Programmation événementielle & interfaces graphiques Java Swing

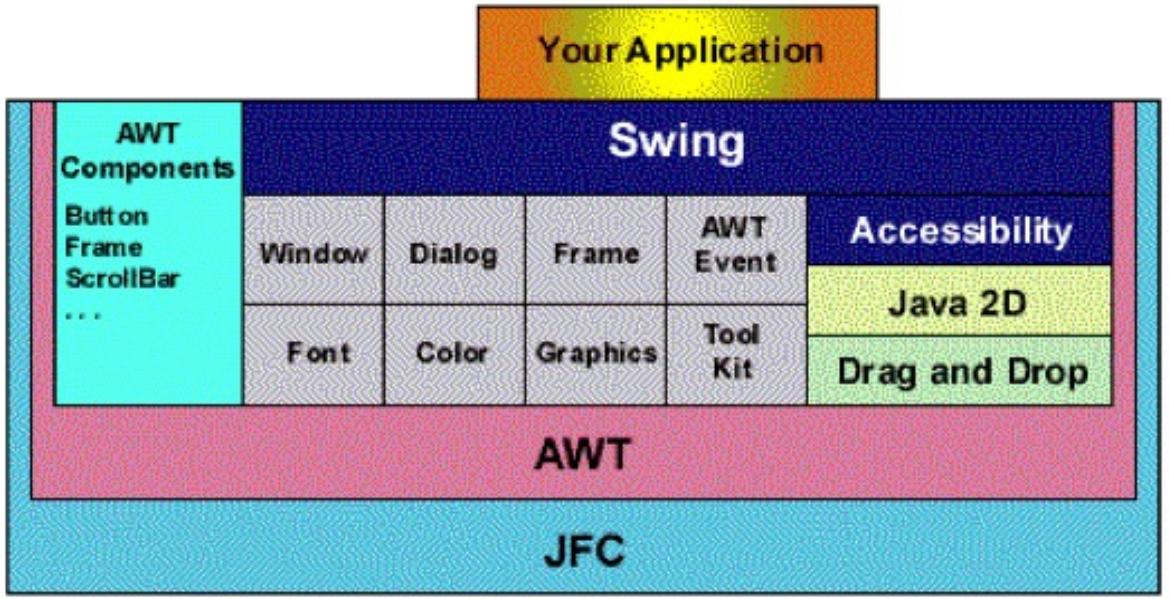
Eric Lecolinet
Télécom Paris – IP Paris
www.telecom-paris.fr/~elc

Oct. 2022

Toolkits graphiques Java

Pour le desktop

- **JavaFX**
  le plus récent, inspiré du Web
- **Swing**
- **AWT Components**
  obsolète
- **SWT**
  Eclipse Foundation



Swing

- multi-plateformes
- repose sur **AWT Components** (à ne pas confondre avec **Swing** **!**)
  - attention : **JButton** != **Button** !

