Chapitre 5 : Les événements

- Gestion des événements
- Evénements de bas niveau: interfaces écouteurs et classes Adapter
- Evénements sémantiques
- Architecture MVC : événements sur des tables ou des listes
- Actions globales sur des menus et barre d'outils

Événements

Programmation par événements

> Les programmes qui possèdent une IHM avec clavier, écran graphique et souris suivent un modèle de programmation appelé programmation par événements.

> Le programme attend que se produisent des événements, (par exemple, sur le clavier, la souris ou l'écran), et, lorsque ces événements se produisent, il y réagit en déclenchant les procédures appropriées dans les objets qui sont intéressés.

Programmation par événements

> Un programme Swing, ou une applet possèdent une interface graphique et sont donc de ce type.

> Mécanisme : si on clique la souris, c'est le système qui découvre l'événement en premier. Il détermine quelle application contrôle la souris au moment où s'est produit l'événement, et il passe les informations à cette application (si elle a déclaré être intéressée par la souris bien sûr; sinon, l'événement est ignoré.)

Les événements en Java

- Supposons que l'on clique sur un bouton de la souris. Cela génère un événement intercepté par le système et redirigé vers l'application concernée, ici écrite en java.
- Java définit des événements de bas niveau fournissant une interface au système de fenêtrage de la plateforme concernée. Il définit aussi des événements plus sémantiques, relatifs aux objets awt et swing.
- Un thread dédié gère les événements java.

Les événements en Java

- Le modèle de programmation de java définit les événements comme ayant une source et une cible.
- Dans notre exemple, le bouton est la **source** de l'événement, i.e. l'objet où le clic s'est produit. Java crée alors un objet **`Event`** qui contient les informations associées à cet événement de clic.
- Le thread de gestion des événements appelle alors la (ou les) procédure(s) associée(s) à l'événement. Ces procédures sont définies par le programmeur.

Les événements en Java

- Les procédures déclenchées sont définies dans les objets **cibles** - et elles ont cet événement en argument.
- si une application est intéressée par une gestion globale, son cadre peut être la cible de tous les événements. Le cadre invoque alors `enableEvents()` pour sélectionner les types d'événements qui l'intéresse, et il implémente les procédures de gestion associées pour chaque type d'événement sélectionné.

Gestion globale (java 1.0): méthode enableEvents()

```
public class toto extends JFrame {
        public toto() {
           enableEvents(AWTEvent.WINDOW_EVENT_MASK);
           protected void processWindowEvent(WindowEvent e) {
                if (e.getID() == WindowEvent.WINDOW_CLOSING){
                                dispose(); // libère les ressources
                                System.exit(0);
                }
                super.processWindowEvent( e ); // passe l'ev à la fenêtre
           }
        }
}
```

Gestion locale (java 1.1): les écouteurs

Mais un objet intéressé par un événement survenant sur un objet source, peut aussi se déclarer écouteur (`listener`) de cet événement sur l'objet. Un `écouteur` est un objet qui capte des événements se produisant sur un autre objet (ou d'un groupe d'objets, par ex. un groupe d'éléments de menu). C'est une cible locale.

> un `écouteur` définit une méthode prenant en argument un événement, et cette méthode sera déclenchée chaque fois que l'événement « écouté » se produira sur l'objet source.

Modèle par délégation

> Cette gestion des événements par des écouteurs suit un modèle appelé

modèle par délégation

La gestion des événements est en effet ici *déléguée* aux objets qui se sont déclarés intéressés par certains événements (et ce sont eux qui en réalisent la gestion).

Rappel: interfaces vs classes

> Les interfaces déclarent des méthodes qui sont destinées à être implémentées dans des classes.

> Une interface est une classe abstraite qui déclare des méthodes (implicitement `public`) sans leur fournir de code, et d'éventuelles constantes (membres `final static`).

Rappel: interfaces vs classes

☺ Les interfaces permettent de compenser partiellement l'absence d'héritage multiple dans Java.

☺ Une classe peut en effet étendre une autre classe et implémenter autant d'interfaces qu'elle le souhaite.

Comment définir un écouteur?

> Tout objet, toute classe peut devenir un écouteur: il suffit que la classe implémente une interface dite écouteur (**`listener`**).

> Il y a plusieurs interfaces `listener` qui distinguent les types d'événements sélectionnés.

Exemple: dans le cas d'un clic de bouton, on a un événement d'action **`ActionEvent`** et on utilise l'interface **`ActionListener.`**

Comment définir un écouteur?

> Le code qui sera invoqué par l'objet cible lors de la survenu de l'événement (sur l'objet source) est une méthode qui doit être définie dans l'objet cible qui implémente l'interface écouteur.

> Ex : dans le cas de l'interface **`ActionListener`** c'est la méthode **`actionPerformed(ActionEvent e)`** (littéralement action effectuée) qui est invoquée quand l'événement d'action (ici de type `ActionEvent`) se produit.

Relier la source à la cible

☞Dans ce modèle, il ne suffit pas d'implémenter l'interface `listener` pour déclencher les procédures sur les événements.

