

Programmation événementielle & interfaces graphiques Java Swing

Eric Lecolinet

Télécom Paristech – Dept. INFRES

www.telecom-paristech.fr/~elc

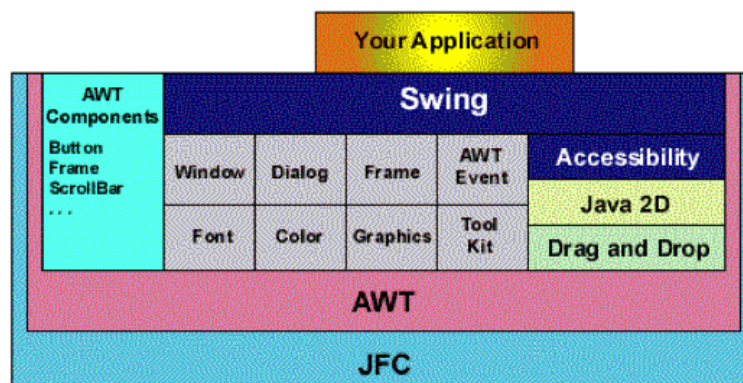
Toolkits graphiques Java

Il y en a trois !

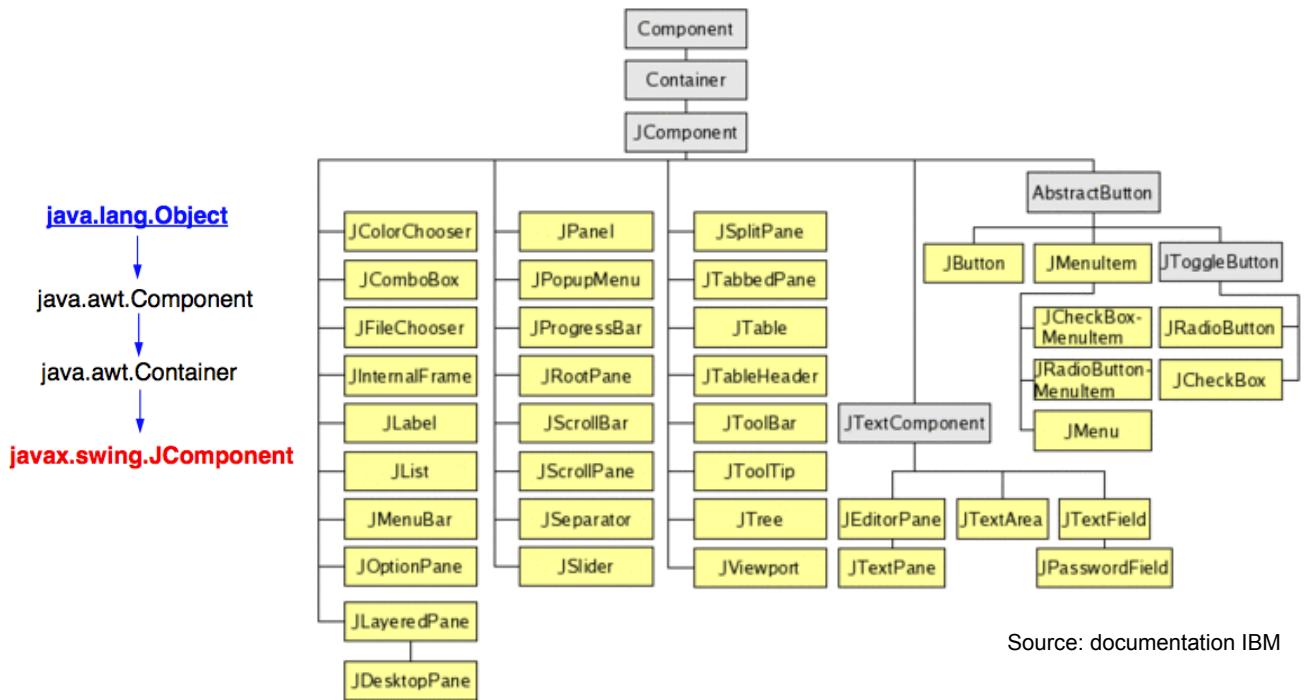
- **AWT Components**, obsolète
- **Swing** supporté par Sun/Oracle
- **SWT** libre, initié par IBM / Eclipse
- tous (+ ou -) multi-plateformes

Swing repose sur AWT

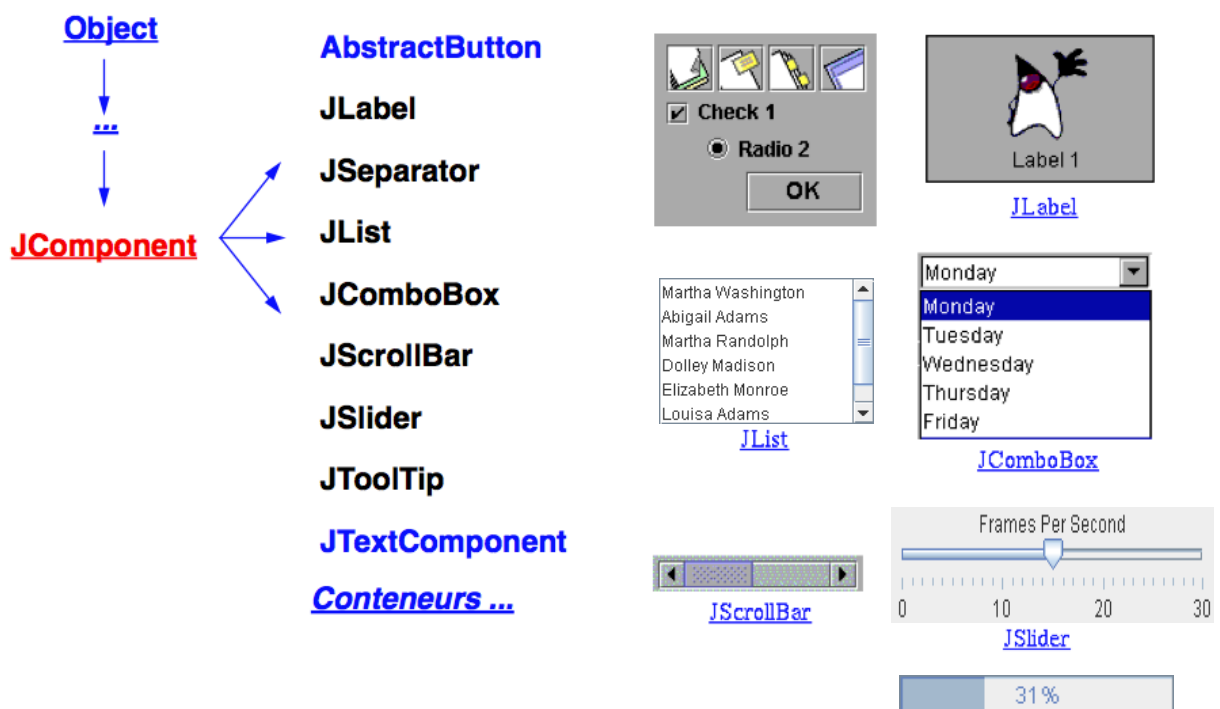
- mais Swing != AWT !
- **JButton != Button !**



