Projet JAVA-POO

UNIVERSITÉ DE NANTES

Borisov Vadim
Seck Abdoulahat

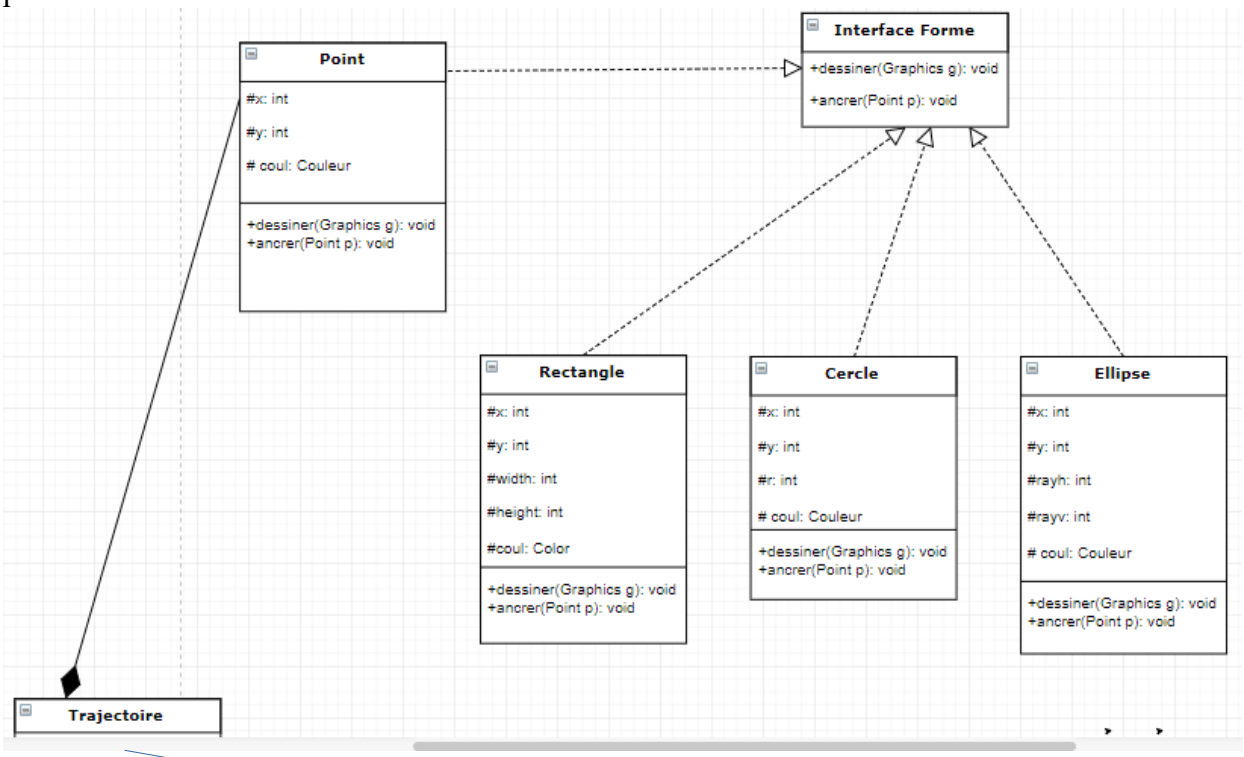
Groupe: 385
Formation: L2 Info
Année: 2017/2018

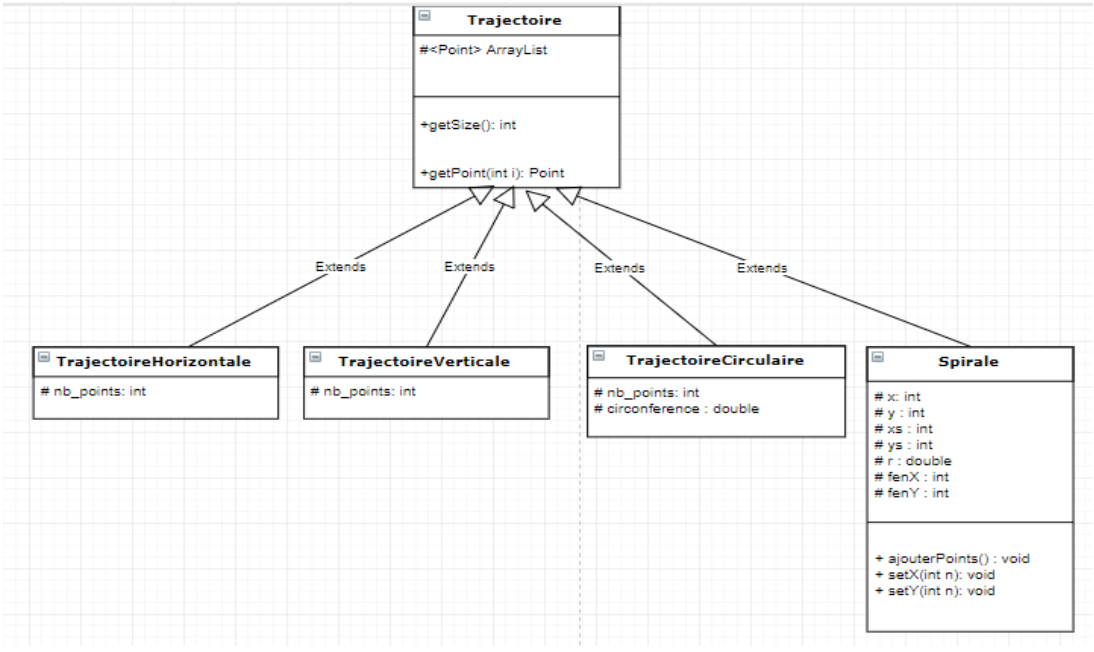
Introduction:

