

# **Programmation événementielle appliquée aux interfaces graphiques**

# Introduction

# Programmation séquentielle

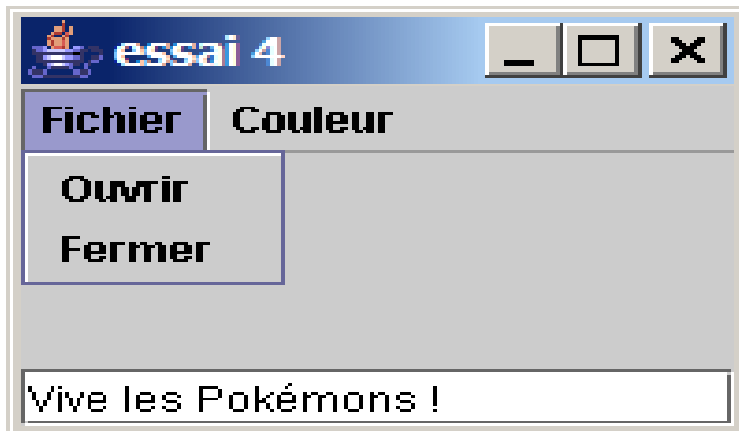
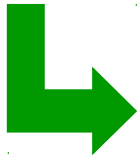
Exécuter une application c'est exécuter la méthode **main** d'une des classes de l'application

```
public static void main(String args[])
{
    Toto t = new Toto();
    for (int i = 0; i < C.maxi; i ++)
        { t.f(i); }
    System.out.println("Fin");
}
```

A la fin de l'exécution de main, **l'application est terminée**

# Programmation événementielle

```
public static void main(String args[])  
{ MaFenêtre f = new MaFenêtre(); }
```



Crée une fenêtre  
graphique et l'affiche  
(via le constructeur)

Et ensuite ?

La méthode main est terminée **mais pas l'application**

**Boucle** d'attente d'actions de l'utilisateur

action → création d'un objet événement  
→ traitement de l'événement  
par les objets intéressés

# **Interfaces graphiques**

## *une première fenêtre*

# Bases sur les **classes graphiques**

- 2 bibliothèques graphiques
  - **AWT** package `java.awt` (1<sup>ère</sup> bibliothèque)
  - **SWING** package `javax.swing`  
dont les composants sont **plus indépendants** du système d'exploitation, et **de plus haut niveau**
- **Frame** remplacée par **JFrame**
- **Button** remplacé par **JButton**
- **Dialog** remplacée par **JDialog**
- **Ne pas mélanger composants AWT et SWING**
- **Lorsque SWING donne une nouvelle version**
- Les événements **de base** restent définis par AWT :  
package `java.awt.event`

**Object**

## Classe racine des composants graphiques

Objets ayant une apparence graphique et capables de réagir à des actions de l'utilisateur

**Component**

*Button*

Composants conteneurs d'autres composants (méthode add)

**Container**

Composants SWING (sauf fenêtres)