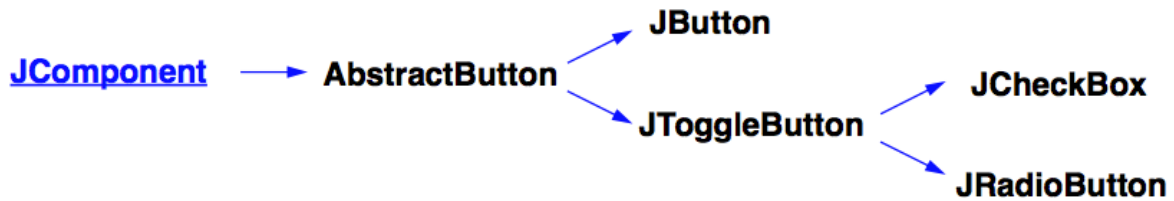
Composants Swing



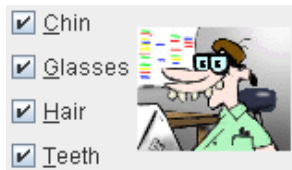
Interacteurs



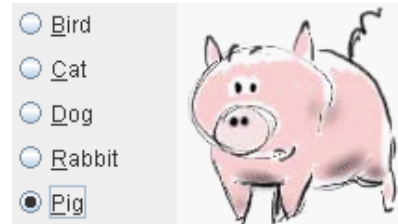
Boutons



JButton



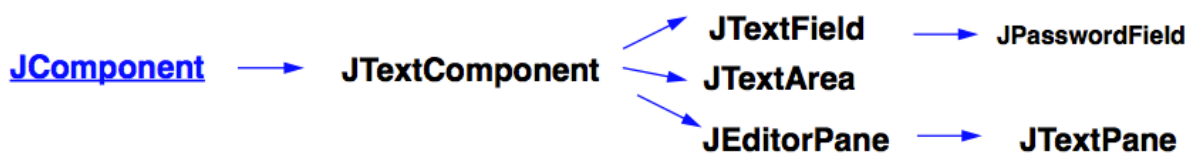
JCheckbox :
choix indépendants



JRadioButton :
choix exclusif : voir **ButtonGroup**

Source: documentation Java Oracle

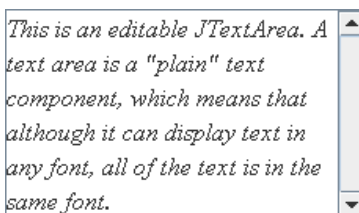
Texte



JTextField

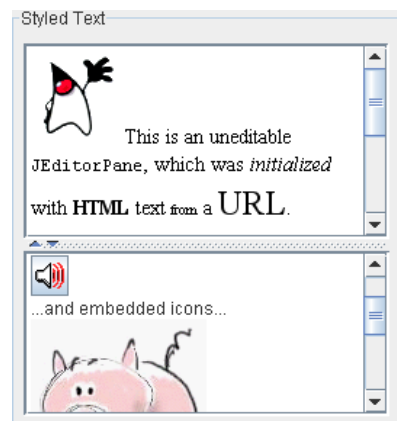


JPasswordField



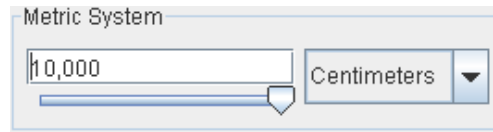
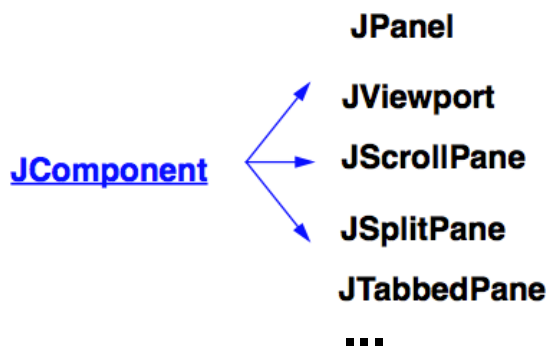
JTextArea :
texte simple multilignes

ascenseur :
voir **JScrollPane**

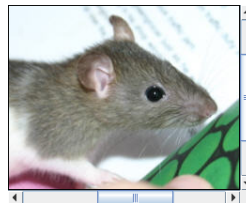


JEditorPane : texte avec styles compatible HTML et RTF

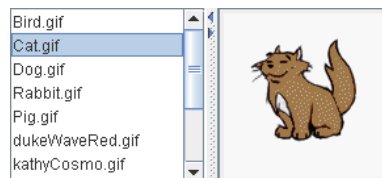
Conteneurs



JPanel: conteneur générique



JScrollPane: avec ascenseurs intégrés

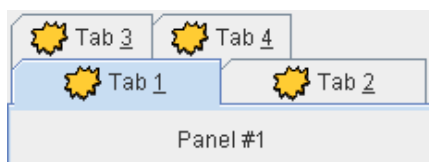


JSplitPane: avec « diviseur » intégré

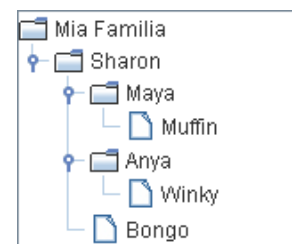
Conteneurs



JToolBar: barre d'outils



JTabbedPane: onglets

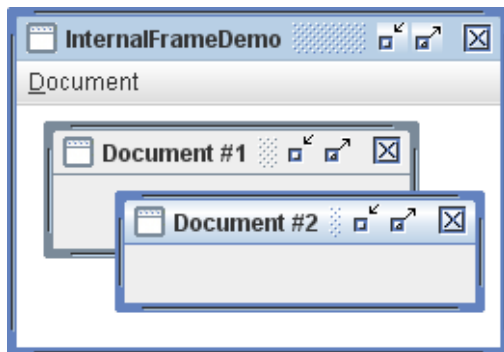


JTree

Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwwzb	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

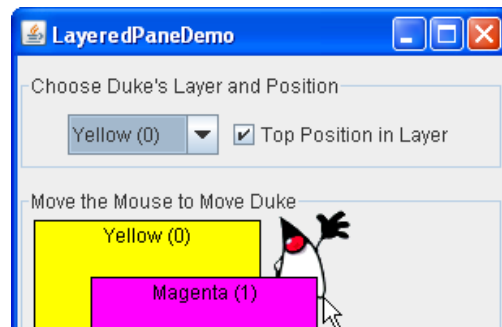
Conteneurs spécifiques



JInternalFrame

à placer dans un **JDesktopPane**
qui joue le rôle de bureau virtuel

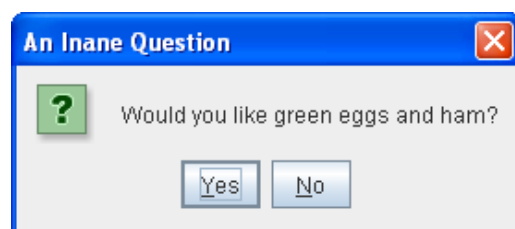
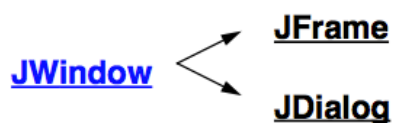
JDesktopPane hérite de **JLayeredPane**



JLayeredPane permet de superposer
des composants

voir aussi **JRootPane** présenté plus loin

Fenêtres

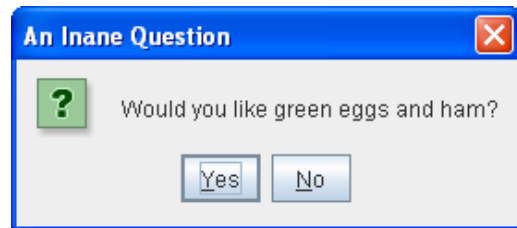
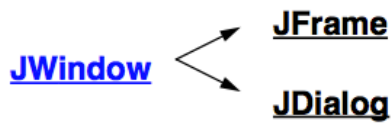


