

FIP/Mise à niveau

S. Rosmorduc

Plan du cours

- Cours 1 : concepts généraux, premiers exemples
- Cours 2 : Graphics et création de nouveaux composants
- Cours 3 : Architecture MVC et composants Swing

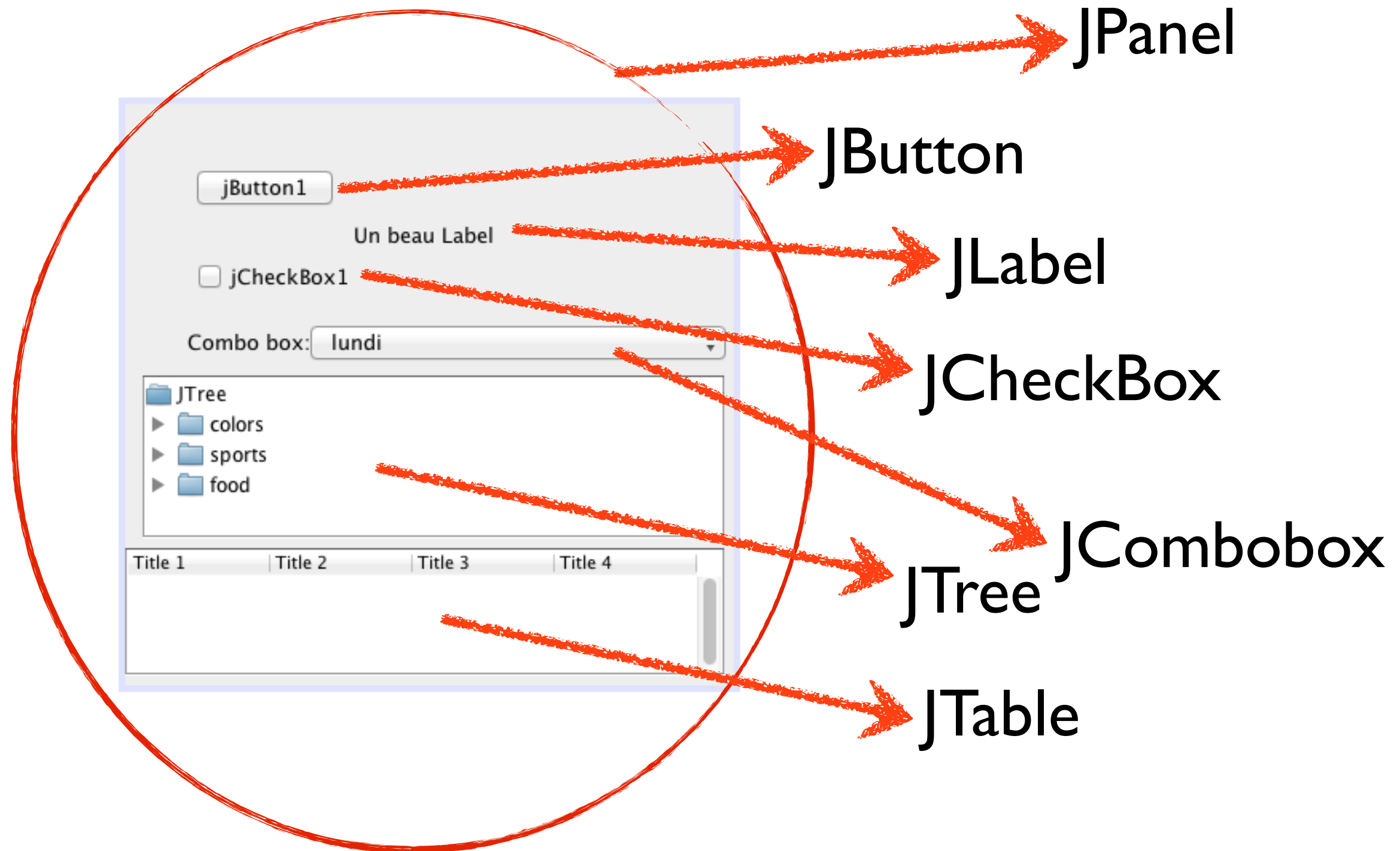
Plan

- introduction (Swing, AWT)
- Notion de composant graphique (et héritage)
- Programmation événementielle
- les conteneurs
- Mise en page
- Quelques objets
- les listeners