**JComponent**

**Window**

Fenêtres

*Frame*

*Dialog*

**JButton**

**JPanel**

**JFrame**

**JDialog**

# Composants **Container**

**Comment sont placés les composants contenus dans un composant ?**

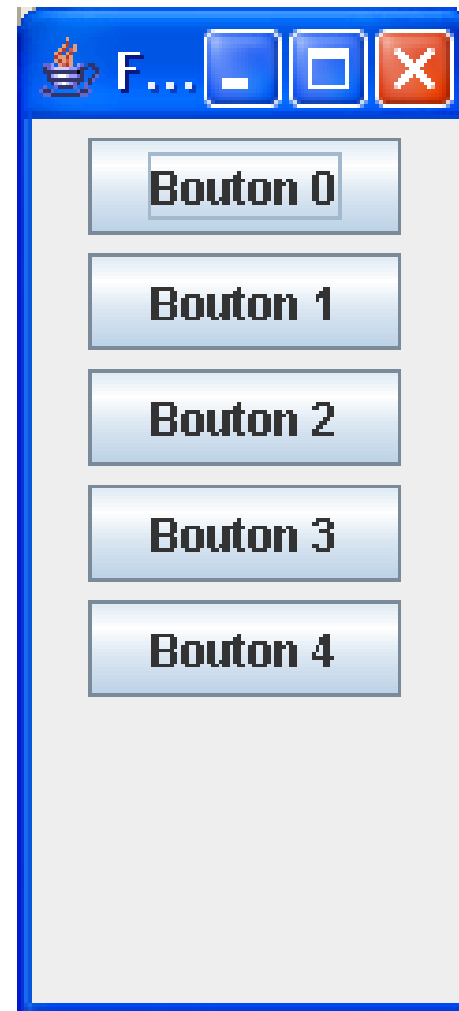
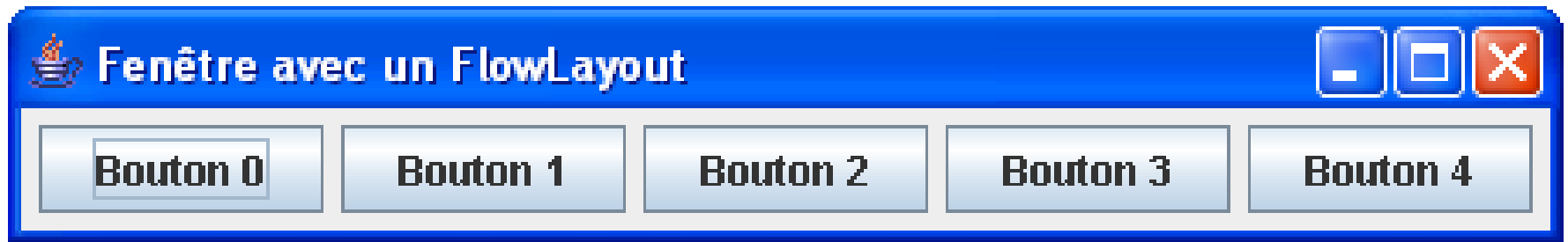
## **Principes :**

- **disposition dynamique** s'adaptant aux modifications de dimensions du conteneur
- chaque conteneur possède un **LayoutManager** qui gère la disposition et la taille de ses composants
- chaque type de LayoutManager a ses propres règles

**Voir tutorial/TP 6, *premières fenêtres graphiques***

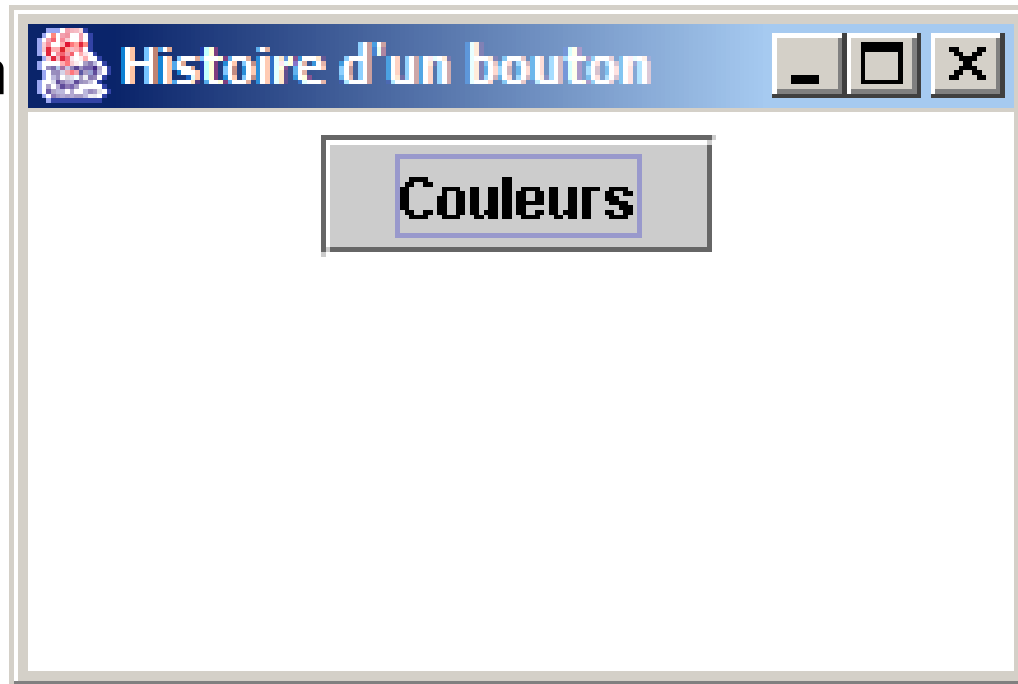
**BorderLayout (régions centre, N, S, E, O)**





# Le minimum pour commencer

- **JFrame** : « fenêtre cadre »
  - classe de base de nos fenêtres
- La fenêtre contient un « panneau »
  - instance de **JPanel** (ou d'une spécialisation)
- Le panneau contient un seul bouton
  - instan



# Problème

Lorsque l'utilisateur actionne le bouton, le panneau doit prendre une couleur différente

(par exemple, en boucle : noir, bleu, jaune, ...)

