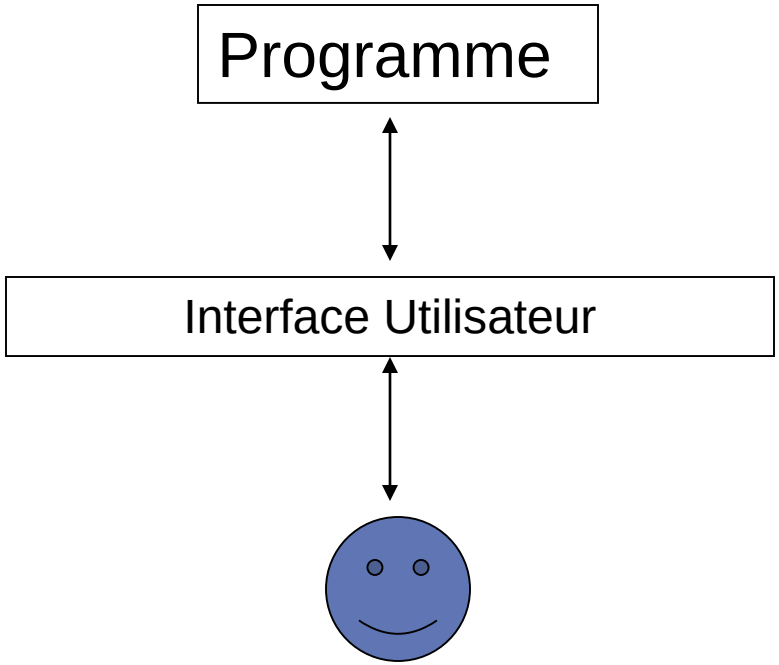
# Les bases de la programmation graphique Java

# AWT
# The Abstract Windowing Toolkit and Swing

# Généralité




Rôles d'une interface utilisateur:

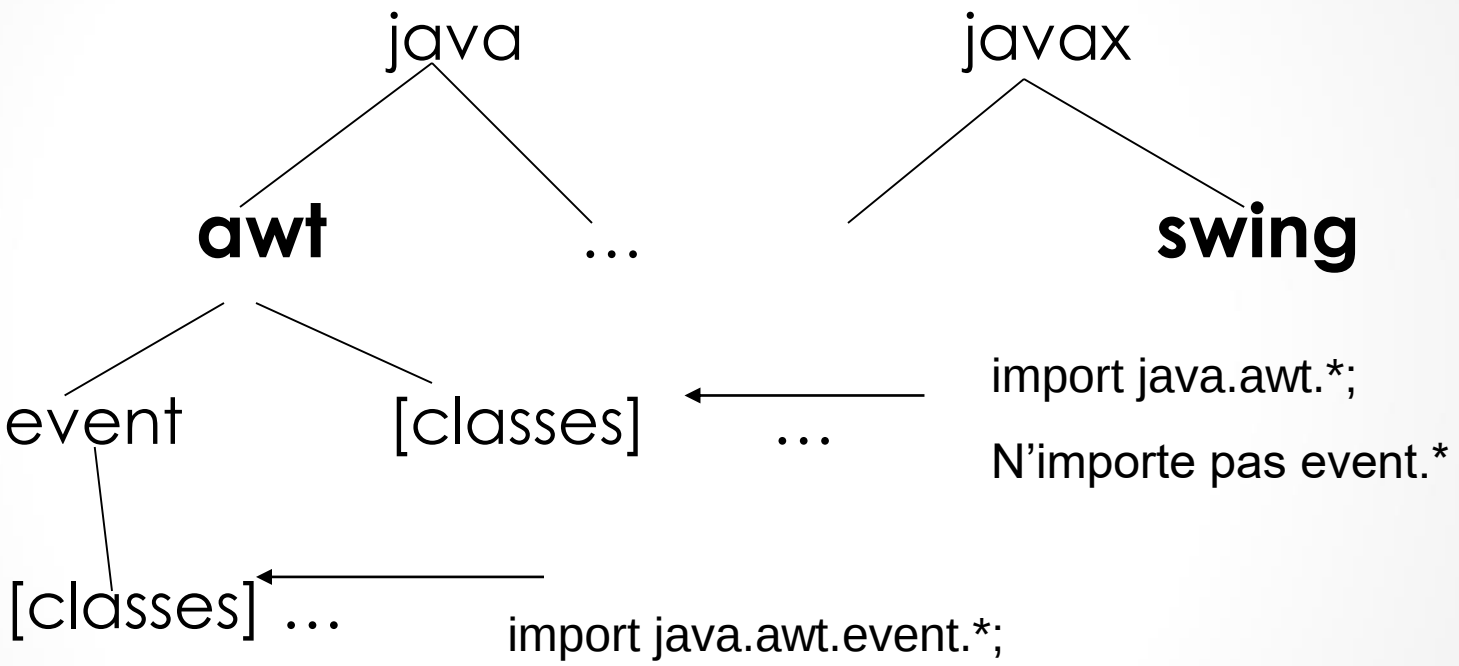
- montrer le résultat de l'exécution
- permettre à l'utilisateur d'interagir
- (1) Montrer – (2) Réagir

# Afficher une interface

- Importer le package (les classes)
  - Les classes sont regroupées en package
  - Importer un package = importer toutes les classes du package
  - `import javax.swing.*;`
- Créer une fenêtre graphique (JFrame, …)
- Définir les paramètres (taille, …)
- Afficher
- Différence:
  - import java.awt.*; les classes dans awt
  - import java.awt.event.*; les classes dans event
  - `import javax.swing.*;`

awt: Abstract Windows Toolkit

# Hiérarchie de package



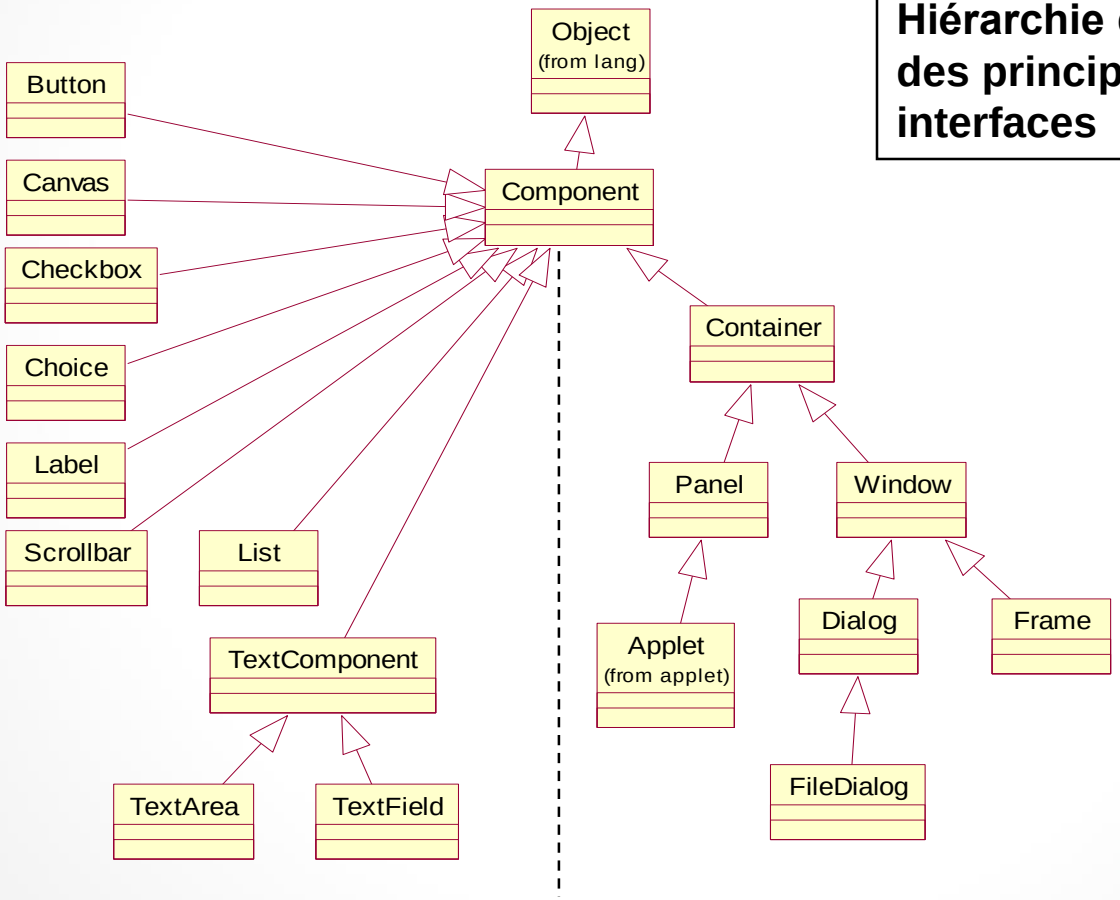
# Conteneurs et composants

- Une interface graphique en Java est un assemblage conteneurs (*Container*) et de composants (*Component*).

- **Un composant** est une partie "visible" de l'interface utilisateur Java.
  - C'est une sous-classes de la classe abstraite **java.awt.Component**.
  - Exemple : les boutons, les zones de textes ou de dessin, etc.

- **Un conteneur** est un espace dans lequel on peut positionner plusieurs composants.
  - Sous-classe de la classe **java.awt.Container**
  - La classe Container est elle-même une sous-classe de la classe Component
  - Par exemple les fenêtres, les applets, etc.

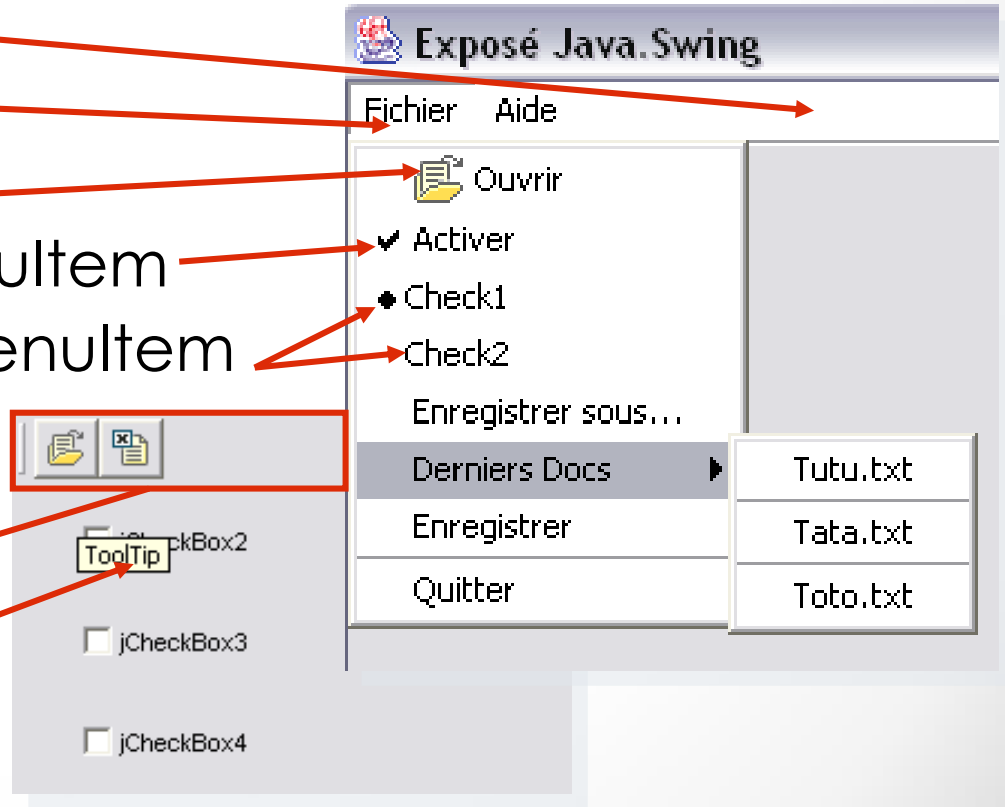
# Conteneurs et composants




**Hiérarchie d'héritage des principaux éléments des interfaces graphiques en Java**

# Les Composants Swing

- Menus, Bar d'outils et ToolTips
  - JMenuBar
  - JMenu
    - JMenuItem
    - JCheckBoxMenuItem
    - JRadioButtonMenuItem
  - JPopupMenu
    - JToolBar
    - JToolTip



# Conteneurs et composants

- Les deux conteneurs les plus courants sont le Frame et le Panel.

- Un **Frame** représente une fenêtre de haut niveau avec un titre, une bordure et des angles de redimensionnement.
  - La plupart des applications utilisent au moins un Frame comme point de départ de leur interface graphique.

- Un **Panel** n'a pas une apparence propre et ne peut pas être utilisé comme fenêtre autonome.
  - Les Panels sont créés et ajoutés aux autres conteneurs de la même façon que les composants tels que les boutons
  - Les Panels peuvent ensuite redéfinir une présentation qui leur soit propre pour contenir eux-mêmes d'autres composants.

# Conteneurs et composants

- On ajoute un composant dans un conteneur, avec la méthode add() :

```
Panel p = new Panel();
Button b = new Button();
p.add(b);
```

- De manière similaire, un composant est retirer de son conteneur par la méthode remove() :

```
p.remove(b);
```

- Un composant a (notamment) :
  - une taille préférée que l'on obtient avec **getPreferredSize**()
  - une taille minimum que l'on obtient avec **getMinimunSize**()
  - une taille maximum que l'on obtient avec **getMaximunSize**()

# Conteneurs et composants

Exemple


```
import java.awt.*;

public class EssaiFenetre1
{
  public static void main(String[] args)
  {
    Frame f =new Frame("Ma première fenêtre");
    Button b= new Button("coucou");
    f.add(b);
    f.pack();
    f.show();
  }
}
```

**Création d'une fenêtre (un objet de la classe Frame) avec un titre**

**Création du bouton ayant pour label « coucou »**

**Ajout du bouton dans la fenêtre**

**On demande à la fenêtre de choisir la taille minimum avec pack() et de se rendre visible avec show()**

# Gestionnaire de présentation

- A chaque conteneur est associé un gestionnaire de présentation (***layout manager***)
- Le gestionnaire de présentation gère le positionnement et le (re)dimensionnement des composants d'un conteneur.
- Le ré-agencement des composants dans un conteneur a lieu lors de :
  - la modification de sa taille,
  - le changement de la taille ou le déplacement d'un des composants.
  - l'ajout, l'affichage, la suppression ou le masquage d'un composant.

