Introduction au dessin en Java

par Christophe Dujardin

Date de publication : 25/02/05

Dernière mise à jour :

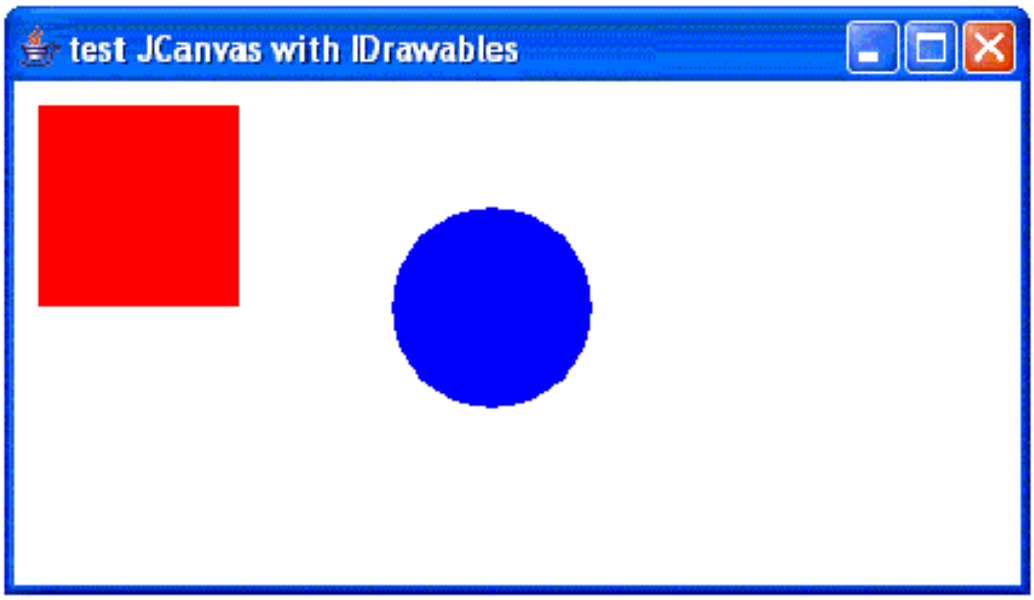
Cet article montre comment construire un système pour afficher et déplacer des figures en Java, en utilisant Java2d.

I - Introduction

L'objectif de ce tutoriel est de présenter une solution pour dessiner des figures à l'écran et de permettre de les sélectionner et de les déplacer. Ce document n'est donc pas à proprement parler un cours sur java2d, il se contente simplement de présenter rapidement les principes de base du dessin en java.

Un exemple d'utilisation de ce framework est un visualisateur de base de donnée, où chaque table est représentée par un rectangle contenant son nom ainsi que ses champs. Dans ce tutoriel, nous resterons modestes et dessinerons des figures géométriques (rond et carré), afin de pouvoir se concentrer sur les principes de base.

Pour distinguer les classes appartenant au JDK de celles que nous créons ici, nous utiliserons la convention suivante: les noms des classes du JDK seront toujours donnés complètement, c'est à dire le nom de la classe précédé par le nom du package (par exemple, la classe JCanvas est créée par nous, tandis que la classe java.awt.Point fait partie du JDK).



Panneau contenant deux figures

II - Dessiner en java : comment ça marche

A - Le système de coordonnées

Le système de coordonnées de Java utilise le pixel comme unité de mesure. Le point d'origine, de coordonnée 0,0, correspond à l'angle supérieur gauche du panneau (alors qu'on le place généralement en bas à gauche sur un schéma). La valeur de la coordonnée x croît à mesure que l'on s'éloigne vers la droite du point 0,0 et la coordonnée y croît à mesure que l'on s'éloigne vers le bas par rapport au point 0,0.

Toutes les valeurs de pixels sont des entiers, et on ne peut donc pas utiliser des nombres à virgule flottante pour représenter une valeur.

Le système de coordonnées

B - Mon premier dessin : JCanvas

Pour dessiner, nous utilisons la classe java.awt.Graphics, qui contient des méthodes permettant de dessiner du texte ou des formes sur un composant graphique comme une Applet ou, dans notre cas, un javax.swing.JPanel.