> Il faut en plus relier la source de l'événement à la cible. Cela se fait en enregistrant l'écouteur dans la source :

> Dans notre exemple, l'écouteur doit être enregistré sur le bouton avec la méthode **`addActionListener`** (cette méthode prend un objet qui implémente l'interface `listener` en argument).

Description des événements Java

Evénements java

- Les événements Java sont classés en deux catégories :
  1. **les événements de bas niveau** (clavier, souris, opérations sur les fenêtres)
  2. **les événements sémantiques** (mouvements des scrollbars, clic sur un objet bouton): il s'agit d'événements spécifiques, liés à des composants swing ou awt spécifiques.

Evénements java

- La plupart des événements et des interfaces écouteurs sont dans

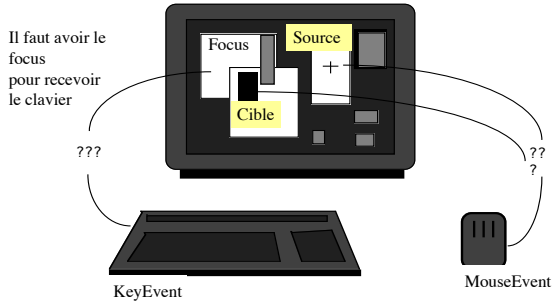
`java.awt.event`

- Les événements spécifiques aux composants Swing sont dans

`javax.swing.event`

1. Evénements de bas niveau

Problématique des Entrées



événements de bas niveau

(définis dans `java.awt.event`)

- > **`FocusEvent:`** activation ou désactivation du focus du clavier sur un composant.
- > **`MouseEvent:`** mouvements et clics de souris, et entrer/sortir d'un composant.
- > **`KeyEvent:`** événements clavier.
- > **`WindowEvent:`** dés/activation, ouverture fermeture, dés/iconification de fenêtres.
- > **`ComponentEvent :`** changement de taille, position, ou visibilité d'un composant.

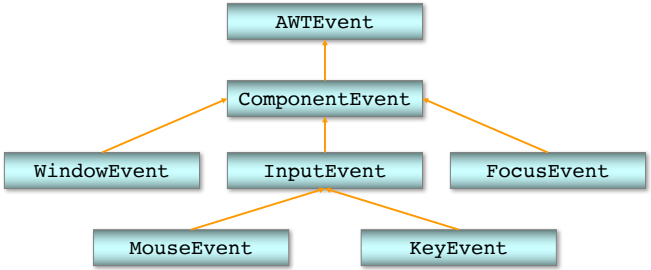
événements de bas niveau

La liste précédente n'est pas exhaustive. Il existe d'autres événements de bas niveau, comme

- > **`PaintEvent:`** qui concerne la gestion du réaffichage (A ne pas utiliser avec des écouteurs -> réécrire la méthode paint ou paintComponent).
- > **`ContainerEvent:`** événements associés à un conteneur (ajouts, suppressions, etc.).

Mais vous n'avez normalement pas besoin de gérer ces événements car ils sont traités automatiquement.

Hiérarchie des événements de bas niveau



AWTEvent

La classe `AWTEvent` dérive de `java.util.EventObject`

> `EventObjet` contient la méthode `getSource()` qui renvoie l'objet source de l'événement (l'objet où l'événement s'est *logiquement* produit).

> `EventObject` implémente l'interface `Serializable`, donc tous ces événements sont sérialisables.

Constantes de `AWTEvent`

Ces constantes (`public final`) permettent d'identifier des classes d'événements. Voici les principales:

`MOUSE_EVENT_MASK`	`MOUSE_MOTION_EVENT_MASK`
`KEY_EVENT_MASK`	`FOCUS_EVENT_MASK`
`ITEM_EVENT_MASK`	`TEXT_EVENT_MASK`
`WINDOW_EVENT_MASK`	`ADJUSTMENT_EVENT_MASK`

> Il y en a d'autres (`COMPONENT_EVENT_MASK`).

Gestion globale: utiliser enableEvents()

```
public class toto extends JFrame {
        public toto() {
          enableEvents(AWTEvent.WINDOW_EVENT_MASK);
          protected void processWindowEvent(WindowEvent e) {
                if (e.getID() == WindowEvent.WINDOW_CLOSING){
                                dispose(); // libère les ressources
                                System.exit(0);
                }
                super.processWindowEvent( e ):  // passe l'ev.
          }
        }
}
```

utiliser enableEvents()

Les ID (types d'événements particuliers) correspondants dans `WindowEvent:`

`WINDOW_OPENED, WINDOW_CLOSING, WINDOW_CLOSED, WINDOW_ACTIVATED, WINDOW_DESACTIVATED, WINDOW_ICONIFIED, WINDOW_DEICONIFIED.`

Pour les autres classes d'evénements, il y a d'autres ID (types plus particuliers).

utiliser enableEvents()

1. Appeler `EnableEvent()` sur le composant avec des masques de AWTEvent liés par un ou bit-à-bit.
2. Implémenter les méthodes correspondantes, ex :
   - `processEvent( AWTEvent e )`
   - `processComponentEvent( ComponentEvent e )`
   - `processFocusEvent( FocusEvent e )`
   - `processKeyEvent( KeyEvent e )`
   - `processMouseEvent( MouseEvent e )`
   - `processMouseMotionEvent( MouseEvent e )`

☞ La méthode `processWindowEvent` n'est disponible que pour `Window` et ses sous-classes.