*action*



*action*



*action*



*action*



```
public class FrameCouleur extends JFrame
{
    private PanneauBouton panneau;

    public FrameCouleur()
    {
        setSize(500, 400);
        setTitle("Histoire d'un bouton");
        panneau = new PanneauBouton();
        add(panneau);
    }

    public static void main(String args[])
    {
        FrameCouleur f = new FrameCouleur();
        f.setVisible(true);
    }
}
```

```
public class PanneauBouton extends JPanel
{
    private static Color [] tCol
    = {Color.black, Color.blue, Color.yellow};

    private JButton b;

    public PanneauBouton()
    {
        b = new JButton("Couleurs");
        add(b);
        setBackground(Color.white);
    }
}
```

*Y'a plus qu'à ... gérer l'action sur bouton*

# **Gestion des événements**

*les sources parlent aux écouteurs*

# Gestion des événements

- Les **événements** sont des objets
- Objets **sources** d'événements :  
gènèrent des événements  
en réponse à des actions de l'utilisateur  
  
ex: fenêtre, bouton, panneau...
- Objets **écouteurs** d'événements
  - **s'inscrivent** auprès d'objets **sources**
  - **sont avertis** lorsque les événements auxquels ils se sont inscrits sont générés

Quel objet peut être écouteur ?

A priori n'importe quel objet à condition que sa classe implémente une certaine **interface**

# Bref rappel sur les interfaces

- **idée** : spécifie des *entêtes de méthodes* à implémenter

ex : définir les opérations sur le type abstrait **pile**

L'interface définit les méthodes qu'une classe doit implémenter pour être conforme à ce type abstrait

- Une classe peut déclarer qu'elle *implémente* une interface : elle doit alors **concrétiser chaque méthode** spécifiée dans l'interface

(sauf si elle est abstraite, auquel cas elle peut ne pas implémenter toutes les méthodes de l'interface)



## Exemple : événement "action sur bouton"

- Objet **source** d'événement

```
b = new JButton("Couleurs");
```

- Certains objets veulent être avertis des **actions sur b**, par ex: **o**

- **b** gère une **liste des écouteurs** d'événements de type « action » sur **b**

→ Il faut que **o** s'inscrive auprès de **b**

```
b.addActionListener(o);
```

```
//ajoute o comme écouteur des actions sur b
```

- Lors d'une action sur le bouton :
  - création d'un événement **e** de type `ActionEvent`
  - pour chaque objet recensé dans la liste d'écouteurs de **b** (donc **o** en particulier)  
une méthode est appelée  
  
`o.actionPerformed(e)`
- La classe de **o** doit donc posséder cette méthode :  
→ ceci est assuré par le fait qu'elle implémente  
l'interface `java.awt.event.ActionListener`  
qui prévoit la méthode :  
  
`public void actionPerformed(ActionEvent e)`

La même chose avec des  
dessins ...

```
interface ActionListener  
{void actionPerformed  
  (ActionEvent e);}
```