Dans un javax.swing.JPanel (ou une applet), il n'est pas nécessaire de créer un objet java.awt.Graphics (ce n'est d'ailleurs pas possible car cette classe est abstraite). En fait, cet objet Graphics est passé en paramètre de la méthode paint. Pour dessiner sur un composant, il suffit donc simplement d'écrire une classe qui hérite d'un composant et qui surcharche la méthode paint.

Dans ce tutoriel, nous travaillerons avec une sous-classe de javax.swing.JPanel que nous appellerons JCanvas. Le code de la classe JCanvas se trouve ci-dessous. Dans notre panneau, nous dessinons un carré rouge ainsi qu'un rond bleu (le panneau est montré en introduction de l'article).

Première version de la classe JCanvas

```
import java.awt.*;
import javax.swing.*;

public class JCanvas extends JPanel {

	public void paint(Graphics g) {
		Color c = g.getColor();
		g.setColor(Color.RED);
		g.fillRect(10,10,80,80);
		g.setColor(Color.BLUE);
		g.fillOval(150,50,80,80);
		g.setColor(c);
	}

}
```

Voici un exemple de code permettant d'utiliser la classe JCanvas. Nous lui donnons une couleur blanche et une taille préférée de 400 sur 200 pixels. La dernière ligne permet d'afficher le panneau à l'écran, dans une javax.swing.JFrame.

Utilisation de la classe JCanvas

```
public class Demo1 {

	public static void main(String[] args) {
		JCanvas jc = new JCanvas();
		jc.setBackground(Color.WHITE);
		jc.setPreferredSize(new Dimension(400,200));
		GUIHelper.showOnFrame(jc,"test");
	}

}
```

Voici la classe GUIHelper, qui contient le code permettant de mettre notre panneau dans une frame, et de la faire se montrer à l'écran:

La classe GUIHelper

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JComponent;
import javax.swing.JFrame;


public class GUIHelper {

	public static void showOnFrame(JComponent component, String frameName) {
		JFrame frame = new JFrame(frameName);
		WindowAdapter wa = new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				System.exit(0);
			}
		};
		frame.addWindowListener(wa);
		frame.getContentPane().add(component);
		frame.pack();
		frame.setVisible(true);
	}

}
```

III - Dessiner plusieurs formes

Jusqu'ici, nous avons appris comment fonctionne le dessin avec java2d et la classe Graphics. Nous allons, dans la suite de ce tutoriel, développer des classes permettant de dessiner, tout en tentant de respecter ces quelques critères:

- Dessiner plusieurs formes différentes,
- Conserver un code clair, simple et maintenable,
- Pouvoir ajouter et enlever facilement des formes durant l'exécution de l'application,
- Pouvoir déterminer la forme se trouvant à une position donnée sur le tableau.

A - L'interface IDrawable

L'interface IDrawable représente un objet qui sait comment il doit être dessiné. Elle contient une méthode draw, qui prend un objet de type java.awt.Graphics en paramètre. Cette méthode est chargée de dessiner la représentation de l'objet, selon son implémentation.

L'interface IDrawable

```
import java.awt.Graphics;
import java.awt.Rectangle;

public interface IDrawable {

	public  void draw(Graphics g);

	public Rectangle getRectangle();
}
```

Pour dessiner une forme, il suffit d'écrire une classe qui implémente cette interface et de l'ajouter sur le JCanvas. Pour dessiner des figures géométriques, nous écrirons une classe par type de figure. On aura donc, par exemple une classe RectangleDrawable pour dessiner un rectangle et CircleDrawable pour dessiner un cercle. Comme vous l'aurez remarqué, il s'agit d'un exemple typique de polymorphisme. Nous verrons comment implémenter ces classes plus tard.

IDrawable contient une seconde méthode, getRectangle. Cette méthode retourne un objet de type java.awt.Rectangle, dont nous reparlerons de l'utilisation plus tard.

B - Adaptation de la classe JCanvas

Il nous reste maintenant à modifier la classe JCanvas, pour qu'elle puisse contenir plusieurs objets de type IDrawable et les faire se dessiner. Nous ajoutons une variable d'instance de type java.util.List et modifions la méthode paint pour qu'elle effectue un itération sur cette liste. Elle appelle la méthode draw sur chaque IDrawable contenu dans cette liste. Nous avons également ajouté des méthodes permettant d'ajouter et d'enlever des IDrawables. La dernière méthode, clear, permet de vider la liste.

seconde version de JCanvas

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
```