Méthode 2: (java 1.1) utiliser des écouteurs délocalisés

Il y a des interfaces écouteurs de bas niveau correspondants aux masques d'événements principaux. Elles sont toutes des extensions de `java.util.EventListener`. Par exemple:

> `WindowListener` // événements fenêtre
> `MouseListener` // clics + entrée/sortie fenêtre
> `MouseMotionListener` // mouvements souris
> `KeyListener` // touches clavier
> `FocusListener` // focus clavier
> `ComponentListener` // configuration

Ecouteurs sur les fenêtres (WindowEvent)

WindowListener

Méthodes de l'interface	Description
windowOpened(WindowEvent e)	Appelée la 1ere fois que la fenêtre s'ouvre
windowClosing(WindowEvent e)	Menu close du système
windowClosed(WindowEvent e)	Appelée quand on a fermé la fenêtre
windowActivated(WindowEvent e)	Quand la fenêtre est activée (ex:clic dessus)
windowDeactivated(WindowEvent e)	Quand la fenêtre est désactivée
windowIconified(WindowEvent e)	Quand la fenêtre est réduite à une icône
windowDeiconified(WindowEvent e)	Quand on restore la fenêtre d'une icône

WindowListener

> Il y a une méthode pour chaque type d'événements, déclaré comme constante dans `WindowEvent.`

Ces constantes définissent le type précis de l'événement - type qui est retourné par `getID()`.

WindowListener

Méthodes de l'interface	ID du WindowEvent
windowOpened(WindowEvent e)	WINDOW_OPENED
windowClosing(WindowEvent e)	WINDOW_CLOSING
windowClosed(WindowEvent e)	WINDOW_CLOSED
windowActivated(WindowEvent e)	WINDOW_ACTIVATED
windowDeactivated(WindowEvent e)	WINDOW_DEACTIVATED
windowIconified(WindowEvent e)	WINDOW_ICONIFIED
windowDeiconified(WindowEvent e)	WINDOW_DEICONIFIED

Écouteurs Souris (MouseEvent)

MouseListener

Méthodes de l'interface	Description
mouseClicked(MouseEvent e)	Enfoncer et relâcher: clic sur un composant
mousePressed(MouseEvent e)	Enfoncer sur un composant
mouseReleased(MouseEvent e)	Relâcher sur un composant
mouseEntered(MouseEvent e)	Entrer dans la zone d'un composant
mouseExited(MouseEvent e)	Quitter la zone d'un composant

MouseMotionListener

Méthodes de l'interface	Description
mouseMoved(MouseEvent e)	Mouvement de la souris
mouseDragged(MouseEvent e)	Drag : mouvement + bouton enfoncé

MouseListener

Fonctionne comme `WindowListener`. Les méthodes de `MouseListener` correspondent aux ID `public final` de l'événement `MouseEvent`:

- `MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_MOVED, MOUSE_DRAGGED, MOUSE_ENTERED, MOUSE_EXITED.`
- `mouseClicked(MouseEvent e), mousePressed(MouseEvent e), mouseReleased(MouseEvent e),` etc.

MouseInputListener

> Cette interface écouteur est dans `javax.swing.event.`

> Elle implémente à la fois un `MouseListener` et un `MouseMotionListener.`

☞Mais attention! Il n'existe pas de méthode addMouseInputListener, et quand on l'utilise, il faut enregistrer l'écouteur deux fois : une fois avec `addMouseListener` et une autre fois avec `addMouseMotionListener`.

Écouteurs Clavier, Focus, et Composant
(KeyEvent, FocusEvent et ComponentEvent)

KeyListener

Méthodes de l'interface	Description
`keyTyped(keyEvent e)`	Clé pressée et relâchée
`keyPressed(keyEvent e)`	Clé pressée
`keyReleased(keyEvent e)`	Clé relâchée

KeyListener

Méthodes de l'interface	ID de KeyEvent
keyTyped(keyEvent e)	KEY_TYPED
keyPressed(keyEvent e)	KEY_PRESSED
keyReleased(keyEvent e)	KEY_RELEASED

FocusListener

Méthodes de l'interface	Description
focusGained(FocusEvent e)	Quand un composant prend le focus clavier
focusLost(FocusEvent e)	Quand il le perd

FocusListener

Méthodes de l'interface	ID de FocusEvent
focusGained(FocusEvent e)	FOCUS_GAINED
focusLost(FocusEvent e)	FOCUS_LOST

ComponentListener

Méthodes de l'interface	Description
componentHidden(ComponentEvent e)	Le composant est rendu invisible
componentMoved(ComponentEvent e)	La position du composant change
componentResized(ComponentEvent e)	La taille du composant change
componentShown(ComponentEvent e)	Le composant est rendu visible

ComponentListener

Méthodes de l'interface	ID de ComponentEvent
componentHidden(ComponentEvent e)	COMPONENT_HIDDEN
componentMoved(ComponentEvent e)	COMPONENT_MOVED
componentResized(ComponentEvent e)	COMPONENT_RESIZED
componentShown(ComponentEvent e)	COMPONENT_SHOWN

