Programmation orientée objets

Leçon 7 : Interactions clavier et souris

Prof. Rolf Ingold

Département d'informatique

UNI
FR

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Contenu de la leçon

- Rappel sur la programmation par événements
- Ecouteurs d'événements multiples
- Ecouteurs et adaptateurs
- Interactions avec le clavier
- Interactions avec la souris
- Notion de focus

Rappel sur la applications interactives

- Les **applications interactives** suivent le paradigme suivant
  - l'utilisateur interagit avec les composants graphiques et déclenche des **événements**
  - ces événements sont transmis aux **écouteurs d'événements** qui décident des commandes à exécuter
  - lorsque l'exécution est terminée, les composants graphiques reçoivent une **notification**
  - ils **mettent à jour l'affichage** afin de refléter le nouvel état

Evénements

- Les composants peuvent engendrer différents types d'événements

- Dans Swing/AWT, un événement est un objet héritant de **`java.awt.AWTEvent`** elle-même sous-classe de **`java.util.EventObject`**

- Chaque événement donne accès à
  - sa **source**, c'est-à-dire au composant qui a déclenché l'événement
  - ses caractéristiques (nom, positionnement, etc.)
  - des informations contextuelles (par exemple les touches de modification SHIFT, CTRL, ...)

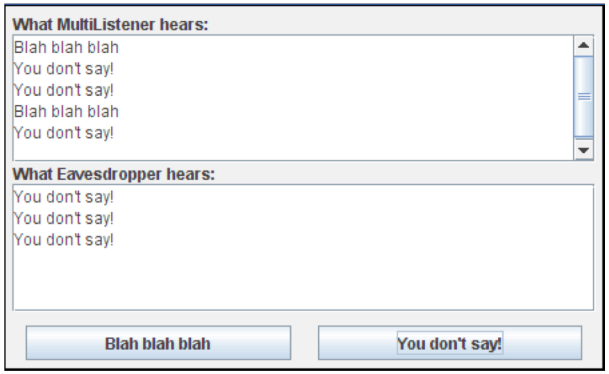
Ecouteurs d'événements

- Les **écouteurs d'événements** sont des objets qui implémentent des interfaces de `java.awt.event` ou de `javax.swing.event`
- Ils sont définis dans le programme d'application
  - dans une classe qui implémentent l'interface correspondant au type d'événement
- Ils doivent être associés aux composants qui sont susceptibles de déclencher les événements correspondants

- Plusieurs écouteurs peuvent être associés à un même événement
- Un même écouteur d'événement peut être associé à plusieurs composants
  - mais il n'a pas d'ambiguïté, car le composant concerné est une caractéristique de l'événement

Exemple

[from docs.oracle.com/javase/tutorial/]




```
public class MultiListener
        ... implements ActionListener {
    ...
    public void init() {
        ...
        button1.addActionListener(this);
        button2.addActionListener(this);
        button2.addActionListener(new Eavesdropper());
        ...
    }
    ...
    public void actionPerformed(ActionEvent e) {
        topTextArea.append(e.getActionCommand() + newline);
    }
    ...
}
public class Eavesdropper implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        bottomTextArea.append(e.getActionCommand() + newline);
    }
    ...
}
```


UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Ecouteurs et adaptateurs

- La librairie Swing/AWT spécifie en tout
  - 37 écouteurs d'événements, sous forme d'interfaces se terminant par `...Listener`
  - 31 classes d'événements se terminant par `...Event`

  voir https://docs.oracle.com/javase/tutorial/uiswing/events/api.html

- Chaque interface définit entre 1 et 8 méthodes
- Ensemble, ils permettent de répondre à quelque 90 types d'événements
- Lorsqu'un écouteur `...Listener` traite plusieurs événements, la librairie offre souvent une classe abstraite appelée **adaptateur** (se terminant par `...Adapter)` qui définit un comportement par défaut pour chacune des méthodes
  - ainsi, en définissant une sous-classe d'un adaptateur, il suffit redéfinir que les méthodes dont on réellement a besoin

Interactions avec le clavier

- Dans une application interactive Swing/AWT, le clavier est logiquement associé à un composant (celui qui a le **focus**)

- Pour traiter les événements du clavier, il faut implémenter l'interface **`KeyListener`** comprenant trois méthodes
  - **`keyPressed(KeyEvent e)`** : qui indique lorsque l'utilisateur a pressé sur une touche
  - **`keyReleased(KeyEvent e)`** : qui indique que l'utilissteur a relâché une touche
  - **`keyTyped(KeyEvent e)`** : qui est la combinaison des deux précédents et qui donne accès au caractère généré (lorsqu'il existe)
- Comme alternative, on peut créer une sous classe de **`KeyAdapter`**