JFrame : fenêtre principale de l'application

JDialog : fenêtre secondaire

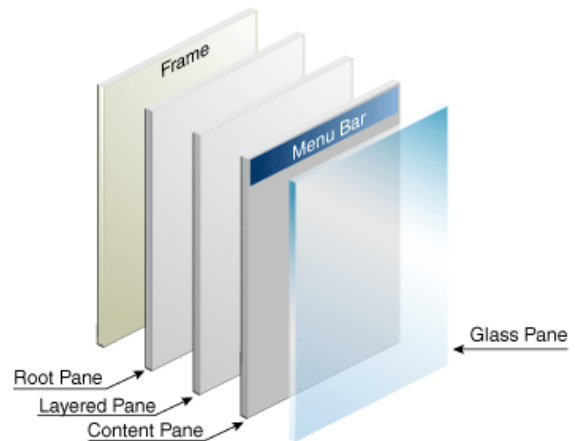
- normalement **dépendante** de la JFrame :
 - pas d'iconification séparée, toujours au dessus de la JFrame
- généralement **temporaire** et **modale** :
 - bloque l'interaction, impose à l'utilisateur de répondre

Fenêtres

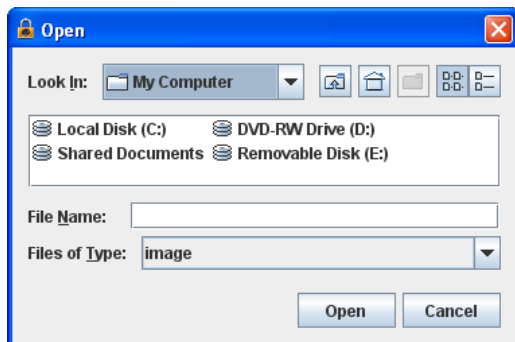


JRootPane implicitement créé

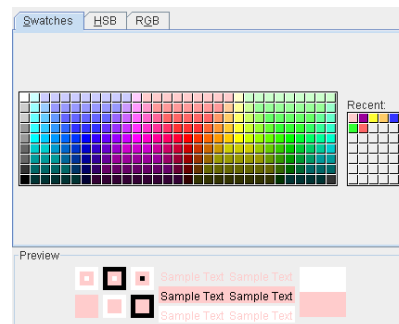
- par JApplet, JDialog, JFrame
- et JInternalFrame



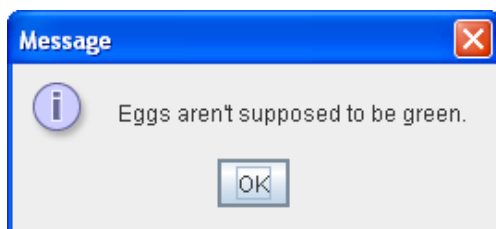
Boîtes de dialogue prédéfinies



JFileChooser



JColorChooser

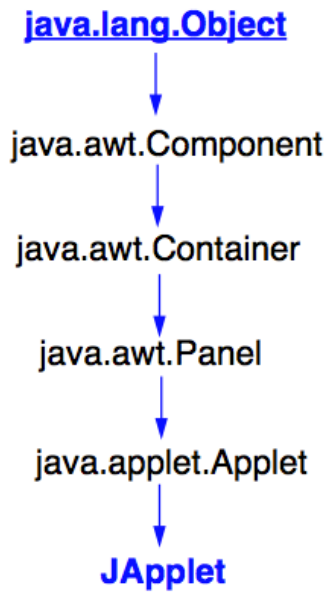


JOptionPane (multiples variantes)

Particularité

- peuvent être créés :
 - comme composants internes
 - ou comme boîtes de dialogue

Applets



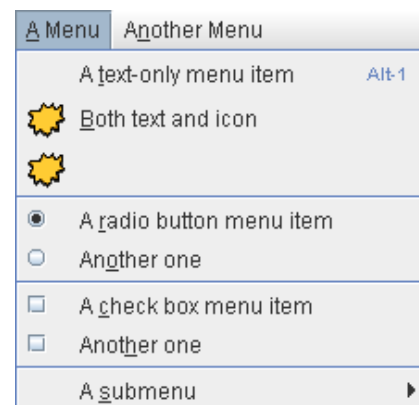
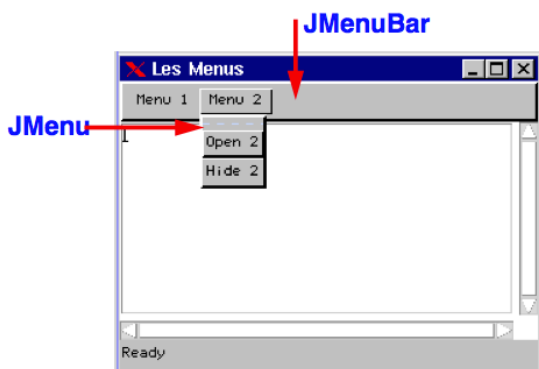
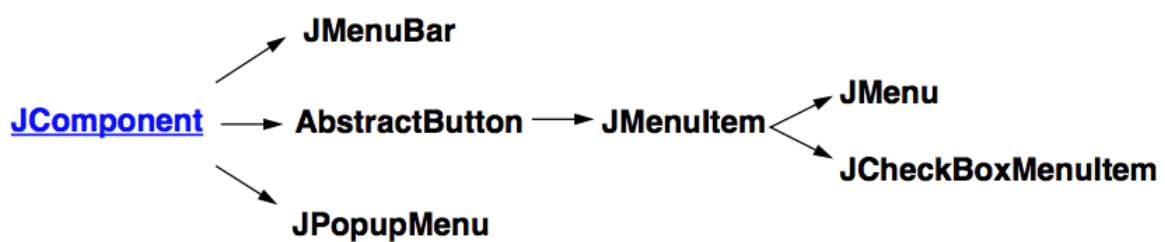
Applet = composant de premier niveau

– pour les applets Web

Attention

– restrictions diverses (accès aux fichiers, sockets...) pour des raisons de sécurité

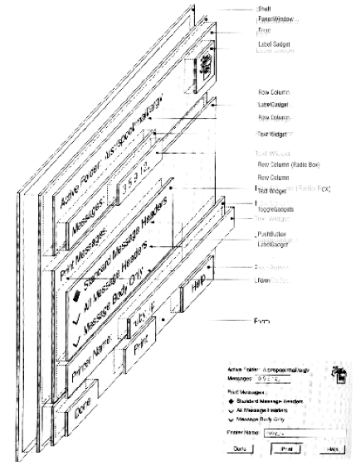
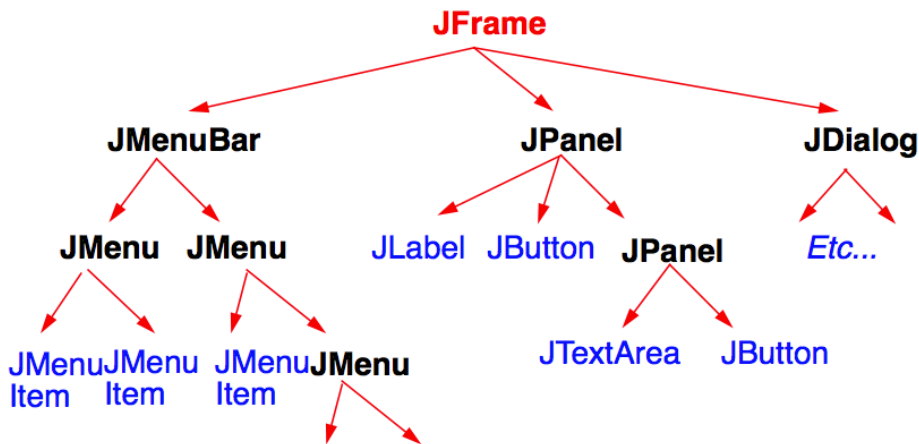
Menus