Composants Swing

java.lang.Object

↓

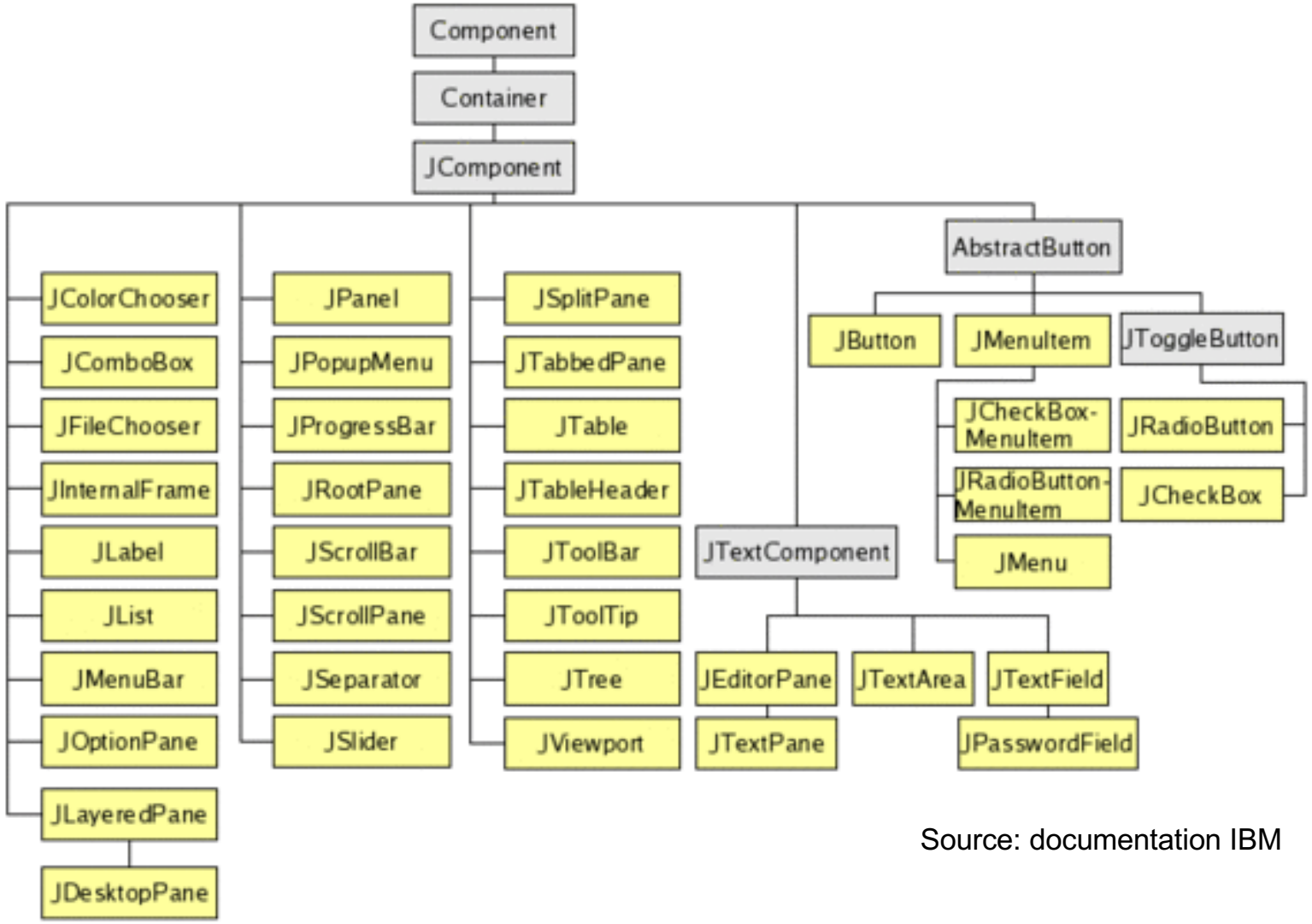
java.awt.Component

↓

java.awt.Container

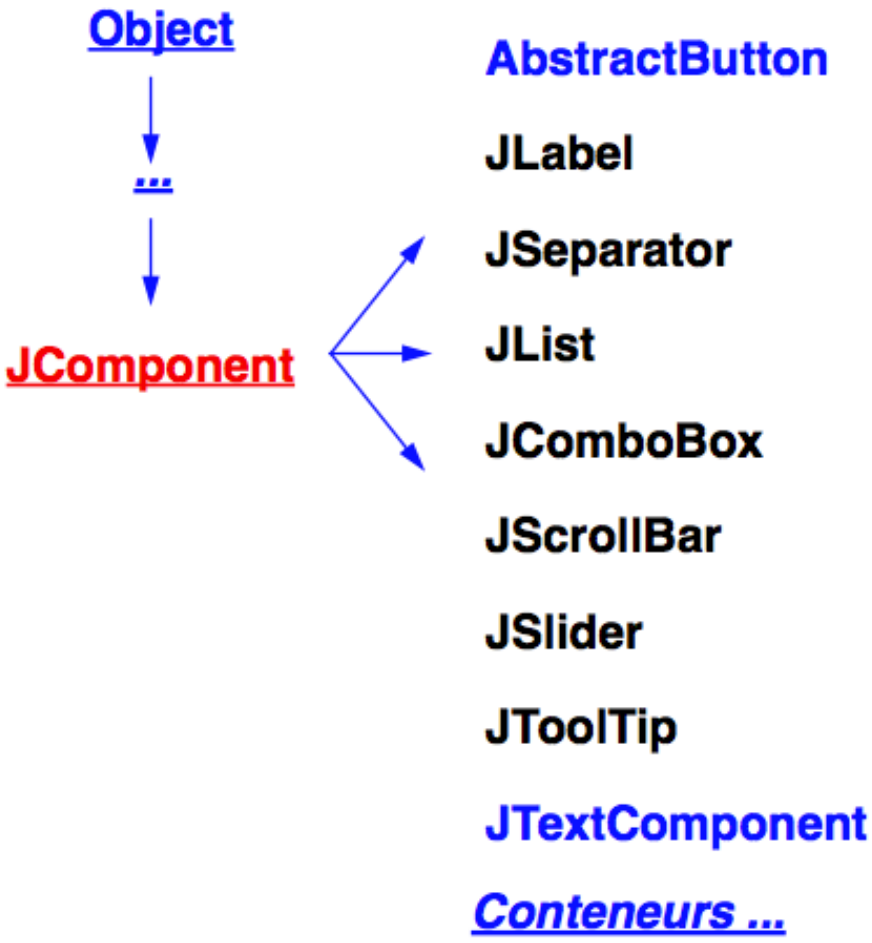
↓

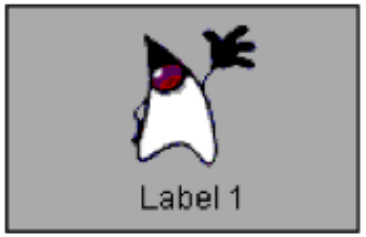
javax.swing.JComponent



Source: documentation IBM

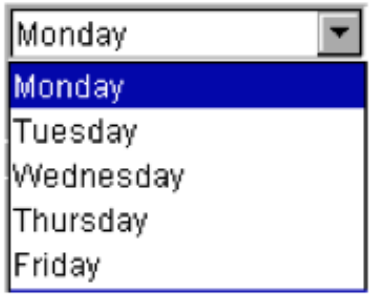
Interacteurs



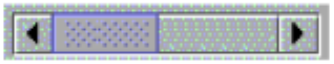


JLabel

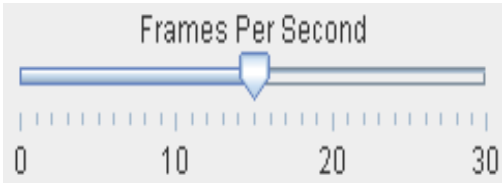
JList



JComboBox

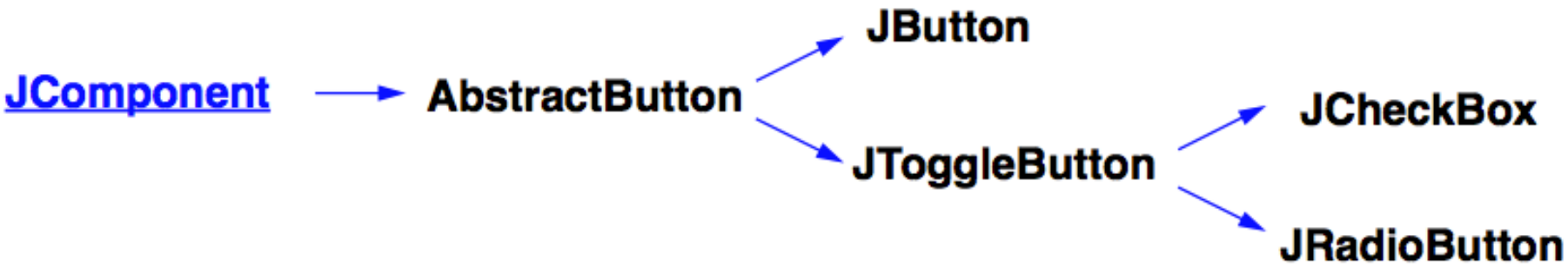


JScrollBar

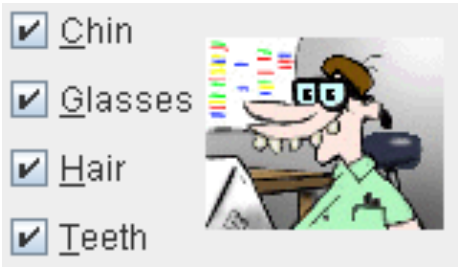


JSlider

Boutons



JButton

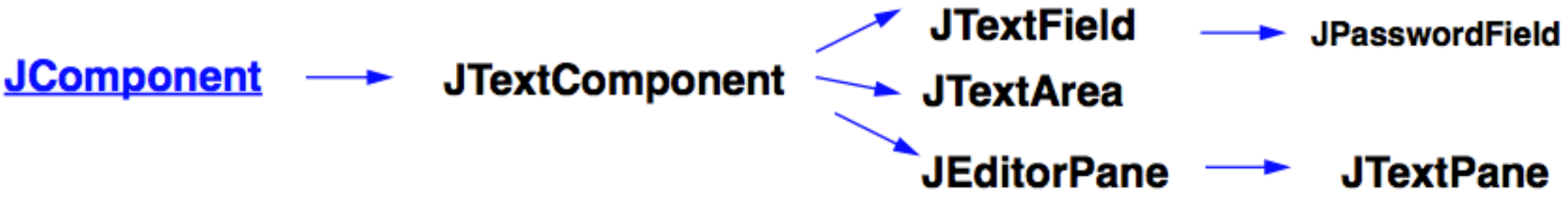


JCheckbox :
choix idépendants

JRadioButton :
choix exclusif : cf. **ButtonGroup**

Source: documentation Java Oracle

Texte

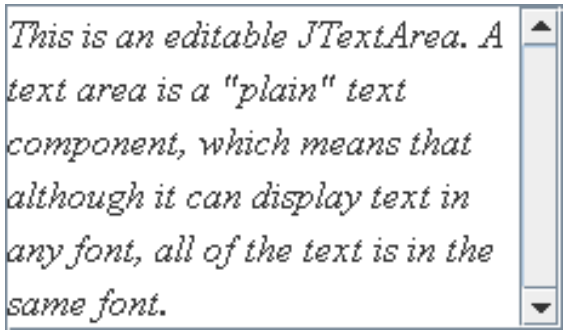


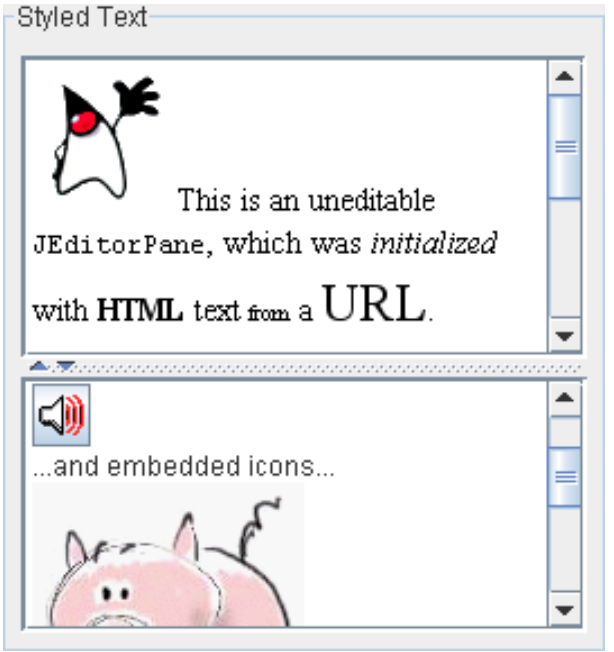
JTextField

JPasswordField

JTextArea :
texte simple multilignes

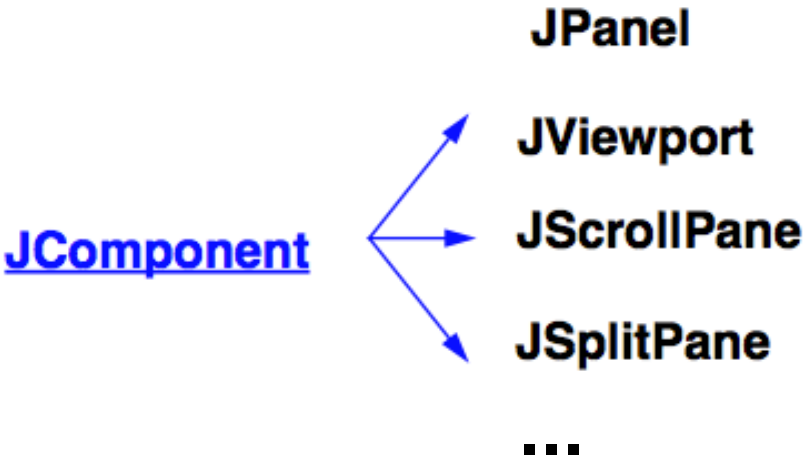
Ascenseurs :
JScrollPane

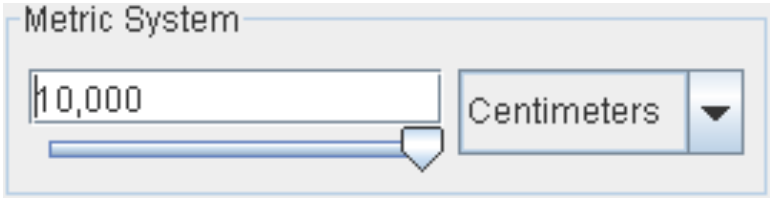




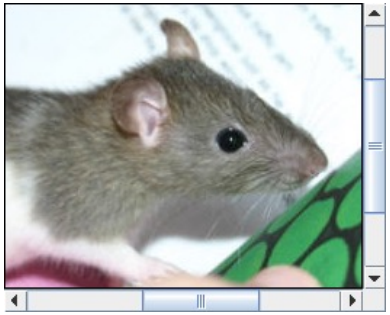
JEditorPane : texte avec styles
compatible HTML et RTF

Conteneurs





JPanel: conteneur générique

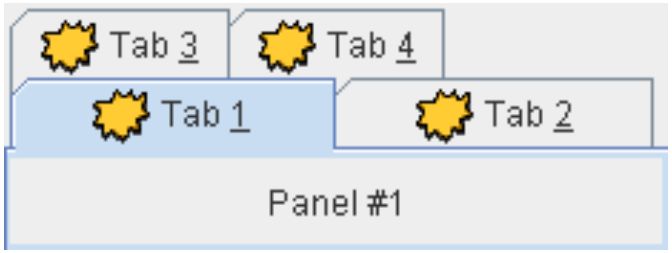


JScrollPane:
avec ascenseurs intégrés

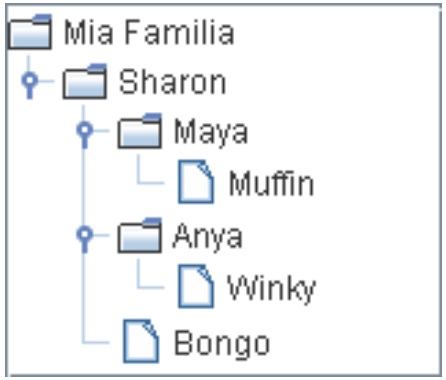
JSplitPane:
avec « diviseur » intégré

Conteneurs

JToolBar: barre d'outils
(sous la barre de menus)



JTabbedPane:
onglets

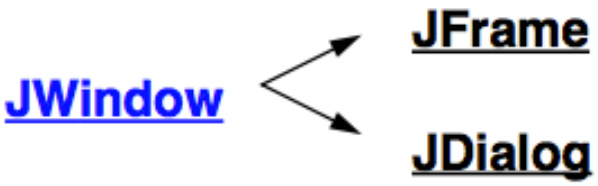


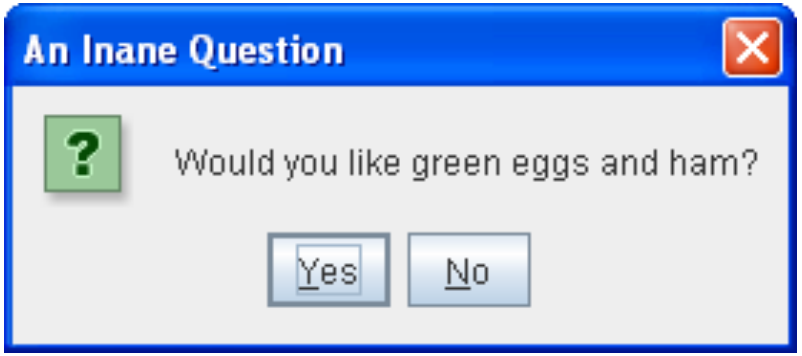
JTree

Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

JTable

Fenêtres





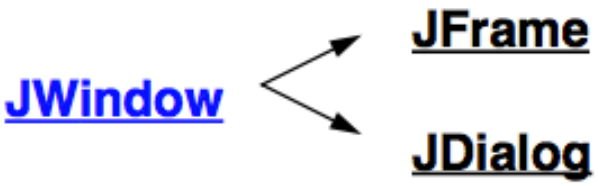
JFrame : fenêtre **principale** de l'application

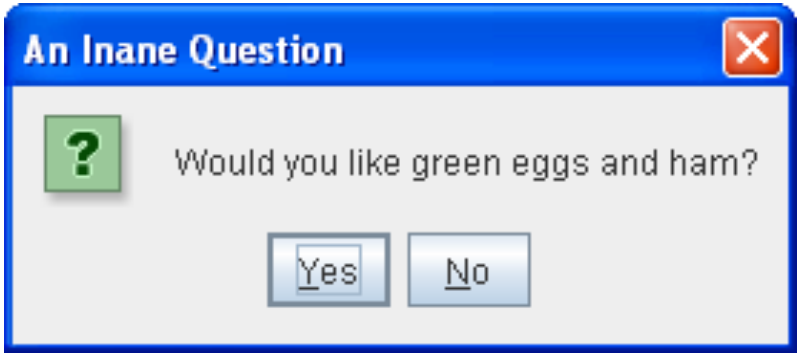
JDialog : fenêtre **secondaire**

- dépendante de la **JFrame** (en théorie pas d'iconification séparée, toujours au dessus)

JDialog modal : **bloque l'interaction** => l'utilisateur de répondre

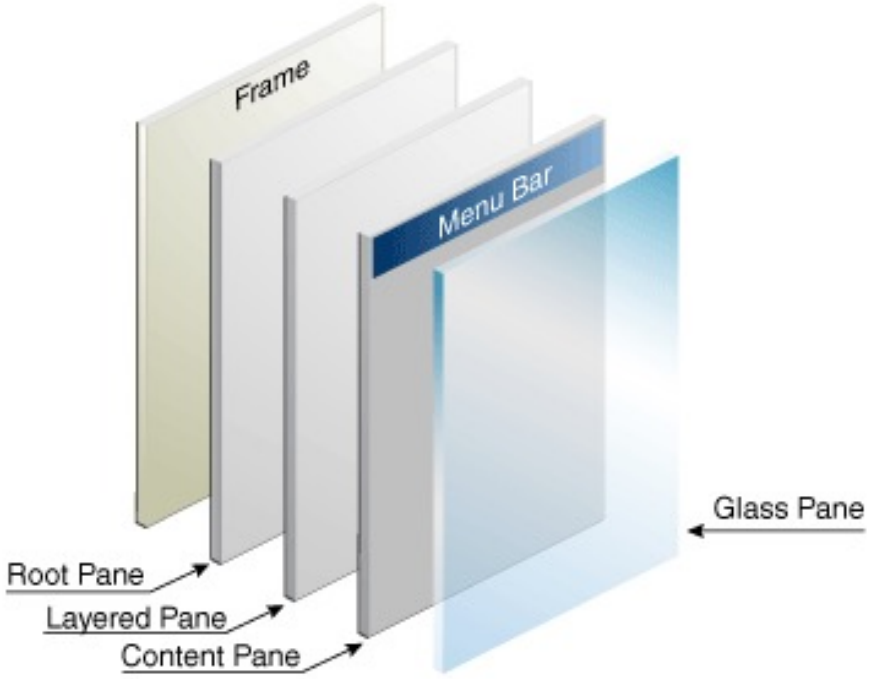
Fenêtres



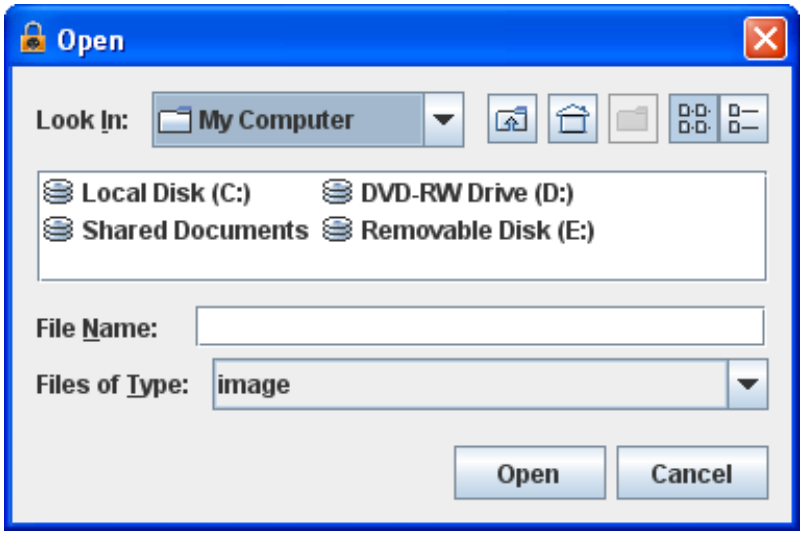


Créent des **paneaux intermédiaires**

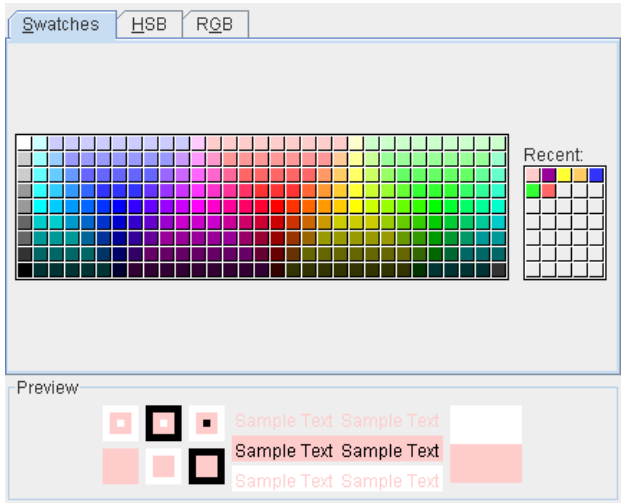
- **ContentPane** : conteneur où on **ajoute** les composants graphiques
- **GlassPane** : conteneur **transparent** superposé (pour usages avancés)