Événements associés clavier

- Les événements de type **`KeyEvent`** permettent notamment d'accéder aux informations suivantes
  - **`getKeyChar()`** : le caractère Unicode correspondant
  - **`getKeyCode()`** : la touche correspondante (un code numérique)
  - **`getModifiers()`** : l'état des touches modificatrices du clavier, à utiliser avec les masques (**`SHIFT_MASK`**, **`CTRL_MASK`**, **`ALT_MASK`**, **`META_MASK`**)

Focus

- Le **focus** détermine quel composant est momentanément lié au clavier et répond à ce dernier
  - le focus est associé au plus à un composant
  - il est également lié au focus de l'application qui est unique aussi
- En principe, le focus est contrôlé par l'utilisateur
  - il est aussi possible de le contrôler par le programme, par exemple en réponse à une autre action
- Le changement de focus génère des événements
  - exemple : attribuer le focus à un champ particulier lorsqu'une fenêtre devient active (reçoit le focus)

```
...
frame.addWindowFocusListener(new WindowAdapter() {
    public void windowGainedFocus(WindowEvent e) {
        myTextField.requestFocusInWindow();
    }
});
...
```


UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Interaction avec la souris : écouteur principal

- La souris est susceptible de générer plusieurs types d'événements qui sont traités par différents écouteurs d'événements
- Le principal type d'écouteur est **`MouseListener`** ; il doit implémenter
  - **`mouseEntered(MouseEvent e)`** : lorsque le curseur arrive sur composant
  - **`mouseExisted(MouseEvent e)`** : lorsque le curseur quitte le composant
  - **`mousePressed(MouseEvent e)`** : lorsque l'utilisateur appuye sur un bouton
  - **`mouseReleased(MouseEvent e)`** : lorsque l'utilisateur relâche un bouton
  - **`mouseClicked(MouseEvent e)`** : combinaison des deux événements précédents s'il surviennent sur le même composant

Autres écouteurs liés à la souris

- Les autres écouteurs sont
  - `MouseMotionListener` : lorsque la souris se déplace sur le même composant et qui doit implémenter les méthodes
    - `mouseMoved(MouseEvent e)` : déplacement simple
    - `mouseDragged(MouseEvent e)` : déplacement avec un bouton enfoncé
  - `MouseWheelListener` : lorsque l'utilisateur actionne la roulette et qui implémente
    - `mouseWheelMoved(MouseWheelEvent e)`

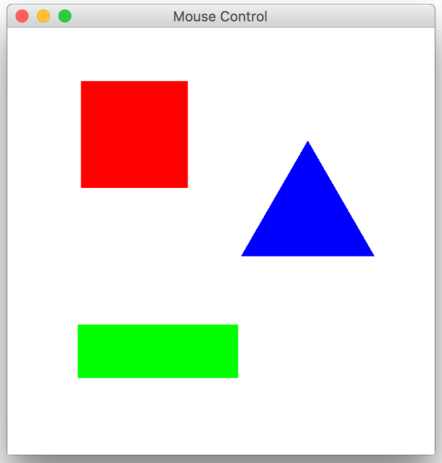
- Exemple : déplacement d'un carré à l'aide de la souris
  - avec utilisation d'une **classe anonyme**

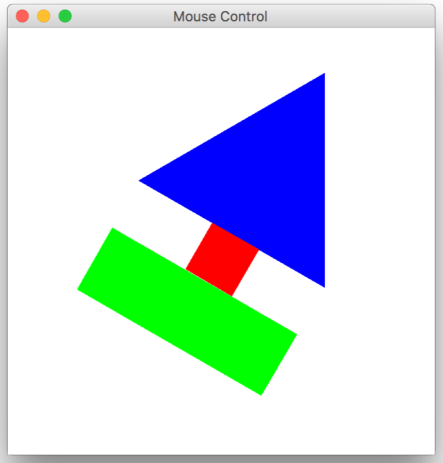
Evénements de bas niveau liés à la souris