Arbre d'instanciation

Arbre d'instanciation

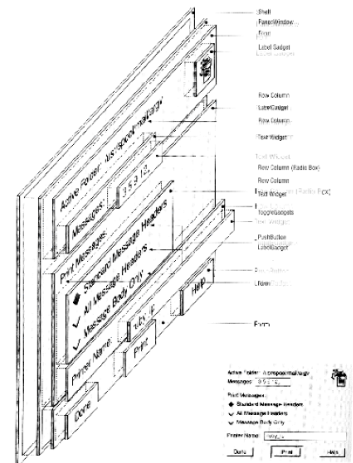
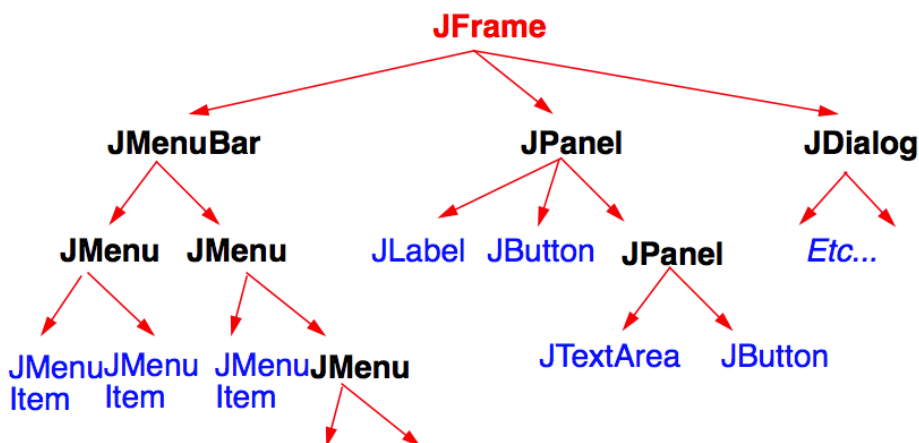
- arbre de **filiation** des instances de composants graphiques



Arbre d'instanciation

Chaque objet « contient » ses enfants

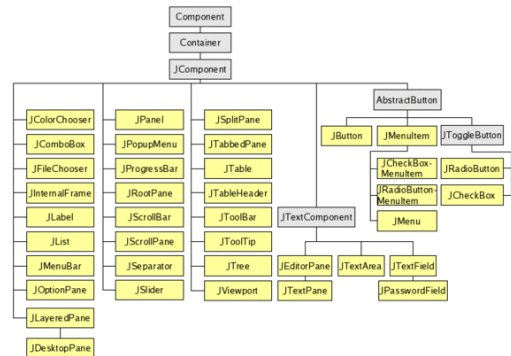
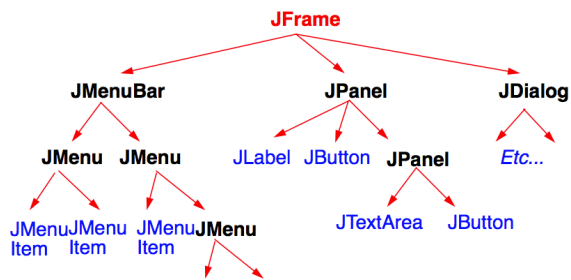
- **superposition** : enfants affichés au dessus des parents
- **clipping** : enfants « découpés » : ne dépassent pas des parents