seconde version de JCanvas

```
import javax.swing.JPanel;
import java.awt.Graphics;


public class JCanvas extends JPanel {

	private List drawables = new LinkedList();

	public void paint(Graphics g) {
		for (Iterator iter = drawables.iterator(); iter.hasNext();) {
			IDrawable d = (IDrawable) iter.next();
			d.draw(g);
		}
	}

	public void addDrawable(IDrawable d) {
		drawables.add(d);
		repaint();
	}

	public void removeDrawable(IDrawable d) {
		drawables.remove(d);
		repaint();
	}

	public void clear() {
		drawables.clear();
		repaint();
	}

}
```

Nous avons choisi d'utiliser une java.util.LinkedList comme implémentation de java.util.List, parce que c'est la classe la plus performante lorsqu'il s'agit de parcourir une liste au moyen d'un itérateur.

Pour permettre de redessiner le JCanvas, après avoir effectué une modification, nous appelons la méthode repaint. Cette méthode fait en sorte que la methode paint soit appelée par le système.

C - Rectangle, Point et Dimension

Dans cette section, nous allons parler de la seconde méthode de l'interface IDrawable: getRectangle. Cette méthode retourne un objet représentant l'emplacement que va occuper la forme sur le JCanvas. En effet, la classe java.awt.Rectangle représente un rectangle ainsi que sa position, selon le système de coordonnées vue dans la première partie de ce tutoriel.

un rectangle avec le système de coordonnées

Un objet de type java.awt.Rectangle est caractérisé par 4 variables d'instances qui représentent 2 concepts :

- Sa position : Il s'agit, comme montré sur le dessin, des coordonnées x et y du coin supérieur gauche de rectangle (sur le dessin, x= 10 et y= 15).
- Sa taille : width correspond à la valeur du côté horizontal et height au côté vertical. Sur le dessin, width= 20 et height= 25

Nous verrons plus tard qu'en plus de contenir ces variables, la classe Rectangle possède de nombreuses méthodes qui nous seront fort utiles.

Bien entendu, le fait d'utiliser cette classe ne nous permet pas de travailler finement.Ainsi, un rond ne sera pas représenté comme un rond, mais comme un carré (le carré tangeant au cercle pour être précis). Nous nous en contenterons pour ce tutoriel, mais il est tout à fait possible, pour des applications plus élaborées, d'utiliser une classe plus complexe, comme un polygone, par exemple.

Remarquons que l'on peut également connaître la position du coin supérieur gauche du rectangle avec la méthode getLocation, qui retourne un objet de type java.awt.Point qui contient les 2 variables x et y ainsi que des méthodes utilitaires. Pour la dimension du rectangle, on peut utiliser la methode getSize, qui retourne un objet de type java.awt.Dimension, contenant les 2 valeurs.

D - Implémenter IDrawable : dessiner des rectangles et des cercles

Dans cette partie, nous allons enfin écrire des classes qui implémentent l'interface IDrawable. Nous allons écrire une classe permettant de dessiner un carré et une seconde un cercle.

Puisqu'il n'y aura que l'implémentation de la méthode draw qui diffère entre les 2 classes, nous allons écrire un classe abstraite, contenant le code commun aux 2 classes. La classe se nomme FormDrawable.

Dans la classe FormDrawable, nous plaçons les informations que chaque type de forme doit connaître : la position de la forme sur le JCanvas, sa dimension et sa couleur. La position et la dimension peuvent être placées dans un objet de type Rectangle, d'autant plus que l'on doit retourner un objet de ce type dans une des méthodes de l'interface IDrawable. La classe FormDrawable contient donc 2 variables d'instances et implémente la méthode getRectangle.

Remarquons que nous avons choisi de retourner systématiquement une copie du rectangle, ce qui a pour conséquences qu'une tierce personne ne peut pas modifier les variables de l'extérieur, sans que l'on soit prévenu. En contre partie, nous construisons un nouvel objet à chaque appel de la méthode.

