Cours Programmation Orientée Objet en Java

Cours 5

Anne Parrain

UFR des Sciences - Université d'Artois
Licence Informatique
Semestre 4

Année universitaire 2018–2019

Plan du cours

1. Les concepts de la programmation d'une interface graphique
   - Les composants graphiques
   - Les gestionnaires de positionnement
   - La gestion des événements

Section 1

Les concepts de la programmation d'une interface graphique

Les différentes API

- `java.awt`
- `javax.swing`
- `java.awt.event`
- `javax.swing.event`
- ...

Elles se trouvent dans le module `java.desktop`

Les composants graphiques

Une interface graphique est constituée de composants :

- des composants simples : *button*, *radiobutton*, *checkbox*, listes déroulantes. . .
- des conteneurs ; *panel*, *frame*, *applet*, *window*. . .

Les composants graphiques élémentaires

Ils héritent tous, directement ou indirectement, de la classe `java.awt.Component`.

- `javax.swing.JButton`
- `javax.swing.JLabel`
- `javax.swing.JTextField`, `javax.swing.JTextArea`
- `java.awt.Canvas`

Les composants graphiques élémentaires

- javax.swing.JCheckBox

```
JCheckBox entree =
    new JCheckBox("Entree");
JCheckBox plat =
    new JCheckBox("Plat principal");
JCheckBox fromage =
    new JCheckBox("Fromage");
JCheckBox dessert =
    new JCheckBox("Dessert");
JCheckBox cafe = new JCheckBox("Cafe");
plat.setSelected(true);
dessert.setSelected(true);
```

Les composants graphiques élémentaires

- javax.swing.JRadioButton

```
JLabel etiquette = JLabel("Civilite :");
JRadioButton mme = new JRadioButton("Mme.");
JRadioButton mr = new JRadioButton("M.");
JRadioButton melle =
    new JRadioButton("Melle.");
ButtonGroup btg = new ButtonGroup();
btg.add(mme);
btg.add(mr);
btg.add(melle);
```

Les composants graphiques élémentaires

- javax.swing.JList

```
String[] data ={"one","two","three","four"};
JList dataList = new JList(data);

// par defaut:
// dataList.setSelectionMode(
//   ListSelectionModel.
          MULTIPLE_INTERVAL_SELECTION);
// on change
dataList.setSelectionMode(
  ListSelectionModel.SINGLE_SELECTION);

dataList.setSelectedIndex(1);//select "two"
dataList.getSelectedValue();//returns "two"
```

Les composants graphiques élémentaires

et d'autres . . .

- `javax.swing.JMenu`
- `javax.swing.JToolTip`
- . . .

Les conteneurs

Une application graphique se déroule dans une instance de `JFrame`.

Sur cette instance sont posés des composants graphiques

- conteneurs
- ou composants graphiques élémentaires

Les conteneurs

Ils héritent tous de la classe `Container` qui hérite de la classe `Component`.

La classe Container contient notamment la méthode `add(Component c)` pour poser des composants sur le conteneur.

Petit tour d'horizon des différents conteneurs :

- `JFrame`
- `JWindow`
- `JInternalFrame`
- `JApplet`
- `JPanel`
- `JTabbedPane`
- `JDialog`
- `JOptionPane`

Les gestionnaires de positionnement

Idée : On délègue à un composant la responsabilité du placement des objets graphiques sur le conteneur.

Avantages :

- on ne calcule pas de position ;
- adaptation en fonction de la taille de la fenêtre principale ;
- plus flexible
- code beaucoup plus facile à maintenir

Les gestionnaires de positionnement

Les gestionnaires de positionnement obéissent à une politique.

- affichage autour d'un espace central : `BorderLayout`
- affichage en ligne horizontale : `FlowLayout`
- affichage vertical ou horizontal : `BoxLayout`
- affichage en tableau : `GridLayout`
- affichage paramétrable dans une grille : `GridBagLayout`

Tous ces gestionnaires sont définis dans `java.awt` (sauf `BoxLayout`).

Les gestionnaires de positionnement

Chaque conteneur a un gestionnaire de positionnement par défaut :

- le panneau central d'un `JFrame` est géré par un `BorderLayout`
- un `Jpanel` est géré par un `FlowLayout`
- ...