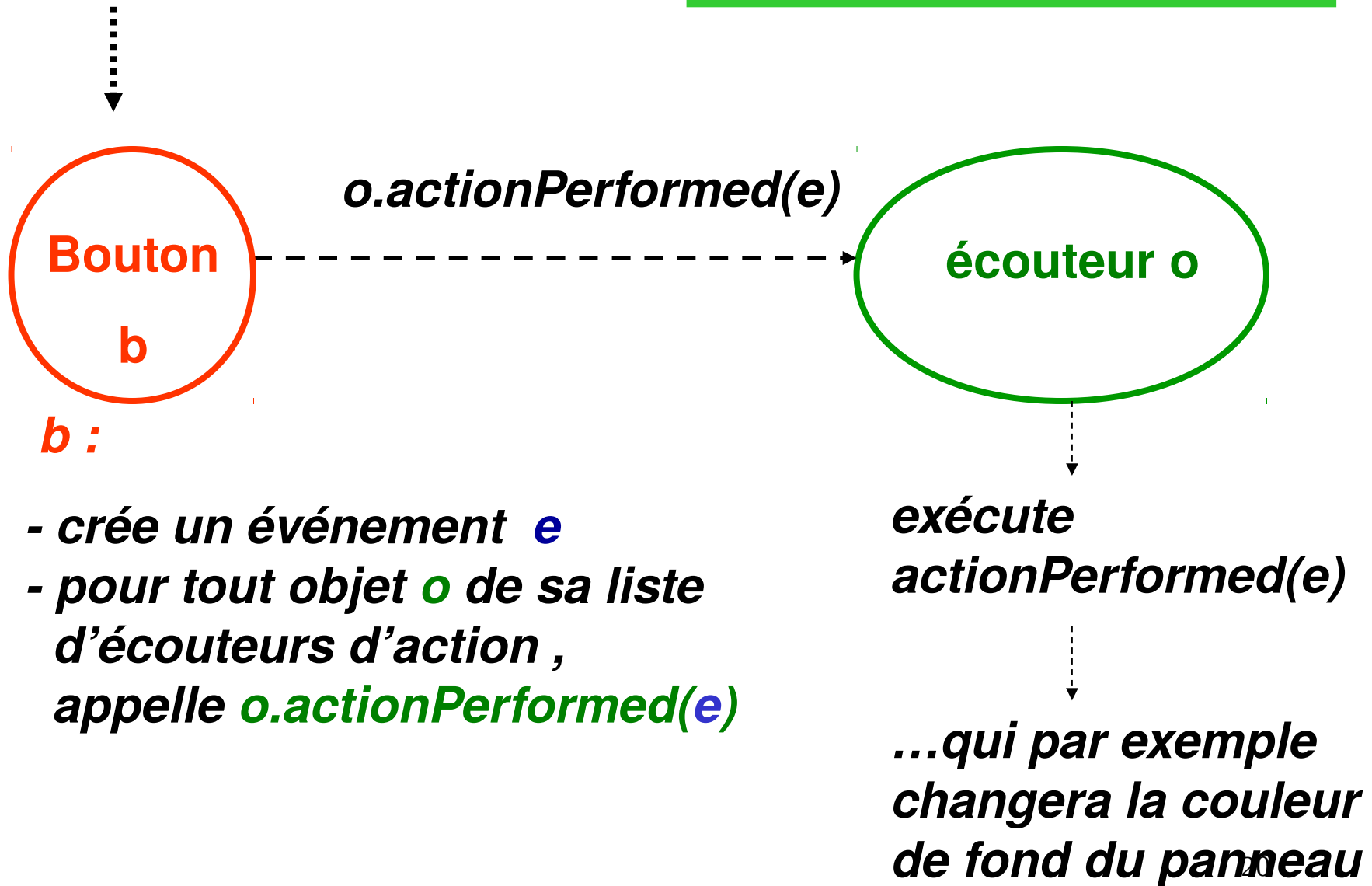


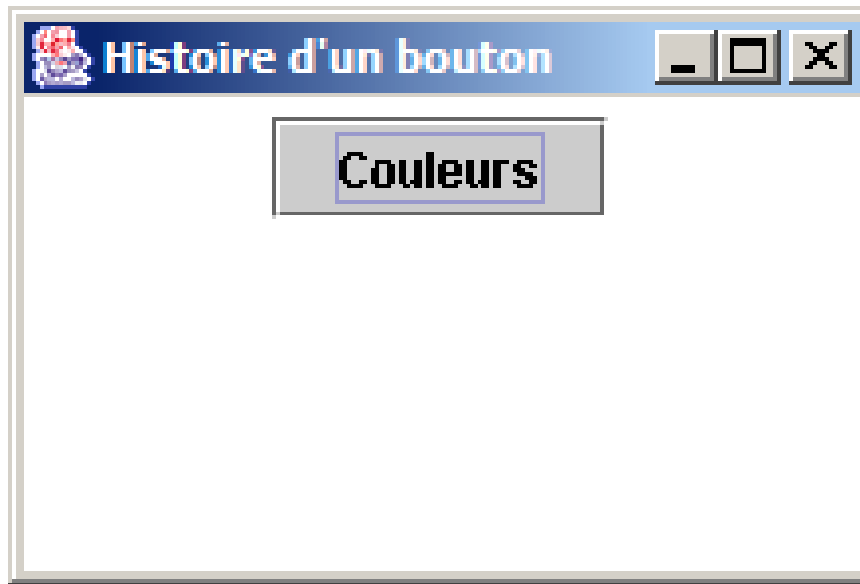
L'écouteur s'enregistre auprès de l'objet source