- Les principaux gestionnaires de présentation de l'AWT sont : **FlowLayout**, **BorderLayout, GridLayout, CardLayout, GridBagLayout**

# Gestionnaire de présentation

- Tout conteneur possède un gestionnaire de présentation par défaut.
  - Tout instance de ***Container*** référence une instance de ***LayoutManager***.
  - Il est possible d'en changer grâce à la méthode **setLayout()**.

- Les gestionnaires de présentation par défaut sont :
  - Le **BorderLayout** pour **Window** et ses descendants (**Frame**, **Dialog**, …)
  - Le **FlowLayout** pour **Panel** et ses descendants (**Applet**, etc.)

- Une fois installé, un gestionnaire de présentation fonctionne "tout seul" en interagissant avec le conteneur.

# FlowLayout

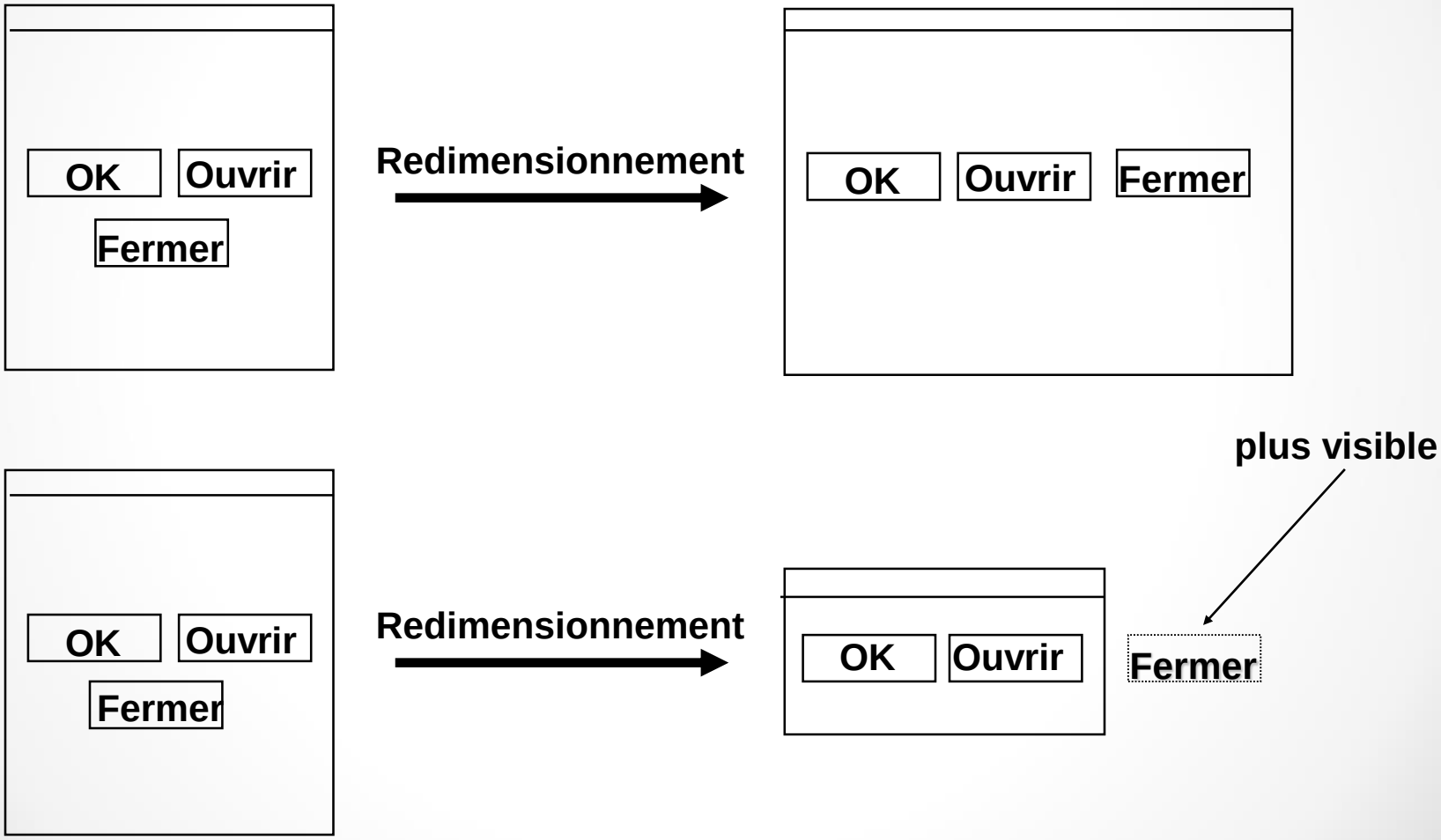
- Le **FlowLayout** est le plus simple des managers de l'AWT
- 
- Gestionnaire de présentation utilisé par défaut dans les Panel si aucun LayoutManager n'est spécifié.

- Un FlowLayout peut spécifier :
  - une justification à gauche, à droite ou centrée,
  - un espacement horizontal ou vertical entre deux composants.
  - Par défaut, les composants sont centrés à l'intérieur de la zone qui leur est allouée.

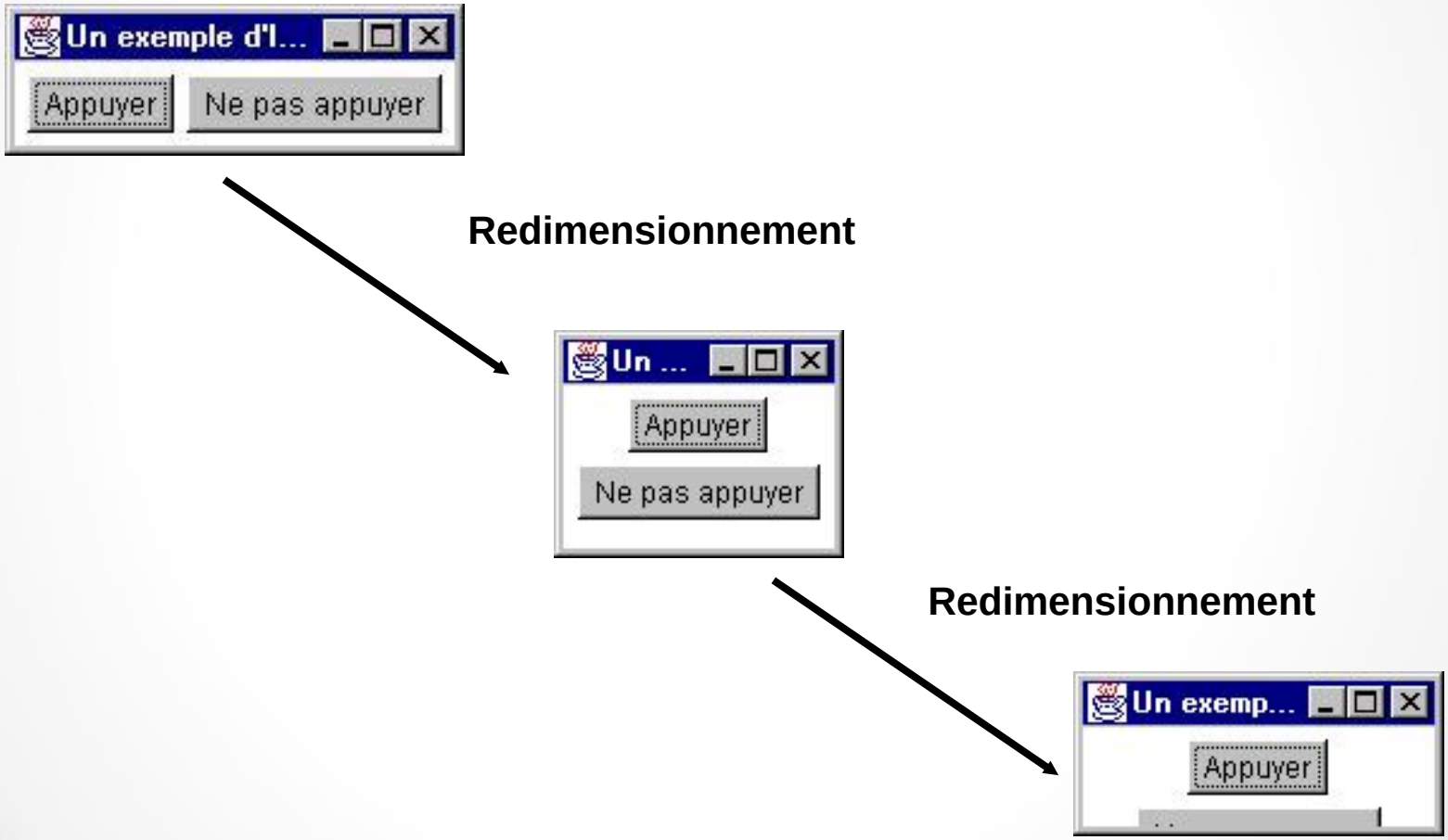
# FlowLayout

- La stratégie de disposition du **FlowLayout** est la suivante :
  - Respecter la **taille préférée** de tous les composants contenus.
  - Disposer autant de composants que l'on peut en faire tenir horizontalement à l'intérieur de l'objet **Container**.
  - Commencer une nouvelle rangée de composants si on ne peut pas les faire tenir sur une seule rangée.
  - Si tous les composants ne peuvent pas tenir dans l'objet **Container**, ce n'est pas géré (c'est-à-dire que les composants peuvent ne pas apparaître).

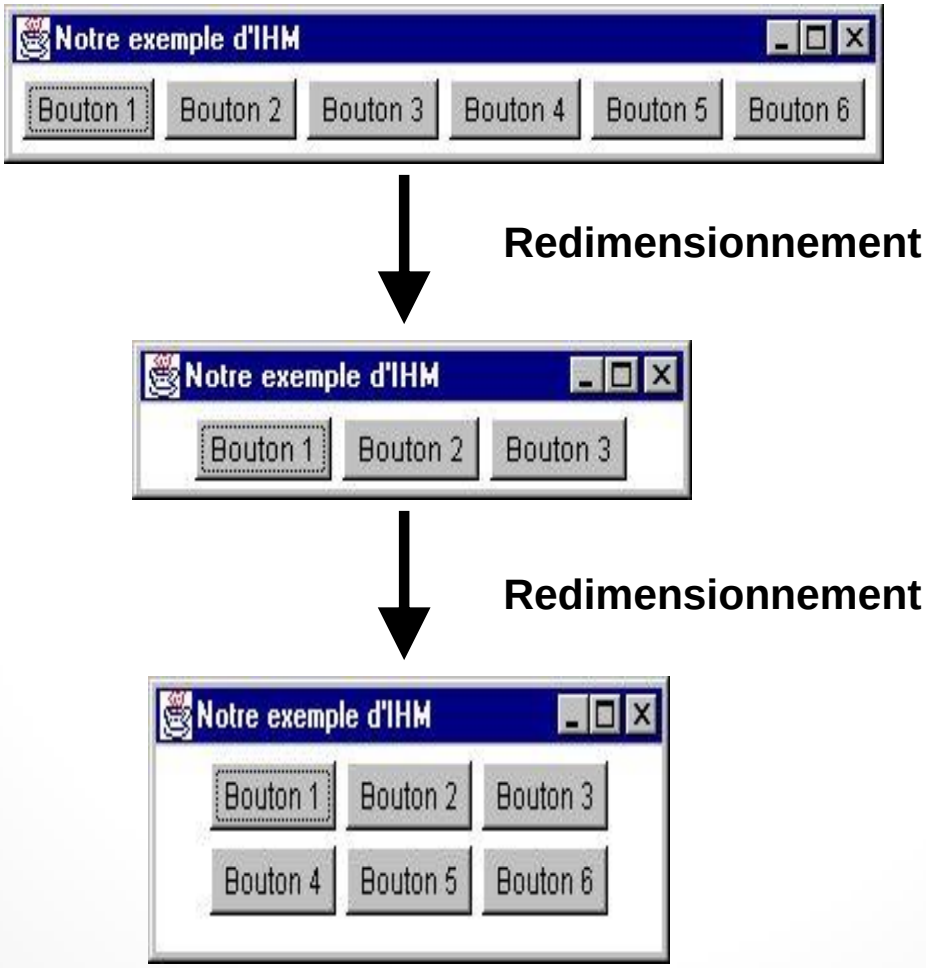
# FlowLayout


OK
Ouvrir
Fermer
Redimensionnement
OK
Ouvrir
Fermer
plus visible
OK
Ouvrir
Fermer
Redimensionnement
OK
Ouvrir
Fermer

# FlowLayout


Un exemple d'l...
Appuyer
Ne pas appuyer
Redimensionnement
Un ...
Appuyer
Ne pas appuyer
Redimensionnement
Un exemp...
Appuyer

# FlowLayout


Notre exemple d'IHM
Bouton 1
Bouton 2
Bouton 3
Bouton 4
Bouton 5
Bouton 6
Redimensionnement
Notre exemple d'IHM
Bouton 1
Bouton 2
Bouton 3
Redimensionnement
Notre exemple d'IHM
Bouton 1
Bouton 2
Bouton 3
Bouton 4
Bouton 5
Bouton 6

# FlowLayout

- Le FlowLayout cache réellement et effectivement les composants qui ne rentrent pas dans le cadre.
- Le FlowLayout n'a d'intérêt que quand il y a peu de composants.

- L'équivalent vertical du FlowLayout n'existe pas
- La présentation FlowLayout positionne les composants ligne par ligne.
  - **Chaque fois qu'une ligne est remplie, une nouvelle ligne est commencée.**
  - **Ces composants sont alignés de gauche à droite**

- Le gestionnaire FlowLayout n'impose pas la taille des composants mais leur permet d'avoir la taille qu'ils préfèrent.

# FlowLayout

La méthode **setLayout** permet de définir un gestionnaire de positionnement qui se chargera de placer correctement les composants insérés.