- Les événements de type **`MouseEvent`** permettent notamment d'accéder aux informations suivantes
  - **`getSource()`** ou **`getComponent()`** : composant qui est l'origine de l'événement
  - **`getX()`**, **`getY()`** : coordonnées de la souris, relatives au composant
  - **`getButton()`** : identification du bouton (de la souris) qui a changé d'état (**`BUTTON1`**, **`BUTTON2`**, **`BUTTON3`**)
  - **`getClickCount()`** : nombre de clicks
  - **`getModifiers()`** : l'état des touches de modification du clavier, à utiliser avec les masques (**`ALT_MASK`**, **`CTRL_MASK`**, **`META_MASK`**, **`SHIFT_MASK`**)
  - **`isShiftDown()`**, **`isControlDown()`**, ... fonctions qui retournent l'état de chaque touche de modification

Exemple : édition d'objets graphiques

- Le paquetage **`oop.demo.swing.mouse`** contient une application interactive utilisant la souris pour
  - déplacer les objets graphiques
  - les faire pivoter (autour de leur centre)
  - les redimensionner (par rapport à leur centre)





Conception de l'interface

- Toute les opérations s'appliquent à l'**objet sélectionné**
  - un objet est sélectionné lorsque l'utilisateur clique dessus (bouton de gauche ou de droite)
    - il reste sélectionné jusqu'au prochain clic de souris
  - l'objet sélectionné est déplacé avec la souris en maintenant le bouton gauche pressé
  - il est pivoté (rotation) avec la souris en maintenant le bouton droit pressé
  - il est redimensionné en appuyant sur la touche SHIFT (majuscule) et en en maintenant le bouton droit pressé
  - il est également redimensionné en utilisant la roulette

Architecture du programme

- Le programme suit une découpe Modèle-Vue-Contrôleur (MVC)
  - la classe **`Model`** contient les objets graphiques
  - la classe **`View`** gère l'affichage
  - la classe **`Controller`** gère l'interaction

- Quatre intéracteurs ont été définis
  - trois d'entre eux sont contrôlés par le déplacement de la souris ; ils héritent de la classe abstraite **`Transformer`**
    - la classe **`Translator`** effectue les translations
    - la classe **`Rotator`** effectue les rotations
    - la classe **`Rescaler`** effectue les redimensionnements
  - le quatrième intéracteur utilise la roulette
    - la classe **`Resizer`** effectue de redimensionnement par sauts de 10%

Le modèle

- Dans cet exemple le modèle contient exactement 3 objets graphiques stockés dans un tableau fixe
  - un carré rouge
  - un triangle bleu
  - un rectangle vert

- Comme les objets peuvent se chevaucher et se superposer, il faut pouvoir les réarranger, i.e. modifier l'ordre
- Dès que l'utilisateur clique sur une partie visible d'un objet, il passe en avant-plan et devient l'objet sélectionné
- Les transformations s'effectuent toujours sur l'objet à l'avant-plan

Contenu de la classe `Model`

- La classe Model contient les primitives suivantes
  - l'implémentation n'est pas précisée

```
public class Model extends Observable {
    ...
    public Model() { ... }
    public Shape[] getShapes() { ... }
    public Shape shapeAt(Point point) { ... }
    public void putAtFront(Shape shape) { ... }
    public Shape frontShape() { ... }
    public void translateFrontShape(Vector vector) { ... }
    public void rotateFrontShape(double angle) { ... }
    public void scaleFrontShape(double factor) { ... }
}
```

La vue

- La vue est responsable du système de coordonnées du modèle
  - l'origine est au centre,
  - l'axe des y est orientée vers le haut
  - l'unité est le pixel

- La vue doit être capable de convertir la position de la souris (i.e. du curseur dans la fenêtre) en coordonnées du modèle

Classe View

```
public class View extends Display implements Observer {
    public static final int WIDTH = 400;
    public static final int HEIGHT = 400;

    private Model model;

    public View(Model model) {
        super(WIDTH, HEIGHT);
        this.model = model;
        model.addObserver(this);
        setOrigin(WIDTH / 2, HEIGHT/2);
        setScale(1, true);
    }

    public Point pointAt(double x, double y) {
        return new Point(x – WIDTH / 2, - (y – HEIGHT / 2));
    }

    @Override
    public void paint(Display display) {
        for (Shape shp : model.getShapes()) { shp.paint(display); }
    }

    @Override
    public void update() { repaint(); }
}
```


UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Le contrôleur

- Le contrôleur contient les gestionnaires d'événements
- Rappel : Java offre trois types d'écouteurs de la souris
  - **`MouseListener`** : pour réagir aux boutons de la souris
  - **`MouseMotionListener`** : pour réagir aux déplacements de la souris
  - **`MouseWheelListener`** : pour réagit à la roulette de la souris
- La classe abstraite **`MouseAdapter`** implémente des comportements par défaut pour les différents écouteurs