Ecouteurs d'événements de bas niveau

```
public class Sketcher implements WindowListener {
        private static SketcherFrame window;   // fenêtre de l'application
        private static Sketcher theApp;                    // l'objet application

        public static void main (String[] args) {
                theApp = new Sketcher();
                theApp.init();
        }
        public void init() {
           window = new SketcherFrame(«Dessin»);
           Toolkit leKit = window.getToolkit();
           Dimension wndSize = leKit.getScreenSize();
           window.setBounds(wndSize.width/6, wndSize.height/6,
                                2*wndSize.width/3,2*wndSize.height/3);
           window.addWindowListener(this);
           window.setVisible(true);
        }
```


Ecouteurs d'événements de bas niveau

```
// Méthode pour l'événement windowClosing
        public void windowClosing(WindowEvent e ) {
           window.dispose();          // libère les ressources de la fenêtre
           System.exit(0);   // termine l'application
        }
// fonctions de l'interface listener que nous devons implémenter
// mais dont nous n'avons pas besoin …
        public void windowOpened(WindowEvent e ) {};
        public void windowClosed(WindowEvent e ) {};
        public void windowIconified(WindowEvent e ) {};
        public void windowDeiconified(WindowEvent e ) {};
        public void windowActivated(WindowEvent e ) {};
        public void windowDeactivated(WindowEvent e ) {};
}
        // dans SketcherFrame
        // on supprime setDefaultCloseOperation(EXIT_ON_CLOSE);
```


Ecouteurs d'événements de bas niveau

☹ Gros inconvénient de la gestion par écouteurs: il faut implémenter toutes les méthodes de l'interface correspondante, y compris celles dont on ne se sert pas.

☺ Cet inconvénient peut être supprimé grâce aux classes `Adapter`.

Classes Adapter

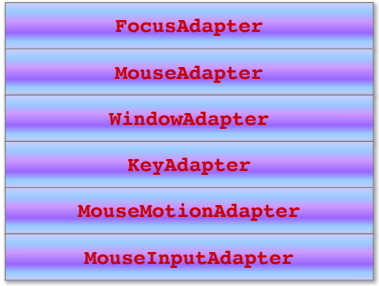
Ce sont des classes qui implémentent une interface (ici écouteur) mais dont les méthodes n'ont pas de code : elles ne font rien.

> On peut alors étendre la classe Adapter choisie en redéfinissant uniquement les méthodes que l'on souhaite utiliser. (Les autres sont déjà définies et vides).

Classes Adapter pour les écouteurs de bas niveau

> Il y a une classe `Adapter` pour chacun des écouteurs de bas niveau définis dans le package `java.awt.event`, plus une définie dans le package `javax.swing.event` étendant l'interface `MouseInputListener`.

Classes Adapter d'écouteurs de bas niveau



1ère méthode: étendre directement une classe Adapter

☞ Etendre une classe Adapter.

```
public class MaClass extends MouseAdapter {
        ...
    unObject.addMouseListener(this);
        ...
    public void mouseClicked(MouseEvent e) {
      ...
            // l'implementation de la méthode
            // associée à l'événement vient ici ...
    }
}
```

2ème méthode: définir des classes internes d'écouteurs

☞ Définir une classe interne.

```
window.addWindowListener(new WindowHandler());
                // classe interne WindowHandler
                //  pour les événements de fermeture
class WindowHandler extends WindowAdapter {
        // Méthode pour WINDOW_CLOSING event
        public void windowClosing( WindowEvent e ) {
                window.dispose();
                System.exit(0);
        }
}
```


2ème méthode: exemple

```
// les import: java.applet.Applet, java.awt.*, java.awt.event.*
public class AdapterSpot extends Applet {
        private Point clickPoint = null;
        private static final int RADIUS = 7;
        public void init() {
                addMouseListener(new MyMouseAdapter());
        }
        public void paint(Graphics g) {
                g.drawRect(0, 0, getSize().width - 1,
                                getSize().height - 1);
```


2ème méthode: exemple

```
                if (clickPoint != null)
                        g.fillOval(clickPoint.x-RADIUS,
                                clickPoint.y-RADIUS,
                                RADIUS*2, RADIUS*2);
        }       // fin de paint
        class MyMouseAdapter extends MouseAdapter  {
                public void mousePressed(MouseEvent event) {
                        clickPoint = event.getPoint();
                        repaint();
                }
}       }
```

2. Evénements sémantiques

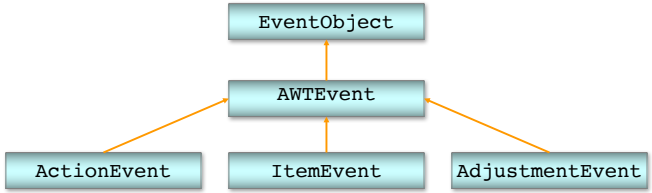
Événements sémantiques

Ce sont des événements spécifiques, liés à des opérations spécifiques sur certains composants : sélection d'éléments de menu, clic sur un bouton, etc.