On peut associer un autre gestionnaire de positionnement :

- via le constructeur du conteneur ;
- via la méthode `setLayout(LayoutManager l)` de la classe `Container`.

Créer une interface graphique

1. dessiner rapidement l'interface souhaitée ;
2. la découper en unité logique de placement : les unités peuvent être côte à côte ou emboitées
3. chaque unité deviendra un conteneur géré par le layout manager qui conviendra ;
4. créer les conteneurs les plus internes ;
5. poser les composants élémentaires sur ces composants ;
6. puis emboiter en remontant vers la fenêtre principale.

La gestion des événements

Idée :

1. les composants graphiques écoutent les événements qui se passent sur eux ;
2. ils préviennent le composant à qui est déléguée la gestion de la réaction.

Avantages :

- un même gestionnaire pour plusieurs composants graphiques
- découplage interface graphique / modélisation

La gestion des événements

À chaque type d'événement est associée une interface.

Elle définit les messages qu'elle peut recevoir pour être informée des événements.

On prévient le composant graphique que pour tel type d'événement, il devra prévenir tel objet.

```
monBtn.addActionListener(new GereBouton());
```

`GereBouton` est une classe qui doit implémenter l'interface `ActionListener` qui contient une seule méthode

```
    public void actionPerformed(ActionEvent evt);
```

Les événements

Tous les événements héritent de la classe `java.util.EventObject`

Field Summary

Fields

Modifier and Type	Field	Description
`protected Object`	`source`	The object on which the Event initially occurred.

Constructor Summary

Constructors

Constructor	Description
`EventObject(Object source)`	Constructs a prototypical Event.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`Object`	`getSource()`	The object on which the Event initially occurred.
`String`	`toString()`	Returns a String representation of this EventObject.

Qui créé les événements ?

- Il existe un thread de la JVM chargé de récupérer les interactions utilisateurs
- Si on clique sur un bouton, ce thread crée un événement (du bon type) en utilisant le bouton comme source pour le constructeur
- Conséquences : généralement on ne crée pas les événements, on ne fait que les gérer.

Les différents événements

- Les événements peuvent être distingués selon 2 types :
  - bas niveau : événements fenêtre/système et interactions bas niveau (un clique de souris sur un composant, la frappe d'une touche du clavier)
  - sémantiques : Tout le reste (un bouton a été pressé)
- Il faut le plus possible traiter les événements sémantiques :
  - ils contiennent une information de plus haut niveau.
  - ils centralisent les événements de bas niveau : qu'on presse un bouton avec le clavier ou la souris, le résultat est le même pour l'application.

Qu'est-ce qu'un écouteur ?

- Un écouteur est un objet destiné à recevoir et à gérer les événements générés par le système
- Les écouteurs principaux se trouvent eux aussi dans le package java.awt.event
- La plupart du temps, il s'agit seulement d'une interface java :
  - n'importe quel objet peut devenir un écouteur du moment qu'il implémente les méthodes définies dans l'interface.
- Exemple :
  `java.awt.event.ActionListener`

Les différents écouteurs d'événement

- `ComponentListener`
- `FocusListener`
- `KeyListener`
- `MouseListener`
- `MouseMotionListener`
- `MouseWheelListener`

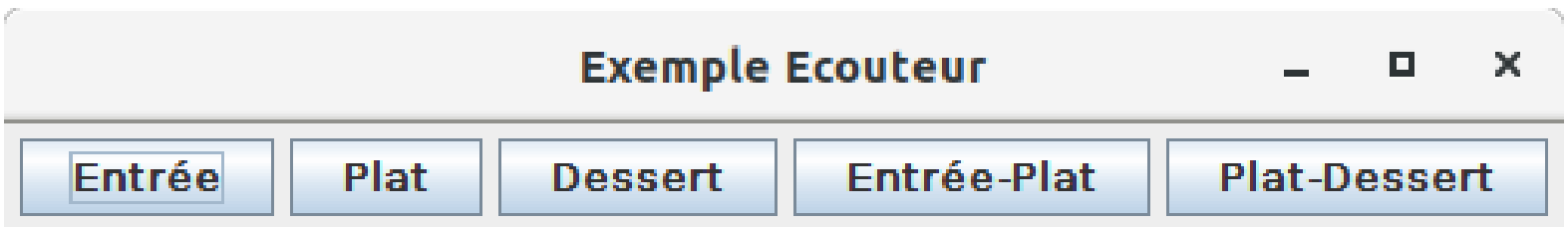
Ces écouteurs sont supportés par tous les composants.