Classe Controller

```
public class Controller {

    private Model model;
    private View view;
    private MouseAdapter listener = new LocalMouseListener();

    public Controller(Model model, View view) {
        this.model = model;
        this.view = view;
        view.addMouseListener(listener);
        view.addMouseMotionListener(listener);
        view.addMouseWheelListener(listener);
    }

    private class LocalMouseListener extends MouseAdapter {

        ...
    }
}
```

Classe Controller.LocalMouseListener (1/2)

```
private class LocalMouseListener extends MouseAdapter {
    private Transformer transformer;
    private Resizer resizer;
    @Override
    public void mousePressed(MouseEvent event) {
        resizer = null;
        Point point = view.pointAt(event.getX(), event.getY());
        Shape selected = model.shapeAt(point);
        if (selected != null) {
            model.putAtFront(selected);
            if (event.getButton() == MouseEvent.BUTTON1) {
                transformer = new Translator(model, point);
            }
            if (event.getButton() == MouseEvent.BUTTON3) {
                if (!event.isShiftDown()) {
                    transformer = new Rotator(model, point);
                } else {
                    transformer = new Rescaler(model, point);
                }
            }
            resizer = new Resizer(model);
        }
    }
    ...
}
```

Classe Controller.LocalMouseListener (2/2)

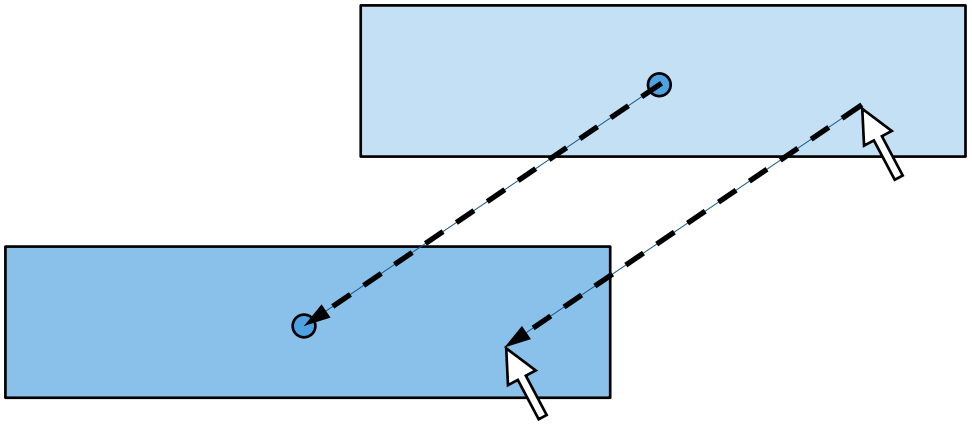
```
private class LocalMouseListener extends MouseAdapter {
    ...
    @Override
    public void mouseDragged(MouseEvent event) {
        if (transformer != null) {
            Point point = view.pointAt(event.getX(), event.getY());
            transformer.controlMoved(point);
        }
    }
    @Override
    public void mouseReleased(MouseEvent event) {
        transformer = null;
    }
    @Override
    public void mouseWheelMoved(MouseWheelEvent event) {
        if (resizer != null) {
            resizer.wheelRotated(event.getWheelRotation());
        }
    }
}
```

Classe abstraite `Transformer`

```
public abstract class Transformer {

    protected Model model;

    protected Transformer(Model model) {
        this.model = model;
    }

    public abstract void controlMoved(Point point);

}
```

Contrôle de la translation

- L'utilisateur contrôle la translation avec le bouton de gauche
- Le vecteur de translation correspond à celui du déplacement de la souris