> Il y a trois classes de base d'événements sémantiques dérivés de AWTEvent.

> Il existe des événements sémantiques spécifiques aux composants Swing dérivés également de cette classe.

Événements sémantiques



Ces trois événements sémantiques sont dans `java.awt.event.`

`ActionEvent`

> il se produit quand on effectue une action sur un composant réactif: clic sur un item de menu ou sur un bouton.

> il est émis par les objets de type

- Boutons: `JButton, JToggleButton, JCheckBox`
- Menus: `JMenu, JMenuItem, JCheckBoxMenuItem, JRadioBoxMenuItem.`.
- Texte: `JTextField`

ItemEvent

> il se produit quand un composant est sélectionné ou désélectionné.

> il est émis par les objets de type

- Boutons: `JButton, JToggleButton, JCheckBox`
- Menus: `JMenu, JMenuItem, JCheckBoxMenuItem,JRadioBoxMenuItem`
- mais aussi `JComboBox, Jlist.`

AdjustmentEvent

> il se produit quand un élément ajustable, comme une JScrollbar, est ajusté.

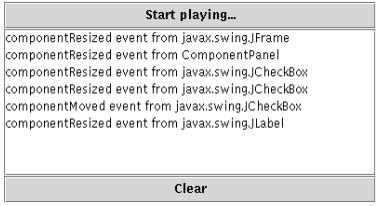
> il est émis par `Scrollbar, JScrollbar`.

Ecouteurs pour événements sémantiques

Comme pour les événements de bas niveau, c'est la façon la plus simple de gérer les événements sémantiques

interface listener	Méthode (unique)
ActionListener	void actionPerformed(ActionEvent e)
ItemListener	void itemStateChanged(ItemEvent e)
AdjustmentListener	void adjustmentValueChanged(AdjustmentEvent e)

Exemple



Cette *applet* contrôle l'apparition d'une fenêtre supplémentaire (un `JFrame`) qui apparaît lorsqu'on enfonce le bouton "Start playing…"

Exemple

Dans le JFrame, on trouvera un label et une checkbox qui permettra de cocher si le label est rendu visible ou invisible.

On pourra aussi iconifier le JFrame ou non.

Les différents événements captés écriront leur source et leur type au centre du cadre de l'applet précédent dans la zone de texte.


```
    JButton b1 = new JButton("Start playing...");
     b1.setActionCommand(SHOW);
     b1.addActionListener(this);
     getContentPane().add(b1, BorderLayout.NORTH);

     JButton b2 = new JButton("Clear");
     b2.setActionCommand(CLEAR);
     b2.addActionListener(this);
     getContentPane().add(b2, BorderLayout.SOUTH);

     aFrame = new JFrame("A Frame"); // une autre fenêtre
     ComponentPanel p = new ComponentPanel(this); // défini + loin
     aFrame.addComponentListener(this); // l'Applet écoute aFrame
     p.addComponentListener(this);// et son panneau
     aFrame.getContentPane().add(p,
                                 BorderLayout.CENTER);
     aFrame.pack();
     aFrame.addWindowListener(
           new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
              showIt = false;
           }
           });
  } // fin de init()
```

Exemple

```
public class ComponentEventDemo extends JApplet
     implements ComponentListener, ActionListener {
    JTextArea display;
    JFrame aFrame; // une fenêtre supplémentaire à afficher
    public boolean showIt = false;
    final static String SHOW = "show";
    final static String CLEAR = "clear";
    String newline = "\n";

    public void init() {
        display = new JTextArea();
        display.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(display);
        scrollPane.setPreferredSize(new Dimension(200, 75));
        getContentPane().add(scrollPane, BorderLayout.CENTER);
```



```
public void actionPerformed(ActionEvent e) { // sur les boutons
     if (e.getActionCommand() == SHOW) {
         showIt = true;
         aFrame.setVisible(true);
     } else { // si ce n'est pas SHOW, c'est donc CLEAR
         display.setText(""); // efface le contenu central
     }
}
public void stop() {
     SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            aFrame.setVisible(false);
        }
     });
}
public void start() {
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
             if (showIt) {
                 aFrame.setVisible(true);
             }
         }
     });
 }
```

```
    protected void displayMessage(String message) {
         display.append(message + newline);
    }

    public void componentHidden(ComponentEvent e) {
         displayMessage("componentHidden event from " +
                        e.getComponent().getClass().getName());
    }
    public void componentMoved(ComponentEvent e) {
         displayMessage("componentMoved event from " +
              e.getComponent().getClass().getName());
    }
    public void componentResized(ComponentEvent e) {
         displayMessage("componentResized event from " +
              e.getComponent().getClass().getName());
    }

    public void componentShown(ComponentEvent e) {
         displayMessage("componentShown event from "+
              e.getComponent().getClass().getName());
     }
}      // fin de ComponentEventDemo
```

```
  public void itemStateChanged(ItemEvent e) { // pour la checkbox

        if (e.getStateChange() == ItemEvent.SELECTED) {
            label.setVisible(true);

         }
         else {
            label.setVisible(false);
        }
    }
}
```



```
class ComponentPanel extends JPanel
                    implements ItemListener { // contenu de "aFrame"
    JLabel label;
    JCheckBox checkbox;

    ComponentPanel(ComponentEventDemo listener) {
        super(new BorderLayout()); // sinon c'est un FlowLayout

        label = new JLabel("This is a Label", JLabel.CENTER);
        add(label, BorderLayout.CENTER);

        checkbox = new JCheckBox("Label visible", true);
        checkbox.addItemListener(this);
        add(checkbox, BorderLayout.SOUTH);

        label.addComponentListener(listener); // listener = le JApplet
        checkbox.addComponentListener(listener);
    }
```