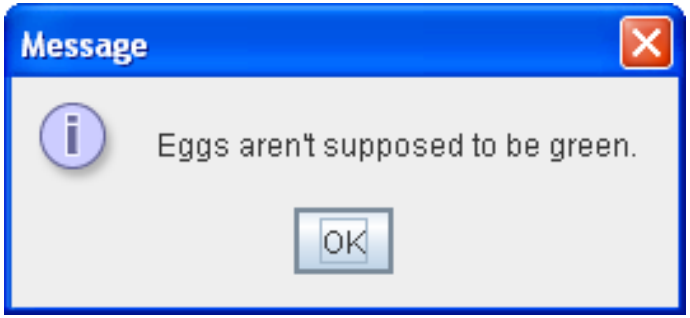
Boîtes de dialogue prédéfinies



JFileChooser



JColorChooser

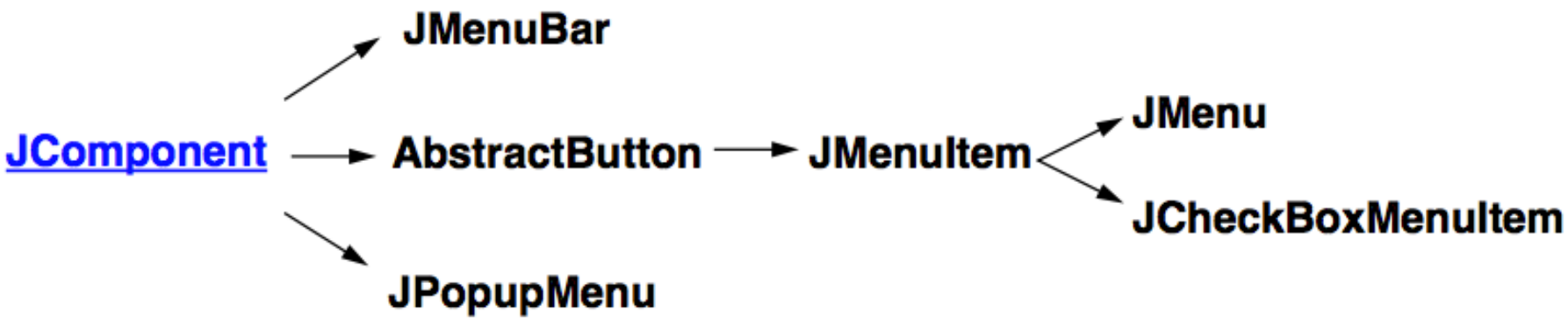


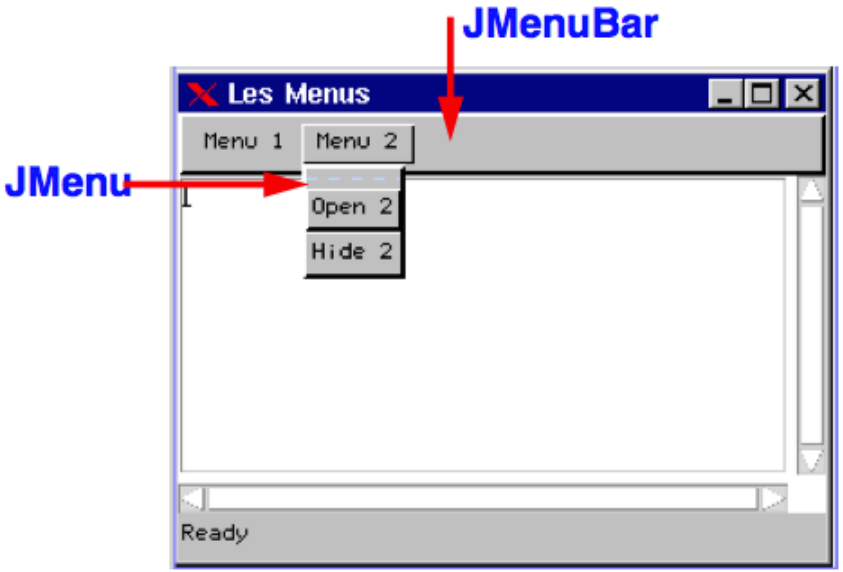
JOptionPane (multiples variantes)

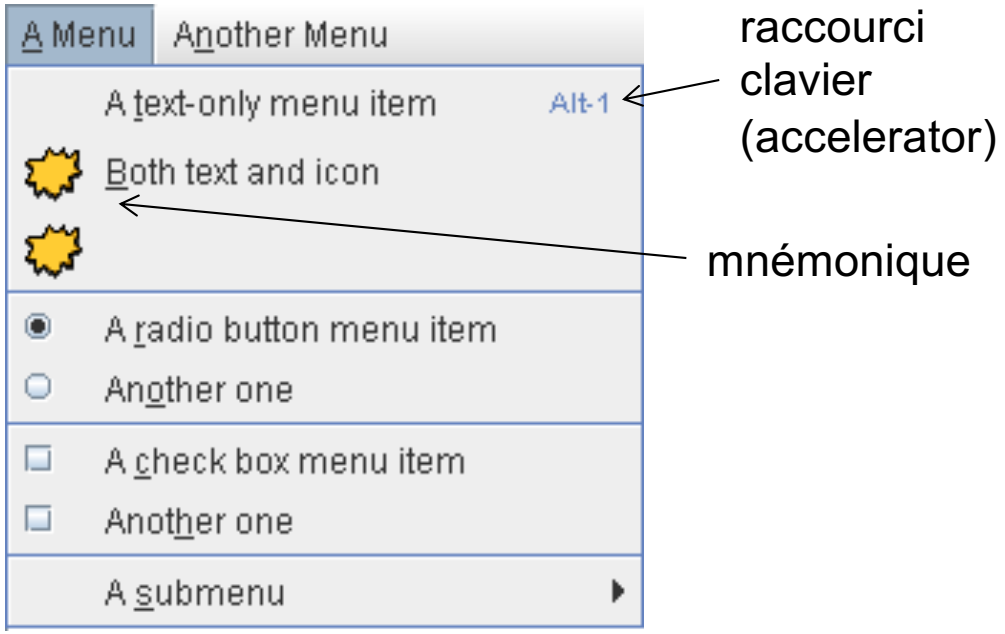
Peuvent être créés :

- comme **boîtes de dialogue**
- ou comme **conteneurs**

Menus


JMenuBar
JComponent
AbstractButton
JMenuItem
JMenu
JCheckBoxMenuItem
JPopupMenu

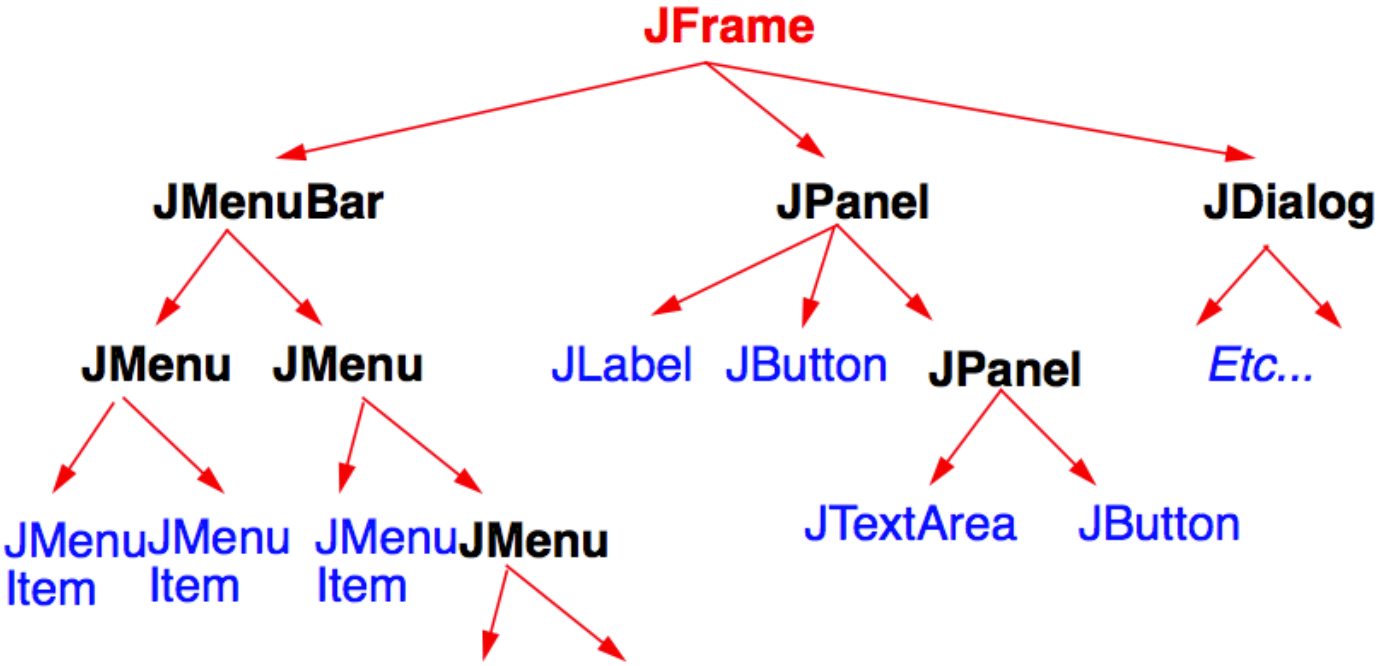

JMenuBar
JMenu
Les Menus
Menu 1
Menu 2
Open 2
Hide 2
Ready

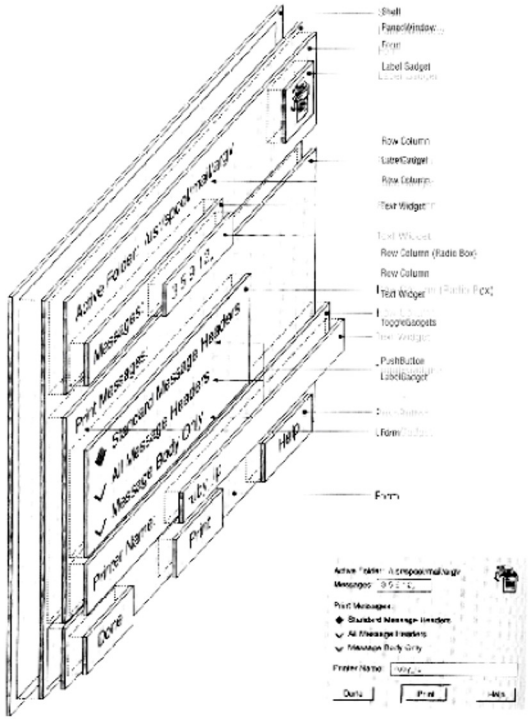

A Menu
Another Menu
A text-only menu item
Alt-1
Both text and icon
A radio button menu item
Another one
A check box menu item
Another one
A submenu
raccourci clavier (accelerator)
mnémonique

Arbre d'instanciation

Arbre d'instanciation

- arbre de **filiation** des **instances** de composants graphiques

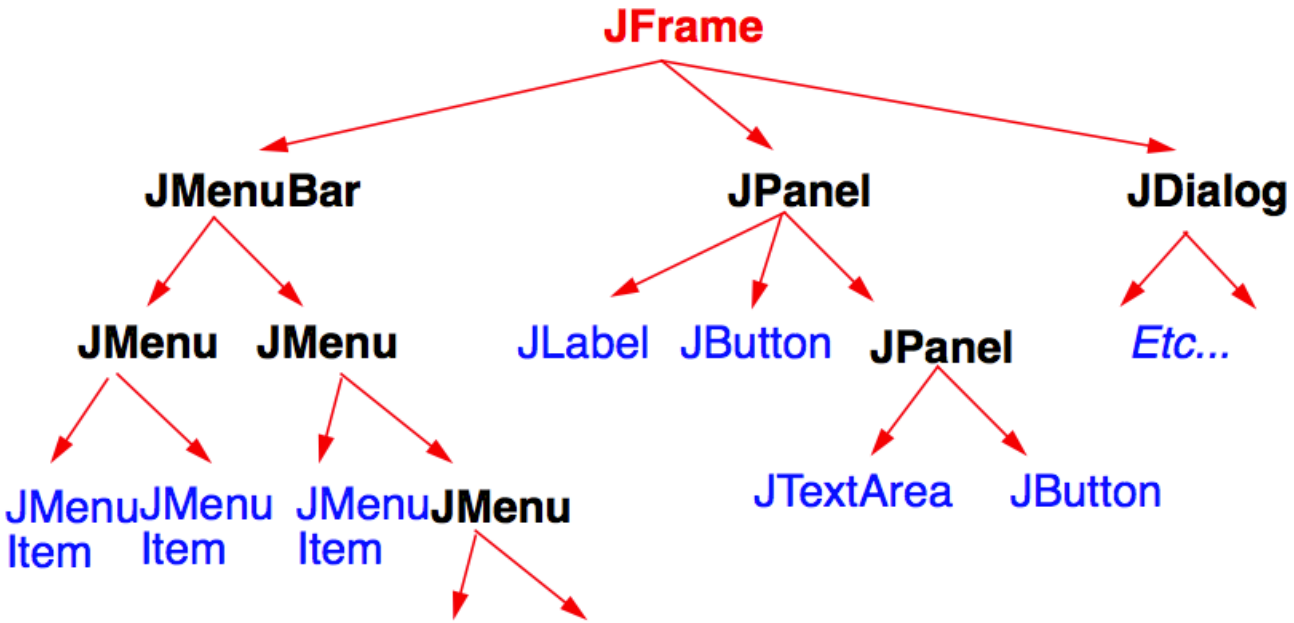


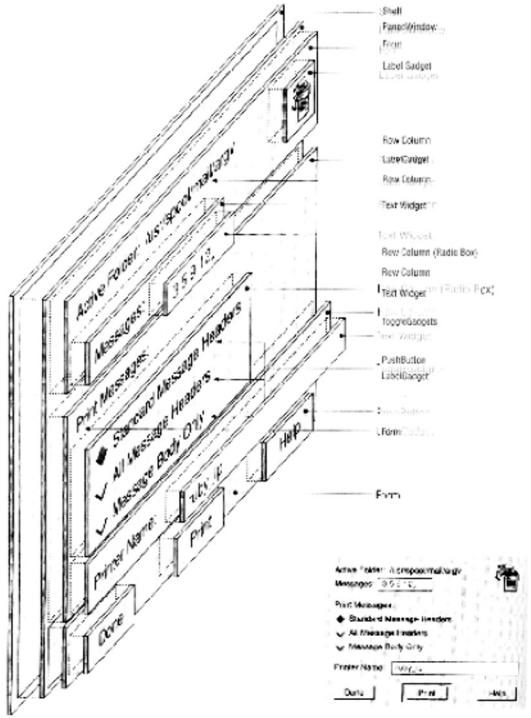


Arbre d'instanciation

Chaque objet graphique « contient » ses enfants

- **superposition** : enfants affichés au dessus des parents
- **clipping** : enfants « découpés » : ne dépassent pas des parents

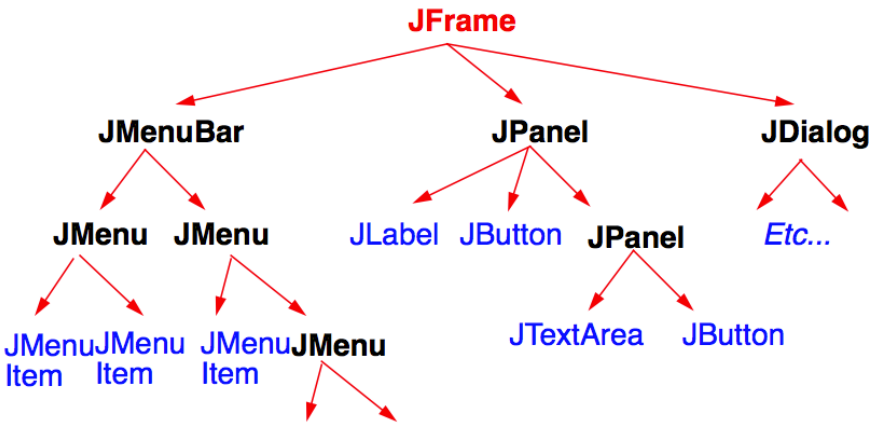


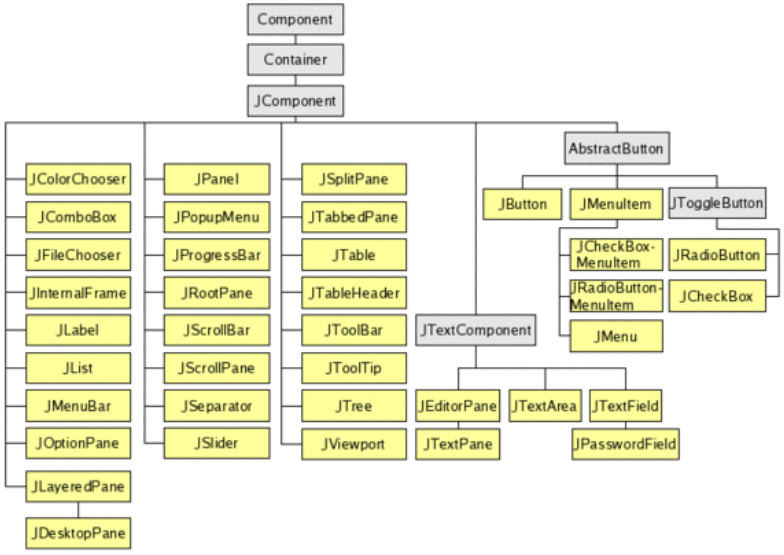


Arbre d'instanciation

Attention : ne pas confondre avec l'arbre d'héritage !