La classe doit absolument être déclarée abstract puisqu'elle n'implémente pas la méthode draw. Nous ajoutons bien entendu les constructeurs permettant de créer nos formes.

FormDrawable

```
import java.awt.*;

public abstract class FormDrawable implements IDrawable {


        protected Rectangle rect ;
        protected Color color;

        public FormDrawable(Color color, Point pos, Dimension dim){
                this.color=color;
                this.rect = new Rectangle(pos,dim);
        }

        public abstract void draw(Graphics g) ;

        public Rectangle getRectangle(){
```

FormDrawable
```
            return (Rectangle) rect.clone();
        }

    }
```

Dans les classes concrètes, il ne reste plus qu'à hériter de FormDrawable et à implémenter la méthode draw, en récupérant les données nécessaires dans les variables d'instance de la superclasse, qui sont déclarées protected.

RectangleDrawable
```
    import java.awt.Color;
    import java.awt.Dimension;
    import java.awt.Graphics;
    import java.awt.Point;

    public class RectangleDrawable extends FormDrawable{

        public RectangleDrawable(Color color, Point pos, Dimension dim) {
            super(color, pos, dim);

        }

        public void draw(Graphics g) {
            Color c = g.getColor();
            g.setColor(color);
            g.fillRect(rect.x,rect.y,rect.height,rect.width);
            g.setColor(c);
        }

    }
```

Remarquons que nous faisons en sorte de remettre l'objet Graphics dans le même état qu'il se trouvait lors du début de l'appel de la méthode.

E - Exemple d'utilisation

Finalement, voici un exemple de code utilisant notre nouvelle version de JCanvas, avec création d'un rectangle et d'un cercle.

Demo2
```
    public static void main(String[] args) {
        JCanvas jc = new JCanvas();
        jc.setBackground(Color.WHITE);
        jc.setPreferredSize(new Dimension(400,200));
        Dimension dim  =new Dimension(20,20);
        IDrawable rect = new RectangleDrawable(Color.RED,new Point(10,10),dim);
        IDrawable circle = new CircleDrawable(Color.RED,new Point(50,10),dim);
        jc.addDrawable(rect);
        jc.addDrawable(circle);
        GUIHelper.showOnFrame(jc,"test JCanvas with IDrawables");
    }
```

IV - Utilisation de la classe java.awt.Rectangle

La classe java.awt.Rectangle contient des méthodes qui permettent de vérifier si un rectangle donné contient un certain point ou un autre rectangle. On peut également vérifier si 2 rectangles se recouvrent. Nous allons profiter de ces méthodes pour ajouter quelques fonctionnalités à la classe JCanvas.

Le principe d'implémentation reste toujours le même : nous créons un itérateur sur la liste de drawables, et, dans la boucle, nous exécutons le code nécessaire, en appelant les méthodes de la classe Rectangle.

Trouver tous les IDrawable qui contiennent un certain point :

```
public List findDrawables(Point p) {
	List l = new ArrayList();
	for (Iterator iter = drawables.iterator(); iter.hasNext();) {
		IDrawable element = (IDrawable) iter.next();
		if(element.getRectangle().contains(p)){
			l.add(element);
		}
	}
	return l;
}
```

Vérifier si une certaine zone du JCanavas n'est pas déjà occupée par un IDrawable:

```
public boolean isFree(Rectangle rect) {
	for (Iterator iter = drawables.iterator(); iter.hasNext();) {
		IDrawable element = (IDrawable) iter.next();
		if(element.getRectangle().intersects(rect)){
			return false;
		}
	}
	return true;
}
```

Vérifier si un Idrawable est bien le seul à se trouver dans sa zone:

```
public boolean isAlone(IDrawable drawable) {
	Rectangle rect = drawable.getRectangle();
	for (Iterator iter = drawables.iterator(); iter.hasNext();) {
		IDrawable element = (IDrawable) iter.next();
		if(!element.equals(drawable) and  element.getRectangle().intersects(rect)) {
			return false;
		}
	}
	return true;
}
```

V - Ajouter un écouteur d'événements de la souris

En java, il est possible d'attacher un écouteur d'événement sur un composant graphique. Un événement peut être, par exemple, un clique de souris ou un déplacement de celle-ci.