Les différents écouteurs d'événement

- `ActionListener`
- `ChangeListener` : Button, RadioButton, CheckBox, ComboBox,...
- `ItemListener`
- `CaretListener` : TextArea, TextField, ...
- `ListSelectionListener`
- `WindowListener`

Il n'y a pas d'exclusivité

- Un composant graphique peut prévenir plusieurs écouteurs d'un même événement.
- Un écouteur peut gérer les événements venant de plusieurs composants.



Exemple écouteur

```
public class ExpleEcouteur {
    ExpleEcouteur(){
        JFrame fen = new JFrame();
        fen.getContentPane().setLayout(new
            FlowLayout(FlowLayout.LEFT));
        fen.setTitle("Exemple Ecouteur");
        JButton btnE = new JButton("Entree");
        JButton btnP = new JButton("Plat");
        JButton btnD = new JButton("Dessert");
        JButton btnEP = new JButton("Entree-Plat");
        JButton btnPD = new JButton("Plat-Dessert");
        JButton btnEPD = new JButton("Entree-Plat-Dessert");
        fen.getContentPane().add(btnE);
        fen.getContentPane().add(btnP);
        // ... ajout des autres boutons
```

Exemple écouteur (2)

```
        ActionListener ge = new GereEntree();
        ActionListener gp = new GerePlat();
        ActionListener gd = new GereDessert();
        btnE.addActionListener(ge);
        btnP.addActionListener(gp);
        btnD.addActionListener(gd);
        btnEP.addActionListener(ge);
        btnEP.addActionListener(gp);
        btnPD.addActionListener(gp);
        btnPD.addActionListener(gd);
        btnEPD.addActionListener(ge);
        btnEPD.addActionListener(gp);
        btnEPD.addActionListener(gd);
        fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fen.pack();
        fen.setVisible(true);
    }
```

Exemple écouteur (3)

```
class GereDessert implements ActionListener {
    public void actionPerformed(ActionEvent evt){
        System.out.println("Dessert");
    }
}
class GerePlat implements ActionListener {
    public void actionPerformed(ActionEvent evt){
        System.out.println("Plat");
    }
}
class GereEntree implements ActionListener {
    public void actionPerformed(ActionEvent evt){
        System.out.println("Entree");
    }
}
```

Remarques

- Un objet peut être son propre écouteur !
- Car un écouteur est une interface
- Donc, il n'y a aucune raison qu'il ne puisse pas implémenter les méthodes nécessaires

Exemple 1

```
...
JButton btn = new JButton("OK");
btn.addActionListener(new GereBouton());
...
public class GereBouton
      implements ActionListener{
  public void actionPerformed(ActionEvent e){
    System.out.println("Coucou");
  }
}
```

Exemple 2

```
public class UnExemple extends JFrame{
  public UnExemple(String titre){
    super(titre);
    Container cont = getContentPane();
    cont.setLayout(new FlowLayout());
    JButton jaune = new JButton("jaune");
    JButton rouge = new JButton("rouge");
    cont.add(jaune);
    cont.add(rouge);
    Gere2Boutons gb = new
        Gere2Boutons(this,jaune.getText(),rouge.getText());
    jaune.addActionListener(gb);
    rouge.addActionListener(gb);
    setVisible(true);
  }
  public void changeCouleur(Color c){
    getContentPane().setBackground(c);
  }
}
```

Exemple 2

```
public class Gere2Boutons
   implements ActionListener{
  private UnExemple fenetre;
  private String jaune;
  private String rouge;
  public Gere2Boutons(UnExemple fenetre,
         String jaune, String rouge){
  ...
  }
  public void actionPerformed(ActionEvent e){
    String texte = ((JButton) e.getSource()).getText();
    if (texte.equals(jaune))
      fenetre.changeCouleur(Color.yellow);
    else if (texte.equals(rouge))
      fenetre.changeCouleur(Color.red);
    else
      System.out.println("y a un probleme!");
  }
}
```

Exemple 2

```
public class Test{
  public static void main(String[] args){
    new UnExemple("Et c'est parti!");
  }
}
```

java.awt.event.MouseListener

```
public interface MouseListener
extends EventListener
```

The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component. (To track mouse moves and mouse drags, use the `MouseMotionListener`.)