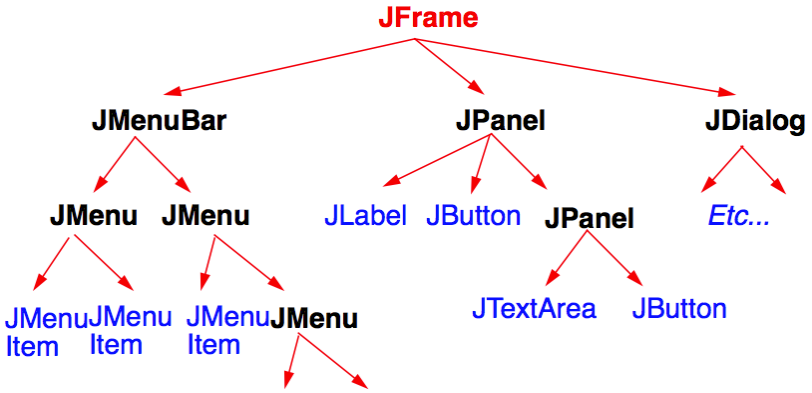
- **arbre d'instanciation** = arbre de filiation des instances
- **arbre d'héritage** = hiérarchie des classes





Arbre d'instanciation

JFrame = objet de plus haut niveau



Les conteneurs peuvent être emboîtés

- en particulier les **JPanels**

Les layout managers assurent la disposition spatiale

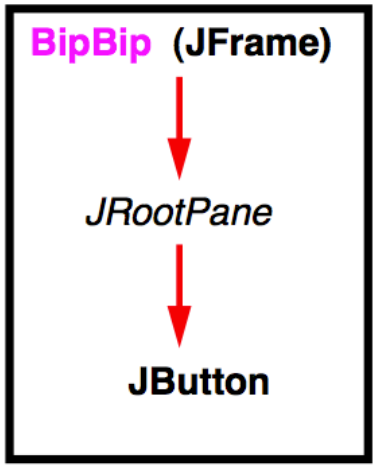
- un layout manager par conteneur
- défaut pour **JPanel** : **FlowLayout**, pour **JWindow** : **BorderLayout**

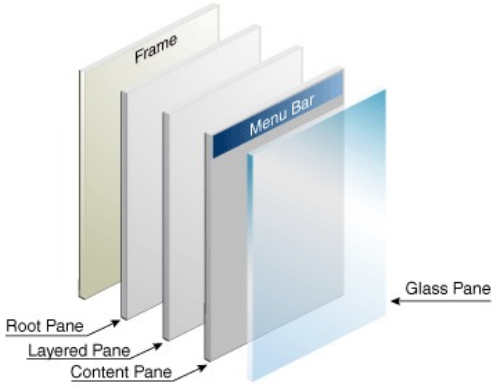
Ne pas oublier d'appeler :

- frame.**pack**( ) // calcul récursif des **positions et des tailles**
- frame.**setVisible**(true) // fait **apparaître** la fenêtre

Exemple : version 0

```
import javax.swing.*;

public class BipBip extends JFrame {      // fenêtre principale
    JButton button = null;

    public static void main(String argv[ ])  {
        BipBip toplevel = new BipBip();      // en gris : optionnel
    }

    public BipBip()  {
        button = new JButton ("Please Click Me !");
        getContentPane().add(button);     // en gris : avant version 5

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Bib Bip");
        pack();                    //  calcule la disposition spatiale
        setVisible(true);          // rend l'interface visible
    }
}
```





Exemple : version 0

Notes :

- package javax.swing
- une seule classe public par fichier, le fichier doit avoir le même nom

- button est une variable d'instance (on peut l'initialiser contrairement à C++)
- toplevel est une variable locale

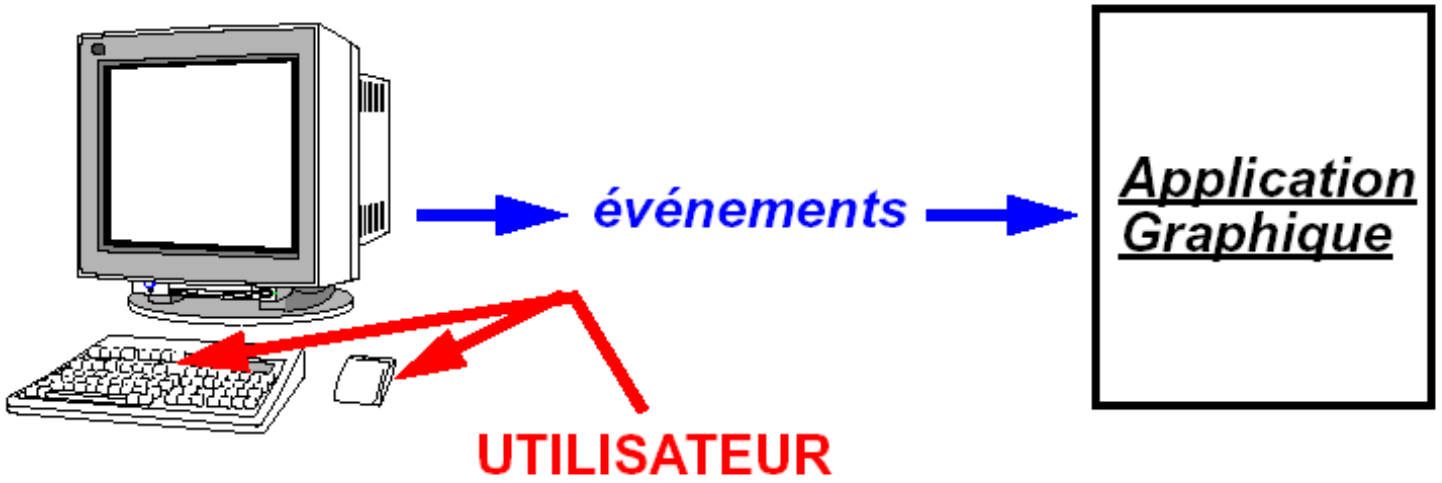
- main() est une méthode de classe (cf. static)
- les méthodes d'instance ont automatiquement accès aux variables d'instance
  elles ont un paramètre caché this qui pointe sur l'instance

- getContentPane() nécesaire avant la version 5 à cause du JRootPane
  JWindow.add() a été redéfini dans les versions ultérieure de Java

Evénements

Evénements

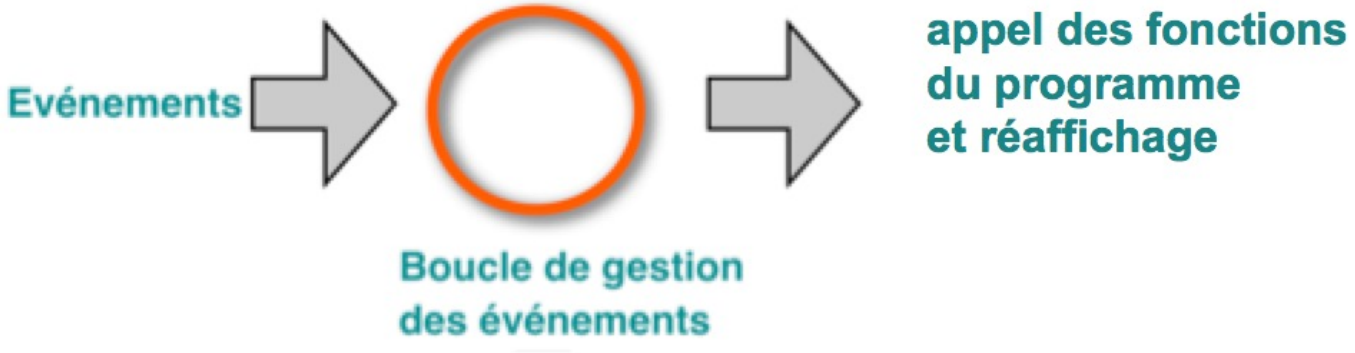
- envoyés à l'application ciblée
- à chaque action élémentaire de l'utilisateur



Boucle de gestion des événements

Boucle infinie qui

- récupère les événements
- notifie les composants graphiques



Lancée automatiquement

- à la fin de la méthode **main**() dans le cas de **Java**

Evénements Java

Evénements AWT et Swing

- **objets** correspondant à des catégories d'évenements
- les principaux héritent de **java.awt.event.AWTEvent**

Evénements de "bas niveau"

- **MouseEvent** appuyer, relacher, bouger la souris ...
- **KeyEvent** appuyer, relacher une touche clavier...
- **WindowEvent** fermeture des fenêtres ....
- **FocusEvent** focus clavier (= où vont les caractères tapés au clavier)

etc.

Evénements de "haut niveau"

- **ActionEvent** **activer** un bouton, un champ textuel ...
  abstraction des événements de bas niveau
- **TextEvent** modification du texte entré

etc.

Evénements Java

Méthodes communes aux AWTEvent

- getSource() **objet** producteur (Object)
- getID() **type** d'événement (int)

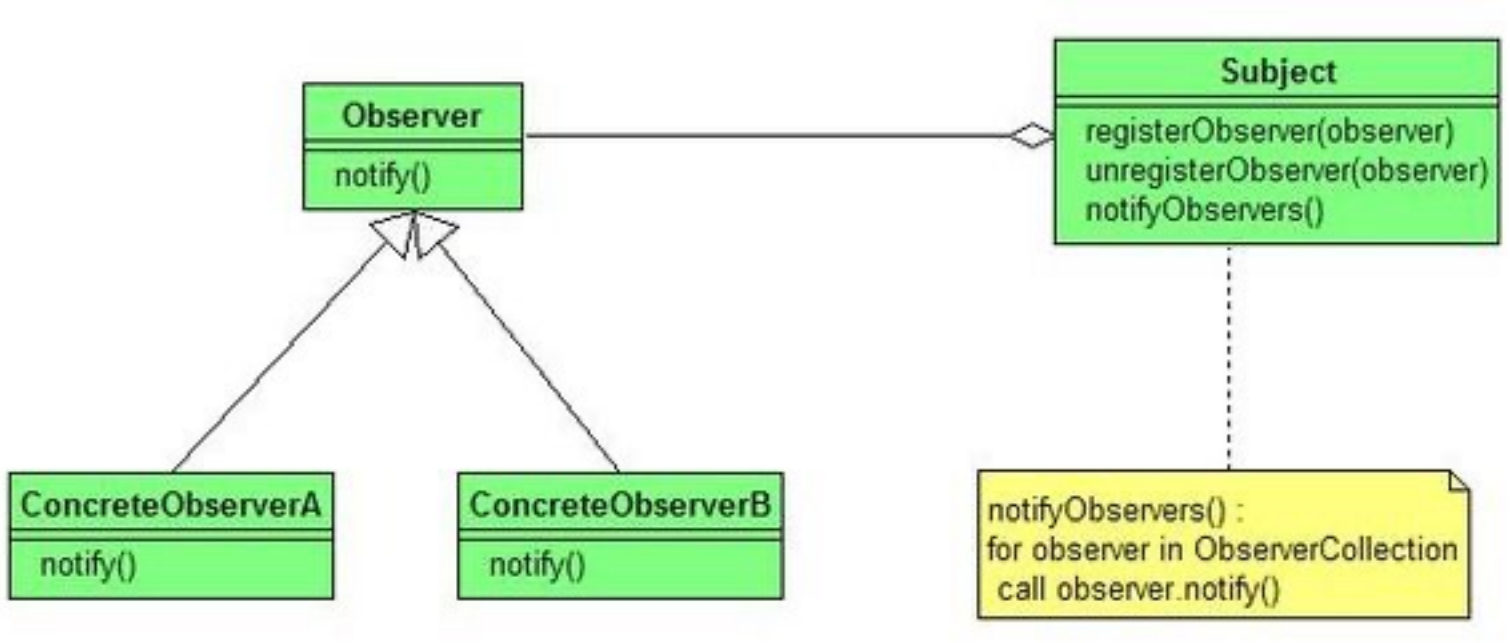
Exemple: méthodes de MouseEvent

- getX(), getY()
- getClickCount()
- getModifiers()
- getWhen()
- etc.