Notre travail consiste à créer un environnement graphique qui permet de déplacer les formes selon différentes trajectoires choisies par l'utilisateur dans la classe principale. Comme le travail est fait en binôme les tâches étaient réparties: il fallait programmer la partie forme indépendemment de la partie trajectoire et les associées après.

Modélisation:

visibilité:
+: public
-: private
#: protected





Il existe une hiérarchie des trajectoires avec une super-classe Trajectoire et des sous-classes TrajectoireHorizontale, TrajectoireVerticale, TrajectoireCirculaire, Spirale. Chaque instance des sous-classes de super classe Trajectoire est toujours l`ensemble des points, donc on hérite simplement ArrayList qui sera la base de contenu des points de la trajectoire à stocker.

Chaque trajectoire est définie par les points, donc on a une relation entre la super classe Trajectoire et la classe Point.

De l'autre coté il existe l'interface Forme. N'importe quel forme doit être capable de dessiner sur l'écran et être ancrer à un point de la trajectoire, donc les classes Rectangle, Cercle, Ellipse et Point vont implémenter l`interface Forme.

Spécification:

Classe: Trajectoire
Rôle: la classe **Trajectoire** permet de stocker une liste de points d'une trajectoire.

Méthodes:

Signature	Description
Trajectoire()	Construit une instance de type **ArrayList**
`int getSize()`	Retourne la taille de la liste des points
`Point getPoint(int i)`	Retourne le point à l'indice **i**

Classe: TrajectoireCirculaire
Rôle: la classe **TrajectoireCirculaire** permet de tracer une trajectoire circulaire de centre (x,y) et de rayon r.

Méthodes:

Signature	Description
`TrajectoireCirculaire(int x, int y, int r)`	Construit une instance de type **ArrayList** tel que nombre de points dépend du rayon **r**

Classe: TrajectoireHorizontale
Rôle: la classe **TrajectoireHorizontale** permet de tracer une trajectoire horizontale avec le point de départ P1 et le point d'arrivée P2.

Méthodes:

Signature	Description
`TrajectoireHorizontale(Point P1, Point P2)`	Construit une instance de type **ArrayList** tel que le nombre de points dépend des abscisses de P1 et P2.

Classe: TrajectoireVerticale
Rôle: la classe **TrajectoireVerticale** permet de tracer une trajectoire verticale avec le point départ P1 et point d'arrivée P2.

Méthodes:

Signature	Description
`TrajectoireVerticale(Point P1, Point P2)`	Construit une instance de type **ArrayList** tel que le nombre de points dépend des ordonnés de P1 et P2.

Classe: Spirale
Rôle: la classe **Spirale** permet de tracer une trajectoire de type spirale d'Archimède avec le centre (x,y) et les coordonnées d'arrêt de la trajectoire sur les bords de la fenêtre graphique.

Méthodes:

Signature	Description
`Spirale(int x, int y)`	Construit une instance de type **ArrayList** vide.
`void setX(int n)`	Affecte un entier **n** passé en paramètre à l'abscisse (**fenX**) qui indique l'arrêt de la trajectoire.
`void setY(int n)`	Affecte un entier **n** passé en paramètre à l'ordonnée (**fenY**) qui indique l'arrêt de la trajectoire.
`void ajouterPoints()`	Ajoute un ensemble de points dans **ArrayList**

Interface: Forme
Rôle: L'interface **Forme** permet de définir de manière abstraite toutes les formes.