La classe java.awt.Component, dont héritent tous les composants graphiques, possède 5 méthodes permettant d'ajouter un écouteur d'événement:

- addComponentListener : pour écouter les changements du composant lui-même (sa taille, sa position, et).
- addFocusListener : lorsque le composant gagne ou perd le focus.
- addKeyListener: lorsque l'utilisateur appuie sur une touche du clavier alors que le composant a le focus.
- addMouseListener: écoute les clics de souris ainsi que l'entrée et la sortie de la souris de la zone de dessin du composant.
- addMouseMotionListener: écoute les changements de position du curseur au dessus du composant.

Reste à savoir ce qui doit être passé en paramètre de ces méthodes: un objet implémentant l'interface dont le nom suit le add, dans le nom de la méthode. En effet, Listener en anglais signifie écouteur, d'où addXXXListener où XXX est remplacé par le type de Listener (MouseListener pour le mouvement de la souris, par exemple).

Chacune de ces interfaces possède des méthodes qui sont appelées lorsqu'un événement la concernant se produit. L'objet passé en paramètre de ces méthodes représente l'événement, càd qu'il contient toutes les informations nécessaires (la coordonnée du clic de souris, par exemple).

Dans la suite de ce tutoriel,nous allons utiliser les 2 Listeners qui concernent la souris : java.awt.event.MouseListener pour les écouter toutes les actions sauf les mouvements) et java.awt.event.MouseMotionListener (pour les mouvements).

A - MouseListener

Nous allons écrire une classe qui implémente cette interface pour réagir aux pressions de l'utilisateur sur les boutons de la souris. Cette classe permettra, selon que l'on clique sur le bouton gauche ou droit, d'ajouter ou d'enlever des formes du panneau.

L'interface java.awt.event.MouseListener contient 5 méthodes. Puisqu'il s'agit d'une interface, toute classe qui l'implémente doit implémenter ces 5 méthodes. Afin de nous faciliter la vie, le JDK contient une classe abstraite, nommée java.awt.event.MouseAdapter, qui implémente java.awt.event.MouseListener, mais avec des méthodes qui ne font rien. Il nous suffit dès lors d'étendre cette classe et de réécrire les méthodes qui nous intéressent.

Nous allons créer notre propre écouteur d'événements de la souris, la classe JCanvasMouseListener. Cette classe s'enregistre comme java.awt.event.MouseListener auprès du JCanvas qu'on passe comme paramètre du constructeur. Elle possède 4 méthodes permettant de distinguer les mouvements sur les boutons gauche et droit de la souris. C'est de cette classe abstraite que tous nos écouteurs hériteront.

la classe JCanvasMouseListener

```
import java.awt.event.*;
import javax.swing.SwingUtilities;

public abstract class JCanvasMouseListener extends MouseAdapter {

	protected JCanvas canvas;

	public JCanvasMouseListener(JCanvas canvas) {
		super();
		this.canvas = canvas;
		canvas.addMouseListener(this);
	}

	public JCanvas getCanvas() {
		return canvas;
	}

	public void mouseClicked(MouseEvent e) {
		if (SwingUtilities.isLeftMouseButton(e)) {
			leftClickAction(e);
		} else {
			rightClickAction(e);
		}
	}


	protected void rightClickAction(MouseEvent e) {

	}

	protected void leftClickAction(MouseEvent e) {

	}

}
```

Pour ce faire, nous surchargons la méthode mouseClicked, et utilisons la méthode getModifier de l'objet de type java.awt.event.MouseEvent passé en paramètre. Pour savoir s'il s'agit du bouton gauche ou droit, nous pouvons utiliser la méthode de la classe javax.swing.SwingUtility. Les méthodes respectives sont alors appelées.

Comme vous pouvez le constater, nous reprenons la même philosophie de le classe MouseAdapter : ces méthodes ne sont pas déclarées abstraites, mais elle ne font rien. De cette façon, nous n'obligeons pas les sous-classes à implémenter des méthodes inutilement.