The class that is interested in processing a mouse event either implements this interface (and all the methods it contains) or extends the abstract `MouseAdapter` class (overriding only the methods of interest).

The listener object created from that class is then registered with a component using the component's `addMouseListener` method. A mouse event is generated when the mouse is pressed, released clicked (pressed and released). A mouse event is also generated when the mouse cursor enters or leaves a component. When a mouse event occurs, the relevant method in the listener object is invoked, and the `MouseEvent` is passed to it.

java.awt.event.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.

java.awt.event.MouseEvent

```
public class MouseEvent
extends InputEvent
```

An event which indicates that a mouse action occurred in a component. A mouse action is considered to occur in a particular component if and only if the mouse cursor is over the unobscured part of the component's bounds when the action happens. Component bounds can be obscurred by the visible component's children or by a menu or by a top-level window. This event is used both for mouse events (click, enter, exit) and mouse motion events (moves and drags).

This low-level event is generated by a component object for:

- Mouse Events
  - a mouse button is pressed
  - a mouse button is released
  - a mouse button is clicked (pressed and released)
  - the mouse cursor enters the unobscured part of component's geometry
  - the mouse cursor exits the unobscured part of component's geometry
- Mouse Motion Events
  - the mouse is moved
  - the mouse is dragged

java.awt.event.MouseEvent

Method Summary	
int	**getButton**() Returns which, if any, of the mouse buttons has changed state.
int	**getClickCount**() Returns the number of mouse clicks associated with this event.
static String	**getMouseModifiersText**(int modifiers) Returns a String describing the modifier keys and mouse buttons that were down during the event, such as "Shift", or "Ctrl+Shift".
Point	**getPoint**() Returns the x,y position of the event relative to the source component.
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.
boolean	**isPopupTrigger**() Returns whether or not this mouse event is the popup menu trigger event for the platform.
String	**paramString**() Returns a parameter string identifying this event.
void	**translatePoint**(int x, int y) Translates the event's coordinates to a new position by adding specified x (horizontal) and y (vertical) offsets.

java.awt.event.MouseListener

```
public class ExpleMouseListener extends JFrame {

    ExpleMouseListener(){
        super();
        setTitle("Exemple Mouse Listener");
        setBounds(50,50,400,200);
        getContentPane().addMouseListener(new
            MonEcouteurSouris());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args){
        new ExpleMouseListener();
    }
}
```

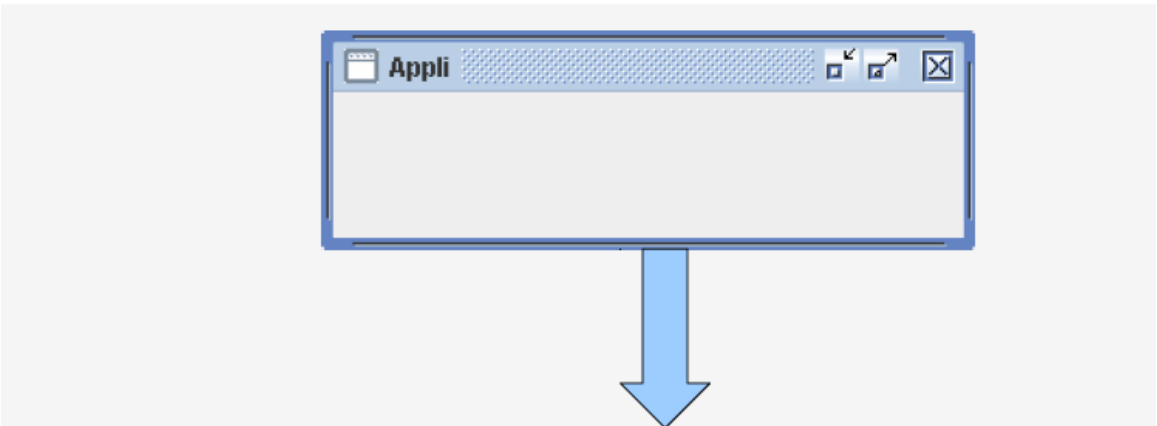
java.awt.event.MouseListener

```
class MonEcouteurSouris implements MouseListener {

    public void mouseClicked(MouseEvent evt){
        System.out.println("Position du click : x =
            "+evt.getX()+" - y = "+evt.getY());
      }

    public void mouseEntered(MouseEvent evt){}
    public void mousePressed(MouseEvent evt){}
    public void mouseReleased(MouseEvent evt){}
    public void mouseExited(MouseEvent evt){}
}
```