(et il ne peut le faire que s'il est instance d'une  
classe qui implémente l'interface appropriée)

***l'utilisateur actionne le bouton***

**Ce qui se passe lors d'une action sur le bouton**





**Action sur le bouton → le panneau change de couleur**

**LA** question : *qui écoute?*

*un objet de l'interface graphique*

*un objet créé juste pour écouter*

*Comment choisir?*

# **Gestion des événements**

*qui sont les écouteurs ?*

L'écouteur doit être capable de **traiter l'événement**  
(ici : changer la couleur de fond de panneau)

*bouton... bof*

L'écouteur doit **s'inscrire** auprès de l'objet source  
(ici : le bouton)

donc **il doit connaître** l'objet source,  
ou bien **l'objet source doit le connaître**

*fenêtre... bof*

Restent : le **panneau** ou un **objet créé exprès**

## Solution 1 : Le panneau écoute

```
public class PanneauBouton extends JPanel
    implements ActionListener
{
    private static Color [] tCol
    = {Color.black, Color.blue, Color.yellow};
    private JButton b;
    private int numCol = -1;

    public PanneauBouton()
    {
        b = new JButton("Couleurs");
        add(b);
        b.addActionListener(this);
        setBackground(Color.white);
    }

    public void actionPerformed(ActionEvent e)
    {
        numCol = (numCol + 1) % tCol.length;
        setBackground(tCol[numCol]);
    }
}
```



## Solution 2 : Un écouteur d'une classe dédiée écoute

```
public class PanneauBouton extends JPanel
{
    private static Color [] tCol
    = {Color.black, Color.blue, Color.yellow};

    private JButton b;

    public PanneauBouton()
    {
        b = new JButton("Couleurs");
        add(b);
        setBackground(Color.white);

        EcouteBouton ecout = new EcouteBouton(...);
        b.addActionListener(ecout);
    }
}
```

```
class EcouteBouton implements ActionListener
{
    private static Color[] tCol
    = {Color.black, Color.blue, Color.yellow};
    private int numCol = -1;
    private JPanel p;

    public EcouteBouton(JPanel p)
    { this.p = p; }

    public void actionPerformed(ActionEvent e)
    {
        numCol = (numCol + 1) % tCol.length;
        p.setBackground(tCol[numCol]);
    }
}
```

## Retour à la classe PanneauBouton :

```
public class PanneauBouton extends JPanel
{
private static Color [] ccol
= {Color.black, Color.blue, Color.yellow};

    private JButton b;

    public PanneauBouton()
    {
        b = new JButton("Couleurs");
        add(b);
        setBackground(Color.white);
        EcouteBouton ecout = new EcouteBouton(this);
        b.addActionListener(ecout);
    }
}
```

*Finalemt, on n'a pas besoin de nommer  
l'instance de EcouteBouton*

## Après avoir simplifié la classe PanneauBouton :

```
public class PanneauBouton extends JPanel
{
    private JButton b;

    public PanneauBouton()
    {
        b = new JButton("Couleurs");
        add(b);
        setBackground(Color.white);
        b.addActionListener(new EcouteBouton(this));
    }
}
```