1 - Exemples d'utilisation

Comme exemple d'utilisation, nous allons créer une classe qui, lors d'un clic gauche de la souris, ajoute un rectangle rouge si il n'y en a pas déjà un. Lors d'un clic droit de la souris, le rectangle contenant le curseur, si il existe, est effacé.

SimpleMouseListener

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Point;
import java.awt.event.MouseEvent;
```

SimpleMouseListener

```
import java.util.List;

public class SimpleMouseListener extends JCanvasMouseListener {

	public SimpleMouseListener(JCanvas canvas) {
		super(canvas);

	}

	protected void rightClickAction(MouseEvent e) {
		List selectedDrawables = canvas.findDrawables(e.getPoint());
		if (selectedDrawables.size() == 0) return;
		IDrawable drawable = (IDrawable) selectedDrawables.get(0);
		canvas.removeDrawable(drawable);
	}

	protected void leftClickAction(MouseEvent e) {
		Point p = e.getPoint();
		IDrawable rect = createDrawable(e);
		if (canvas.isFree(rect.getRectangle())) {
			canvas.addDrawable(rect);
		}

	}

	private IDrawable createDrawable(MouseEvent e) {
		Point p = e.getPoint();
		Dimension dim = new Dimension(40, 40);
		return new RectangleDrawable(Color.RED, p, dim);

	}

}
```

La méthode leftClickAction , récupère les coordonnées de la position du curseur grâce à la méthode getLocation de la classe MouseEvent. Il ne reste plus après qu'à créer un nouveau RectangleDrawable dont la position est définie par ce point. Si le JCanvas nous dit que le rectangle du drawable ne contient pas d'autre drawable, nous pouvons alors ajouter ce drawable sur le JCanavas.

La méthode rightClickAction est implémentée de la même manière. Comme la classe JCanavas retourne une liste de drawables sensés se trouver à la position passée en paramètre, nous vérifions si la liste n'est pas vide, et ensuite prenons son premier élément. Il ne reste plus qu'à l'enlever du JCanvas.

Démonstration

```
public static void main(String[] args) {
		JCanvas jc = new JCanvas();
		jc.setBackground(Color.WHITE);
		jc.setPreferredSize(new Dimension(400, 200));
		Dimension dim = new Dimension(40, 40);
		IDrawable rect = new RectangleDrawable(Color.RED, new Point(10, 10),dim);
		IDrawable circle = new CircleDrawable(Color.BLUE, new Point(60, 30),dim);
		jc.addDrawable(rect);
		jc.addDrawable(circle);
		new SimpleMouseListener(jc);
		GUIHelper.showOnFrame(jc, "test JCanvas");
	}
```

B - MouseMotionListener

Dans cette partie, nous allons tenter de faire se déplacer des figures sur notre panneau, au moyen de la souris. Nous allons donc implémenter un MouseMotionListener, qui détecte les mouvements de souris.

Avant cela, nous allons créer un nouvelle interface, qui permet de faire se déplacer un drawable : IMovableDrawable.

1 - L'interface IMovableDrawable

L'interface IMovableDrawable, qui étends IDrawable, contient simplement des accesseurs pour un objet de type Point. Ce type correspond simplement à la position que l'on souhaite donner à une figure. Bien entendu, cette position correspond au centre de la figure.

L'interface IMovableDrawable

```
import java.awt.Point;

public interface IMovableDrawable extends IDrawable{

	void setPosition(Point p);

	Point getPosition();

}
```

Nous allons modifier la classe FormDrawable pour la faire implémenter cette interface. Remarquons qu'il ne s'agit pas de simples accesseurs puisque nous modifions les coordonnées pour représenter le centre du rectangle.

FormDrawable

```
public abstract class FormDrawable implements IMovableDrawable {


	protected Rectangle rect ;
	protected Color color;


	public FormDrawable(Color color, Point pos, Dimension dim){
		this.color=color;
		this.rect = new Rectangle(dim);
		setPosition(pos);