Événements sémantiques

Il y en a beaucoup d'autres, ainsi par exemple:

- Les sous-classes de `AbstractButton` peuvent générer des événements **`ChangeEvent`** quand on modifie l'état d'un bouton.
- Les dérivés de `JMenuItem` génèrent des **`MenuDragMouseEvent`** et des **`MenuKeyEvent`**.
- Une `JList` génère des **`SelectionEvent.`**
- Les modèles associés aux listes et aux tables génèrent des **`ListDataEvent`** et des **`TableModelEvent`** (envoyés aux vues quand des changements se produisent sur le modèle).

Événements sur des tables ou des listes

Architecture Modèle/Vue

- On peut faire partager le modèle de données d'une JTable ou d'une JList à d'autres composants, considérés comme d'autres « vues » de l'objet JTable ou JList (en fait ce sont d'autres vues du modèle sous-jacent à cette table ou cette liste)
- ex: une liste de noms, affichés dans une JList. La JList est une vue (interactive) de la liste de noms, permettant la sélection d'élément(s). Un morceau de texte, indiquant en permanence le nombre d'éléments de la liste, peut aussi être considéré comme une vue du modèle.

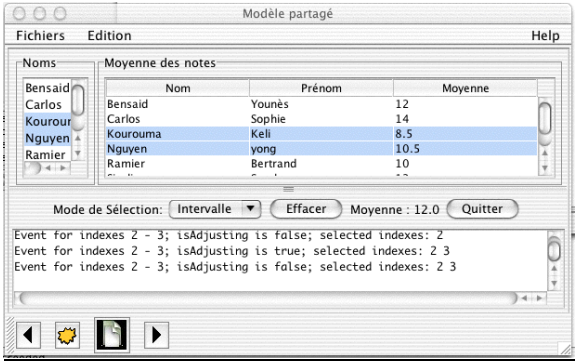
Architecture Modèle/Vue

On a vu que la plupart des classes de `JComponent` encapsulent un modèle, une vue et un contrôleur.

- Si le modèle est implémenté par une interface `<X>Model` prédéfinie, il existe généralement une classe Adapter nommée `Default<X>Model.` C'est le cas ici avec `<X>=List ou Table.`
- L'interface `<X>Model` permet de gérer les données du modèle et d'ajouter des vues (écouteurs de changements sur le modèle).
- Si le modèle change, il génère un événement et notifie les vues (via des méthodes en `fire<...>`).

Architecture Modèle/Vue

> Pour qu'un autre composant soit considéré comme une vue, il doit :
  - implémenter l'interface adéquate pour être écouteur de l'événement de changement du modèle.
  - se faire ajouter à la liste des écouteurs du modèle.
> On peut ainsi faire partager le modèle associé à une JList (ou une JTable) à un autre composant (cf. exemple précédent).
> On pourra également faire partager un même modèle de données à une JTable et une JList.



La classe JList

```
> JList(Vector listData)        // modèle fixe
> JList(Object[] listData)      // modèle fixe
> JList(ListModel dm)           // modèle modifiable
```

Exemple avec un modèle modifiable :

```
String listData[]= {«Younes»,…, «Sarah»};
DefaultListModel model = new
     DefaultListModel();
For (int i=0; i<listData.length; i++)
  model.addElement(listData[i]);
JList dataList = new JList(model);
JScrollPane listeScroll = new
                    JScrollPane(dataList);
```


La classe JList

> `setSelectionMode(int mode);` les modes de sélection sont dans `ListSelectionModel` et peuvent valoir `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION`, `MULTIPLE_INTERVAL_SELECTION`.
> `addListSelectionListener(…);` un `ListSelectionListener` écoute les `ListSelectionEvent e` (chaque fois qu'une sélection change) et implémente une méthode : `ValueChanged`.
> `Boolean = e.getValueIsAdjusting()`

Modèle d'une JList

Pour le modèle, utiliser la classe `DefaultListModel` ou la sous-classer. Ce modèle stocke les objets sous forme de vecteur et fournit les méthodes :

> `addElement(Object), boolean contains(Object), Object get(index),`
> `boolean removeElement(Objet),`
> `Object remove(index), int size().`
> `addListDataListener(ListDataListener l), removeListDataListener(ListDataListener l).`
> `fireContentsChanged, fireIntervalAdded, fireIntervalRemoved (Object source, int index0, int index1).`

Vue d'une JList

Une vue d'une JList implémentera l'interface `ListDataListener.` Il y a trois méthodes :

> `void contentsChanged(ListDataEvent e)`
> `void intervalAdded(ListDataEvent e)`
> `void intervalRemoved(ListDataEvent e)`