*L'écouteur est ici un objet **anonyme***

On pourrait faire de la **classe** EcouteBouton une classe

- **interne** à la classe PanneauBouton
- et même une **classe interne anonyme**

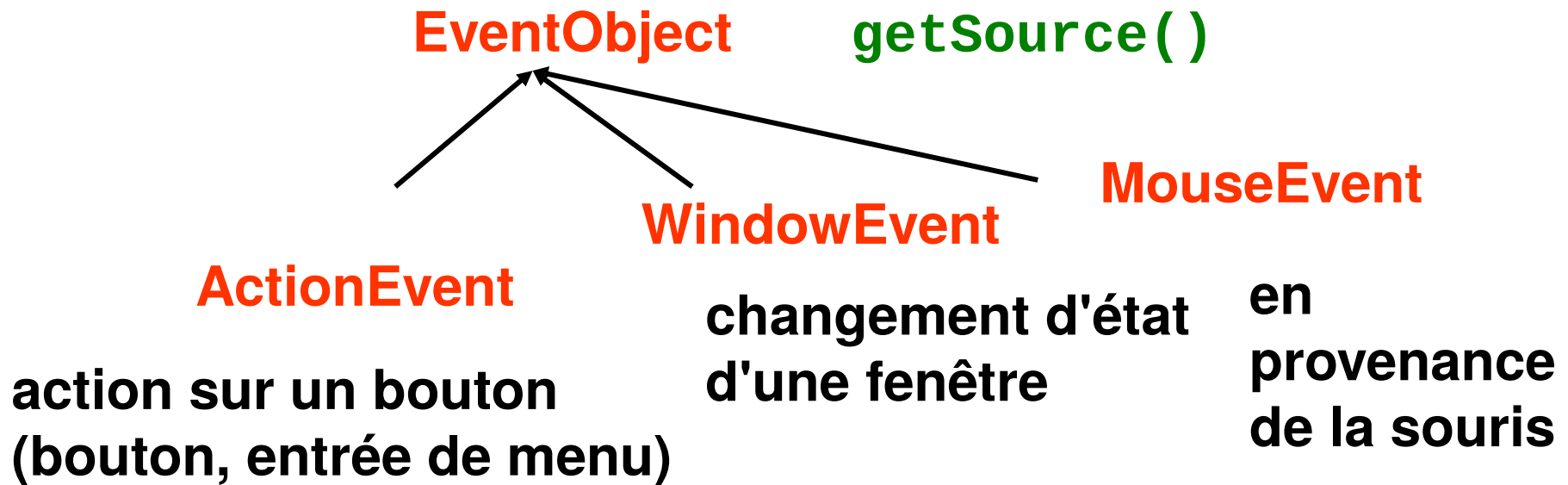
*Mais ce sera pour le prochain cours ...*

# **Gestion des événements**

## *l'objet événement*

# Mécanisme général

- Différentes classes d'événements



- Différentes interfaces prévoyant des méthodes pour traiter certains types d'événements

package `java.awt.event`

- Interface **ActionListener**

`void actionPerformed(ActionEvent e)`

- Interface **MouseListener**

`void mousePressed(MouseEvent e)`

`void mouseReleased(MouseEvent e)`

`void mouseClicked(MouseEvent e)`

`void mouseEntered(MouseEvent e)`

`void mouseExited(MouseEvent e)`

# Événements de **bas** niveau / **haut** niveau

**Bas niveau** : correspond à une action physique de l'utilisateur

*ex : un clic souris (MouseEvent)  
appui sur une touche (KeyEvent)*

**Haut niveau** : a été interprété  
une signification lui a été donnée

*ex : une **action** sur bouton (ActionEvent)*

*peut-être due à des événements de bas niveau  
différents : un clic souris  
ou appui de la touche **return***



- Un objet qui peut être **source** d'événements possède une **liste d'objets écouteurs** (et même des listes de différents types d'écouteurs)
- Lorsque le système l'avertit d'une certaine action :
  - il génère un **événement** d'un certain type,
  - il déclenche la **méthode appropriée**(prévue dans l'interface correspondante)  
sur **tous les objets de cette liste**  
en passant **l'événement** en paramètre

## Exemple 2 : gérer des événements souris (dits "de bas niveau")

- Objet **source** : n'importe quel composant graphique
- 3 interfaces possibles pour les **écouteurs** :

**MouseListener**

**MouseMotionListener**

**MouseListener**

- Événement **souris** :

instance de **MouseEvent**

méthodes **getX()** et **getY()**

[ coordonnées de la souris sur l'objet source,  
au moment de l'événement ]

- Interface **MouseListener**

`void mousePressed(MouseEvent e)`

`void mouseReleased(MouseEvent e)`

`void mouseClicked(MouseEvent e)`

`void mouseEntered(MouseEvent e)`

`void mouseExited(MouseEvent e)`

- Interface **MouseMotionListener**

`void mouseMoved(MouseEvent e)`

`void mouseDragged(MouseEvent e)`

- Interface **MouseListener** (depuis version 5)

