestLaby

Exceptions

FileFormatException
ImpossibleMoveException

controleur

Class TestLaby

java.lang.Object
controleur.TestLaby

```
public class TestLaby
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description
TestLaby(java.io.File fic) Constructeur qui initialise le labyrinthe à partir du fichier en paramètre

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	deplacerAuto() Déplacement aléatoire dans le labyrinthe
boolean	deplacerDFS(int ligne, int colonne) Déplacement récursif en profondeur dans le labyrinthe
static void	main(java.lang.String[] args)

4) Résultats en mode console

Ci-dessous, un extrait des résultats obtenus en mode console. A vous de tester le reste…

```
run:
Entrez le nom du fichier du labyrinthe :
labyrinthe.txt
5 5
_X___
_X___
_XX__
_____
__X__
1 Déplacer automatique en profondeur
2 Déplacer aléatoire
0 Quitter
Entrez votre choix :
1
ligne = 0 colonne=0
VX___
_X___
_XX__
_____
__X__
ligne = 1 colonne=0
VX___
VX___
_XX__
_____
__X__
ligne = 2 colonne=0
VX___
VX___
VXX__
_____
__X__
ligne = 3 colonne=0
VX___
VX___
VXX__
V____
__X__
ligne = 4 colonne=0
```

TRAVAIL A FAIRE :

Exercice 1 : conception du diagramme de classes

Pour la conception du diagramme de classes, appuyez-vous sur le « Support conception orientée objet » sur la page campus du cours : http://campus.ece.fr/course/view.php?id=124.

A l'aide des explications ci-dessus sur le pattern MVC du code et de la javadoc, donnez le diagramme de classes de ce code. N'oubliez pas les niveaux de visibilité pour les attributs et les méthodes (**public private** ou **protected**) et les multiplicités des relations inter-classes.

Pour créer un diagramme de classes avec Draw.io :

1. Sur DropBox, cliquer sur « Create new diagram »
2. Laisser « Blank Diagram » et faites « Create »
3. Dans le menu à gauche, un sous menu se nomme UML.

Tous les éléments UML sont disponibles dans ce sous-menu. Pour enregistrer la progression « File > Save as ».
Pour exporter en image, pdf, html, etc .. « File > Export as ».

Exercice 2 : implémentation du labyrinthe en mode graphique

Des extraits utiles de la documentation Javadoc sont fournis en **ANNEXE** : pour plus de détails, consulter la documentation des API Java *(javadoc)* sur la page campus du cours.
Commentez votre code : devant les classes, attributs, constructeurs et méthodes vos commentaires doivent respecter le format Javadoc /** commentaires */ pour que ceux-ci apparaissent dans la javadoc générée

2.1) Dessinez votre fenêtre

Avant de dessiner le labyrinthe, vous devez créer une fenêtre en mode graphique. Pour cela, il existe une classe nommée **JFrame** dans le package **javax.swing**. Ci-dessous, deux approches possibles.

Première approche : dans la classe *TestLaby* du *controleur*, on crée un objet de type **JFrame**.

```
import javax.swing.*;

public class TestLaby
{
  private JFrame fen = new JFrame();

 public static void main (String args[])
 {
   fen.setSize (300, 150);
   fen.setTitle ("Ma premiere fenetre");
   fen.setVisible (true);
   ...
 }
}
```

Seconde approche : dans la *vue*, il faut définir une nouvelle classe *LabyGraphique* dérivée de **JFrame** et dans la classe *TestLaby* du *controleur*, on crée un objet de type *LabyGraphique*.