Exemple

```
import java.awt.*;
class FflowLayout {
   static public void main (String arg [ ]) {
     Frame w = new Frame("Exemple de fenetre avec un FlowLayout");
     w.setLayout(new FlowLayout ());
     w.add(new Button ("UN"));
     w.add(new Label ("DEUX"));
     w.add(new TextField("Trois"));
     w.show();
     w.pack();
   }
}
```

Résultat à l'exécution :

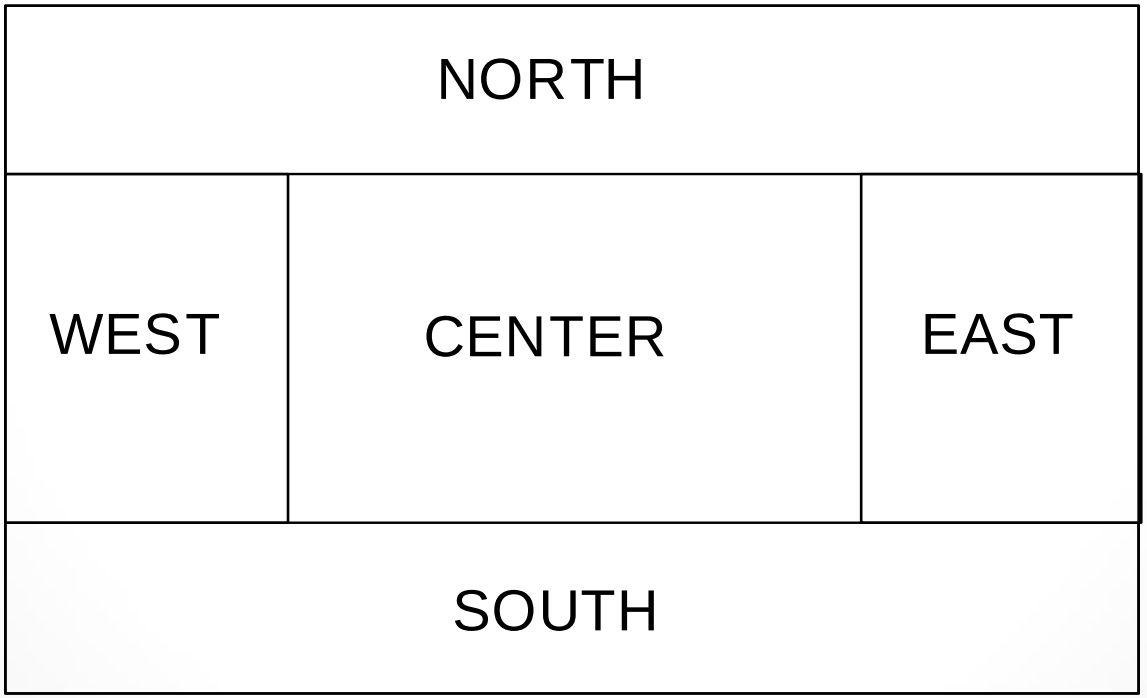
# BorderLayout

- **BorderLayout** divise son espace de travail en cinq zones géographiques : **North, South, East, West et Center.**
- Les composants sont ajoutés par nom à ces zones (un seul composant par zone).

  - Exemple
    add("North", new Button("Le bouton nord !"));
  - Si une des zones de bordure ne contient rien, sa taille est 0.

# BorderLayout

- Division de l'espace avec le **BorderLayout**



# BorderLayout

**Exemple**

```
import java.awt.*;

public class EssaiBorderLayout extends Frame
{
  private Button b1,b2,b3,b4, b5;
   public EssaiBorderLayout() {
    setLayout(new BorderLayout());
    b1 = new Button ("Nord");
    b2 = new Button ("Sud");
    b3 = new Button ("Est");
    b4 = new Button ("Ouest");
    b5 = new Button ("Centre");
    this.add(b1, BorderLayout.NORTH);
    this.add(b2 , BorderLayout.SOUTH);
    this.add(b3, BorderLayout.EAST);
    this.add(b4, BorderLayout.WEST);
    this.add(b5, BorderLayout.CENTER);
  }
```

Résultat à l'exécution :


```
public static void main (String args []) {
EssaiBorderLayout Test = new
EssaiBorderLayout();
    Test.pack ();
    Test.setVisible(true) ;          }}
```

# BorderLayout

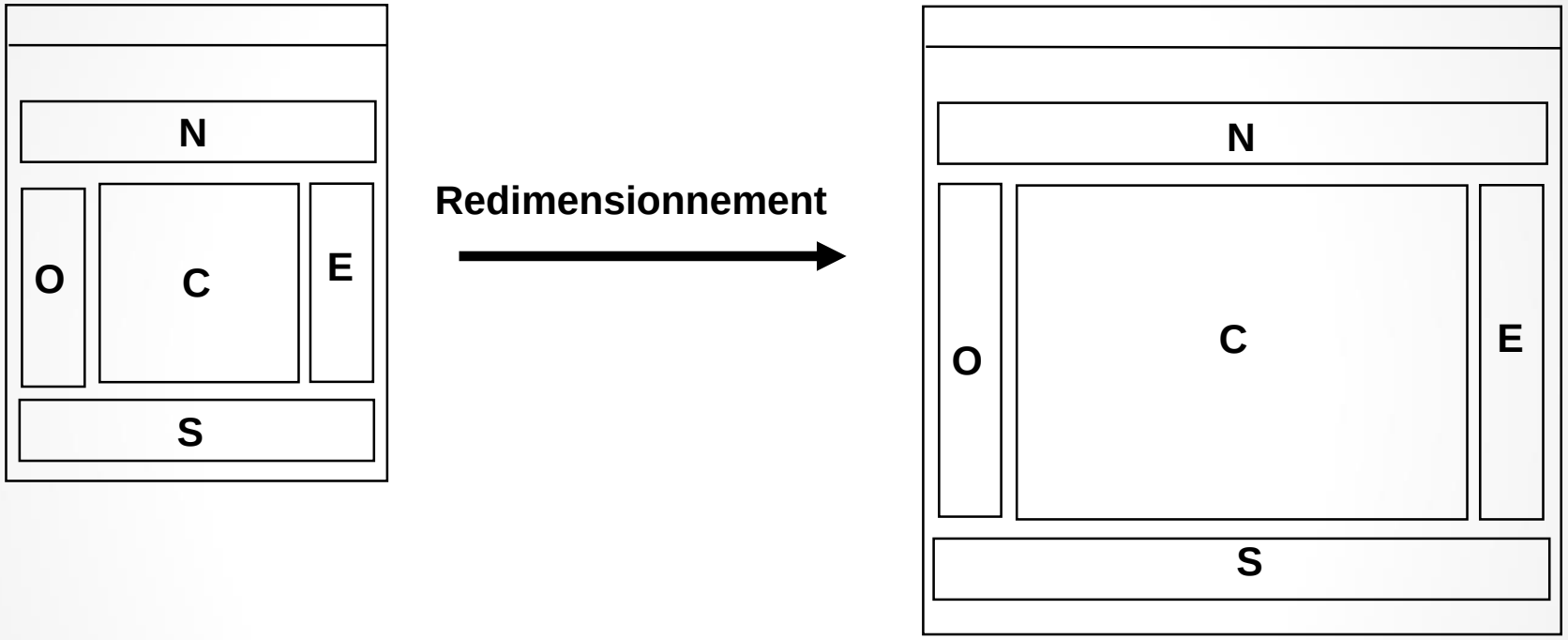
- Stratégie de disposition du BorderLayout

  - S'il y a un composant dans la partie placée dans la partie **NORTH**, il récupère sa ***taille préférée***, respecte sa ***hauteur préférée*** si possible et fixe sa largeur à la totalité de la largeur disponible de l'objet Container.
  - S'il y a un composant dans la partie placée dans la partie **SOUTH**, il fait pareil que dans le cas de la partie **NORTH**.
  - S'il y a un composant dans la partie placée dans la partie **EAST**, il récupère sa **taille préférée**, respecte sa ***largeur préférée*** si possible et fixe sa hauteur à la totalité de la hauteur encore disponible.
  - S'il y a un composant dans la partie placée dans la partie **WEST**, il fait pareil que dans le cas de la partie **EAST**.
  - S'il y a un composant dans la partie **CENTER**, il lui donne la place qui reste, s'il en reste encore.

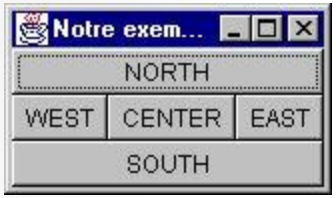
# BorderLayout

- Lors du redimensionnement, le composant est lui-même redimensionné en fonction de la taille de la zone, c-à-d :
  - les zones nord et sud sont éventuellement **élargies** **mais pas** **allongées**.
  - les zones est et ouest sont éventuellement **allongées** **mais pas** **élargies**,
  - la zone centrale est **étirée dans les deux sens**.

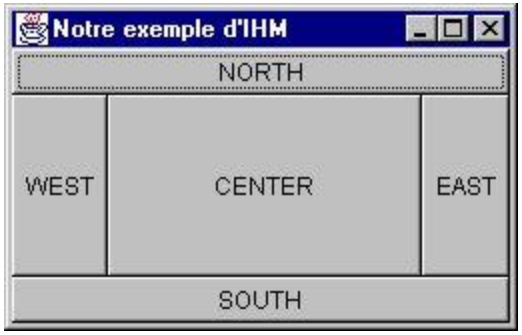
# BorderLayout



# BorderLayout



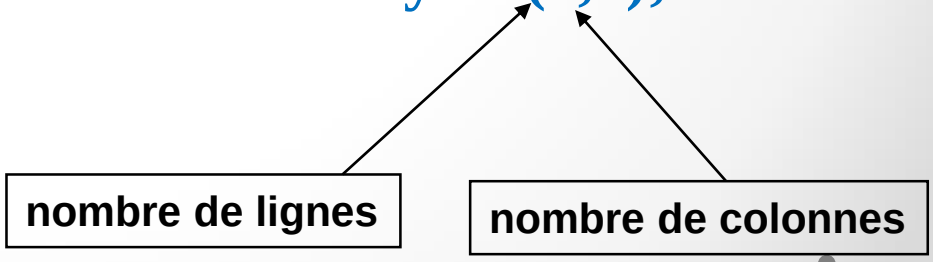

**Redimensionnement**




**Redimensionnement**

# GridLayout

- Le **GridLayout** dispose les composants dans une grille.
  - Découpage de la zone d'affichage en lignes et en colonnes qui définissent des cellules de dimensions égales.
  - Chaque composant à la même taille
    - quand ils sont ajoutés dans les cellules le remplissage s'effectue de gauche à droite et de haut en bas.
  - Les 2 paramètres sont les rangées et les colonnes.
  - Construction d'un **GridLayout** : **new GridLayout(3,2);**



# GridLayout

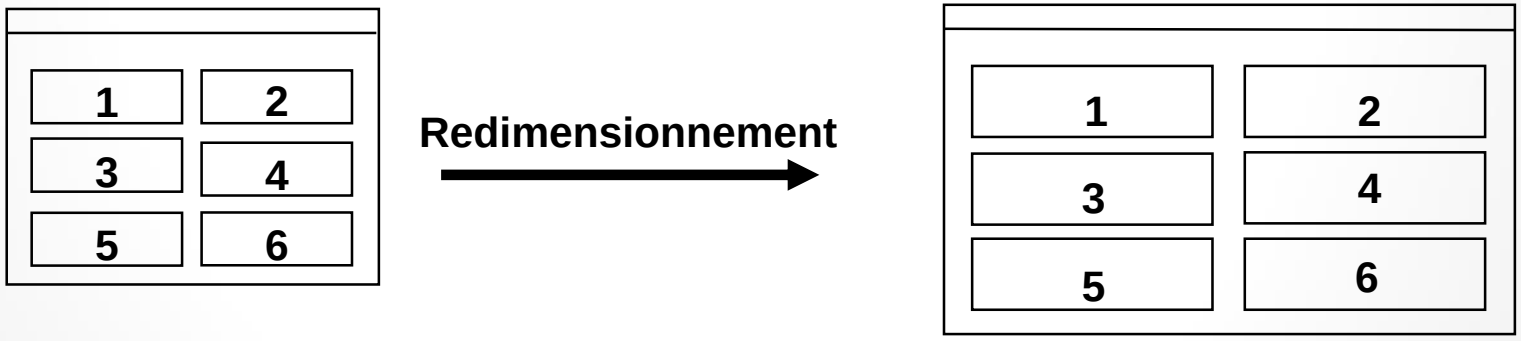
Exemple

```
import java.awt.*;

public class AppliGridLayout extends Frame
{
public AppliGridLayout()
 {
  super("AppliGridLayout");
  this.setLayout(new GridLayout(3,2));
  for (int i = 1; i < 7; i++)
    add(new Button(Integer.toString(i)));
  this.pack();
  this.show();
 }

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

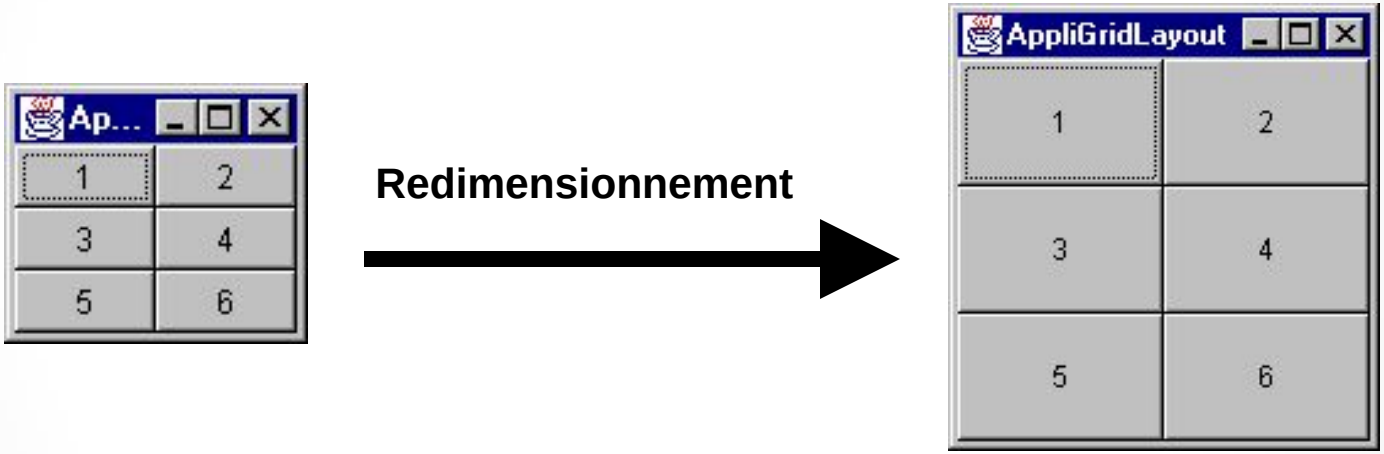

Résultat à l'exécution :

# GridLayout

- Lors d'un redimensionnement les composants changent tous de taille mais leurs positions relatives ne changent pas.



# GridLayout



# CardLayout

- Le **CardLayout** n'affiche qu'un composant à la fois :
  - les composants sont considérées comme empilées, à la façon d'un tas de cartes.
- La présentation **CardLayout** permet à plusieurs composants de partager le même espace d'affichage de telle sorte que seul l'un d'entre eux soit visible à la fois.
- Pour ajouter un composant à un conteneur utilisant un CardLayout il faut utiliser add(String cle, Component monComposant)
- Permet de passer de l 'affichage d 'un composant à un autre en appelant les méthodes **first, last, next, previous** ou **show**

# GridBagLayout

- Le gestionnaire **GridBagLayout** fournit des fonctions de présentation complexes
  - basées sur une grille dont les lignes et les colonnes sont de taille variables.
  - permet à des composants simples de prendre leur taille préférée au sein d'une cellule, au lieu de remplir toute la cellule.
  - permet aussi l'extension d'un même composant sur plusieurs cellules.
- Le GridBagLayout est compliqué à gérer.
  - Dans la plupart des cas, il est possible d'éviter de l'utiliser en associant des objets **Container** utilisant des gestionnaires différents.

# GridBagLayout

- Le gestionnaire **GridBagLayout** est associé à un objet **GridBagConstraints**
  - l'objet **GridBagConstraints** définit des contraintes de positionnement, d'alignements, de taille, etc. d'un composant dans un conteneur géré par un **GridBagLayout**.
  - On associe chaque composant que l'on place dans le **GridBagLayout** avec un objet **GridBagConstraints**
    - Un même objet **GridBagConstraints** peut-être associé à plusieurs composants.
    - Définir les objets **GridBagConstraints** en spécifiant les différents paramètres est assez fastidieux…
    - Voir la doc

# Mise en forme complexe