Détecter les événements

Principe : patron Observateur / Observé

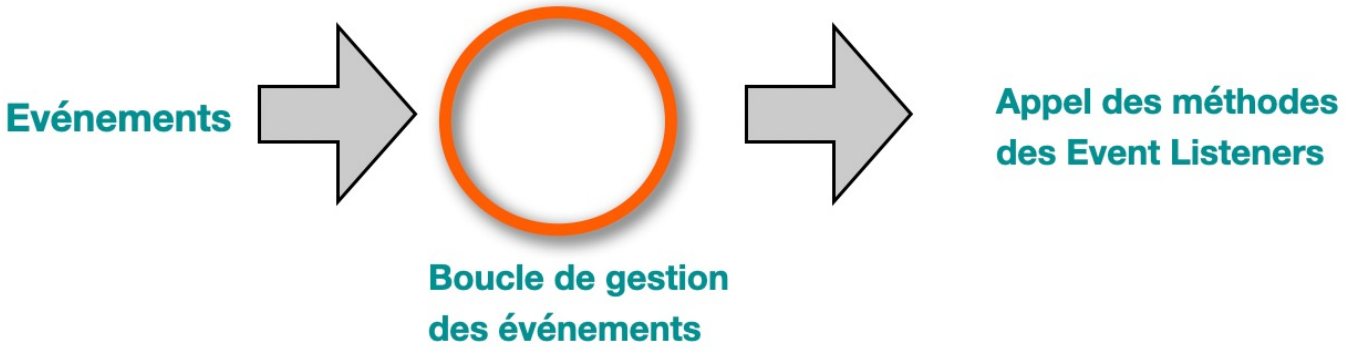
- Associer un ou des **observateurs** aux objets **observés**
- Les **observateurs** sont **notifiés** automatiquement
  - quand une certaine condition se produit sur un **observé**



source: Wikipedia

Event listeners

A chaque classe d'**événement** correspond une classe d'**Event Listener**
(sauf cas particuliers)

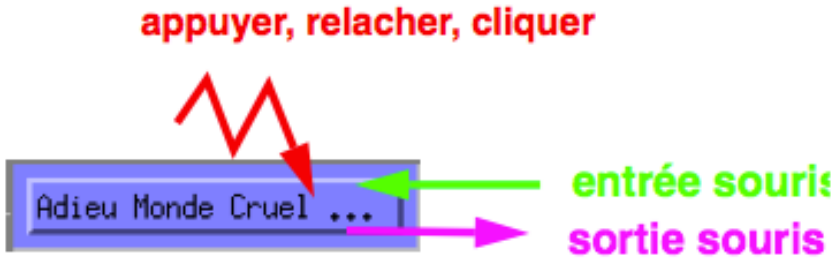


Exemple : ActionEvent

- **Evénement** : ActionEvent
- **Listener** : ActionListener
- **Méthode** : actionPerformed(ActionEvent)

Event listeners

Exemple : MouseEvent



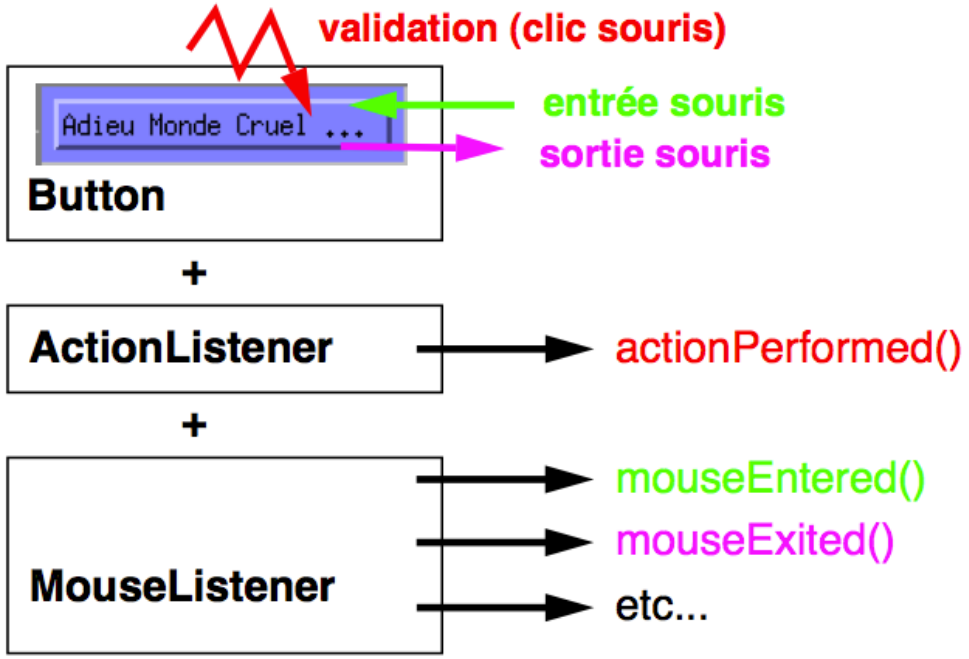
- **Evénement** : MouseEvent
- **Listener** : MouseListener
- **Méthodes** :
  - mouseClicked(MouseEvent)
  - mouseEntered(MouseEvent)
  - mouseExited(MouseEvent)
  - mousePressed(MouseEvent)
  - mouseReleased(MouseEvent)

- **Listener** : MouseMotionListener
- **Méthodes** :
  - mouseDragged(MouseEvent)
  - mouseMoved(MouseEvent)

Remarque

- **toutes** les méthodes doivent être **implémentées**
- car les Listeners sont des **interfaces** (au sens du langage Java)

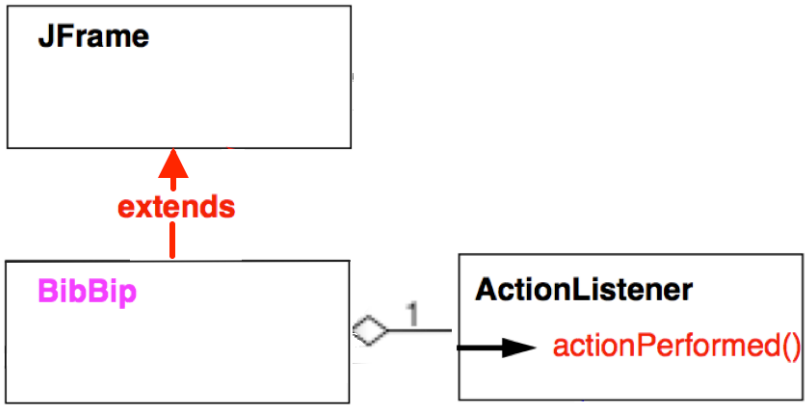
Rendre les composants réactifs



Associer des Listeners aux composants graphiques

- un **composant** peut avoir plusieurs **listeners**
- un même **listener** peut être associé à plusieurs **composants**

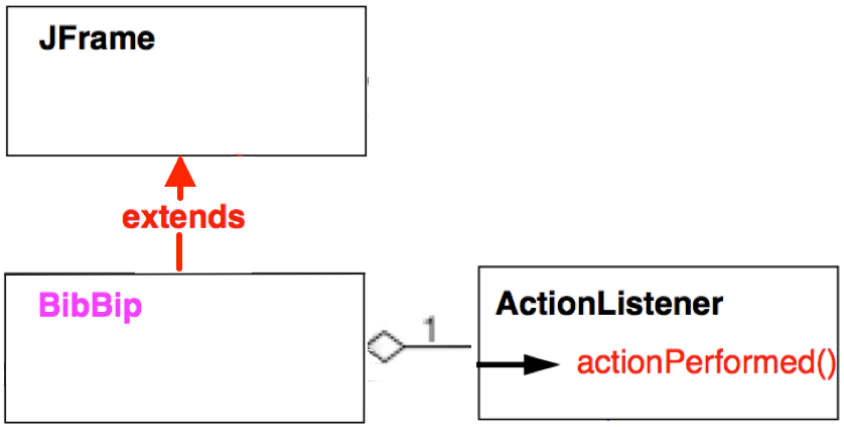
Exemple : version 1




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

public class BipBip extends JFrame {
    JButton button;

    public static void main(String argv[ ])  {
        new BipBip();
    }

    public BipBip()  {
        button = new JButton ("Do It");
        add(button);

        // créer et associer un ActionListener
        Ecoute elc = new Ecoute();
        button.addActionListener(elc);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}
```

```
class Ecoute implements ActionListener {
    // méthode appelée quand on active le bouton
    public void actionPerformed(ActionEvent e) {
        System.out.println("Done!");
    }
}
```

Inconvénients ?




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

public class BipBip extends JFrame {
    JButton button;
    JLabel label = new JLabel();

    public static void main(String argv[ ])  {
        new BipBip();
    }

    public BipBip()  {
        button = new JButton ("Do It");
        add(button);

        Ecoute elc = new Ecoute();
        button.addActionListener(elc);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}
```

```
class Ecoute implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Done!");
        label.setText("Done!");   // ne compile pas !
    }
}
```

Communication entre objets

comment le Listener peut-il **agir** sur les composants graphiques ?

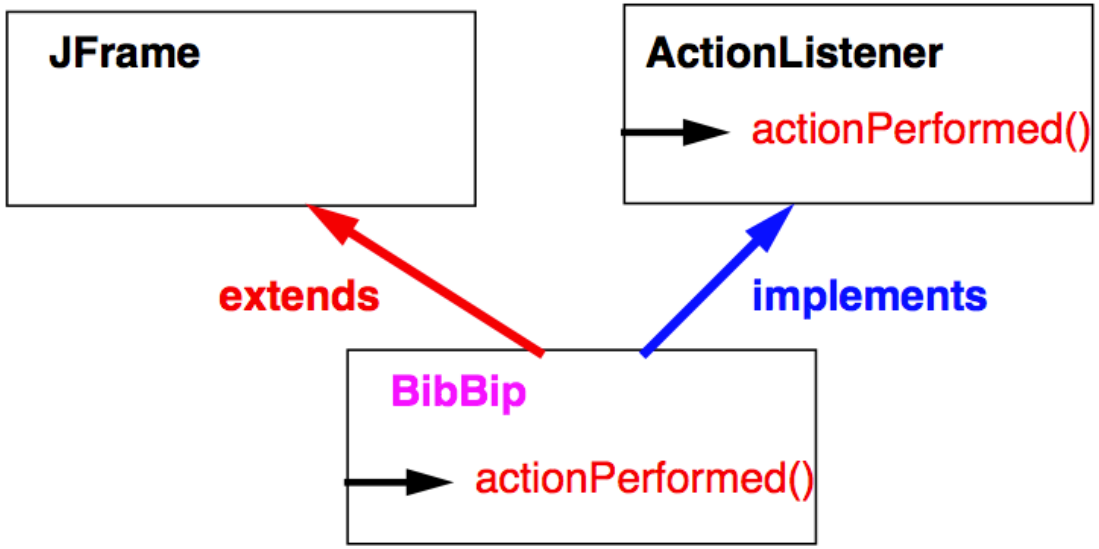
```
import javax.swing.*;
import java.awt.event.*;

public class BipBip extends JFrame {
    JButton button;
    JLabel label = new JLabel();

    public static void main(String argv[ ])  {
        new BipBip();
    }

    public BipBip()  {
        button = new JButton ("Do It");
        add(button);

        Ecoute elc = new Ecoute();
        button.addActionListener(elc);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}
```

```
class Ecoute implements ActionListener {
    BipBip bipbip;

    public Ecoute (BipBip bipbip) {
        this.bipbip = bipbip;
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("Done!");
        bipbip.label.setText("Done!");
    }
}
```

Solution

- le **Listener** a une **référence** vers la partie graphique
- solution **flexible** mais **lourde**

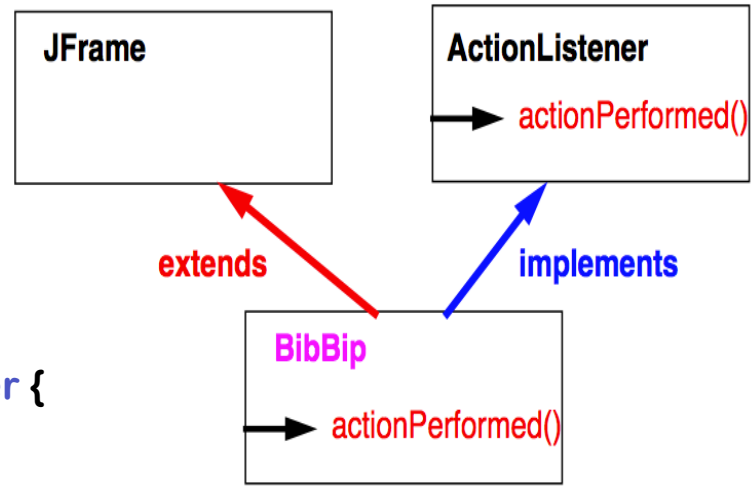
Objets hybrides



A la fois composant graphique et Listener

- **un seul** objet => plus de problème de **communication** entre objets !
- principe de l'**héritage multiple**
  - restreint avec **Java** : on peut « hériter » de plusieurs **interfaces**
  - (mais pas de plusieurs classes)

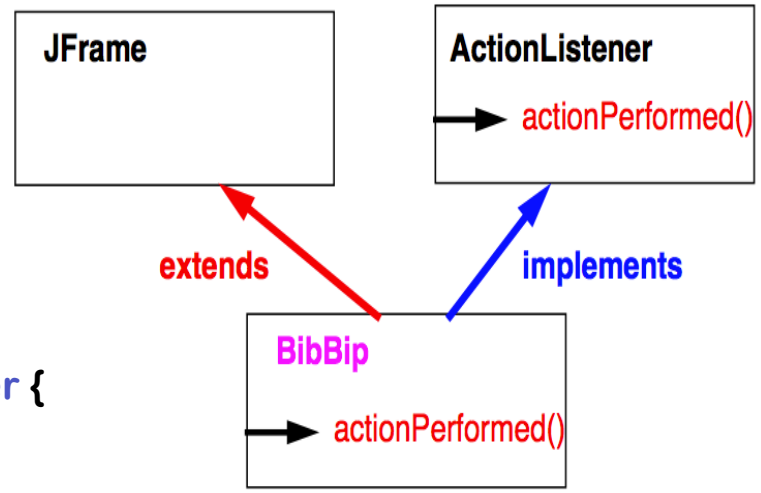
Exemple : version 2




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

public class BipBip extends JFrame implements ActionListener {
    JButton button;
    JLabel label = new JLabel();

    public static void main(String argv[ ])  {
        new BipBip();
    }

    public BipBip()  {
        add( button = new JButton ("Do It") );

        // BibBip est à la fois un JFrame et un Listener
        button.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        label.setText("Done!");
    }
}
```

actionPerformed() à accès à label car c'est une **méthode d'instance** de **BibBip**

Inconvénients ?




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

public class BipBip extends JFrame implements ActionListener {
    JButton button;
    JLabel label = new JLabel();

    public static void main(String argv[ ])  {
        new BipBip();
    }

    public BipBip()  {
        add( button = new JButton ("Do It") );

        // BibBip est à la fois un JFrame et un Listener
        button.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        label.setText("Done!");
    }
}
```

Plusieurs boutons ?

comment avoir **plusieurs** comportements avec **un seul** Listener ?