```
import javax.swing.*;

class LabyGraphique extends JFrame
{
 public LabyGraphique ()  { // constructeur
     setTitle ("Ma premiere fenetre");
     setSize (300, 150);
 }
}

import vue.LabyGraphique;

public class TestLaby
{
 private LabyGraphique fen = new LabyGraphique ();

 public static void main (String args[]) {
     fen.setVisible (true);
     ...
 }
}
```

En utilisant l'une de ces deux méthodes, compléter ce code pour obtenir une fenêtre à votre goût

2.2) Dessinez votre labyrinthe avec style (layout)

Pour mettre en forme (layout) votre futur labyrinthe, vous devez d'abord importer la librairie *java.awt.**. Par exemple, l'extrait de code suivant montre comment dessiner une grille pour un labyrinthe composé de boutons :

```
import javax.swing.*;
import java.awt.*;
import modele.*;
public class LabyGraphique extends JFrame
{
   private JPanel pan ; // panneau
   private JButton boutons[][]; // matrice de boutons

   public LabyGraphique (){ // constructeur
     setTitle ("Mon labyrinthe");
     setSize (300, 150);
     pan = new JPanel(); // instancier le panneau
     getContentPane().add(pan); // ajouter le panneau dans la fenêtre
   }

  // Méthode qui affiche la grille du labyrinthe
  public void affiche(Labyrinthe laby) {
      pan.setLayout(new GridLayout(laby.getTailleY(), laby.getTailleX())); // mise en forme avec une grille
      boutons = new JButton[laby.getTailleY()][]; // instancier les lignes de la matrice de boutons

      for (int i = 0; i < laby.getTailleY(); i++)
         boutons[i] = new JButton[laby.getTailleX()];// Pour chaque ligne de la matrice, instancier les boutons

      // Ajouter les boutons dans le panneau
      for (int i = 0; i < laby.getTailleY(); i++) {
         for (int j = 0; j < laby.getTailleX(); j++) {
            boutons[i][j] = new JButton(); // instancier chaque bouton
            pan.add(boutons[i][j]);
         }
      }
      // rendre la fenetre visible
      this.setVisible(true);
}

import vue.LabyGraphique;
public class TestLaby
{
  private  LabyGraphique fen = new LabyGraphique(); // définir et instancier un objet de LabyGraphique
  public static void main (String args[]) {
    try {   ...
            test = new TestLaby(new File(nomlaby));
            fen.affiche(laby);
      ...
    } catch ...
  }
}
```

Complétez ce code pour effectuer les traitements suivants dans la classe *LabyGraphique* :

- Avec la méthode *getCase* de *Labyrinthe*, récupérer chaque case et afficher dans chaque bouton un texte ou une image associée à la case, soit de type *CaseMur* soit de type *CaseTrou*.
- Ajouter des boutons pour les déplacements en profondeur, aléatoire et tout autre déplacement que vous souhaitez mettre en place : par exemple, déplacement manuel en interaction avec l'utilisateur, ou automatique avec BFS (Breadth First Search) pour un parcours en largeur avec une file d'attente etc. ☺
- Toute autre idée qui pourrait améliorer le style de votre labyrinthe, comme par exemple y ajouter un personnage ☺

2.3) Votre labyrinthe écoute les évènements (listener)

Les interfaces graphiques sont fondées sur le principe des événements qui permettent à l'utilisateur d'interagir avec le clavier ou la souris. La plupart des événements sont créés par des composants graphiques qu'on aura introduits dans la fenêtre (menu, boutons, etc).
Pour cela, il est nécessaire d'utiliser le package **java.awt.event**. Les fenêtres doivent utiliser les événements via des écouteurs (listener) comme **ActionListener**. La classe *TestLaby* du *controleur* doit implémenter l'interface ActionListener, et éventuellement MouseListener, comme le montre l'exemple ci-dessous :

```
import javax.swing.*;
import java.awt.event.*; // pour les listener et les event : ActionListener, ActionEvent, MouseListener, MouseEvent etc.
import vue.LabyGraphique ;
import java.io.File ;
import modele.*;

public class TestLaby implements ActionListener
{
  private  LabyGraphique fen = new LabyGraphique();
  private static Labyrinthe laby;

  public TestLaby (File fic) { // constructeur avec le fichier en paramètre
     laby = new Labyrinthe(fic); // instancier le labyrinthe à partir du fichier