```
super("AppliComplexeLayout");
setLayout(new BorderLayout());
Panel pnorth = new Panel();
pnorth.add(b1); pnorth.add(b2);
pnorth.add(b3); pnorth.add(b4);
this.add(pnorth,BorderLayout.NORTH);

Panel pcenter = new Panel();
pcenter.setLayout(new GridLayout(2,2));
pcenter.add(gr1); pcenter.add(gr2);
pcenter.add(gr3); pcenter.add(gr4);
this.add(pcenter,BorderLayout.CENTER);

Panel psouth = new Panel();
psouth.setLayout(new FlowLayout());
psouth.add(ch); psouth.add(tf);
this.add(psouth, BorderLayout.SOUTH);
```

# D'autres gestionnaires?

- On peut imposer à un objet « container » de n'avoir pas de gestionnaire en fixant son **LayoutManager** à la valeur null
  - **Frame f = new Frame(); f.setLayout(null);**
  - A la charge alors du programmeur de positionner chacun des composants « manuellement » en indiquant leur position absolue dans le repère de la fenêtre.
  - C'est à éviter, sauf dans des cas particuliers,.
- Il est possible d'écrire ses propres **LayoutManager**…

# Récapitulatif

- **FlowLayout**
  - Flux : composants placés les uns derrière les autres
- **BorderLayout**
  - Ecran découpé en 5 zones (« North », « West », « South », « East », « Center »)
- **GridLayout**
  - Grille : une case par composant, chaque case de la même taille
- **CardLayout**
  - « Onglets » : on affiche un élément à la fois
- **GridBagLayout**
  - Grille complexe : plusieurs cases par composant

# Les événements graphiques

- L'utilisateur effectue
  - une action au niveau de l'interface utilisateur (clic souris, sélection d'un item, etc)
  - alors un **événement graphique** est émis.
- Lorsqu'un événement se produit
  - il est reçu par le composant avec lequel l'utilisateur interagit (par exemple un bouton, un curseur, un champ de texte, etc.).
  - Ce composant transmet cet événement à un autre objet, un **écouteur** qui possède une méthode pour traiter l'événement
    - cette méthode reçoit l'objet événement généré de façon à traiter l'interaction de l'utilisateur.

# Les événements graphiques

- La gestion des événements passe par l'utilisation d'objets "**écouteurs d'événements**" (les ***Listener***) et d'objets sources d'événements.
  - Un objet écouteur est l'instance d'une classe implémentant l'interface ***EventListener*** (ou une interface “fille”).
  - Une source d'événements est un objet pouvant recenser des objets écouteurs et leur envoyer des objets événements.
- Lorsqu'un événement se produit,
  - la source d'événements envoie un objet événement correspondant à tous ses écouteurs.
  - Les objets écouteurs utilisent alors l'information contenue dans l'objet événement pour déterminer leur réponse.

# Les événements graphiques

```
import java.awt.*;
import java.awt.event.*;

class MonAction implements ActionListener {
        public void actionPerformed (ActionEvent e)  {
                System.out.println ("Une action a eu lieu") ;}
}

public class TestBouton {
        public TestBouton(){
                Frame f = new Frame ("TestBouton");
                Button b = new Button ("Cliquer ici");
                f.add (b) ;
                f.pack (); f.setVisible (true) ;
                b.addActionListener (new MonAction ());}

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

# Les événements graphiques

- Les **écouteurs** sont des **interfaces**
- Donc une même classe peut implémenter plusieurs interfaces écouteur.

  - Par exemple une classe héritant de Frame implémentera les interfaces **MouseMotionListener** (pour les déplacements souris) et **MouseListener** (pour les clics souris).

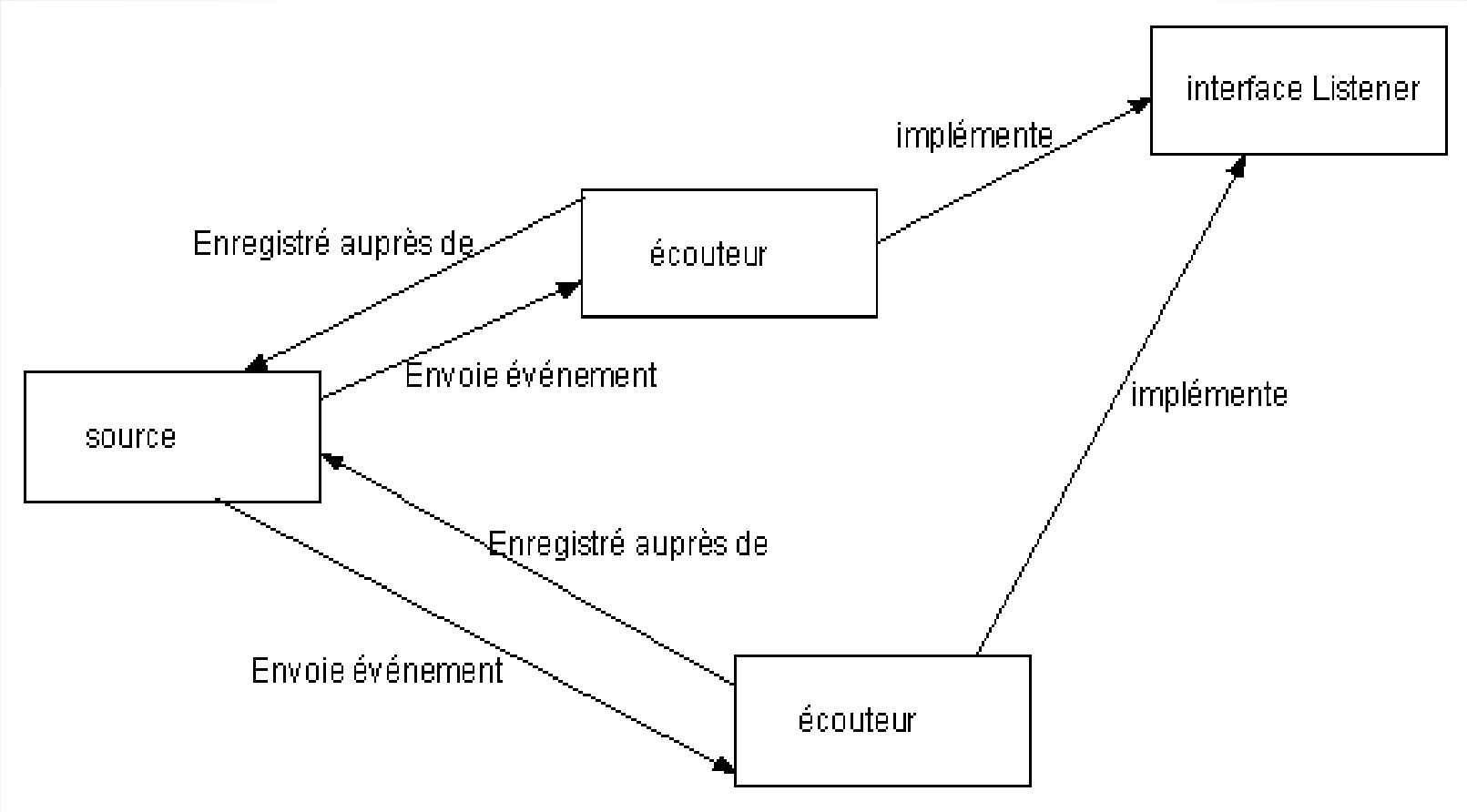
- Chaque composant de l'AWT est conçu pour être la source d'un ou plusieurs types d'événements particuliers.

  - Cela se voit notamment grâce à la présence dans la classe de composant d'une méthode nommée **addXXXListener().**

# Les événements graphiques

- L'objet événement envoyé aux écouteurs et passé en paramètres des fonctions correspondantes peut contenir des paramètres intéressants pour l'application.
  - Par exemple,  getX() et getY() sur un MouseEvent retournent les coordonnées de la position du pointeur de la souris.
  - Une information généralement utile quelques soit le type d'événement est la source de cet événement que l'on obtient avec la méthode getSource().

# Les événements graphiques



# Catégories d'événements graphiques

- Plusieurs types d'événements sont définis dans le package java.awt.event.

- Pour chaque catégorie d'événements, il existe une interface qui doit être définie par toute classe souhaitant recevoir cette catégorie d'événements.

  - Cette interface exige aussi qu'une ou plusieurs méthodes soient définies.
  - Ces méthodes sont appelées lorsque des événements particuliers surviennent.

# Catégories d'événements graphiques

| Catégorie | Nom de l'interface | Méthodes |
|---|---|---|
| Action | ActionListener | actionPerformed (ActionEvent) |
| Item | ItemListener | itemStateChanged (ItemEvent) |
| Mouse | MouseMotionListener | mouseDragged (MouseEvent)<br>mouseMoved (MouseEvent) |
| Mouse | MouseListener | mousePressed (MouseEvent)<br>mouseReleased (MouseEvent)<br>mouseEntered (MouseEvent) (MouseEvent)<br>mouseExited<br>mouseClicked |
| Key | KeyListener | keyPressed (KeyEvent)<br>keyReleased (KeyEvent)<br>keyTyped (KeyEvent) |
| Focus | FocusListener | focusGained (FocusEvent)<br>focusLost (FocusEvent) |

# Catégories d'événements graphiques

| | | |
|---|---|---|
| Adjustment | AdjustmentListener | adjustmentValueChanged (AdjustmentEvent) |
| Component | ComponentListener | componentMoved (ComponentEvent)componentHiddent (ComponentEvent)componentResize (ComponentEvent)componentShown (ComponentEvent) |
| Window | WindowListener | windowClosing (WindowEvent)<br>windowOpened (WindowEvent)<br>windowIconified (WindowEvent<br>windowDeiconified (WindowEvent)<br>windowClosed (WindowEvent)<br>windowActivated (WindowEvent)<br>windowDeactivated (WindowEvent) |
| Container | ContainerListener | componentAdded (ContainerEvent)<br>componentRemoved(ContainerEvent) |
| Text | TextListener | textValueChanged (TextEvent) |

# Catégories d'événements graphiques

- ActionListener
  - Action (clic) sur un bouton, retour chariot dans une zone de texte, « tic d'horloge » (Objet Timer)
- WindowListener
  - Fermeture, iconisation, etc. des fenêtres
- TextListener
  - Changement de valeur dans une zone de texte
- ItemListener
  - Sélection d'un item dans une liste

# Catégories d'événements graphiques

- MouseListener
  - Clic, enfoncement/relâchement des boutons de la souris, etc.
- MouseMotionListener
  - Déplacement de la souris, drag&drop avec la souris, etc.
- AdjustmentListener
  - Déplacement d'une échelle
- ComponentListener
  - Savoir si un composant a été caché, affiché …
- ContainerListener
  - Ajout d'un composant dans un Container

# Catégories d'événements graphiques

- FocusListener
  - Pour savoir si un élément a le "focus"
- KeyListener
  - Pour la gestion des événements clavier

# Catégories d'événements graphiques

```
import java.awt.*;
import java.awt.event.*;
public class EssaiActionEvent1 extends Frame
                        implements ActionListener
{
  public static void main(String args[])
  {EssaiActionEvent1 f= new EssaiActionEvent1();}
  public EssaiActionEvent1()
  {
   super("Utilisation d'un ActionEvent");
   Button b = new Button("action");
   b.addActionListener(this);
   add(BorderLayout.CENTER,b);pack();show();
  }
  public void actionPerformed( ActionEvent e )
 {
   setTitle("bouton cliqué !");
 }}
```


Implémentation de l'interface ActionListener

On enregistre l'écouteur d'evt action auprès de l'objet source "b"

Lorsque l'on clique sur le bouton dans l 'interface, le titre de la fenêtre change

# Catégories d'événements graphiques

```
public class EssaiActionEvent2 extends Frame
                        implements ActionListener
{ private Button b1,b2;
  public static void main(String args[])
  {EssaiActionEvent2 f= new EssaiActionEvent2();}
  public EssaiActionEvent2(){
   super("Utilisation d'un ActionEvent");
   b1 = new Button("action1");
   b2 = new Button("action2");
   b1.addActionListener(this);
   b2.addActionListener(this);
   add(BorderLayout.CENTER,b1);
   add(BorderLayout.SOUTH,b2);
   pack();show(); }
  public void actionPerformed( ActionEvent e ) {
   if (e.getSource() == b1) setTitle("action1 cliqué");
   if (e.getSource() == b2) setTitle("action2 cliqué");
  }}
```


Les 2 boutons ont le même écouteur (la fenêtre)

e.getSource()" renvoie l'objet source de l'événement. On effectue un test sur les boutons (on compare les références)

# Catégories d'événements graphiques

```
import java.awt.*; import java.awt.event.*;
public class WinEvt extends Frame
      implements WindowListener
{
 public static void main(String[] args) {
   WinEvt f= new WinEvt();}
   public WinEvt() {
     super("Cette fenêtre se ferme");
     addWindowListener(this);
     pack();show();}
    public void windowOpened(WindowEvent e){}
    public void windowClosing(WindowEvent e)
    {System.exit(0);}
    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){} }
```


Implémenter cette interface impose l'implémentation de bcp de méthodes

La fenêtre est son propre écouteur

WindowClosing() est appelé lorsque l'on clique sur la croix de la fenêtre

"System.exit(0)" permet de quitter une application java

# Les adapteurs

- Les classes « adapteur » permettent une mise en œuvre simple de l'écoute d'événements graphiques.
  - Ce sont des classes qui implémentent les écouteurs d'événements possédant le plus de méthodes, en définissant un corps vide pour chacune d'entre elles.
  - Plutôt que d'implémenter l'intégralité d'une interface dont une seule méthode est pertinente pour résoudre un problème donné, une alternative est de sous-classer l'adapteur approprié et de redéfinir juste les méthodes qui nous intéressent.
    - Par exemple pour la gestion des événements fenêtres...