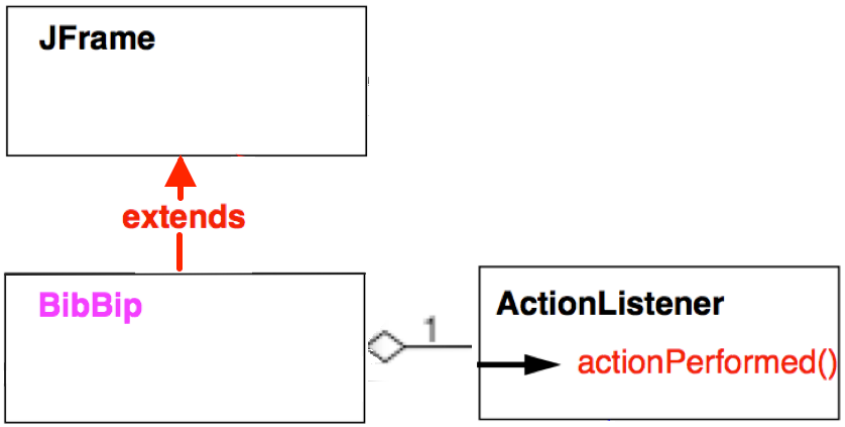
```
import javax.swing.*;
import java.awt.event.*;

public class BipBip extends JFrame
                    implements ActionListener {
    JButton doIt, close;
    JLabel label = new JLabel();

    public static void main(String argv[ ])  {
        new BipBip();
    }

    public BipBip()  {
        add(doIt = new JButton("Do It"));
        add(close = new JButton("Close"));

        doIt.addActionListener(this);
        close.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
```

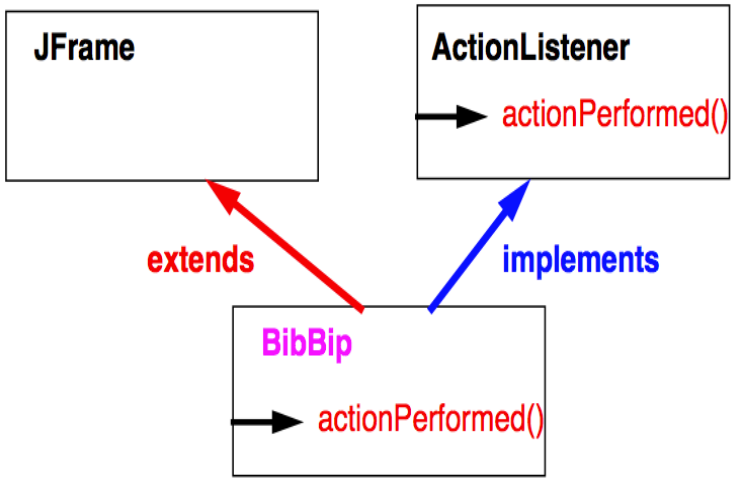
```
// suite de la classe BipBip

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == doIt)
            label.setText("Done!");
        else  if (e.getSource() == close)
            System.exit(0);
    }
}     // fin de la classe !
```

On peut distinguer les boutons via getSource()

Autre solution getActionCommand()
(moins sûre car dépend des noms)

Avantages et inconvénients





Version 1

- **plus souple :**
  autant de listeners que l'on veut
- **peu concise :**
  on multiplie les objets et les lignes de code

Version 2

- **plus simple mais limitée :**
  on ne peut avoir qu'**une seule** méthode **actionPerformed()**
- **peu adaptée** si beaucoup de commandes

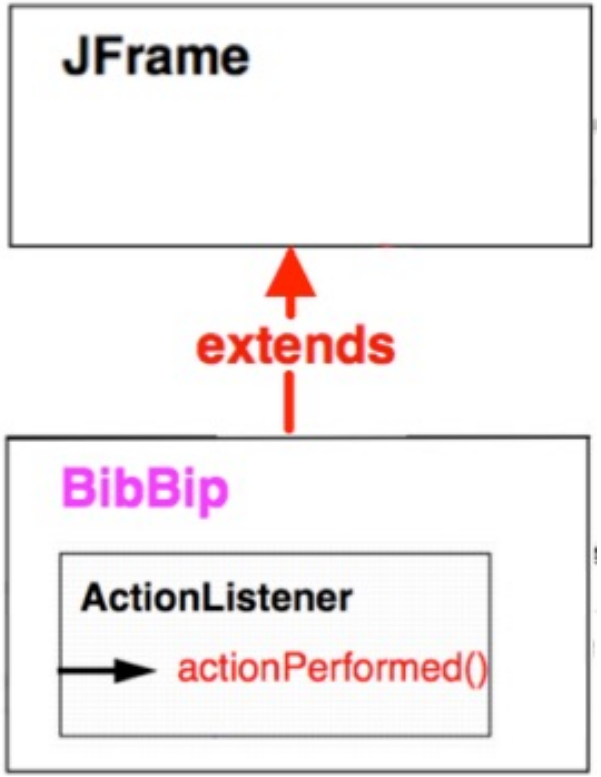
Classes imbriquées

Classes définies à l'intérieur d'une autre classe

- **ont accès aux variables d'instance**
  des classes les contenant
- **rappel** : pas en **C++ !**

Avantages des 2 solutions précédentes

- souplesse sans la lourdeur !



Notes

- le terme exact est **inner classes**
- elles peuvent être **static**
  (sert à structurer en sous-parties)

Exemple : version 3

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

BipBip extends JFrame {
    JButton doIt, close;
    JLabel label = new JLabel();

    public static void main(String argv[ ])  {
        new BipBip();
    }

    public BipBip()  {
        add( doIt = new JButton("Do It") );
        add( close = new JButton("Close!") );

        doIt.addActionListener(new DoItListener( ));
        close.addActionListener(new CloseListener( ));

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
```

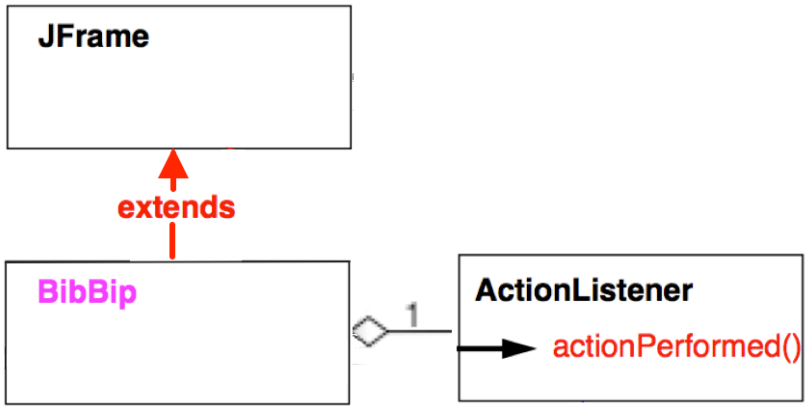
```
class DoItListener implements ActionListener {
        public void actionPerformed(ActionEvent e){
            label.setText("Done!");
        }
    }

class CloseListener implements ActionListener {
        public void actionPerformed(ActionEvent e){
            System.exit(0);
        }
    }
}   // fin de la classe BibBip
```

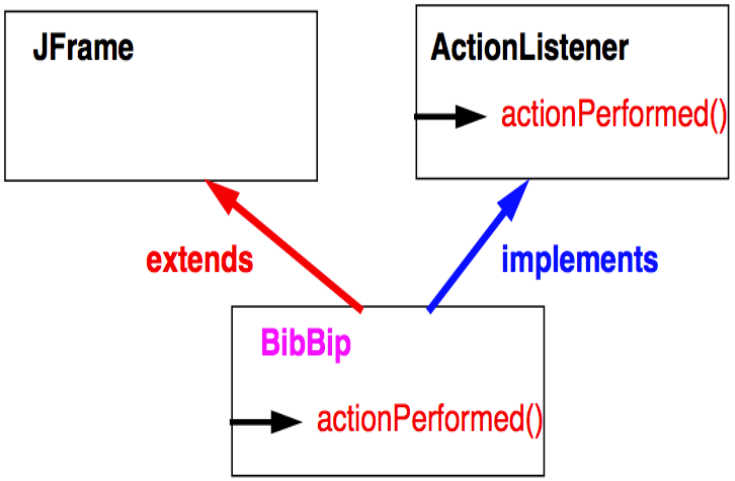
actionPerformed() à accès à label

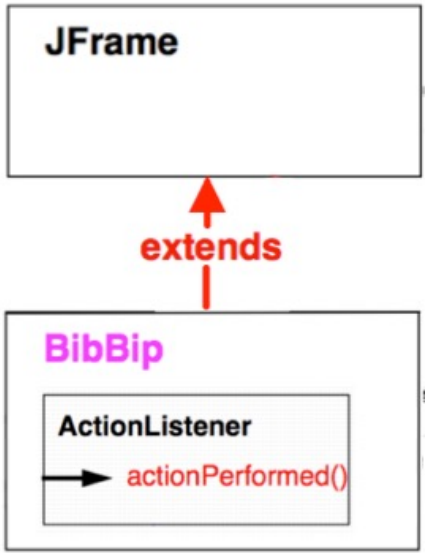
car DoItListener est une

classe imbriquée de BibBip

Synthèse


JFrame
extends
BibBip
1
ActionListener
actionPerformed()


JFrame
ActionListener
actionPerformed()
extends
implements
BibBip
actionPerformed()


JFrame
extends
BibBip
ActionListener
actionPerformed()

Classes imbriquées anonymes & lambdas

```
public class BipBip extends JFrame {
    JLabel label = new JLabel();

    public BipBip()  {
        var doIt = new JButton("Do It");                    // var = inférence de type (comme  auto en C++)
        add(doIt);

        doIt.addActionListener( new ActionListener( ) {       // sous-classe anonyme de ActionListener
             public void actionPerformed( ActionEvent event ) {
                 label.setText("Done!");
             }
        } );
OU  :                                                                  // expression lambda
        doIt.addActionListener( (ActionEvent event)  ->  label.setText("Done!") );
OU  :
        doIt.addActionListener( (event)  ->  label.setText("Done!") );        // si le type peut etre inféré
OU  :
        doIt.addActionListener( event  ->  label.setText("Done!") );              // si un seul argument
    }
    ......
```

Mélanger les plaisirs !

```
abstract class MyButton extends JButton implements ActionListener {
     MyButton(String name) {
          super(name);
          addActionListener(this);
     }
}


public class BipBip extends JFrame {
     JLabel label = new JLabel();
     ......

     public BipBip()  {
          add(new MyButton("Do It")) {
               public void actionPerformed(ActionEvent e) {
                    label.setText("Done!");
               }
          });

          ......
```

Conflits

```
public class BipBip extends JFrame {
   JButton close = new JButton("Close");

   class CloseListener implements ActionListener {
      boolean close = false

      public void actionPerformed(ActionEvent e) {
           setVisible(close);                    // OK
           setVisible(BipBip.close);             // FAUX : pas le bon « close »
           this.setVisible(close);               // ERREUR : pas le bon « this »
           BipBip.this.setVisible(close);        // OK
      }
   }
```

Même nom de variable dans classe imbriquante et classe imbriquée

⇒ 1) à éviter !

⇒ 2) préfixer par le nom de la classe

Compléments sur les constructeurs

```
abstract class MyButton extends JButton implements ActionListener {
    public MyButton(String name, Icon icon) {
        super(name, icon);
        ........
    }

    public MyButton (String name) {
        this(name, null);
    }

}
```

Un constructeur peut en appeler un autre !

Note : en C++11:

- MyButton(string name) : MyButton(name, nullptr) {}
- on pourrait aussi utiliser les **paramètres par défaut**

```
abstract class Toto {
    static {
        ........
    }

}
```

Constructeur de classe

- Pour initaliser des variables de classes
- Pas d'équivalent direct en C++

Dessin

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

class Canvas extends JPanel {
  private int last_x, last_y;

  Canvas() {
    setBackground(Color.white);

    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        last_x = e.getX();
        last_y = e.getY();
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        var g = getGraphics();
        g.drawLine(last_x, last_y, x, y);
        last_x = x;
        last_y = y;
      }
    });
  }
```

si on utilisait MouseListener ou MouseMotionListener il faudrait implémenter :

```
public void mouseReleased (MouseEvent e) { }
public void mouseClicked (MouseEvent e) { }
public void mouseEntered (MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
```

Dessin

```
  // fin de la classe Canvas

  void clear() {
    var g = getGraphics();
    g.setColor(Color.white);
    g.fillRect(0, 0,
               getSize().width,
               getSize().height);
  }
}
```

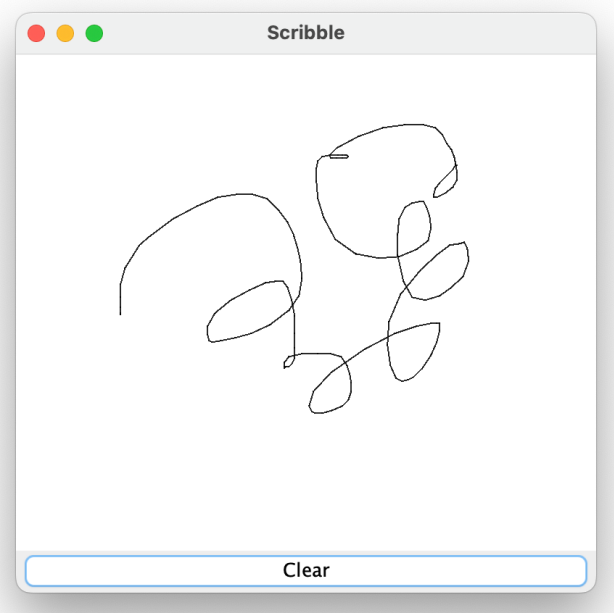
```
public class Scribble1 extends JFrame {

  public static void main(String argv[]) {
    new Scribble1();
  }

  Scribble1( ) {
    var canvas = new Canvas();
    var clear = new JButton("Clear");
    clear.addActionListener(e -> canvas.clear());

    add(BorderLayout.CENTER, canvas);
    add(BorderLayout.SOUTH, clear);

    setTitle("Scribble");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setMinimumSize(new Dimension(400, 400));
    pack();
    setVisible(true);
  }
}
```

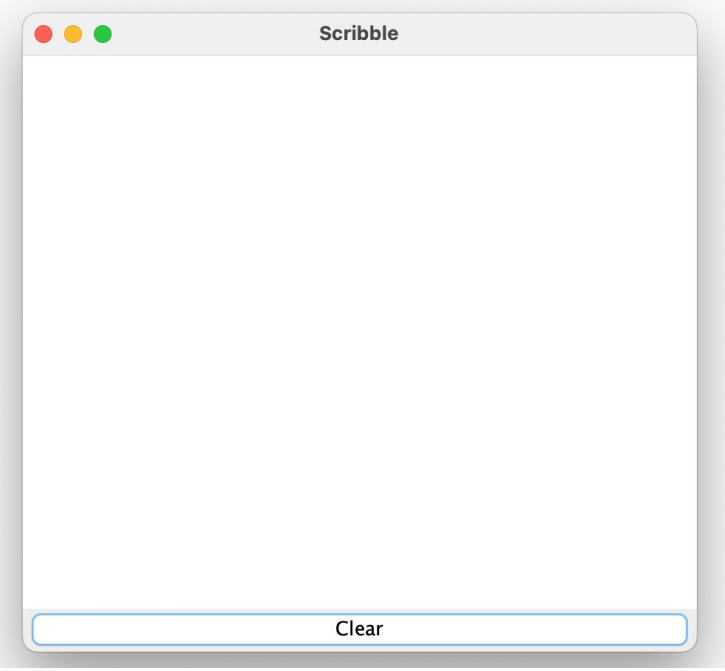
Problème ?