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



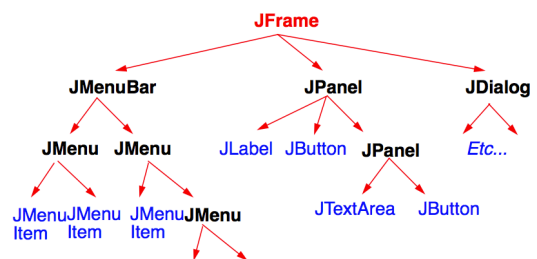
Arbre d'instanciation

Objet de plus haut niveau de l'arbre

- JFrame ou JApplet

Les conteneurs peuvent être emboîtés

- en particulier les JPanel



Les layout managers assurent la disposition spatiale

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

Ne pas oublier d'appeler

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

Exemple : version 0

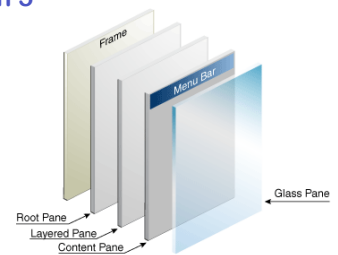
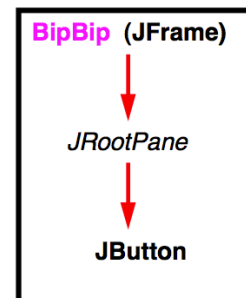
```
// donne accès aux composants Swing
import javax.swing.*;
```

```
public class BipBip extends JFrame { // fenetre 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 : nécessaire avant version 5

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



Exemple : version 0

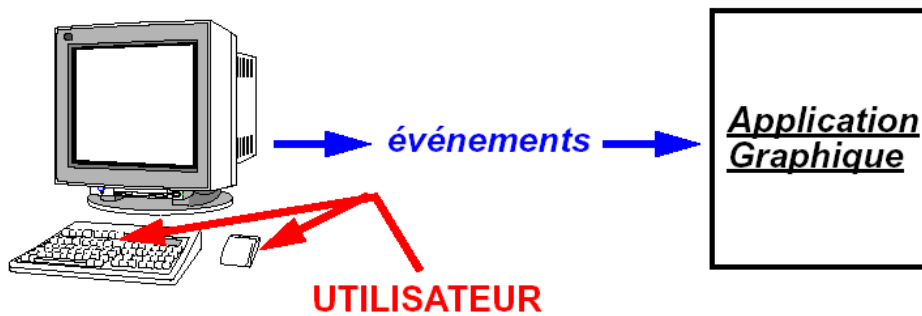
Notes et rappels

- package `javax.swing`
- une seule classe `public` par fichier, le fichier doit avoir le même nom
- `button` est une **variable d'instance** (auto-initialisée en Java contrairement à C/C++)
- `toplevel` est une **variable locale** (ne peut être auto-initialisée en Java)
- `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()` nécessaire avant la version 5 à cause du `JRootPane`
`JWindow.add()` a été redéfini dans les versions ultérieure de Java

Evénements

Evénements

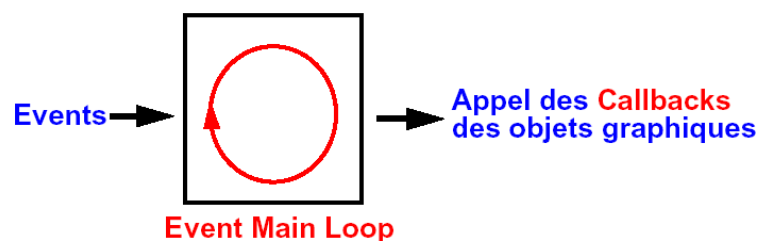
- envoyés à l'application ciblée
- à chaque action élémentaire de l'utilisateur



Boucle de gestion des événements

Boucle infinie qui

- récupère les événements
- appelle les fonctions de « callback » des composants graphiques



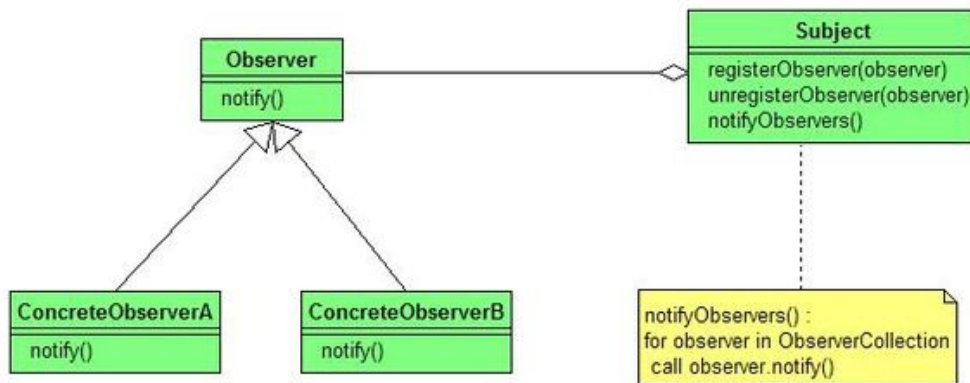
Lancée automatiquement

- à la fin de la fonction `main()` dans le cas de Java

Pattern Observateur/Observé

Principe

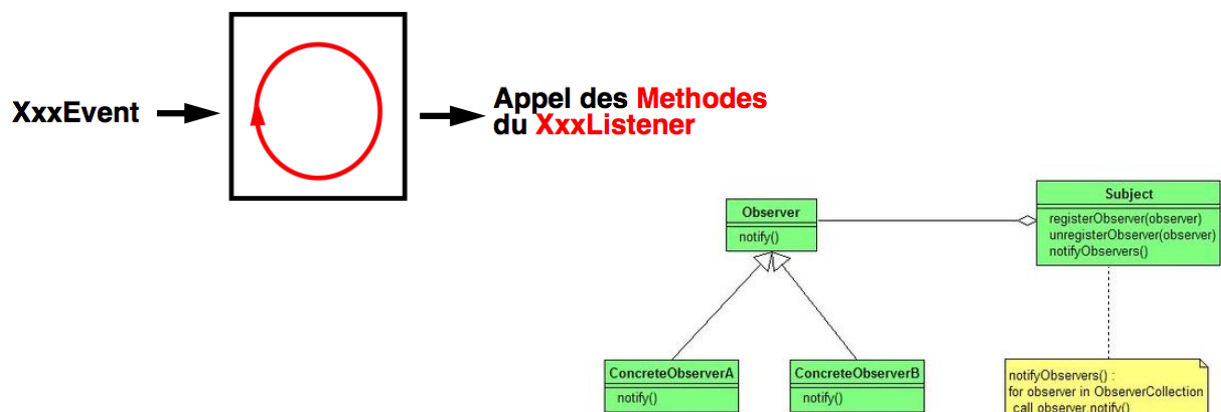
- associer un(des) observateurs(s) à un(des) objet(s) observé(s)
- observateur(s) notifié(s) automatiquement quand une certaine condition se produit sur un observé



Application aux interfaces graphiques

Les observateurs détectent les événements

- ce sont des **fonctions de callback** (en C, C++...)
- ou des objets comme les **Listeners** de Java
 - leurs méthodes servent alors de fonctions de callback



Événements Java

Événements AWT et Swing

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

Événements 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.

Événements de “haut niveau”

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

Événements Java

Méthodes communes aux AWTEvent

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

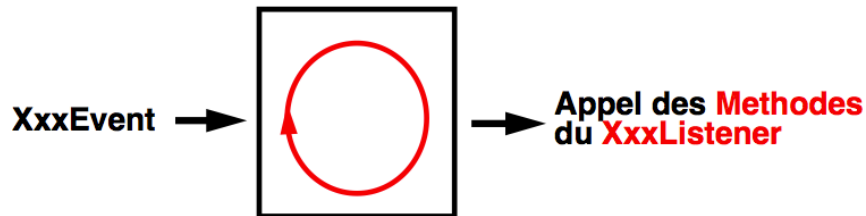
Exemple: méthodes de MouseEvent

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

Observateurs d'événements

Event Listeners

- à chaque sous-classe d'**AWTEvent** correspond un **EventListener** (en général)



Exemple : ActionEvent

- événement : **ActionEvent**
- listener : **ActionListener**
- méthode : **actionPerformed(ActionEvent)**

validation bouton:
-- clic ou space

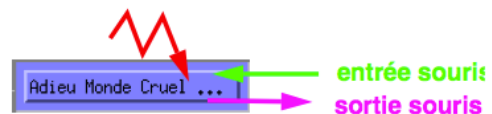


Observateurs d'événements

Exemple : MouseEvent

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

appuyer, relacher, cliquer



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

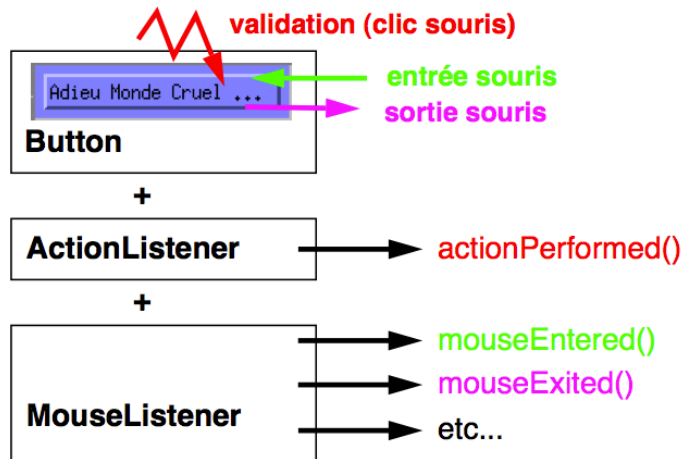
Remarque

- 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 listener peut être associé à plusieurs composants



Exemple : version 1

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

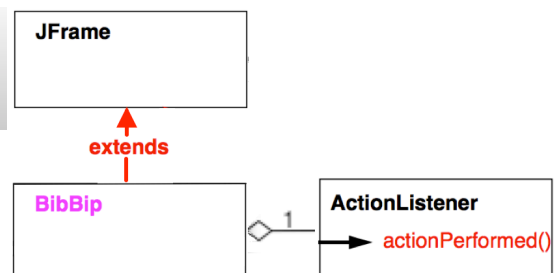
```
public class BipBip extends JFrame {
    JButton button;

    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!");
    }
}
```

Problème ?

Exemple : version 1

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

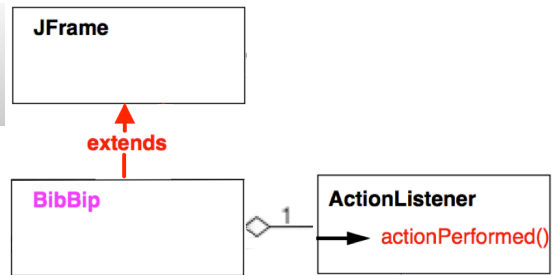
```
public class BipBip extends JFrame {
    JButton button;
    JLabel label = new JLabel();

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

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

        // créer et associer un ActionListener
        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 !
    }
}
```

Problème : communication entre objets

- comment le Listener peut-il agir sur la partie graphique ?

Exemple : version 1

```
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);

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

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

```
class Ecoute implements ActionListener {
    BipBip mainWin;

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

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

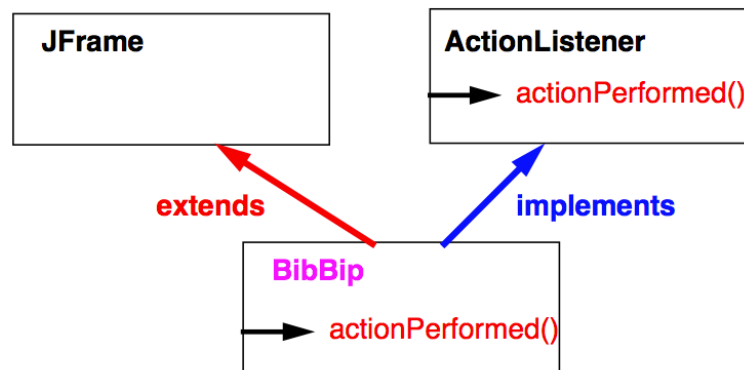
Solution

- le Listener doit conserver 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
 - simplifié dans le cas de **Java** : on ne peut « hériter » multiplement que des spécifications (i.e. des **interfaces**)



Exemple : version 2

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

```
public class BibBip extends JFrame
    implements ActionListener {
```

```
    JButton button;
    JLabel label = new JLabel();
```

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

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

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

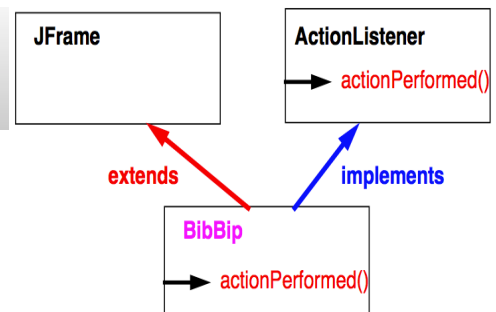
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
```

```
        public void actionPerformed(ActionEvent e) {
            label.setText("Done!");
        }
    } // fin de la classe BibBip
```

Remarque

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

Autre problème ?