# Les adapteurs (2)

**En implémentant l'interface**

```
class Test implements WindowListener
{
   public void windowClosing (WindowEvent e) {System.exit(0);}
   public void windowClosed (WindowEvent e) {}
   public void windowIconified (WindowEvent e) {}
   public void windowOpened (WindowEvent e) {}
   public void windowDeiconified (WindowEvent e) {}
   public void windowActivated (WindowEvent e) {}
   public void windowDeactivated (WindowEvent e) {}
}
```

**En utilisant un WindowAdapter**

```
class Test extends WindowAdapter
{
  public void windowClosing (WindowEvent e) {System.exit(0);}
}
```

# Les adapteurs

- Il existe 7 classes d'adapteurs (autant que d'interfaces d'écouteurs possédant plus d'une méthode) :
  - **ComponentAdapter**
  - **ContainerAdapter**
  - **FocusAdapter**
  - **KeyAdapter**
  - **MouseAdapter**
  - **MouseMotionAdapter**
  - **WindowAdapter**

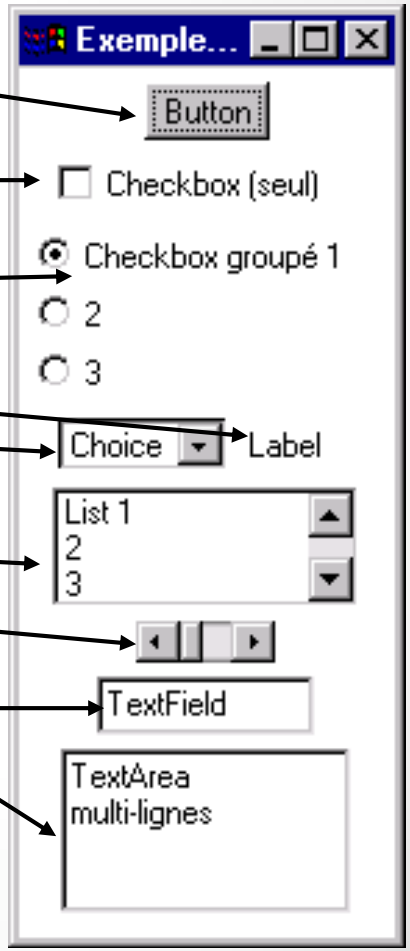
# Les adapteurs

- En pratique, et notamment avec la classe WindowAdapter, on utilise très souvent une classe anonyme

```
Frame fen = new Frame("Machin")
fen.addWindowListener(new WindowAdapter() {
                                public void windowClosing(WindowEvent e)
                                {
                                 System.exit(0);} });
```

# Les composants graphiques

- Button
- Canvas (zone de dessin)
- Checkbox (case à cocher)
- CheckboxGroup
- Label
- Choice (Sélecteur)
- List
- Scrollbar (barre de défilement)
- TextField (zone de saisie d'1 ligne)
- TextArea (zone de saisie multilignes)



# Les composants graphiques

- Button
  - C'est un composant d'interface utilisateur de base de type "appuyer pour activer".
  - Il peut être construit avec une étiquette de texte (un label) précisant son rôle à l'utilisateur.
  - Un objet de la classe Button est une source d'ActionEvent
  - Les écouteurs associés à des objets de la classe Button doivent implémenter interface ActionListener
  - Il n'y a qu'une méthode dans l'interface ActionListener, c 'est la méthode public void actionPerformed(ActionEvent e).

```
Button b = new Button ("Sample") ;
add (b) ;
b.addActionListener (...) ;
```

# Les composants graphiques

- CheckBox
  - La case à cocher fournit un dispositif d'entrée "actif / inactif" accompagné d'une étiquette de texte.
  - La sélection ou la déselection est notifiée par un ItemEvent à un écouteur implémentant l'interface ItemListener.
    - la méthode getStateChange() de ItemEvent retourne une constante : ItemEvent.DESELECTED ou ItemEvent.SELECTED.
    - le méthode getItem() de ItemEvent renvoie la chaîne contenant l'étiquette de la case à cocher considérée.

```
Checkbox one = new Checkbox("One", false);
add(one);
one.addItemListener(...);
```

# Les composants graphiques

- CheckboxGroup
  - On peut regrouper des cases à cocher dans un CheckboxGroup pour obtenir un comportement de type boutons radio
    - On ne peut sélectionner qu'une seule case du groupe de cases à cocher en même temps.
    - Sélectionner une case fera que toute autre case précédemment cochée sera désélectionnée

```
CheckboxGroup cbg = new CheckboxGroup();
Checkbox one = new Checkbox("One", cbg, false);
...
add(one);
...
```

# Les composants graphiques

- Choice
  - Ce composant propose une liste de choix.
    - On ajoute des éléments dans l'objet Choice avec la méthode addItem(String nomItem).
      - La chaîne passée en paramètre sera la chaîne visible dans la liste
    - On récupère la chaîne de caractère correspondant à l'item actuellement sélectionné avec la méthode String getSelectedItem()
    - Cet objet est source de ItemEvent, l'écouteur correspondant étant un ItemListener

```
Choice c = new Choice();
c.addItem("First");
c.addItem("Second");
...
c.addItemListener (...);
```

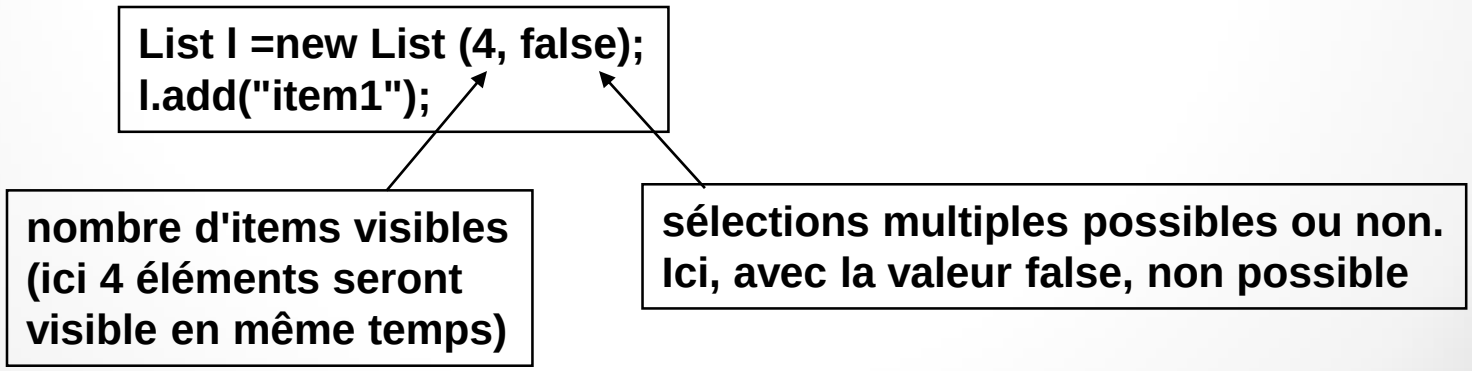
# Les composants graphiques

- Label
  - Un Label affiche une seule ligne de texte (étiquette) non modifiable.
  - En général, les étiquettes ne traitent pas d'événements.

```
Label l = new Label ("Bonjour !");
add(l);
```

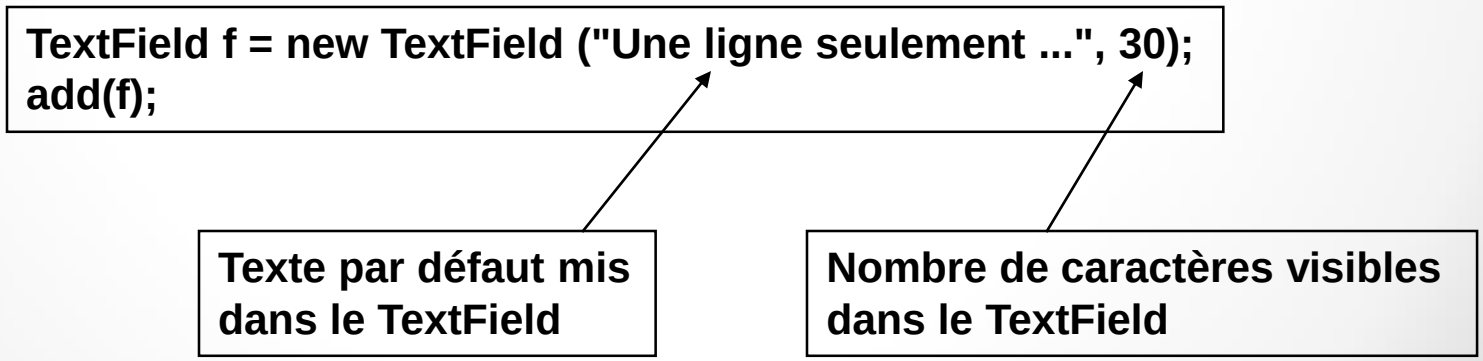
# Les composants graphiques

- List
  - Un objet de la classe List permet de présenter à l'utilisateur plusieurs options de texte parmi lesquelles il peut sélectionner un ou plusieurs éléments.
  - Source d'ActionEvent et d'ItemEvent
  - méthode String getSelectedItem() et String[] getSelectedItems() pour récupérer des items.



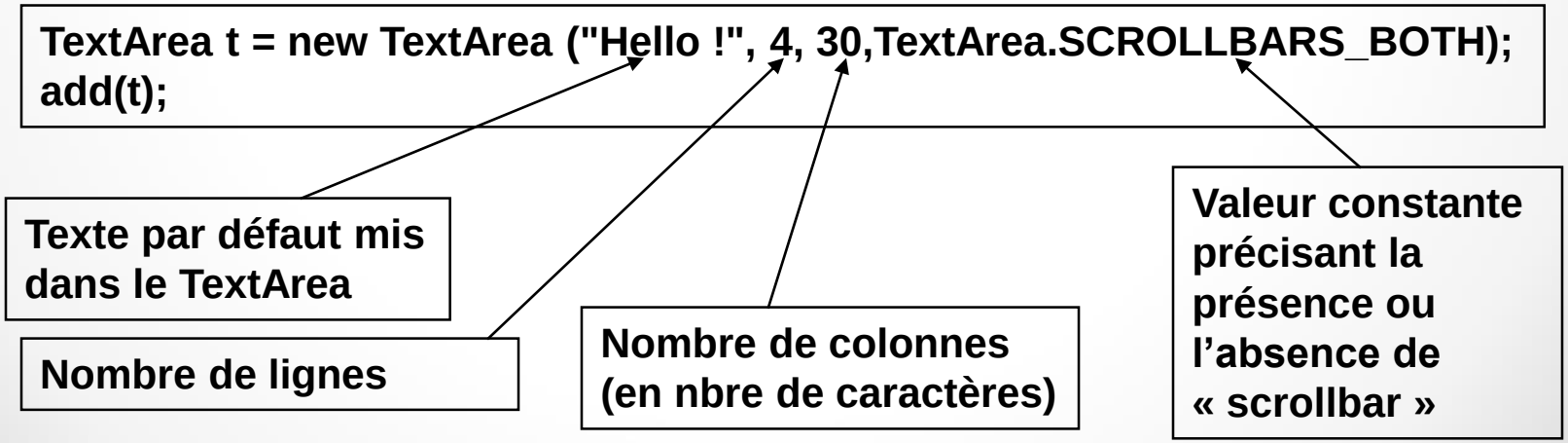
# Les composants graphiques

- TextField
  - Le champ de texte est un dispositif d'entrée de texte sur une seule ligne.
  - Il est source d'ActionEvent
  - On peut définir un champ de texte comme étant éditable ou non.
  - Méthodes void setText(String text) et String getText() pour mettre ou retirer du texte dans le TextField



# Les composants graphiques

- TextArea
  - La zone de texte est un dispositif d'entrée de texte multi-lignes, multi-colonnes avec éventuellement la présence ou non de « scrollbars » (barres de défilement) horizontal et/ou vertical.
  - Il peut être ou non éditable.
  - Méthode setText(), getText() et append() (pour ajouter du texte à la fin d 'un texte existant déjà dans le TextArea)



# Les composants graphiques

- Menu :
  - menu déroulant de base, qui peut être ajoutée à une barre de menus (MenuBar) ou à un autre menu.
  - Les éléments de ces menus sont
    - des MenuItem
      - ils sont rajoutés à un menu
      - En règle générale, ils sont associés à un ActionListener.
    - des CheckBoxMenuItem, ie. des éléments de menus à cocher
      - ils permettent de proposer des sélections de type "activé / désactivé " dans un menu.

- PopupMenu
  - des menus autonomes pouvant s'afficher instantanément sur un autre composant.
    - Ces menus doivent être ajoutés à un composant parent (par exemple un Frame), grâce à la méthode add(...) .
    - Pour afficher un PopupMenu, il faut utiliser la méthode show(...).

# Les composants graphiques

- PopupMenu
  - des menus autonomes pouvant s'afficher instantanément sur un autre composant.
    - Ces menus doivent être ajoutés à un composant parent (par exemple un Frame), grâce à la méthode add(...).
    - Pour afficher un PopupMenu, il faut utiliser la méthode show(...).