L'événement `e` de type `ListDataEvent` fournit :

> `public int getType(): CONTENTS_CHANGED, INTERVAL_ADDED, INTERVAL_REMOVED.`
> `public int getIndex0()`
> `public int getIndex1()`

Plusieurs vues d'une JList

> Créer un modèle instance de `DefaultListModel` et le mémoriser. Créer ensuite la liste avec ce modèle.
> Pour chaque composant désirant être informé des changements (= d'autres vues ou le contrôleur) :
  - Mémoriser le modèle (=le stocker dans un membre).
  - Implémenter `ListDataListener`. Il y a trois méthodes: `contentsChanged, intervalAdded` et `intervalRemoved (ListDataEvent e)`.
  - Enregistrer le composant dans le modèle avec `addListDataListener`.

Modèle d'une JTable

On dispose de 3 éléments pour créer un modèle: l'interface `TableModel,` la classe `AbstractTableModel` qui implémente `TableModel` et la classe `DefaultTableModel.`

`DefaultTableModel` est le plus simple à utiliser :

> `DefaultTableModel(int row, int col)`
> `DefaultTableModel(Object[][] data, Object[] columnNames)`
> `DefaultTableModel(Vector data, Vector columnNames), etc.`

La classe JTable

> `JTable(Object[][] rowData, Object[] columnNames)`

On peut accéder au modèle sous-jacent avec

`TableModel getModel()`

ou appeler le constructeur initial avec un modèle:

```
String nomsCol[]={«Prenom», «Nom»};
String rows[][] = { {«Dinah»,«Cohen»}, … ,
                    {«Said», «Kharrazen»}};
DefaultTableModel model = new
        DefaultTableModel(rows, nomsCol);
JTable table = new JTable(model);
```


DefaultTableModel

> `Vector getDataVector(), addRow(Vector data), insertRow(int, Vector data), Object getValueAt(int row, int col), setValueAt(Object, int row, int col), removeRow(int), int getRowCount(), int getColumnCount(), String getColumnName(int), addColumn(Object columnName), etc.`

> `addTableModelListener(TableModelListener l), removeTableModelListener(l)`

> `fireTableChanged(TableModelEvent e), fireTableRowsInserted(int firstRow, lastRow), fireTableRowsUpdated(int firstRow, lastRow), fireTableRowsDeleted(int firstRow, lastRow), etc.`

Vue d'une JTable

Une vue implémentera l'interface `TableDataListener`. Il y a une méthode :

> `void tableChanged(TableModelEvent e)`

L'événement associé `TableModelEvent` :

> `int getType(): INSERT, UPDATE, DELETE.`

> `int getColumn()`

> `int getFirstRow()`

> `int getLastRow()`

Plusieurs vues d'une JTable

> Créer un modèle (par ex. une instance de `DefaultTableModel`) et le mémoriser.

> Pour chaque composant désirant être informé des changements (les vues, et/ou le controleur) :

- Mémoriser le modèle.
- Implémenter `TableModelListener`. Une méthode: `tableChanged (TableModelEvent e)`.
- Enregistrer le composant dans le modèle avec `addTableModelListener`.

Les Actions

Les Actions

> Nous allons appliquer en TP le modèle par délégation avec écouteurs pour la gestion des événements liés à une barre de menus.

☹ Défaut de cette méthode : si on veut ajouter ensuite une barre d'outils, les boutons de cette barre devront engendrer les mêmes actions que les items de menus. Le code pour cette action devra donc en partie être dupliqué et on pourra introduire des incohérences.

Les Actions

> Une action est un objet de n'importe quelle classe qui implémente l'interface **`Action`**. L'interface déclare des méthodes qui opèrent sur un objet **`Action.`**

> L'interface **`Action`** étend l'interface `ActionListener` donc un objet **`Action`** est aussi un écouteur d'événement d'action de type `ActionEvent`.

Les Actions

> Les composants Swing, comme `JMenu` et `JToolBar` ont une méthode `add()` qui accepte un argument de type `Action.`

> Quand on fait `add()` d'un objet `Action` sur un menu ou une barre d'outils, la méthode `add` crée un composant à partir de l'objet `Action` qui est ***automatiquement du bon type*** par rapport au composant auquel on l'ajoute (i.e. item de menu ou bouton).

Les Actions

> Ainsi, si on ajoute un objet `Action` à un `JMenu`, `add()` ajoute un `JMenuItem` au menu. Si on ajoute ***ce même objet*** à une `JToolBar`, c'est un `JButton` qui lui sera ajouté.

> Comme l'objet `Action` est comme son propre écouteur, le bouton et l'item de menu supporteront la même procédure d'action.

Les 7 propriétés des Actions

Nom	String (aussi label)
Petite icône	Icon (pour la toolbar)
Description courte	String (pour tooltip)
Accélérateur	KeyStroke (accélérateur)
Description longue	String (Aide contextuelle)
Mnémonique	int (mnémonique de l'action)
Clé de commande	(keymap associée au composant)

Les Actions

> L'interface `Action` déclare en outre des méthodes permettant d'accéder aux propriétés liées aux objets Action.