Exemple : version 2

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

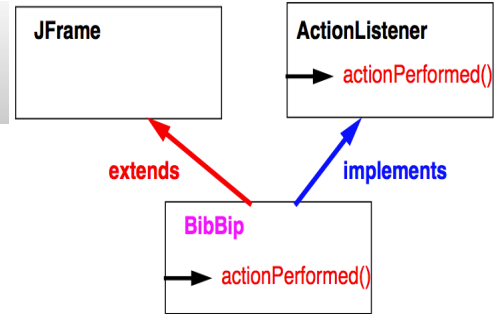
```
public class BipBip extends JFrame
    implements ActionListener {
    JButton button;
    JLabel label = new JLabel();

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

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

        // BipBip 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!");
}
```

```
} // fin de la classe BipBip
```

Problème : plusieurs boutons

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

Exemple : version 2

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

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

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

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

        dolt.addActionListener(this);
        close.addActionListener(this);

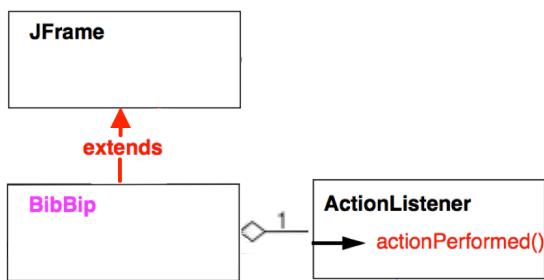
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
}
```

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == dolt)
        label.setText("Done!");
    else if (e.getSource() == close)
        System.exit(0);
}
```

Solution

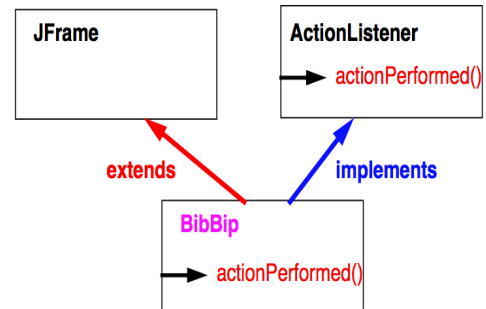
- distinguer les boutons grâce à
 - `getSource()` ou
 - `getActionCommand()`
- la 1ere solution est plus sûre
- peu adapté si beaucoup de commandes

Avantages et inconvénients



Version 1

- plus souple :
 autant de listeners que l'on veut
- mais lourd et peu concis :
 on multiplie les objets et les lignes de code



Version 2

- plus simple mais limité :
 on ne peut avoir qu'une seule méthode **actionPerformed**
- peu adapté si beaucoup de commandes

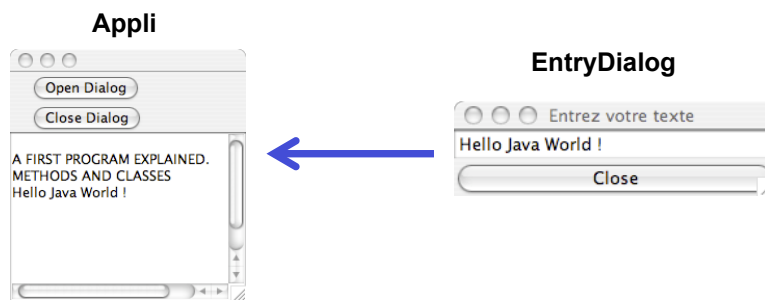
Rappels sur l'envoi de messages



Envoi de message

- Appli veut « **envoyer un message** » à EntryDialog (ex: pour modifier sa zone de texte)

Rappels sur l'envoi de messages



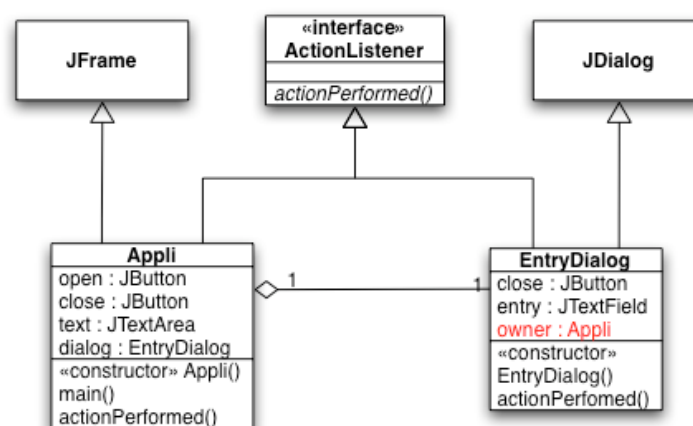
Dans tous les cas

- Appli doit conserver l'adresse d'EntryDialog
- typiquement via une **variable d'instance** (pas une variable de classe !)

Avec les langages à typage statique (Java, C++, C#...)

- l'envoi de message se traduit par un **appel de méthode**
- ⇒ Appli doit connaître le type d'EntryDialog et cette classe doit avoir la méthode adéquate

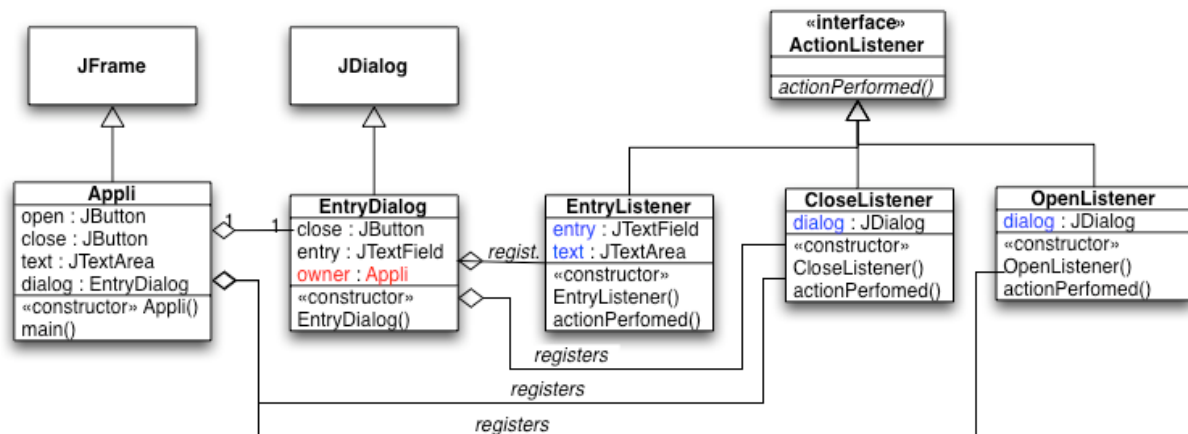
Une possible implémentation



Pseudo héritage multiple des Listeners via **implements**

- comme dans « Version 2 »
- même problème : solution limitée, peu adaptée si beaucoup de commandes

Une autre implémentation



Autant de Listeners que d'actions à effectuer

- comme dans « Version 1 »
- même avantage : souplesse
- même inconvénient : relative complexité

3eme solution : classes imbriquées

Classes définies à l'intérieur d'une autre classe

- ont accès aux **variables d'instance** des classes qui les contiennent
- attention : ce n'est pas le cas en C++ !