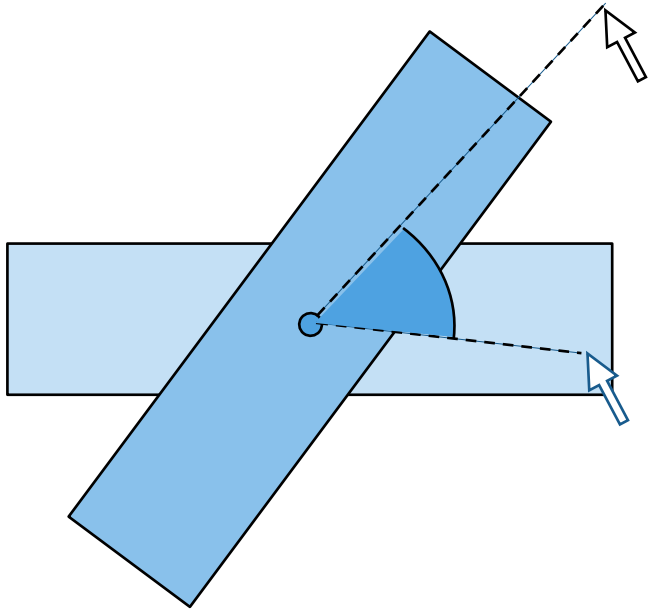


Classe Translator

```
public class Translator extends Transformer {

    private Point prevPoint;

    public Translator(Model model, Point point) {
        super(model);
        prevPoint = point;
    }

    @Override
    public void controlMoved(Point point) {
        double dx = point.x() - prevPoint.x();
        double dy = point.y() - prevPoint.y();
        Vector vector = new Vector(dx, dy);
        model.translateFrontShape(vector);
        prevPoint = point;
    }

}
```

Contrôle de la rotation

- L'utilisateur contrôle la rotation avec le bouton de droite
- L'angle de rotation est défini par l'angle parcouru par le déplacement de la souris par rapport au centre de rotation

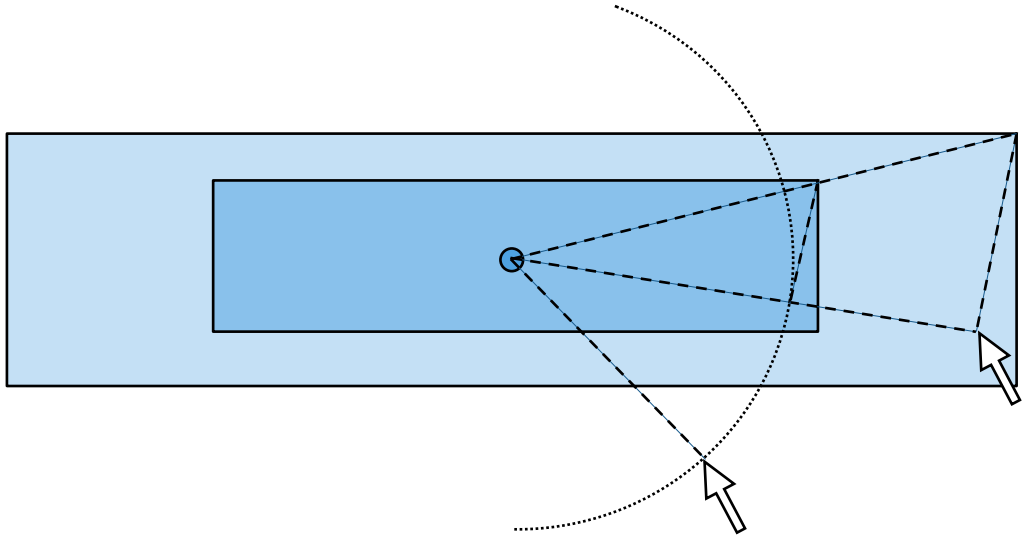


Classe Rotator

```
public class Rotator extends Transformer {

    private double prevAngle;

    public Rotator(Model model, Point point) {
        super(model);
        prevAngle = angle(point);
    }

    @Override
    public void controlMoved(Point point) {
        double delta = Math.round((angle(point) - prevAngle));
        model.rotateFrontShape(delta);
        prevAngle += delta;
    }

    private double angle(Point point) {
        double dx = point.x() - model.frontShape().getAnchor().x();
        double dy = point.y() - model.frontShape().getAnchor().y();
        return Degrees.atan2(dy, dx);
    }

}
```

Contrôle du redimensionnement

- L'utilisateur contrôle la rotation avec le bouton de droite combinée avec la touche SHIFT (majuscule)
- La dimension est déterminer par le rapport des distances entre la souris et le centre de l'objet



Classe Rescaler

```
public class Rescaler extends Transformer {

    private double prevDistance;

    public Rescaler(Model model, Point point) {
        super(model);
        prevDistance = distanceToAnchor(point);
    }

    @Override
    public void controlMoved(Point point) {
        double distance = distanceToAnchor(point);
        model.scaleFrontShape(distance / prevDistance);
        prevDistance = distance;
    }

    private double distanceToAnchor(Point point) {
        return model.frontShape().getAnchor().distance(point);
    }

}
```

Classe `Resizer`

```
public class Resizer {

    protected Model model;

    public Resizer(Model model) {
        this.model = model;
    }

    public void wheelRotated(int nbSteps) {
        model.scaleFrontShape(Math.pow(1.1, nbSteps));
    }

}
```