     // Ajouter un ActionListener pour chacun des boutons de déplacement dans la fenêtre qui écoute
     ….addActionListener (fen); // … à compléter pour chacun des boutons de déplacement
  }

 // Méthode appelée en fonction de chaque événement
 public void actionPerformed(ActionEvent e) {…} // à compléter

 public static void main (String args[]) {
    fen.setVisible(true);
    … // à compléter pour ouvrir le fichier du labyrinthe avec JFileChooser et instancier l'objet de TestLaby
  }
}
```

- Par vous entrainer, commencer à écrire un programme qui réagit au clic de la souris dans la fenêtre en affichant "clic souris". Pour cela, il faut implémenter l'interface **MouseListener** et ses 5 méthodes associées aux 5 actions possibles sur la souris (voir annexes) : *mouseClicked* (clic), *mouseEntered* (entrée dans la fenêtre), *mouseExited* (sortie de la fenêtre), *mousePressed* (pression sur la souris) et *mouseReleased* (libération de la souris). Récupérer et afficher les coordonnées d'un clic généré par l'événement *e* avec *e.getX()* et *e.getY()*. Vous pouvez effectuer les mêmes actions pour afficher l'entrée, la sortie, la pression et la libération de la souris. ☺

- Dans le *main* de la classe *TestLaby*, compléter le code pour ouvrir un fichier avec la classe **JFileChooser** de swing etc.

- Dans la méthode *actionPerformed* de la classe *TestLaby*, compléter le code ci-dessus pour interagir avec les boutons de déplacement de votre labyrinthe et visualiser le déplacement du personnage en visite.

ANNEXE : extraits de la documentation API

java.awt

Class Component

Method Summary	
void	**addMouseListener**(MouseListener l) Adds the specified mouse listener to receive mouse events from this component.

Class Container

Method Summary	
Component	**add**(Component comp) Appends the specified component to the end of this container.

javax.swing

Class JButton

Method Summary	
String	**getText**() Returns the button's text.
void	**setText**(String text) Sets the button's text
void	**addActionListener**(ActionListener l) Adds an ActionListener to the button.

Class JFrame

Constructor Summary
JFrame(String title) Creates a new, initially invisible Frame with the specified title.

Field Summary	
static int	**EXIT_ON_CLOSE** The exit application default window close operation.

Method Summary	
Container	**getContentPane**() Returns the contentPane object for this frame.
void	**setContentPane**(Container contentPane) Sets the contentPane property.
void	**setDefaultCloseOperation**(int operation) Sets the operation that will happen by default when the user initiates a "close" on this frame.
void	**setSize**(int width, int height) Resizes this component so that it has width width and height height.
void	**setVisible**(boolean b) Shows or hides this Window depending on the value of parameter b.

java.awt.event

Interface ActionListener

Method Summary	
void	**actionPerformed**(ActionEvent e) Invoked when an action occurs.

Class ActionEvent

Method Summary	
Object	**getSource**() The object on which the Event initially occurred.
String	**toString**() Returns a String representation of this EventObject.

Interface MouseListener

Method Summary	
void	**mouseClicked**(MouseEvent e) Invoked when the mouse button has been clicked (pressed and released) on a component.
void	**mouseEntered**(MouseEvent e) Invoked when the mouse enters a component.
void	**mouseExited**(MouseEvent e) Invoked when the mouse exits a component.
void	**mousePressed**(MouseEvent e) Invoked when a mouse button has been pressed on a component.
void	**mouseReleased**(MouseEvent e) Invoked when a mouse button has been released on a component.

Class MouseEvent

Method Summary	
int	**getX**() Returns the horizontal x position of the event relative to the source component.
int	**getY**() Returns the vertical y position of the event relative to the source component.