Méthodes:

Signature	Description
`void dessiner(Graphics g)`	Dessine la forme dans la fenêtre graphique
`void ancrer(Point p)`	Ancre la forme à un point de la trajectoire

Classe: Cercle
Rôle: la classe **Cercle** implémente l'interface **Forme** et permet de créer une forme ovale, notamment le cercle de centre (x,y) et de rayon (r,r).

Méthodes:

Signature	Description
`Cercle(int r, Color coul)`	Construit une instance d'un cercle de rayon **r** avec la couleur **coul**
`void dessiner(Graphics g)`	Dessine l'oval `drawOval(x,y,r,r)`
`void ancrer(Point p)`	Centre le cercle sur le point p (ancrage)

Classe: Rectangle
Rôle: la classe **Rectangle** implémente l'interface **Forme** et permet de créer une forme rectangulaire

Méthodes:

Signature	Description
`Rectangle(int width, int height, Color coul)`	Construit une instance d'un rectangle de largeur **width**, de hauteur **height** et de couleur **coul**
`void dessiner(Graphics g)`	Dessine le rectangle `drawRect(x,y,width,height)`
`void ancrer(Point p)`	Ancre le coin en haut à gauche du rectangle sur **p**

Classe: Ellipse
Rôle: la classe **Ellipse** implémente l'interface Forme et permet de créer une ellipse.

Méthodes:

Signature	Description
`Ellipse(int rh,int rv, Color c)`	Construit une instance d'une ellipse de rayon horizontale **rh**, de rayon verticale **rv** et de couleur **c**
`void dessiner(Graphics g)`	Dessine un ellipse `drawOval(x,y,rayh*2,rayv*2)`
`void ancrer(Point p)`	Centre un ellipse sur le point p (ancrage)

Classe: Point
Rôle: la classe **Point** implémente l'interface **Forme** et permet de créer un point.

Méthodes:

Signature	Description
`Point(int x, int y, Color coul)`	Construit une instance d'un point de coordonnée (x,y) et de couleur **coul**
`Point(int x, int y)`	Construit une instance d'un point de coordonnées (x,y)
`void dessiner(Graphics g)`	Dessine un point `drawOval(x,y,1,1)`
`int getAbs()`	Retourne la valeur d'abscisse
`int getOrd()`	Retourne la valeur d'ordonée
`void ancrer(Point p)`	Ancre le point sur **p**

Programme principale:

On utilise l'exemple suivant: déplacer un rectangle bleu de largeur 30 et de hauteur 20 sur une trajectoire circulaire de centre (200, 100) et de rayon 50

```
import java.awt.Color;

public class AppliDessin {

	// effectuer une pause d'une certaine durée en millisecondes
	static void Pause(int millisecondes) {
		try {
			Thread.sleep(millisecondes);
		}
		catch(Exception e) {}
	}

	// ---------------------PROGRAMME PRINCIPAL-----------------------
	public static void main(String[] args) {

		// CONSTRUCTION DE LA FENETRE
		Fenetre fen = new Fenetre("window",400,400);

		// CONSTRUCTION DU RECTANGLE
		Rectangle rect = new Rectangle(30,20, Color.BLUE);

		// CONSTRUCTION DE LA TRAJECTOIRE CIRCULAIRE
		TrajectoireCirculaire t = new TrajectoireCirculaire(200,100,50);
		int nb_points = t.getSize();
		for (int i = 0; i < nb_points; ++i){
			// obtenir i-eme point de la trajectoire
			Point courant = t.getPoint(i);
			// ancrer la forme a un point courant
			rect.ancrer(courant);
			// dessiner la forme
			fen.dessiner(rect);
			// dessiner le point blanc pour visualiser la trajectoire
			fen.dessiner(new Point(courant.getAbs(),courant.getOrd(),
			Color.WHITE));
			Pause(30);
		}

	}
}
```

Conclusion:

L'existence de l'interface **Forme** et de la super-classe **Trajectoire** permet de montrer que la solution proposée est extensible, car on peut aisément créer de nouveaux formes ayant les méthodes obligatoires **dessiner** et **ancrer** qui implémentent l'interface **Forme**, ainsi que créer de nouveaux types de trajectoires qui sont toutes représentées comme la liste de points consécutifs.

Les réussites: on a réussi a créer tous les formes et trajectoires demandés, proposer une structure hiérarchique claire et propre. Le principe de l'encapsulation est respecté.

Les échecs: la représentation de la trajectoire comme la liste de points consécutifs n'est peut être pas la meilleure solution, vu qu'on est obligé de calculer tous ses points avant de déplacer la forme sur elle.
On avait réussi de construire la trajectoire du spirale d'Archimède, mais en ayant certains difficultés avec les formules mathématiques: notre rayon dépend de la vitesse du changement de l'angle.