# Les composants graphiques

- Canvas
  - Il définit un espace vide
    - Sa taille par défaut est zéro par zéro (Ne pas oublier de la modifier avec un setSize(...) ) et il n'a pas de couleur.
      - pour forcer un canvas (ou tout autre composant) à avoir une certaine taille il faut redéfinir les méthodes getMinimumSize() et getPreferredSize().
  - On peut capturer tous les événements dans un Canvas.
    - Il peut donc être associé à de nombreux écouteurs : KeyListener, MouseMotionListener, MouseListener.
    - On l'utilise en général pour définir une zone de dessin

# Les composants graphiques


```
class Ctrait extends Canvas implements
MouseListener
{
  Point pt;
  public Ctrait() { addMouseListener(this); }
  public void paint(Graphics g)
  {g.drawLine(0,0,pt.x,pt.y);
    g.setColor(Color.red);

g.drawString("("+pt.x+";"+pt.y+")",pt.x,pt.y+5);}
 public Dimension getMinimumSize()
 {return new Dimension(200,100);}
 public Dimension getPreferredSize()
 {return getMinimumSize();}
 public void mouseClicked(MouseEvent e){}
 public void mousePressed(MouseEvent e){}
 public void mouseReleased(MouseEvent e)
 {pt=e.getPoint();repaint();}
 public void mouseEntered(MouseEvent e){}
 public void mouseExited(MouseEvent e){}}
```

```
import java.awt.*;
import java.awt.event.*;
public class Dessin extends Frame
{
  public static void main(String[] args)
  {
    Dessin f= new Dessin();
  }
  public Dessin()
  {
    super("Fenêtre de dessin");
    Ctrait c= new Ctrait();
    add(BorderLayout.CENTER,c);
    pack();
    show();
  }
}
```

# Les composants graphiques

- Contrôle des couleurs d'un composant
  - Deux méthodes permettent de définir les couleurs d'un composant
    - setForeground (Color c) : la couleur de l'*encre* avec laquelle on écrira sur le composant
    - setBackground (Color c) : la couleur du fond
  - Ces deux méthodes utilisent un argument instance de la classe java.awt.Color.
    - La gamme complète de couleurs prédéfinies est listée dans la page de documentation relative à la classe Color.
  - Il est aussi possible de créer une couleur spécifique (RGB)

```
int r = 255, g = 255, b = 0 ;
Color c = new Color (r, g, b) ;
```

# Les composants graphiques

- Contrôle des polices de caractères
  - La police utilisée pour afficher du texte dans un composant peut être définie avec setFont(...) avec comme argument une instance de java.awt.Font.
    - **Font f = new Font ("TimesRoman", Font.PLAIN, 14) ;**
  - Les constantes de style de police sont en réalité des valeurs entières, parmi celles citées ci-après :
    - **Font.BOLD**
    - **Font.ITALIC**
    - **Font.PLAIN**
    - **Font.BOLD + Font.ITALIC**
  - Les tailles en points doivent être définies avec une valeur entière.

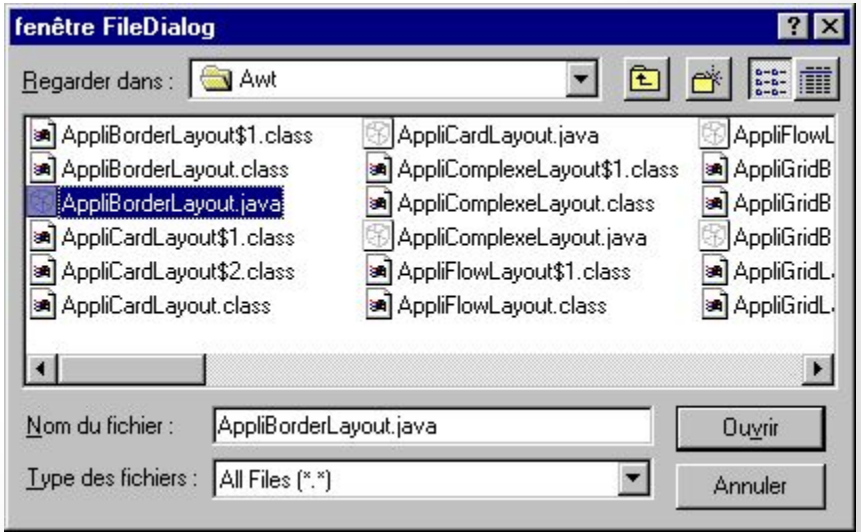
# Conteneurs particuliers

- Dialog
  - Un objet Dialog ressemble à un objet Frame mais ne sert qu'à afficher des messages devant être lus par l'utilisateur.
    - Il n'a pas de boutons permettant de le fermer ou de l'iconiser.
    - On y associe habituellement un bouton de validation.
    - Il est réutilisable pour afficher tous les messages au cours de l'exécution d'un programme.
  - Un objet Dialog dépend d'un objet Frame (ou héritant de Frame)
    - ce Frame est passé comme premier argument au constructeur).
  - Un Dialog n'est pas visible lors de sa création. Utiliser la méthode show() pour la rendre visible (il existe une méthode hide() pour la cacher).

# Conteneurs particuliers

- **FileDialog**
  - C'est une sous-classe de **Dialog** ; par défaut elle n'est pas visible.
  - C'est un dispositif de sélection de fichier : on peut préciser si c'est en vue d'une sauvegarde ou du chargement d'un fichier
  - Un FileDialog ne gère généralement pas d'événements.
  - Comme pour un objet Dialog, un FileDialog dépend d'un objet Frame
  - Un FileDialog est une fenêtre modale : à partir du moment ou la fenêtre a été rendue visible par la méthode show(...), la main n'est rendu à l'utilisateur que quand un fichier a été sélectionné.

**FileDialog**

# Swing

# Introduction à Swing

- La bibliothèque Swing est une nouvelle bibliothèque de composants graphiques pour Java.
  - Swing est intégré à Java 1.2.
  - Swing peut être téléchargé séparément pour une utilisation avec des versions de Java antérieures (1.1.5+)

- Cette bibliothèque s'ajoute à celle qui était utilisée jusqu'alors (AWT) pour des raisons de compatibilité.
  - Swing fait cependant double emploi dans beaucoup de cas avec AWT.
  - L'ambition de Sun est que, progressivement, les développeurs réalisent toutes leurs interfaces avec Swing et laissent tomber les anciennes API graphiques.

# Introduction à Swing

## Avantages et inconvénients :

- **Swing**, plus récent que AWT, est également plus optimisé, et donc plus rapide.

De son côté, AWT existe depuis plus longtemps, il est donc reconnu par plus de systèmes/outils.

- **Swing** prend en compte un plus grand nombre de widgets. De plus, étant le remplaçant officiel d'AWT, il est donc encore amélioré avec les nouvelles versions de Java.

# Introduction à Swing

- **AWT** garde encore pour lui son intégration à J2ME, et donc sa reconnaissance  au sein d'un plus grand nombre de téléphones.

Les inconvénients de Swing sont susceptibles de disparaître avec les évolutions de Java.

Existent également les gestionnaires **SWT** et JFace, créés par Eclipse pour palier les manques des gestionnaires de Sun pour le propre usage d'Eclipse.

**SWT** se veut plus léger que **AWT** et **Swing** réunis, tout en fournissant un large jeu de widgets et bibliothèques graphiques.

# Première fenêtre

**La classe Jframe**

- Pour créer une fenêtre graphique, on dispose, dans le paquetage nommé ***javax.swing***, d'une classe standard nommée *JFrame*, possédant un constructeur sans arguments.

Par exemple, avec :

**JFrame fen = new JFrame() ;**

- Il est en effet nécessaire de demander l'affichage de la fenêtre en appelant la méthode ***setVisible*** :

- **fen.setVisible (true) ; // rend visible la fenetre de reference fen**

```
fen.setSize (300, 150) ; // donne a la fenetre une hauteur de 150 pixels
                         // et une largeur de 300 pixels
```

# Première fenêtre

```
import javax.swing.* ;
public class Premfen0
{ public static void main (String args[])
{ JFrame fen = new JFrame() ;
fen.setSize (300, 150) ;
fen.setTitle ("Ma premiere fenetre") ;
fen.setVisible (true) ;
}
}
```

Importer le package

Créer un objet graphique

Définir la taille

afficher


*Utilisation d'une classe fenêtre personnalisée*

```
import javax.swing.* ;
class MaFenetre extends JFrame
{ public MaFenetre () // constructeur
{ setTitle ("Ma premiere fenetre") ;
setSize (300, 150) ;
}}
public class Premfen1
{ public static void main (String args[])
{ JFrame fen = new MaFenetre() ;
fen.setVisible (true) ;
}}
```

# Composants graphiques lourds

- Un composant graphique lourd (*heavyweight GUI component*) s'appuie sur le gestionnaire de fenêtres local, celui de la machine sur laquelle le programme s'exécute.
  - awt ne comporte que des composants lourds.
  - Ce choix technique a été initialement fait pour assurer la portabilité.

# Composants graphiques lourds

- Exemple :
  - Un bouton de type java.awt.Button intégré dans une application Java sur la plate-forme Unix est représenté grâce à un vrai bouton Motif (appelé son pair - *peer* en anglais).
  - Java communique avec ce bouton Motif en utilisant la Java Native Interface. Cette communication induit un coût.
  - C'est pourquoi ce bouton est appelé composant lourd.

# Composants légers de Swing

- Un composant graphique léger (en anglais, *lightweight GUI component*) est un composant graphique indépendant du gestionnaire de fenêtre local.
  - Un composant léger ressemble à un composant du gestionnaire de fenêtre local mais n'en est pas un : un composant léger émule les composants de gestionnaire de fenêtre local.
  - Un bouton léger est un rectangle dessiné sur une zone de dessin qui contient une étiquette et réagit aux événements souris.
  - Tous les composants de Swing, **exceptés** **JApplet**, **JDialog**, **JFrame** et **JWindow** sont des composants légers.

# Atouts de Swing

- Plus de composants, offrant plus de possibilités.
- Les composants Swing dépendent moins de la plate-forme :
  - Il est plus facile d'écrire une application qui satisfasse au slogan "*Write once, run everywhere*"
  - Swing peut pallier aux faiblesses (bogues ?) de chaque gestionnaire de fenêtre.

# Conventions de nommage

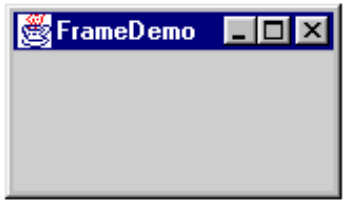
- Les composants Swing sont situés dans le paquetage javax.swing et ses sous paquetages.
  - Un certain nombre de paquetages d'extensions du langage java 1.1 (par opposition aux paquetages standards java) sont regroupés sous javax.
    - Cette convention permet de télécharger ces paquetages dans un environnement navigateur avec une machine virtuelle java 1.1. Ces navigateurs ne sont, en effet, pas autorisés à télécharger des paquetages dont le nom commence par **java**.
- Ils portent des noms similaires à leurs correspondants de AWT précédés d'un J.
  - JFrame, JPanel, JTextField, JButton, JCheckBox, JLabel, etc.