Combinent les avantages des 2 solutions précédentes

- souplesse sans la complexité

Exemple : version 3

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

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

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

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

        dolt.addActionListener(new DoltListener ());
        close.addActionListener(new CloseListener ());

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

class DoltListener 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 BipBip
```

Remarque

- `actionPerformed()` à accès à `label` car `DoltListener` est une classe imbriquée de `BipBip`

Remarques

Remarques

- les classes imbriquées peuvent également servir à **encapsuler l'implémentation**
- le terme exact est plutôt **classes internes** (**inner classes** en Anglais)
- il existe aussi des classes imbriquées statiques = **nested classes**
 - servent surtout à structurer en sous-parties

Classes imbriquées anonymes

Classes imbriquées... qui n'ont pas de nom !

```
public class BipBip extends JFrame {
    JLabel label = new JLabel();    // label doit être une variable d'instance (pas une variable locale)
    .....                          // car elle est référencée dans une méthode d'instance

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

        doIt.addActionListener( new ActionListener() {    // sous-classe anonyme de ActionListener
            public void actionPerformed(ActionEvent e) {
                label.setText("Done!");
            }
        });
    }
    .....
}
```

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

Remarques sur les constructeurs

Un constructeur peut en appeler un autre

```
abstract class MyButton extends JButton implements ActionListener {

    public MyButton(String name, Icon icon) {
        super(name, icon);
        .....
    }

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

Remarques :

- pas possible en C++ (sauf pour C++11)
- par contre C++ accepte les paramètres par défaut

Remarques sur les constructeurs

Constructeur de classe

```
abstract class Toto {  
  
    static {          // constructeur de classe  
        .....  
    }  
}
```

Remarques :

- sert à effectuer des opérations sur les variables de classe
- n'existe pas en C++

Applets

Application

- Programme indépendant
 - interprété par la commande **java**
- Structure
 - hérite de **JFrame** (pour une GUI)
 - méthode **main()**
 - peut être interfacée avec un autre langage (interfaces **natives**)

Applet (appliquette)

- Programme pour navigateur Web
 - interprété par navigateur Web ou commande **appletviewer**
 - **attention**: restrictions d'accès (fichiers, sockets ...)
- Structure
 - hérite de **JApplet**
 - méthode **init()** (pas de new !)

Scribble : dessin dans une applet

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

public class Scribble extends JApplet
    implements MouseListener, MouseMotionListener
{
    private int last_x, last_y;

    public void init() {
        // enregistrer les Listeners
        this.addMouseListener(this);
        this.addMouseMotionListener(this);
        this.setBackground(Color.white);
    }

    // méthode de l'interface MouseListener
    public void mousePressed(MouseEvent e) {
        last_x = e.getX();
        last_y = e.getY();
    }
}
```



Scribble : dessin dans une applet

....(suite)....

```
// méthode de l'interface MouseMotionListener
public void mouseDragged(MouseEvent e) {
    Graphics g = getGraphics();
    int x = e.getX();
    int y = e.getY();

    g.drawLine(last_x, last_y, x, y);
    last_x = x;
    last_y = y;
}

// méthodes inutilisées des Listeners
public void mouseReleased (MouseEvent e) {}
public void mouseClicked (MouseEvent e) {}
public void mouseEntered (MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseMoved(MouseEvent e) {}
}
```

Scribble : dessin dans une applet

Dans la page Web

```
<!-- Applet Scribble : code a inserer dans le fichier HTML -->
<html>
  <body>
    <applet code="Scribble.class" width="300" height="300"> </applet>
  </body>
</html>
```

Pour tester l'applet

- **appletviewer** fichier-html
- ou avec votre navigateur favori

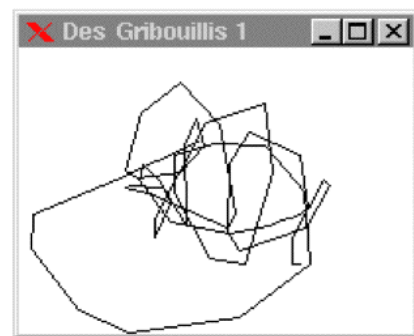
Avec les classes imbriquées

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

public class Scribble extends JApplet {
  private int last_x, last_y;

  public void init() {
    !! NB: rajouter getContentPane(). avant Java 5 !
    setBackground(Color.white);

    // définir, instancier et enregistrer le Listener
    addMouseListener(
      new MouseAdapter () {
        public void mousePressed(MouseEvent e) {
          last_x = e.getX();
          last_y = e.getY();
        }
      }
    );
  }
}
```



Avec les classes imbriquées

....(suite)....

```
addMouseListener(  
    new MouseMotionAdapter ( ) {  
        public void mouseDragged(MouseEvent e) {  
            // même code que dans la précédente version  
            ....  
        }  
    });  
  
// ajouter un bouton qui efface tout  
JButton b = new JButton("Clear");  
add(b);  
b.addActionListener(  
    new ActionListener ( ) {  
        public void actionPerformed(ActionEvent e) {  
            Graphics g = getGraphics( );  
            g.setColor (getBackground( ));  
            g.fillRect (0, 0, getSize( ).width, getSize( ).height);  
        }  
    });
```

Persistance de l'affichage

Problèmes de l'exemple précédent

- 1) l'affichage du dessin n'est pas persistant !
 - il est effacé si on déplace une fenêtre dessus (en fait ça dépend des plateformes)

- 2) normalement les méthodes des listeners ne doivent pas dessiner

Persistance de l'affichage

Problèmes de l'exemple précédent

- 1) l'affichage du dessin n'est pas persistant !
 - il est effacé si on déplace une fenêtre dessus (en fait ça dépend des plateformes)
- 2) normalement les méthodes des listeners ne doivent pas dessiner

Solution

- mémoriser la liste des opérations graphiques dans une « display list »
- réafficher le dessin quand le composant qui le contient est rafraîchi

Réafficher le dessin

Pour réafficher le dessin avec AWT

- redéfinir la méthode `paint()`

Pour réafficher le dessin avec Swing

- redéfinir la méthode `paintComponent()`
- car `paint()` appelle `paintComponent()` puis `paintBorder()` puis `paintChildren()`

```
class Dessin extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);           // ne pas oublier cette ligne !  
        .....  
    }  
}
```

Réafficher le dessin

Pour indiquer qu'il faut réafficher le dessin

- dans une méthode d'un Listener il faut appeler `repaint()`
- les `repaint()` sont « compactés » :
 - un seul réaffichage même si on appelle `repaint()` plusieurs fois