> Il y a 7 propriétés standards dans l'interface `Action` qui caractérisent un objet implémentant l'interface `Action`.

> La classe `Property` est dans `java.util.`

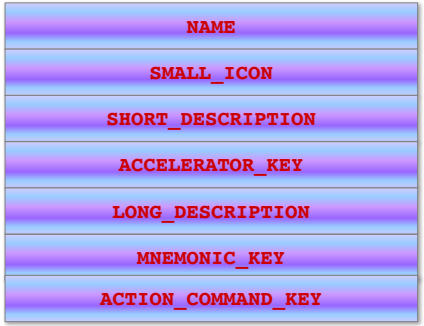
Les propriétés sont stockées dans un dictionnaire (*key*,*property*) organisé en table d'association.

Interface **`Action`**

> Il y a des constantes dans l'interface `Action` qui jouent le rôle de clés pour accéder aux propriétés. Ces constantes sont de type `String.`

> On les utilise pour affecter ou récupérer les valeurs des propriétés associées à un objet Action.

Constantes d'accès (clés)



Interface **Action**

> `void putValue(String key, Object value)`

Associe l'objet `value` avec la clé `key` dans le dictionnaire pour l'objet `Action`.

> Exemple: pour stocker un nom d'action `putValue(NAME, leNom).`

(On utilise la clé standard `NAME` pour stocker l'objet `leNom`).

Interface **Action**

> `Object getValue( String Key )`

Retourne l'objet correspondant à la clé `Key` dans le dictionnaire . Pour retrouver l'icône de la toolbar dans une méthode d'une classe d'action :

```
> Icon lineIcon =
        (Icon) getValue(SMALL_ICON)
```


Interface **Action**

> `boolean isEnabled()`

> Retourne true si l'object d'action est actif et accessible.

> `void setEnabled(boolean state)`

☺ Agit à la fois sur le bouton de la barre d'outils et l'élément de menu si tous deux ont été créés à partir de la même action (éléments grisés).

Interface **Action**

- `void addPropertyChangeListener(PropertyChangeListener listener)`
- Ajoute le listener qui écoute les changements de propriétés (comme l'état d'activation(accessibilité) de l'objet. Utilisé par les conteneurs.
- `void removePropertyChangeListener( PropertyChangeListener listener)`

Classe **AbstractAction**

- Le package `javax.swing` définit la classe `AbstractAction` qui implémente l'interface `Action`.
- Cette classe fournit des mécanismes pour stocker les propriétés d'une action.

`AbstractAction` a trois constructeurs :

- `AbstractAction()`
- `AbstractAction(String name)`
- `AbstractAction(String name,Icon icon)`

Exemple

```
public class SketchFrame extends JFrame {
        public SketchFrame(String title) {
                …
                setJMenuBar(menuBar);
                JMenu fileMenu = new JMenu("File");
                JMenu elementMenu = new JMenu("Elements");
                fileMenu.setMnemonic('F');
                elementMenu.setMnemonic('E');
// On définit des actions pour les items de chaque menu
// et on construit les menus en ajoutant leurs items avec ces actions
// (ensuite on pourra construire la toolbar avec les mêmes actions)
        }
}
```


Exemple

```
class FileAction extends AbstractAction {    // définit les actions
        FileAction(String name) {             // pour les items
                super(name);                          // du menu File
        }
        FileAction(String name, KeyStroke keystroke) {
                this(name);
                if(keystroke != null)
                        putValue(ACCELERATOR_KEY, keystroke);
        }
        public void actionPerformed(ActionEvent e) {
                // code pour les actions liées aux items du menu File
        }
}
```

Exemple

```
public class SketchFrame extends JFrame {
private FileAction newAction, openAction, saveAction;
        public SketchFrame(String title) {
                setTitle(title);
                setJMenuBar(menuBar);
                setDefaultCloseOperation(EXIT_ON_CLOSE);
                JMenu fileMenu = new Jmenu("File");
                JMenu elementMenu = new JMenu("Elements");
                fileMenu.setMnemonic('F');
                elementMenu.setMnemonic('E');
                // On construit ici le file menu avec des actions…
                newAction = new FileAction("New",
                Keystroke.getKeystroke('N',Event.CTRL_MASK));
                addMenuItem(fileMenu, newAction);
```


Exemple

```
private JMenuItem addMenuItem(JMenu menu, Action action) {
        JMenuItem  item = menu.add(action);
        KeyStroke keystrocke = (KeyStroke)
                 action.getValue(action.ACCELERATOR_KEY);
        if (keystroke != null)
                item.setAccelerator(keystroke);
        return item;
}
```


Exemple

```
// pour rajouter l'action à la barre d'outils:
toolBar.add(newAction);
// remarque: pour rajouter son icône, il faudra d'abord l'ajouter à l'action
newAction.putValue(Action.SMALL_ICON,
                                        new ImageIcon("new.gif"));
JButton bouton = toolBar.add(newAction);
// et supprimer le label du bouton d'icône
bouton.setText(null);
// si on avait joué sur le nom (NAME) de l'action, on
// aurait perdu le label de texte dans l'item du menu
```