# Aperçu des composants Swing

**Top-Level Containers**

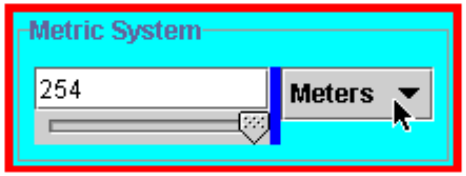
Applet


Dialog



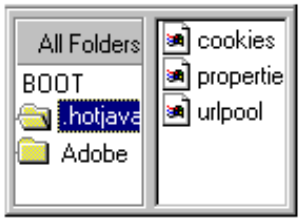

Frame

**General-Purpose Containers**

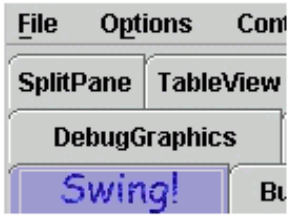



Panel


Scroll pane



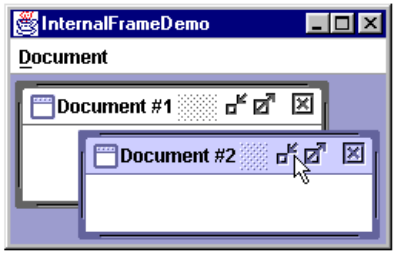

Split pane



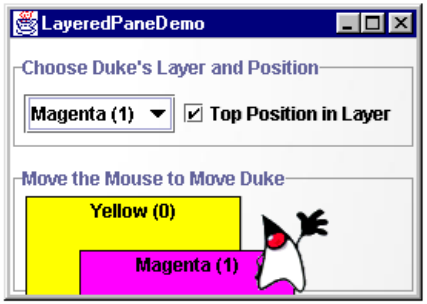

Tabbed pane

Tool bar

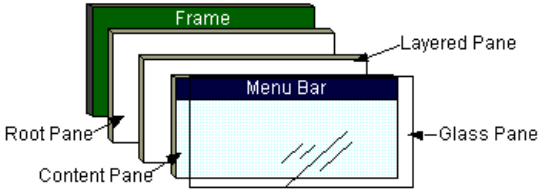
**Special-Purpose Containers**




Internal frame

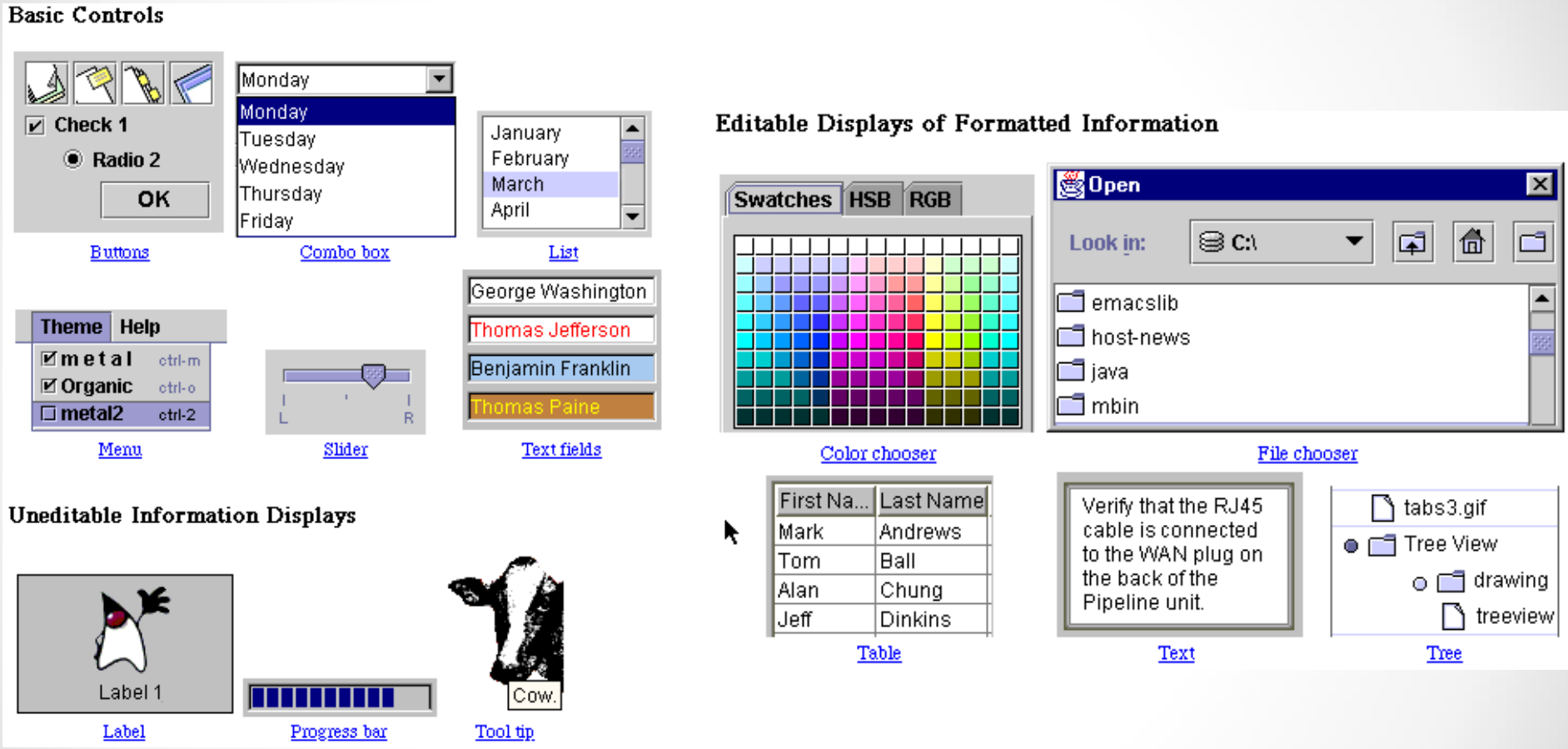



Layered pane




Root pane

# Aperçu des composants Swing



# Principaux paquetages Swing (1)

- javax.swing
  - le paquetage général
- javax.swing.border
  - pour dessiner des bordures autour des composants
- javax.swing.colorchooser
  - classes et interfaces utilisées par le composant JColorChooser
- javax.swing.event
  - les événements générés par les composants Swing
- javax.swing.filechooser
  - classes et interfaces utilisées par le composant JFileChooser

# Principaux paquetages Swing

- javax.swing.table
  - classes et interfaces pour gérer les Jtable
- javax.swing.text
  - classes et interfaces pour la gestion des composants « texte »
- javax.swing.tree
  - classes et interfaces pour gérer les Jtree
- javax.swing.undo
  - pour la gestion de undo/redo dans une application

# JFrame

- Ancêtre commun : classe **`JComponent`**
  - java.lang.Object
  - |
  - +--java.awt.Component
  - |
  - +--java.awt.Container
  - |
  - +--javax.swing.JComponent
- Le JFrame
  - Un objet JFrame a un comportement par défaut associé à une tentative de fermeture de la fenêtre.
    - Contrairement à la classe Frame, qui ne réagissait pas par défaut, l'action de fermeture sur un JFrame rend par défaut la fenêtre invisible.
    - Ce comportement par défaut peut être modifié par setDefaultCloseOperation().

# JFrame

- 4 comportements sont possibles lors de la fermeture de la fenètre
- **DO_NOTHING_ON_CLOSE**
- **HIDE_ON_CLOSE**
- **DISPOSE_ON_CLOSE**
- **EXIT_ON_CLOSE**

```
import javax.swing.*;
public class Simple {
public static void main(String[] args) {
JFrame cadre = new JFrame("Ma fenetre");
cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
cadre.setSize(300, 200);
cadre.setVisible(true);
}
}
```

détermine que l'application sera terminée
Lorsqu'on fermera la fenêtre

# JFrame

- Le conteneur d'un JFrame n'est plus confondu avec le cadre lui-même.
  - Il possède une couche de contenu, dans laquelle on peut ajouter les composants graphiques et dont on peut changer le gestionnaire de présentation.
  - Cette couche est obtenue par la méthode **getContentPane();**

**Avec AWT**

```
Frame monFrame = new Frame ("Mon Frame");
Button monBouton = new Button ("Mon Bouton");
monFrame.add(monBouton);
```

**Avec Swing**

```
JFrame monFrame = new JFrame ("Mon Frame");
JButton monBouton = new JButton ("Mon Bouton");
Container panneauContenu = monFrame.getContentPane();
panneauContenu.add(monBouton);
```

# Quelques composants Swing

- La classe **ImageIcon** et l'interface **Icon**
  - Les objets de cette classe permettent de définir des icônes à partir d'images (gif, jpg, etc.) qui peuvent être ajoutés au classique texte d'un **JLabel**, d'un JButton, etc.

```
Icon monIcone = new ImageIcon("Image.gif");

// Un JLabel
JLabel monLabel = new JLabel("Mon Label");
monLabel.setIcon(monIcone);
monLabel.setHorizontalAlignment(JLabel.RIGHT);

// Un JButton
JButton monBouton = new JButton("Mon bouton", monIcone);
```

# Quelques composants Swing

- JTextPane
  - un éditeur de texte qui permet la gestion de texte formaté, le retour à la ligne automatique (word wrap), l'affichage d'images.
- JPasswordField
  - un champ de saisie de mots de passe : la saisie est invisible et l'affichage de chaque caractère tapé est remplacé par un caractère d'écho (* par défaut).
- JEditorPane
  - un JTextComponent pour afficher et éditer du code  HTML 3.2 et des formats tels que RTF.

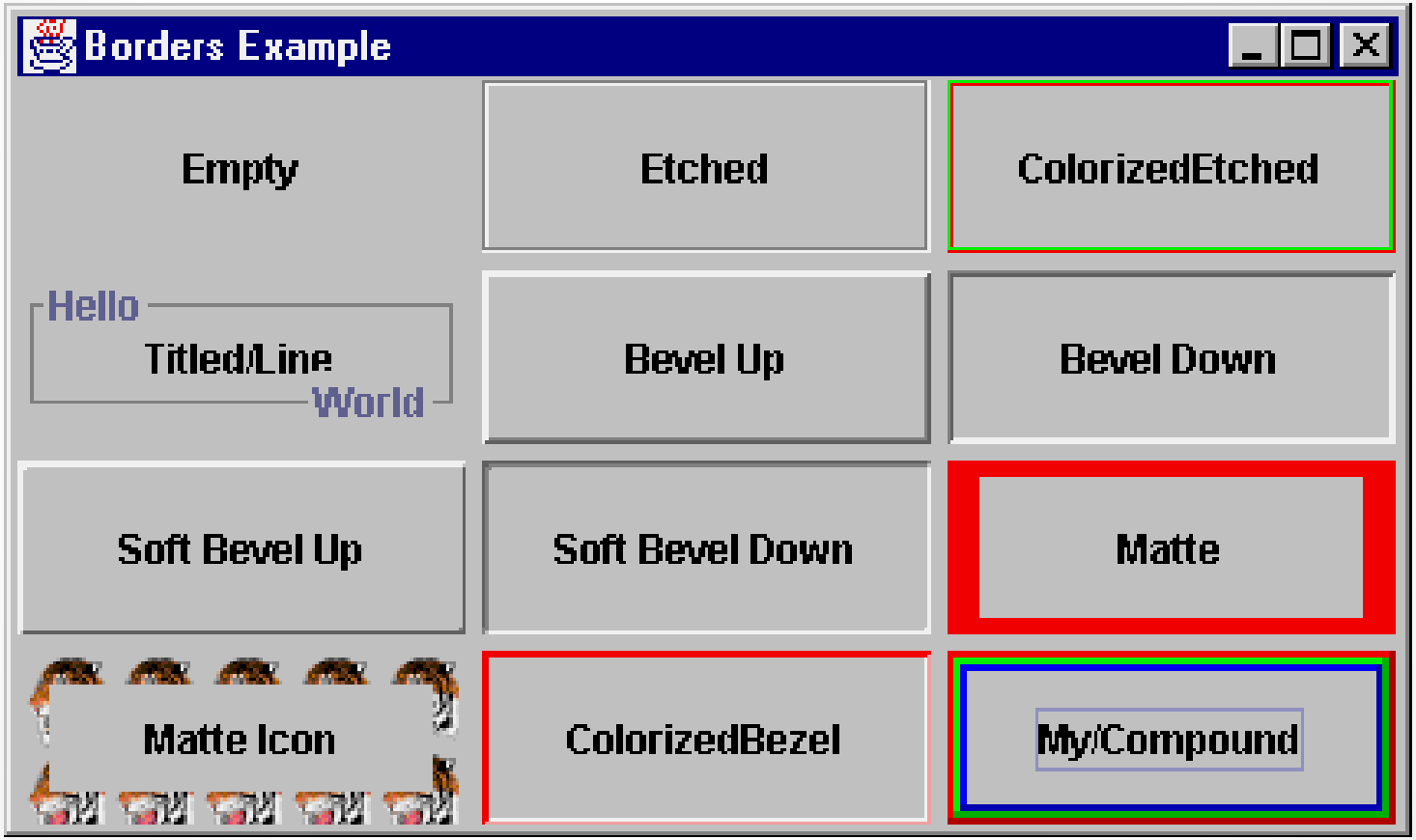
# Exemple de JEditorPane

```
import javax.swing.*;
import java.awt.*;
public class EditeurWEB {
  public static void main(String[] args)
  {
JFrame monFrame = new JFrame ("Mon Frame");
monFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
try {
   JEditorPane monEditeur = new
                    JEditorPane("http://www.google.fr");
   Container panneauContenu = monFrame.getContentPane();
   panneauContenu.setLayout(new FlowLayout());
   panneauContenu.add(monEditeur);
   monFrame.pack();
   monFrame.show();}
catch (Exception e) {System.out.println(e.getMessage());};
  }  }
```

# Quelques composants Swing

- Des bordures peuvent être dessinées autour de tous composants graphiques. Swing en définit 9 types :
    - **AbstractBorder** : ne fait rien
    - **BevelBorder** : une bordure 3D en surépaisseur ou en creux
    - **CompoundBorder** : permet de composer des plusieurs bordures
    - **EmptyBorder**
    - **EtchedBorder**
    - **LineBorder** : bordures d'une seule couleur
    - **MatteBorder**
    - **SoftBevelBorder** : une bordure 3D aux coins arrondis
    - **TitledBorder** : une bordure permettant l'inclusion d'une chaîne de caractères

# Quelques composants Swing



# Quelques composants Swing

- **JLayeredPane**
  - un conteneur qui permet de ranger ses composants en couches (ou transparents).
  - Cela permet de dessiner les composants, selon un certain ordre: premier plan, plan médian, arrière plan, etc.
  - Pour ajouter un composant, il faut spécifier la couche sur laquelle il doit être dessiné

**monJlayeredPane.add (monComposant, new Integer(5));**

- des « layer » sont définis en natif et des constantes pour y accéder existent dans la classe.

# Quelques composants Swing

- **ToolTipText**
  - permet de créer des aides en lignes qui apparaissent lorsque la souris passe sur un composant.

```
JButton monBouton = new JButton ("Un bouton");
monBouton.setToolTipText ("Aide de mon bouton");
```

- Représenter des listes : classe **`JList`**
- Représenter des arbres : classe **`JTree`**
- Représenter des tables (Excel) : classe **`JTable`**
- Ces quelques nouveautés ne sont qu'un aperçu de ce que propose Swing.

Il y a beaucoup de composants, de nouveaux gestionnaires de présentation, de nouveaux événements graphiques que l'on ne peut présenter et détailler dans le cadre de ce cours.

# Gestion d'un clic dans la fenêtre

Implémentation de l'interface MouseListener

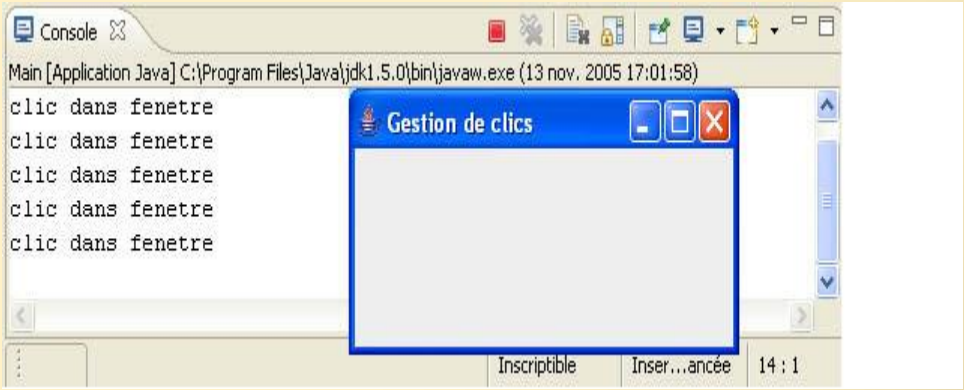
Pour traiter un événement, on associe à la source un objet de son choix dont la classe implémente une interface particulière correspondant à une ***catégorie d'événements***. On dit que cet objet est un ***écouteur*** de cette catégorie d'événements. Chaque méthode proposée par **l'interface** correspond à un événement de la catégorie

un objet d'une classe implémentant l'interface ***MouseListener***. Cette dernière comporte cinq méthodes correspondant chacune à un événement particulier : ***mousePressed***, ***mouseReleased***, ***mouseEntered***, ***mouseExited*** **et** ***mouseClicked*****.**

```
class EcouteurSouris implements MouseListener
{
public void mouseClicked (MouseEvent ev) { ..... }
public void mousePressed (MouseEvent ev) { ..... }
public void mouseReleased(MouseEvent ev) { ..... }
public void mouseEntered (MouseEvent ev) { ..... }
public void mouseExited (MouseEvent ev) { ..... }
// autres methodes et champs de la classe
}
```