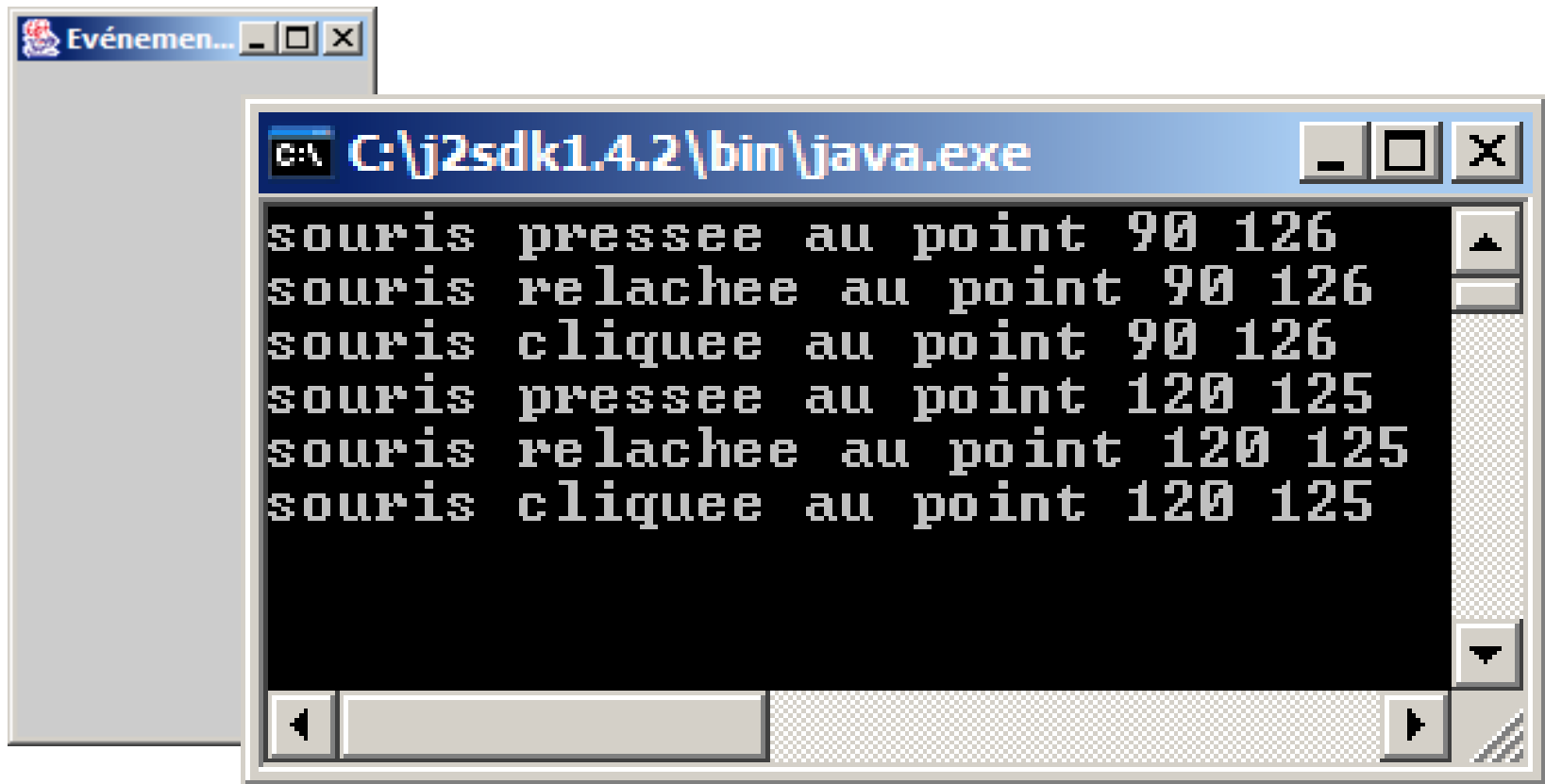
**hérite des deux interfaces ci-dessus**