	}
	public abstract void draw(Graphics g) ;



	public Rectangle getRectangle(){
		return (Rectangle) rect.clone();
	}

	public Point getPosition() {
		Point p=  rect.getLocation();
		p.x = (p.x+rect.width/2);
```

FormDrawable
```
                p.y = (p.y+rect.width/2);
                return p;
        }

        public void setPosition(Point p) {
                rect.x = (p.x-rect.width/2);
                rect.y = (p.y-rect.height/2);

        }
    }
```

2 - Faire bouger les figures

Le code suivant contient le code complet de notre écouteur d'événement de souris, qui implémente à la fois java.awt.event.MouseListener et java.awt.event.MouseMotionListener. Comme les autres, cette classe est abstraite, puisqu'en elle-même, elle ne sert à rien.

la classe JCanvasMouseAdapter
```
    public class JCanvasMouseAdapter extends JCanvasMouseListener implements MouseMotionListener{

        public JCanvasMouseAdapter(JCanvas canvas) {
                super(canvas);
                canvas.addMouseMotionListener(this);

        }

        public void mouseDragged(MouseEvent e) {

        }


        public void mouseMoved(MouseEvent e) {

        }

    }
```

Il ne reste plus qu'à écrire le code qui fera bouger les figures. C'est ce que nous montrons dans la partie suivante.

3 - Exemples d'utilisation

La classe suivante, qui hérite de de JCanvasMouseAdapter, permet de récupérer le drawable lorsque qu'un bouton de la souris est pressé (méthode mousePressed). Lorsque le bouton est relâché (mouse released), la variable est mise à Null. Entre ces 2 opérations, si la souris est déplacée, la méthode mouseDragged est appelée. A chaque fois, nous donnons au drawable la position du curseur et rafraichissons l'écran. Ce qui donne l'impression que la figure suit le curseur.

la classe MoveDrawableMouseListener
```
    import java.awt.event.MouseEvent;
    import java.util.List;

    public class MoveDrawableMouseListener extends JCanvasMouseAdapter {
```

la classe MoveDrawableMouseListener

```
	protected IMovableDrawable drawable;


	public MoveDrawableMouseListener(JCanvas canvas) {
		super(canvas);
	}


	public void mouseDragged(MouseEvent e) {
		if (drawable != null) {
			drawable.setPosition(e.getPoint());
			canvas.repaint();
		}
	}


	public void mousePressed(MouseEvent e) {
		List selectedDrawables = canvas.findDrawables(e.getPoint());
		if (selectedDrawables.size() == 0)
			return;
		drawable = (IMovableDrawable) selectedDrawables.get(0);
	}

	public void mouseReleased(MouseEvent e) {
		drawable = null;
	}

}
```

Dans cette classe, nous vérifions, avant de relâcher la figure, qu'elle ne sera pas déposée à un endroit où il y a déjà une autre figure. Si c'est le cas, le drawable est remis à sa position initiale.

la classe NonOverlapMoveAdapter

```
import java.awt.Point;
import java.awt.event.MouseEvent;

public class NonOverlapMoveAdapter extends MoveDrawableMouseListener{

	private Point initialLocation;


	public NonOverlapMoveAdapter(JCanvas canvas) {
		super(canvas);

	}


	public void mouseReleased(MouseEvent e) {
		if(drawable== null) return ;
		if( !canvas.isAlone(drawable)){
			drawable.setPosition(initialLocation);

		}
		initialLocation =null;
		drawable=null;
		canvas.repaint();
	}


	public void mousePressed(MouseEvent e) {
		super.mousePressed(e);
```

la classe NonOverlapMoveAdapter

```
                if(drawable!=null) {
                        initialLocation=drawable.getPosition();
                }
        }

    }
```

VI - Conclusion et Perspectives

Dans ce tutoriel, nous vous avons montré comment développer un framework simple permettant de dessiner des figures sur un panneau, de les sélectionner et de les faire se déplacer.

Sur base de ce framework, il est possible de créer des applications beaucoup élaborées. Nous l'avons utilisé pour créer un visualisateur de base de données, où chaque table de la table était montrée, avec ses différents champs (plus de détails dans un prochain article).

VII - télécharger les sources et la version pdf de l'article

Les sources du programme peuvent être téléchargées via FTP ou HTTP

Le pdf peut être téléchargé via FTP ou HTTP