Persistance de l'affichage

Problème : l'affichage du dessin n'est pas persistant

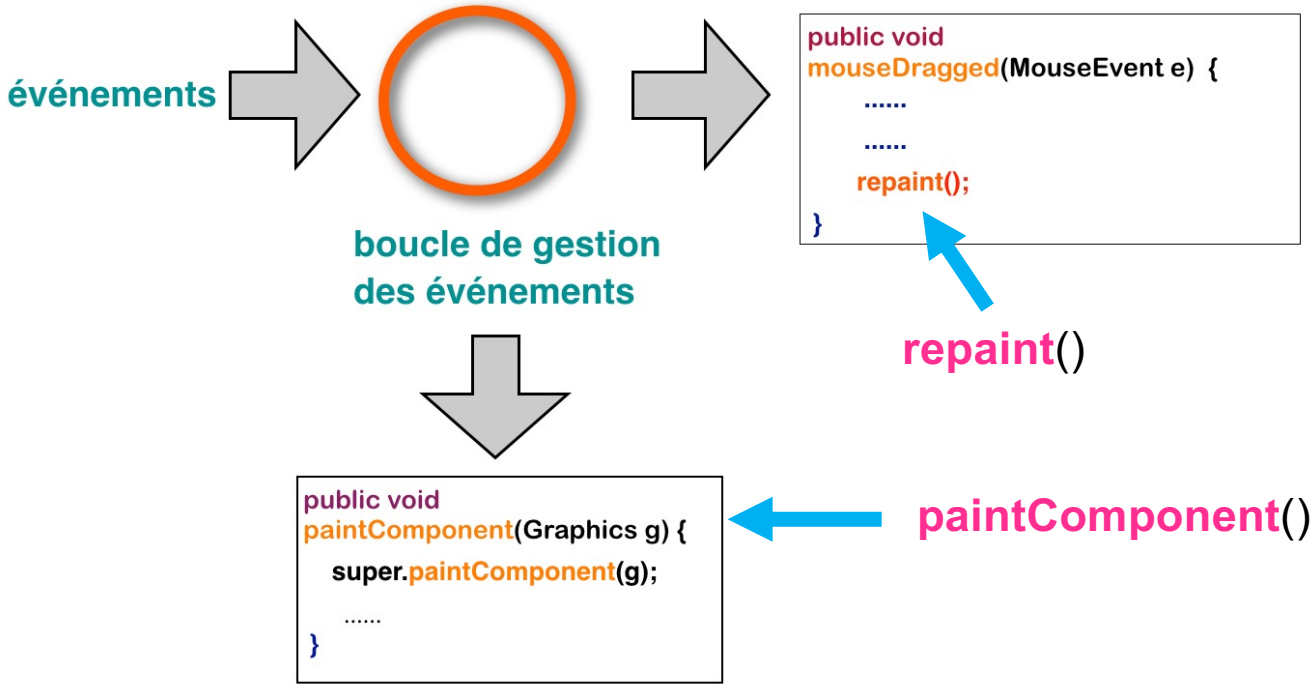
- **dessin effacé** si on retaille, iconifie, etc.
- les méthodes des listeners **ne devraient pas dessiner !**





si on retaille...

Modèle "damaged / repaint"



Principe

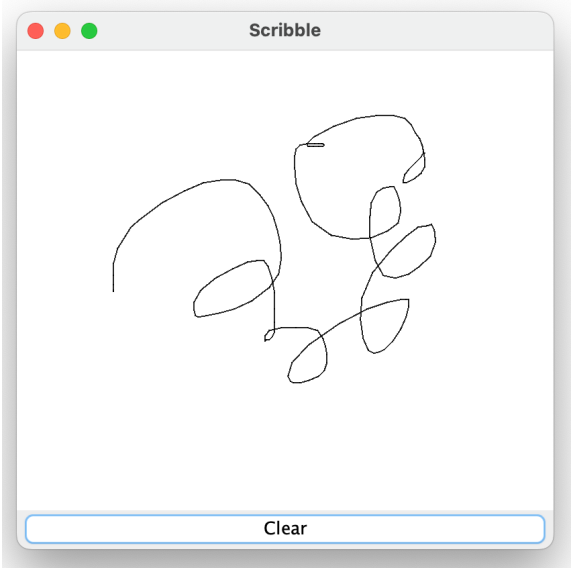
Les listeners :

- **sauvegardent** les opérations **sans afficher**
- **notifient** le toolkit en appelant **repaint**

La méthode **paintComponent** **repaint** le widget (et le dessin !)

Persistance de l'affichage

```
class Canvas extends JPanel
{
  private ArrayList<Point> points = new ArrayList<Point>();

  Canvas() {
    setBackground(Color.white);

    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        points.add(new Point(e.getX(), e.getY()));
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        points.add(new Point(e.getX(), e.getY()));
        repaint();
      }
    });
  }

  void clear() {
    points.clear();
    repaint();
  }
```




```
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    for (int k = 0; k < points.size()-1; ++k) {
      g.drawLine((int)points.get(k).getX(),
                 (int)points.get(k).getY(),
                 (int)points.get(k+1).getX(),
                 (int)points.get(k+1).getY());
    }
  }
}
```

Notes

Swing vs. AWT

- avec **AWT** redéfinir la méthode **paint**()
- avec **Swing**, **paint**() appelle :
  - **paintComponent**() puis **paintBorder**() puis **paintChildren**()

Divers

- Appeler **revalidate**() dans certains cas de **changements de taille**
- Taille des **bords** : **getInsets()**
- **Opacité** des widgets
  - certains composants sont **opaques**, d'autres sont **transparents**
  - **setOpaque**() rend le composant opaque

JFileChooser

Ouvre la boîte de dialogue et bloque l'interaction (dialogue modal) :

```
int returnVal = chooser.showOpenDialog(parent);

if (returnVal == JFileChooser.APPROVE_OPTION) {
    System.out.println("You chose to open this file: "
                        + chooser.getSelectedFile().getName());
}
```

Filtre

```
JFileChooser chooser = new JFileChooser();
var filter = new FileNameExtensionFilter("JPG & GIF", "jpg", "gif");
chooser.setFileFilter(filter);
```

Disposition spatiale

Les LayoutManagers

- calculent **automatiquement** la **disposition spatiale** des enfants des **conteneurs**

A chaque conteneur est associé un LayoutManager

- qui dépend du type de conteneur
- qui peut être changé par la méthode : **setLayout()**

Pour faire le calcul "à la main"

- à éviter sauf cas particuliers : **setLayout(null)**

Avantages des LayoutManagers

C'est plus simple

- **pas de calculs** compliqués à programmer !

Configurabilité

- **accessibilité** : indépendance par rapport aux **tailles des polices**
- **internationalisation** : indépendance par rapport à la **taille du texte**
  - langues orientales : texte **~1/3 plus petit** que l'anglais
  - français, allemand : texte **~1/3 plus grand** que l'anglais

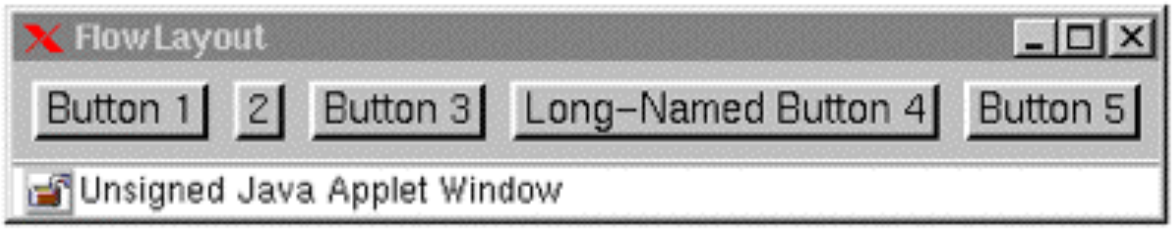
Adaptativité

- les composants graphiques se **retaillent automatiquement**
- quand l'utilisateur **retaille** les fenêtres

Principaux LayoutManagers

FlowLayout

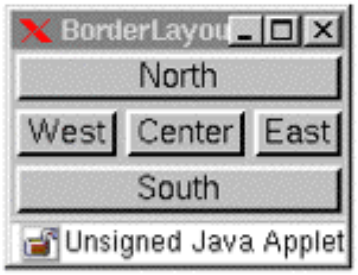
- défaut des **JPanel**
- met les objets **à la suite** comme un "flux textuel" dans une page
    - de gauche à droite puis à la ligne

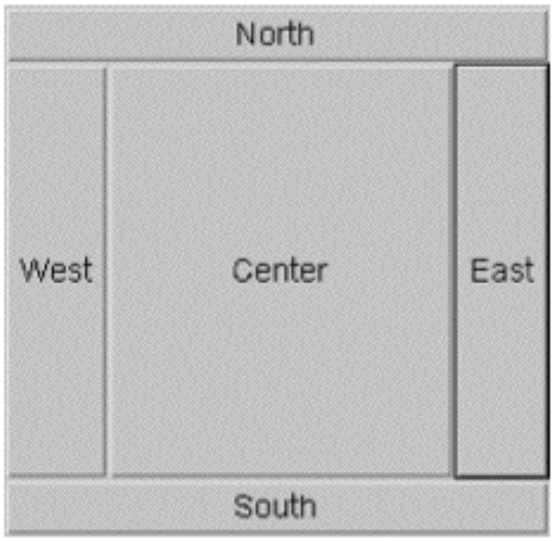


Principaux LayoutManagers

BorderLayout

- défaut des **JFrame** et **JDialog**
- disposition de type **points cardinaux**
    - via constantes: **BorderLayout.CENTER, EAST, NORTH, SOUTH, WEST**
- **retaille automatiquement** les enfants

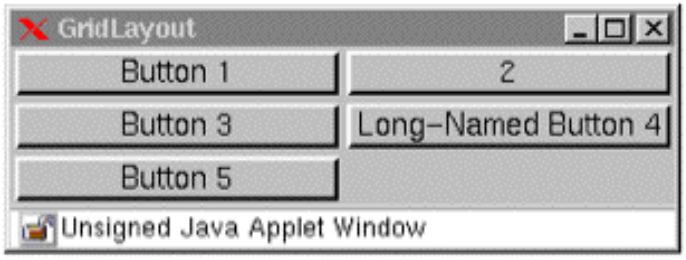




Principaux LayoutManagers (2)

GridLayout

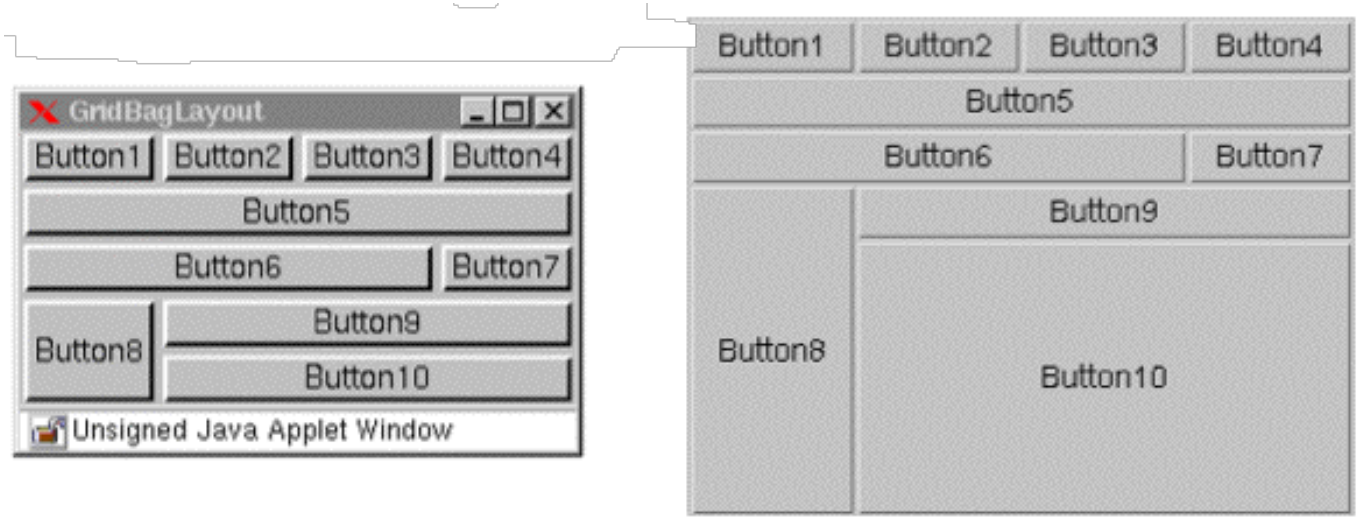
- divise le conteneur en **cellules de même taille** (grille virtuelle)
  - de gauche à droite et de haut en bas
- **retaille automatiquement** les enfants



Principaux LayoutManagers (2)

GridBagLayout

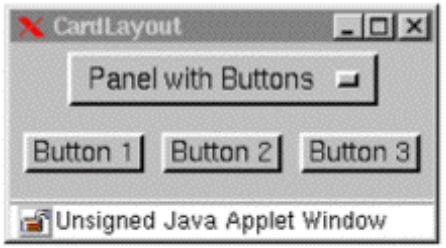
- **grille + contraintes spatiales**
  - les enfants n'ont **pas tous la même taille**
  - spécification par des **GridBagConstraints**