# **Gestion des événements**

## *classes Adapter*

# Problème

étant donnée une fenêtre contenant un panneau,  
**réagir à certaines actions de la souris sur le panneau :**  
*souris pressée, souris relâchée, souris cliquée*  
**par un affichage dans la fenêtre console :**  
*type d'action au point x y*



```

class Panneau extends JPanel
{ ... }

public class Fenetre extends JFrame
{
    private JPanel pan;
    public Fenetre()
    {
        .....
        pan = new Panneau();
        add(pan);
    }
    .....
}

```

*QUI écoute?*

**Solution 1 :** un objet instance d'une **classe dédiée**

**Solution 2 :** le panneau ou la fenêtre

## Solution 1. Ecouteur d'une classe *dédiée*

```
class EcouteSouris implements MouseListener
{
    public void mousePressed(MouseEvent e)
    {
        System.out.println("Souris pressée au pt "
            + e.getX() + ' ' + e.getY());
    }
}
```

On implémente aussi **mouseReleased** et **mouseClicked**

Mais aussi **mouseEntered** et **mouseExited**  
... avec un corps vide

```
}
```

*On doit implémenter 5 méthodes alors 3 seulement nous intéressent...*

Notion de **classe "adaptateur"** :

classe qui **implémente l'interface**

avec un **corps vide** pour chaque méthode

Il existe une **classe adaptateur** pour la plupart des interfaces qui spécifient **plusieurs** méthodes

**[voir documentation du SDK]**



[java.awt.event](#)

## Class MouseAdapter

[java.lang.Object](#)

└ [java.awt.event.MouseAdapter](#)

### All Implemented Interfaces:

[EventListener](#), [MouseListener](#)

### Direct Known Subclasses:

[BasicComboPopup.InvocationMouseHandler](#), [BasicComboPopup.ListMouseHandler](#),

[BasicFileChooserUI.DoubleClickListener](#), [BasicScrollBarUI.ArrowButtonListener](#),

[BasicScrollBarUI.TrackListener](#), [BasicSplitPaneDivider.MouseHandler](#), [BasicTabbedPaneUI.MouseHandler](#),

[BasicTreeUI.MouseHandler](#), [FormView.MouseEventListener](#), [HTMLEditorKit.LinkController](#),

[MetalFileChooserUI.SingleClickListener](#), [ToolTipManager](#)

public abstract class **MouseAdapter**

extends [Object](#)

implements [MouseListener](#)

**implements MouseListener**

An abstract adapter class for receiving mouse events. The methods in this class are empty. This class exists as convenience for creating listener objects.

Mouse events let you track when a mouse is pressed, released, clicked, when it enters a component, and when it exits. (To track mouse moves and mouse drags, use the [MouseMotionAdapter](#).)

**public abstract class **MouseListenerAdapter****  
**extends Object**  
**implements **MouseListener****

**An abstract **adapter class** for receiving mouse events. **The methods in this class are empty.** This class exists as **convenience for creating listener objects.****

**Mouse events let you track when a mouse is pressed, released, clicked, when it enters a component, and when it exits. (To track mouse moves and mouse drags, use the **MouseMotionAdapter**.)**

**Extend this class to create a **MouseEvent** listener and **override the methods for the events of interest.** (If you **implement the **MouseListener** interface, you have to define all of the methods in it. This abstract class defines null methods for them all, so you can only have to define methods for events you care about.****)

**MouseListener**



*implements*

**EcouteSouris**

**MouseListener**



*implements*

**MouseListener**



*extends*

**EcouteSouris**

```
class EcouteSouris extends MouseAdapter
```

```
{ redéfinition des 3 méthodes qui nous  
  intéressent }
```

```
class Panneau extends JPanel  
{ ... }
```

```
public class Fenetre extends JFrame
```

```
{  
    private JPanel pan;  
    public Fenetre()  
    {  
        .....  
        pan = new Panneau();  
        add(pan);
```

```
        EcouteSouris écoute = new EcouteSouris();  
        pan.addMouseListener(écoute);
```

```
    }  
}
```

```
class Panneau extends JPanel { ... }  
    implements MouseListener  
  
{ implémentation des 5 méthodes de l'interface }
```

```
public class Fenetre extends JFrame  
{  
    private JPanel pan;  
    public Fenetre()  
    {  
        .....  
        pan = new Panneau();  
        add(pan);  
        pan.addMouseListener(pan);  
    }  
}
```