Divers

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()` => plus rapide

JFileChooser

```
JFileChooser chooser = new JFileChooser();

ExampleFileFilter filter = new ExampleFileFilter();
filter.addExtension("jpg");
filter.addExtension("gif");
filter.setDescription("JPG & GIF Images");
chooser.setFileFilter(filter);

int returnVal = chooser.showOpenDialog(parent);

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

Disposition spatiale

Les **LayoutManagers**

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

A chaque conteneur est associé un **LayoutManager**

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

Pour faire le calcul "à la main"

- à éviter sauf cas particuliers
 - conteneur.**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 longueur du texte
 - langues orientales : texte ~1/3 plus **petit** que l'anglais
 - français, allemand : texte ~1/3 plus **grand** que l'anglais

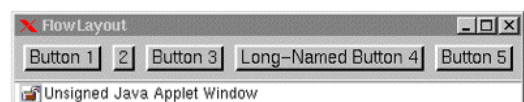
Adaptativité des interfaces

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

Principaux LayoutManagers

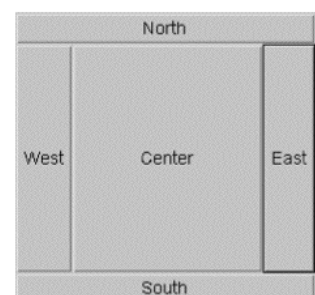
FlowLayout

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



BorderLayout

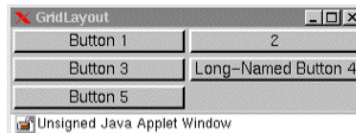
- défaut des **JFrame** et **JDialog**
- retaille **automatiquement** les enfants du conteneur
- disposition de type points cardinaux
 - via constantes:
BorderLayout.CENTER,
EAST, NORTH, SOUTH, WEST



Principaux LayoutManagers (2)

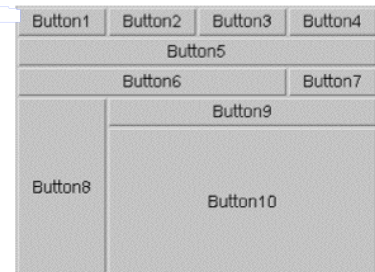
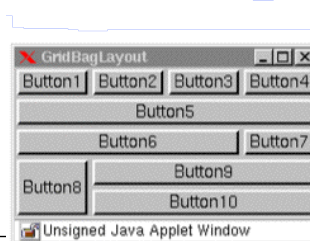
GridLayout

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



GridBagLayout

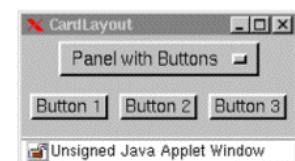
- grille + **contraintes spatiales**
 - les enfants n'ont pas tous la même taille
 - spécification par des **GridBagConstraints**



Principaux LayoutManagers (3)

CardLayout

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



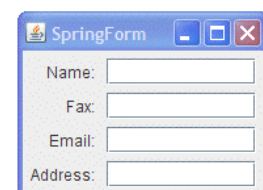
BoxLayout

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



SpringLayout

- contraintes entre les bords des enfants



Autres toolkits graphiques Java

AWT Components

- "Abstract Widget Toolkit"
- plus ancien et moins puissant que **Swing**
- attention: même noms que **Swing** mais ... **sans le J !**
 - exemple: **JButton** (Swing) et **Button** (AWT)

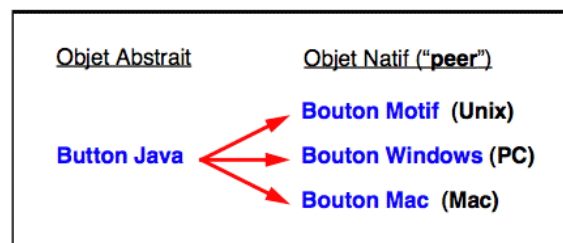
SWT

- "Standard Widget Toolkit"
- développé pour **Eclipse**
- open source
- même type d'architecture que AWT

AWT versus Swing

■ AWT

- couche "abstraite" qui encapsule les **widgets natifs** de la plateforme
- **look & feel** différent suivant l'OS



■ Swing

- réimplémente tous les widgets en **simulant** les look & feel natifs
 - => look & feel indépendant de l'OS
 - => comportement (à peu près) homogène qq soit l'OS
- architecture logicielle plus sophistiquée
- bien plus puissant !

Architecture Swing

■ Swing est inspiré du modèle MVC

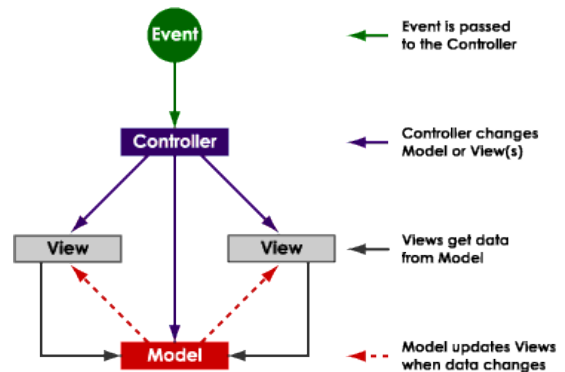
- **Model** : données de l'application
- **View** : représentation visuelle
- **Controller** : gestion des entrées

■ But de MVC

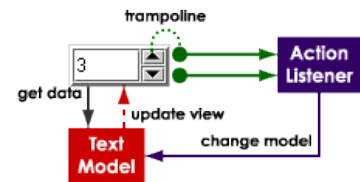
- mieux structurer les applications
- représentations **multi-vues**
 - un **modèle** peut être associé à plusieurs **vues**
 - la synchronisation est implicite

■ Remarques

- en pratique, **V** est fortement lié à **C**
- il existe des **variantes** de MVC !



source: enode.com



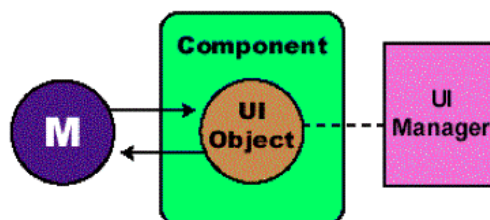
Architecture Swing (2)

■ "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 :

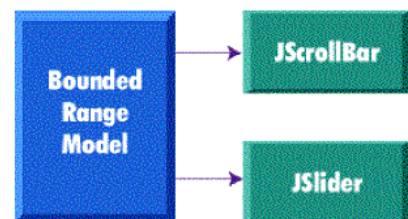
```
public BoundedRangeModel getModel();  
public void setModel(BoundedRangeModel);
```

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();}
```



Pluggable Look and Feel

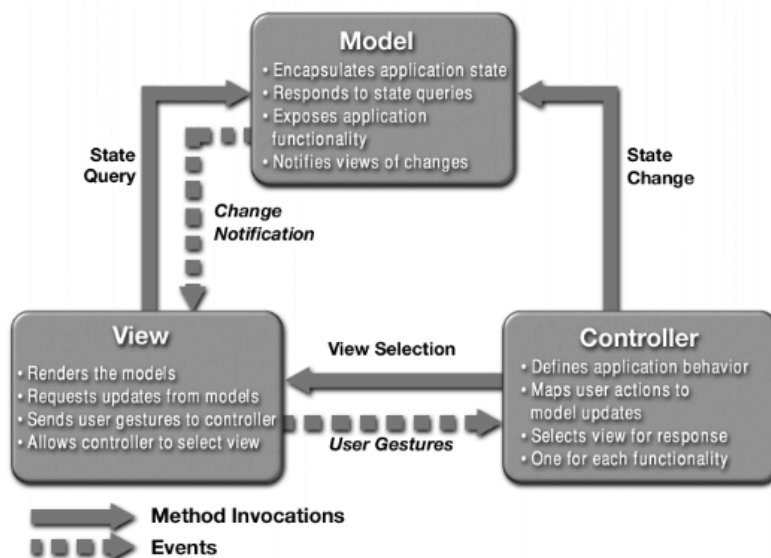
Java Metal

```
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"
);
```

Une variante de MVC



source: Sun

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>

■ Pour aller plus loin : UEs liées à INF224

- **UE INF222** : Modèles et vérification
- **UE INF355** : Paradigmes et langages non classiques
- **UE INFSI351**: Interfaces Homme-Machine, conception, visualisation, Flash, rendu/manipulation des images