Principaux LayoutManagers (3)

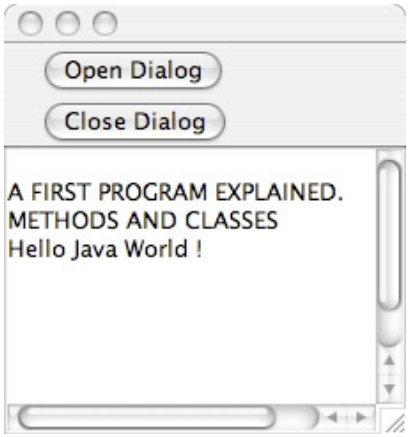
CardLayout

- **empile** les enfants (et les met à la même taille)
- usage typique: **onglets**



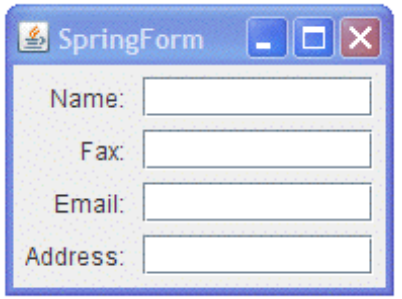
BoxLayout

- disposition **verticale** ou **horizontale**
- exemple :
  panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));



SpringLayout

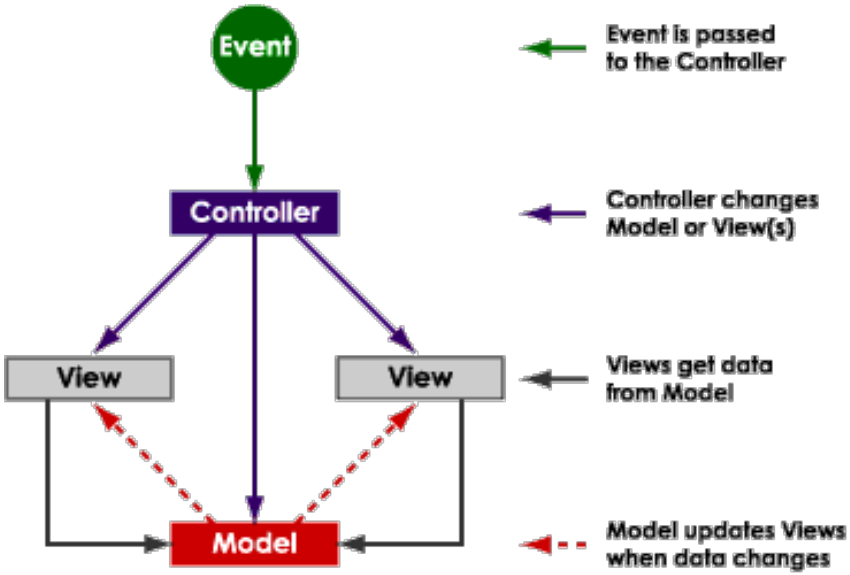
- contraintes entre les **bords** des enfants
- usage typique : **formulaires**



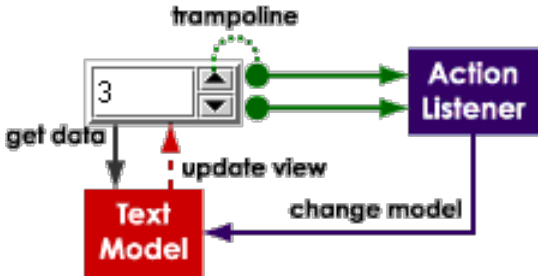
Architecture Swing

Swing est inspiré du modèle MVC

- **Model** : données de l'application
- **View** : représentation visuelle
- **Controller** : gestion des entrées



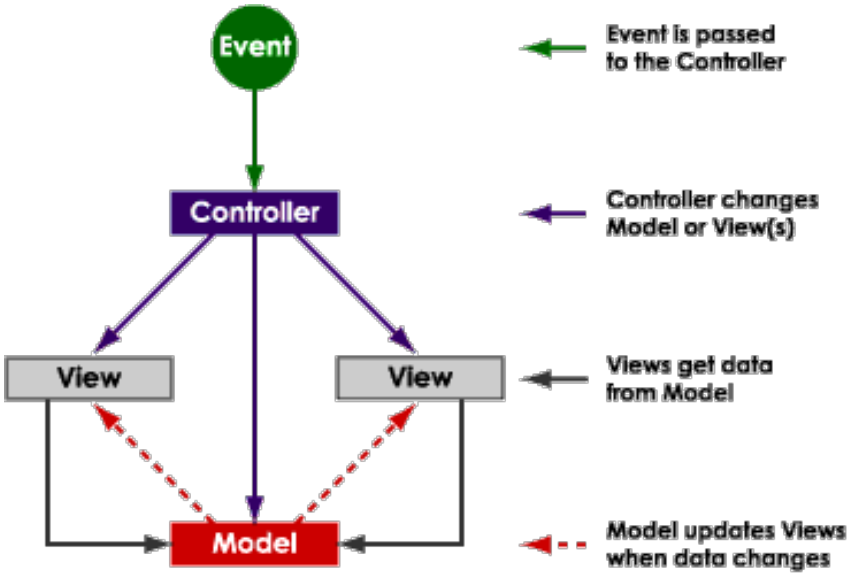
source: enode.com



Architecture Swing

Swing est inspiré du modèle MVC

- **Model** : données de l'application
- **View** : représentation visuelle
- **Controller** : gestion des entrées



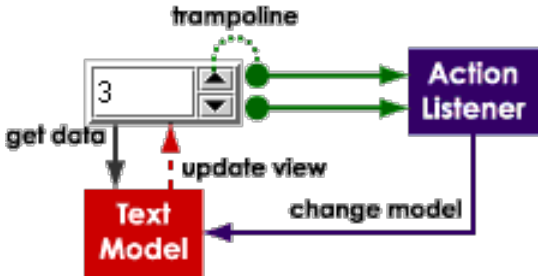
But de MVC

- mieux structurer les applications
- représentations **multi-vues**
  - un **modèle** peut être associé à plusieurs **vues**
  - la synchronisation est implicite

source: enode.com

En pratique

- **V** fortement lié à **C**
- plusieurs **variantes** de **MVC** !



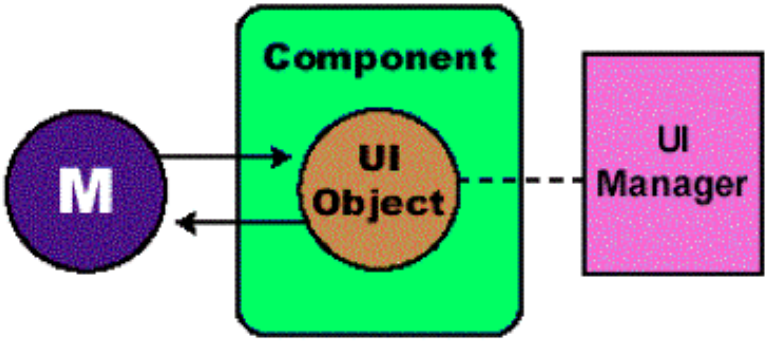
Architecture Swing (2)

"Separable Model Architecture"

- **View** et **Controller** regroupés dans un **UIComponent**
- **Model** : reste séparé

"Pluggable Look and Feel"

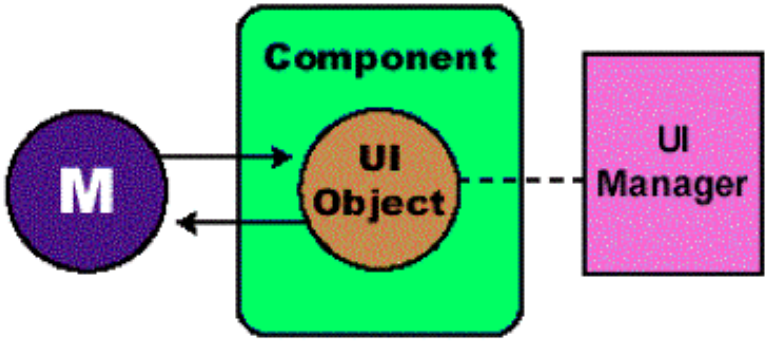
- chaque **JComponent** **Swing** encapsule un **UIComponent**
- les **UIComponent** peuvent être changés dynamiquement par le **UIManager**

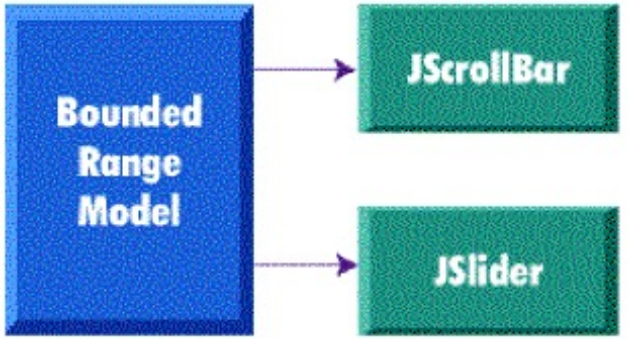


Architecture Swing (3)

Modèles et multi-vues

- (la plupart des) **JComponent Swing** créent implicitement un **Modèle**
- qui peut être "exporté" et partagé avec un autre **JComponent**





Exemple

- **JSlider** et **JScrollbar** : même modèle **BoundedRangeModel**
- mise commun du modèle => **synchronisation** automatique

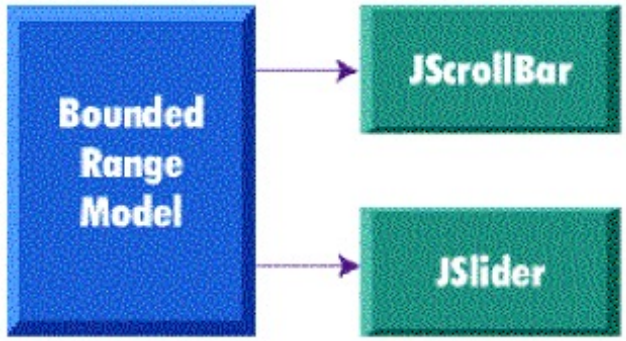
Exemple

Dans l'API de JSlider et JScrollBar :

```
public BoundedRangeModel getModel();
public void setModel(BoundedRangeModel);
```

Changer le modèle du slider et du scrollbar :

```
JSlider slider = new JSlider();
BoundedRangeModel myModel = new DefaultBoundedRangeModel( ) {
                                    public void setValue(int n) {
                                        System.out.println("SetValue: "+ n);
                                        super.setValue(n);
                                    }
                                });
slider.setModel(myModel);
scrollbar.setModel(myModel);
```



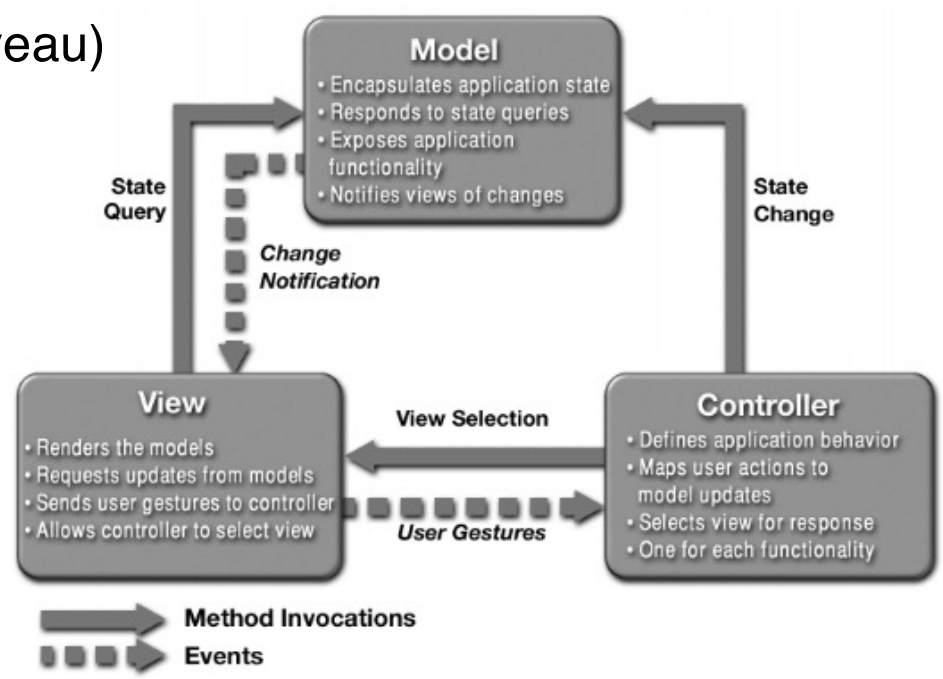
On peut aussi ignorer l'existence des modèles :

```
JSlider slider = new JSlider();
int value = slider.getValue();

// cohérent car dans l'API de JSlider :
public int getValue()  {return getModel().getValue(); }
```

Autres visions de MVC

A plus haut niveau, autre vision possible

- **Controllers = Listeners Swing**
  (Controleurs de haut niveau)
- **Views = Components Swing**
  (incluant des Controleurs de bas niveau)
- **Models**



Vision Web

- cf. illustration

Autres variantes

- model–view–adapter (MVA)
- model–view–presenter (MVP)
- model–view–view model (MVVM)
- hierarchical model–view–controller (HMVC)

Pluggable Look and Feel

Java Metal

```
public static void main(String[ ] args) {
    try {
      UIManager.setLookAndFeel(
            UIManager.getCrossPlatformLookAndFeelClassName());
    } catch (Exception e) { }
    //Create and show the GUI...
     .....
 }
```

Windows

```
      UIManager.setLookAndFeel(
          "com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
          );
```

Graphics2D

Couche graphique évoluée

- plus sophistiquée que **Graphics**

Quelques caractéristiques

- système de coordonnées indépendant du type de sortie (écran, imprimante)
- et transformations affines : translations, rotations, homothéties
  - *package java.awt.geom*
- transparence
  - *AlphaComposite, BITMASK , OPAQUE, TRANSLUCENT ...*
- Composition
- Paths et Shapes
- Fonts et Glyphs
- etc... (voir démo Java2D de Sun)

Pour en savoir plus

Site pédagogique de l'UE INF224

- http://www.telecom-paristech.fr/~elc/cours/inf224.html

UEs liées à INF224

- **IGR201**: Développement d'applications interactives 2D, 3D, Mobile et Web
- **IGR203**: Interaction Homme-Machine