# Gestion d'un clic dans la fenêtre

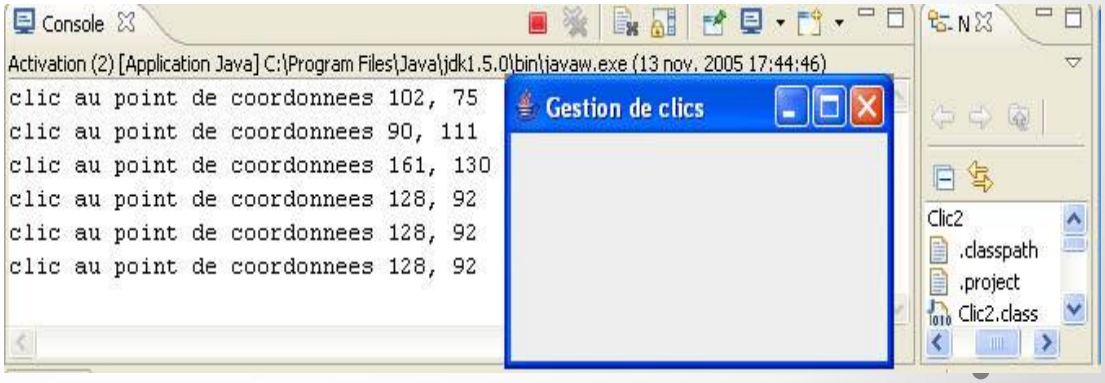
```
import javax.swing.* ; // pour JFrame
import java.awt.event.* ; // pour MouseEvent et MouseListener

class MaFenetre extends JFrame implements MouseListener

{ public MaFenetre ()        // constructeur
{ setTitle ("Gestion de clics") ;
setBounds (10, 20, 300, 200) ;
addMouseListener (this) ;        // la fenêtre sera son propre écouteur d'evenements souris
}
public void mouseClicked(MouseEvent ev)        // méthode gérant un clic souris
{ System.out.println ("clic dans fenetre") ;
}
public void mousePressed (MouseEvent ev) {}
public void mouseReleased(MouseEvent ev) {}
public void mouseEntered (MouseEvent ev) {}
public void mouseExited (MouseEvent ev) {}
}
public class Clic1
{ public static void main (String args[])
{ MaFenetre fen = new MaFenetre() ;
fen.setVisible(true) ;
}}
```




Gestion de l'événement "clic dans la fenêtre"

# Gestion de l'événement "clic dans la fenêtre" avec affichage des coordonnées du clic

```
import javax.swing.* ;
import java.awt.event.* ;
class MaFenetre extends JFrame implements MouseListener
{ MaFenetre () // constructeur
{ setTitle ("Gestion de clics") ;
setBounds (10, 20, 300, 200) ;
addMouseListener (this) ;                    //la fenetre sera son propre ecouteur d'evenements souris
}
public void mouseClicked(MouseEvent ev)          // methode gerant un clic souris
{ int x = ev.getX() ;
int y = ev.getY() ;
System.out.println ("clic au point de coordonnees " + x + ", " + y ) ;
}
public void mousePressed (MouseEvent ev) {}
public void mouseReleased(MouseEvent ev) {}
public void mouseEntered (MouseEvent ev) {}
public void mouseExited (MouseEvent ev) {}
}
public class Clic2
{ public static void main (String args[])
{ MaFenetre fen = new MaFenetre() ;
fen.setVisible(true) ;
}
}
```



# Gestion de plusieurs composants

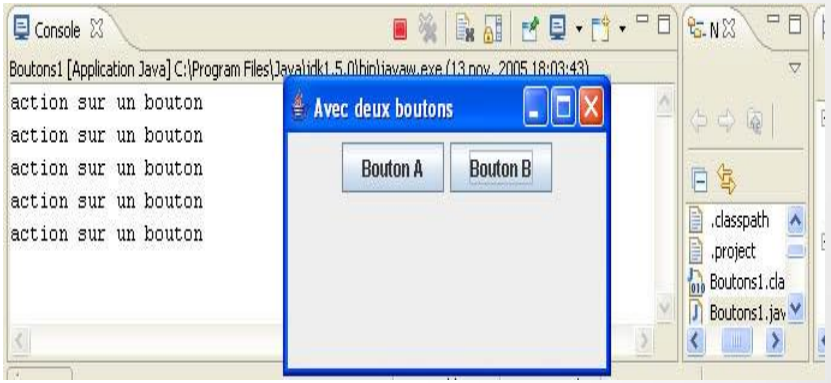
```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class Fen2Boutons extends JFrame implements ActionListener
{ public Fen2Boutons ()
{ setTitle ("Avec deux boutons") ;
setSize (300, 200) ;
monBouton1 = new JButton ("Bouton A") ;
monBouton2 = new JButton ("Bouton B") ;
Container contenu = getContentPane() ;
contenu.setLayout(new FlowLayout()) ;
contenu.add(monBouton1) ;
contenu.add(monBouton2) ;
monBouton1.addActionListener(this); // la fenetre ecoute monBouton1
monBouton2.addActionListener(this); // la fenetre ecoute monBouton2
}
public void actionPerformed (ActionEvent ev) // gestion commune a
{ System.out.println ("action sur un bouton") ; // tous les boutons
}
private JButton monBouton1, monBouton2 ;
}
```

```
public class Boutons1
{ public static void main (String args[])
{ Fen2Boutons fen = new Fen2Boutons() ;
fen.setVisible(true) ;
}
}
```

# Gestion d'un clic dans la fenêtre

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class Fen2Boutons extends JFrame implements
ActionListener
{ public Fen2Boutons ()
{ setTitle ("Avec deux boutons") ;
setSize (300, 200) ;
monBouton1 = new JButton ("Bouton A") ;
monBouton2 = new JButton ("Bouton B") ;
Container contenu = getContentPane() ;
contenu.setLayout(new FlowLayout()) ;
contenu.add(monBouton1) ;
contenu.add(monBouton2) ;
monBouton1.addActionListener(this); // la fenetre ecoute
monBouton1
monBouton2.addActionListener(this); // la fenetre ecoute
monBouton2
}
public void actionPerformed (ActionEvent ev) // gestion
commune a
{ System.out.println ("action sur un bouton") ; // tous les
boutons
}
private JButton monBouton1, monBouton2 ;
}
```

```
public class Boutons1
{ public static void main (String args[])
{ Fen2Boutons fen = new Fen2Boutons() ;
fen.setVisible(true) ;
}}
```

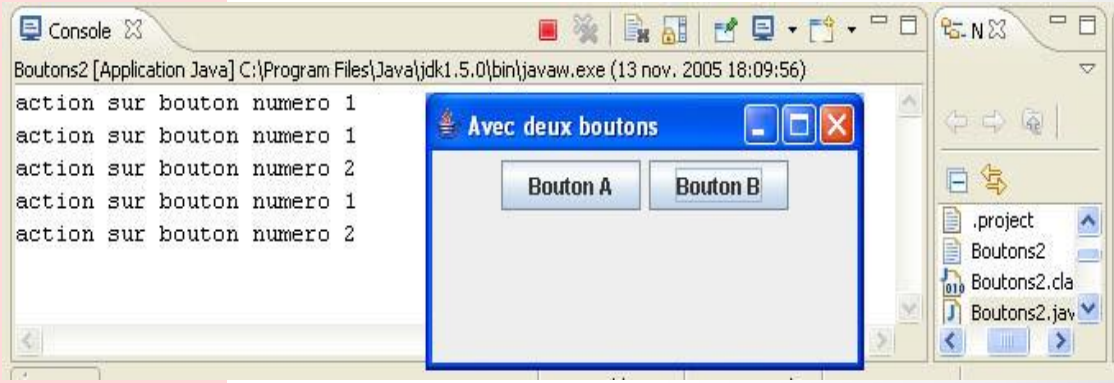


# Gestion d'un clic dans la fenêtre

## La méthode getSource

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class Fen2Boutons extends JFrame implements
ActionListener
{
public Fen2Boutons ()
{ setTitle ("Avec deux boutons") ;
setSize (300, 200) ;
monBouton1 = new JButton ("Bouton A") ;
monBouton2 = new JButton ("Bouton B") ;
Container contenu = getContentPane() ;
contenu.setLayout(new FlowLayout()) ;
contenu.add(monBouton1) ;
contenu.add(monBouton2) ;
monBouton1.addActionListener(this);
monBouton2.addActionListener(this);
}
public void actionPerformed (ActionEvent ev)
{ if (ev.getSource() == monBouton1)
System.out.println ("action sur bouton numero 1") ;
if (ev.getSource() == monBouton2)
System.out.println ("action sur bouton numero 2") ;
}
private JButton monBouton1, monBouton2 ;
}
```

```
public class Boutons2
{ public static void main (String args[])
{ Fen2Boutons fen = new Fen2Boutons() ;
fen.setVisible(true) ;
}
}
```

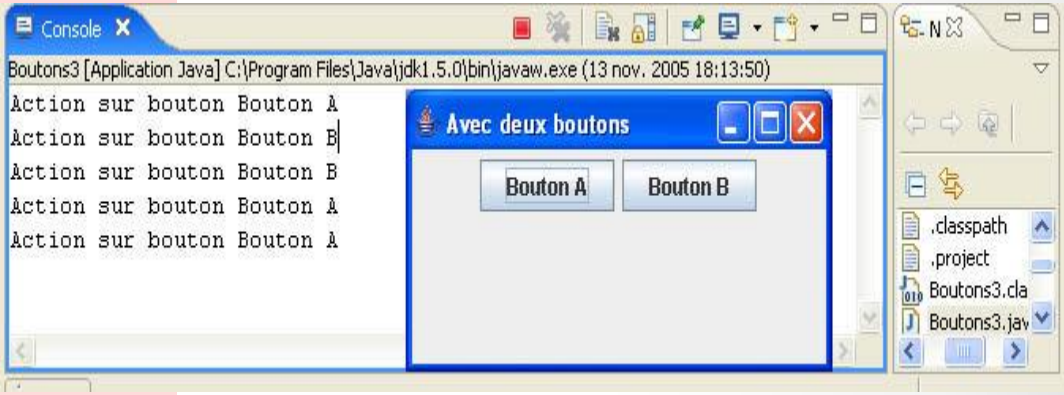


# Gestion d'un clic dans la fenêtre

## La méthode getActionCommand

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class Fen2Boutons extends JFrame implements
ActionListener
{ public Fen2Boutons ()
{ setTitle ("Avec deux boutons") ;
setSize (300, 200) ;
monBouton1 = new JButton ("Bouton A") ;
monBouton2 = new JButton ("Bouton B") ;
Container contenu = getContentPane() ;
contenu.setLayout(new FlowLayout()) ;
contenu.add(monBouton1) ;
contenu.add(monBouton2) ;
monBouton1.addActionListener(this);
monBouton2.addActionListener(this);
}
public void actionPerformed (ActionEvent ev)
{ String nom = ev.getActionCommand() ;
System.out.println ("Action sur bouton " + nom) ;
}
private JButton monBouton1, monBouton2 ;
}
```

```
public class Boutons3
{ public static void main (String args[])
{ Fen2Boutons fen = new Fen2Boutons() ;
fen.setVisible(true) ;
}
}
```




La méthode ***getActionCommand*** permet d'obtenir la chaîne de commande associée à la source d'un événement: c'est-à-dire une chaîne de caractères (*String*) associée à l'action

# Dynamique des composants

Dans les exemples précédents, les boutons étaient créés en même temps que la fenêtre et ils restaient affichés en permanence. Il en ira souvent ainsi. Néanmoins, il faut savoir qu'on peut, à tout instant :

- **créer** un nouveau composant,

- **supprimer** un composant,

- **désactiver** un composant, c'est-à-dire faire en sorte qu'on ne puisse plus agir sur lui ; dans le cas d'un bouton, cela revient à le rendre inopérant,

- **réactiver** un composant désactivé.

# Dynamique des composants

On crée un nouveau composant comme nous avons déjà appris à le faire :

création de l'objet et ajout au contenu de la fenêtre par la méthode ***add***.

Cependant, si cette opération est effectuée après l'affichage de la fenêtre, il faut forcer le gestionnaire de mise en forme à recalculer les positions des composants dans la fenêtre, de l'une des façons suivantes:

- en appelant la méthode ***revalidate*** pour le composant concerné,
- en appelant la méthode ***validate*** pour son conteneur.

# Dynamique des composants

On supprime un composant avec la méthode ***remove*** de son conteneur. Là encore, un appel à ***validate*** est nécessaire (ici, il n'est plus possible d'appeler ***revalidate*** pour le composant qui n'existe plus).

L'activation d'un composant de référence *compo* se fait simplement par :
compo.setEnabled (false) ; // le composant est desactive

La réactivation du même composant se fait par :
compo.setEnabled (true) ; // le composant est reactive
On peut savoir si un composant donné est activé ou non à l'aide de :
compo.isEnabled() ; // true si le composant est active

Notez bien que toutes ces opérations s'appliquent à n'importe quel composant, et pas seulement aux boutons1.

# Dynamique des composants

L'activation d'un composant de référence *compo* se fait simplement par :

**compo.setEnabled (false)** ; // le composant est desactive

La réactivation du même composant se fait par :
**compo.setEnabled (true)** ; // le composant est reactive

On peut savoir si un composant donné est activé ou non à l'aide de :
**compo.isEnabled()** ; // true si le composant est active

Notez bien que toutes ces opérations s'appliquent à n'importe quel composant, et pas seulement  aux boutons.

# Dynamique des composants

Exemple

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class FenBoutonsDyn extends JFrame
{ public FenBoutonsDyn ()
{ setTitle ("Boutons dynamiques") ;
setSize (500, 150) ;
Container contenu = getContentPane() ;
contenu.setLayout (new FlowLayout()) ;
crBouton = new JButton ("CREATION BOUTON") ;
contenu.add(crBouton) ;
EcoutCr ecoutCr = new EcoutCr (contenu) ;
crBouton.addActionListener (ecoutCr) ;
}
private JButton crBouton ;
}
class EcoutCr implements ActionListener
{ public EcoutCr (Container contenu)
{ this.contenu = contenu ;
}
public void actionPerformed (ActionEvent ev)
{ JButton nouvBout = new JButton ("BOUTON") ;
contenu.add(nouvBout) ;
contenu.validate(); // pour recalculer
}
private Container contenu ;
}
```

```
public class BoutDy0
{ public static void main (String args[])
{ FenBoutonsDyn fen = new
FenBoutonsDyn () ;
fen.setVisible (true) ;
}}
```


*Création dynamique de boutons dans une fenêtre*

# Merci de votre attention