java.awt.event.MouseListener



Autre exemple

```
class MonEcouteurSouris implements MouseListener {

    public void mouseClicked(MouseEvent evt){
        System.out.println("Position du click : x =
            "+evt.getX()+" - y = "+evt.getY());
    }

    public void mouseEntered(MouseEvent evt){
        System.out.println("Entree dans la zone de l'ecouteur :
            x = "+evt.getX()+" - y = "+evt.getY());
    }

    public void mousePressed(MouseEvent evt){}
    public void mouseReleased(MouseEvent evt){}

    public void mouseExited(MouseEvent evt){
        System.out.println("Sortie de la zone de l'ecouteur : x
            = "+evt.getX()+" - y = "+evt.getY());
    }
}
```

Listeners concrets dans le JDK

Coder un **java.awt.MouseListener** peut être fastidieux car plusieurs des méthodes de l'interface sont souvent vides :

```
class MonEcouteurSouris implements MouseListener {
    public void mouseClicked(MouseEvent evt){
        System.out.println("Position du click : x =
            "+evt.getX()+" - y = "+evt.getY());
      }

    public void mouseEntered(MouseEvent evt){}
    public void mousePressed(MouseEvent evt){}
    public void mouseReleased(MouseEvent evt){}
    public void mouseExited(MouseEvent evt){}
}
```

java.awt.event.MouseAdapter

```
public abstract class MouseAdapter
extends Object
implements MouseListener, MouseWheelListener, MouseMotionListener
```

An abstract adapter class for receiving mouse events. The methods in this class are empty.

Mouse events let you track when a mouse is pressed, released, clicked, moved, dragged, wi

Extend this class to create a `MouseEvent` (including drag and motion events) or/and `MouseW` implement the `MouseListener`, `MouseMotionListener` interface, you have to define all of th only have to define methods for events you care about.)

Package java.awt.event

```
class MonEcouteurSouris extends MouseAdapter {

    public void mouseClicked(MouseEvent evt){
        System.out.println("Position du click : x =
            "+evt.getX()+" - y = "+evt.getY());
      }
}
```

Les adaptateurs de java.awt.event

Class Summary	
ActionEvent	A semantic event which indicates that a component-defined action occurred.
AdjustmentEvent	The adjustment event emitted by Adjustable objects.
AWTEventListenerProxy	A class which extends the `EventListenerProxy`, specifically for adding an `AWTEventListener` for
ComponentAdapter	An abstract adapter class for receiving component events.
ComponentEvent	A low-level event which indicates that a component moved, changed size, or changed
ContainerAdapter	An abstract adapter class for receiving container events.
ContainerEvent	A low-level event which indicates that a container's contents changed because a comp
FocusAdapter	An abstract adapter class for receiving keyboard focus events.
FocusEvent	A low-level event which indicates that a Component has gained or lost the input focus.
HierarchyBoundsAdapter	An abstract adapter class for receiving ancestor moved and resized events.
HierarchyEvent	An event which indicates a change to the `Component` hierarchy to which a `Component` belong
InputEvent	The root event class for all component-level input events.
InputMethodEvent	Input method events contain information about text that is being composed using an in
InvocationEvent	An event which executes the `run()` method on a `Runnable` when dispatched by the AWT ev
ItemEvent	A semantic event which indicates that an item was selected or deselected.
KeyAdapter	An abstract adapter class for receiving keyboard events.
KeyEvent	An event which indicates that a keystroke occurred in a component.
MouseAdapter	An abstract adapter class for receiving mouse events.
MouseEvent	An event which indicates that a mouse action occurred in a component.
MouseMotionAdapter	An abstract adapter class for receiving mouse motion events.
MouseWheelEvent	An event which indicates that the mouse wheel was rotated in a component.
PaintEvent	The component-level paint event.
TextEvent	A semantic event which indicates that an object's text changed.
WindowAdapter	An abstract adapter class for receiving window events.
WindowEvent	A low-level event that indicates that a window has changed its status.