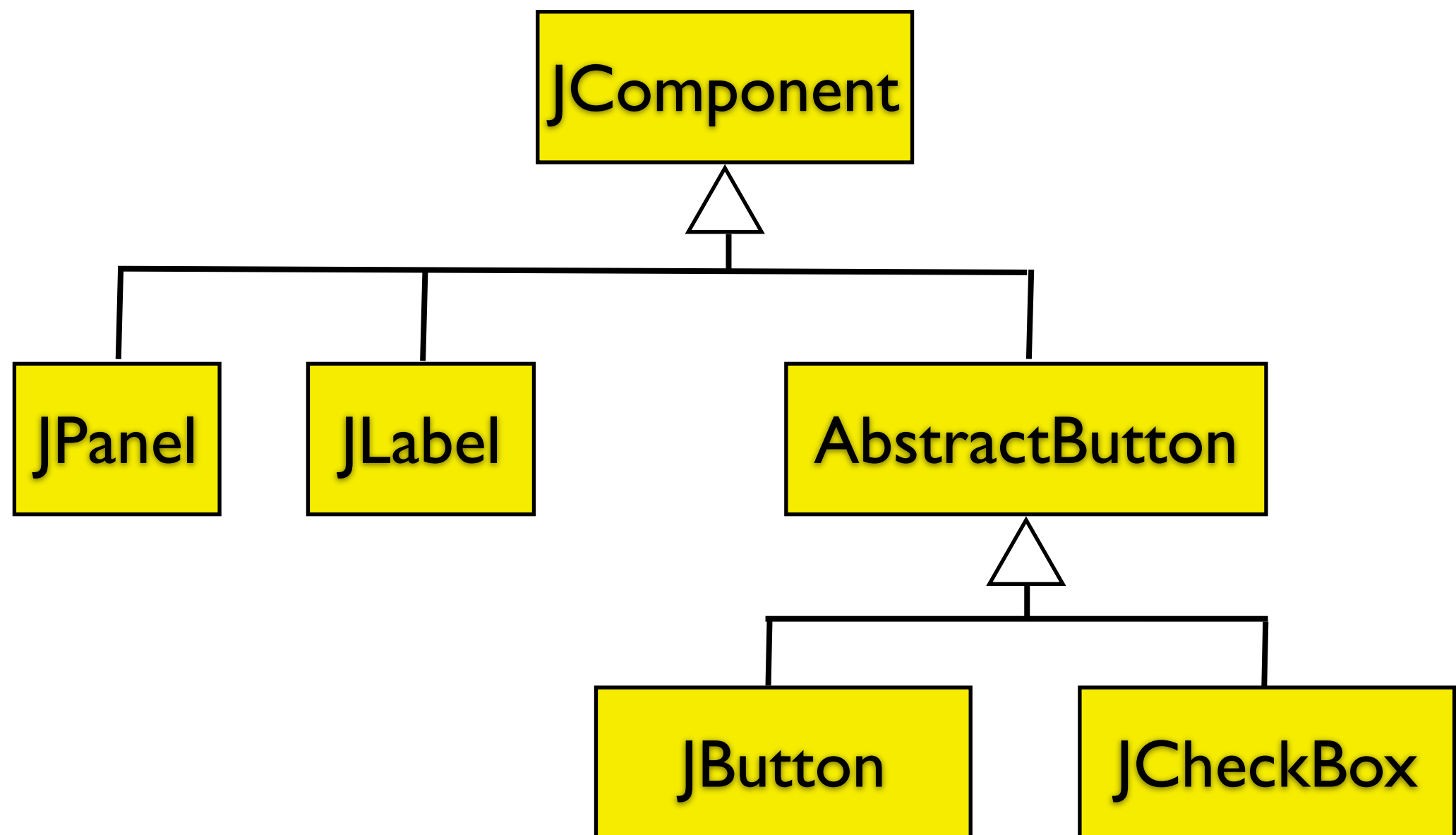
Un peu d'histoire

- java 1.0 : bibliothèque graphique AWT, composants «lourds» (ceux du système d'exploitation). Uniquement des composants qui existent partout
- java 2.0 : Swing, construit sur la couche graphique d'AWT. Composants écrits en java, très (trop ?) modulables.
- actuellement, javafx. Compatible avec Swing, mais pas encore assez riche en composants.
- Pourquoi enseigner Swing: principes très similaires dans tous les systèmes graphiques.

Composants graphiques

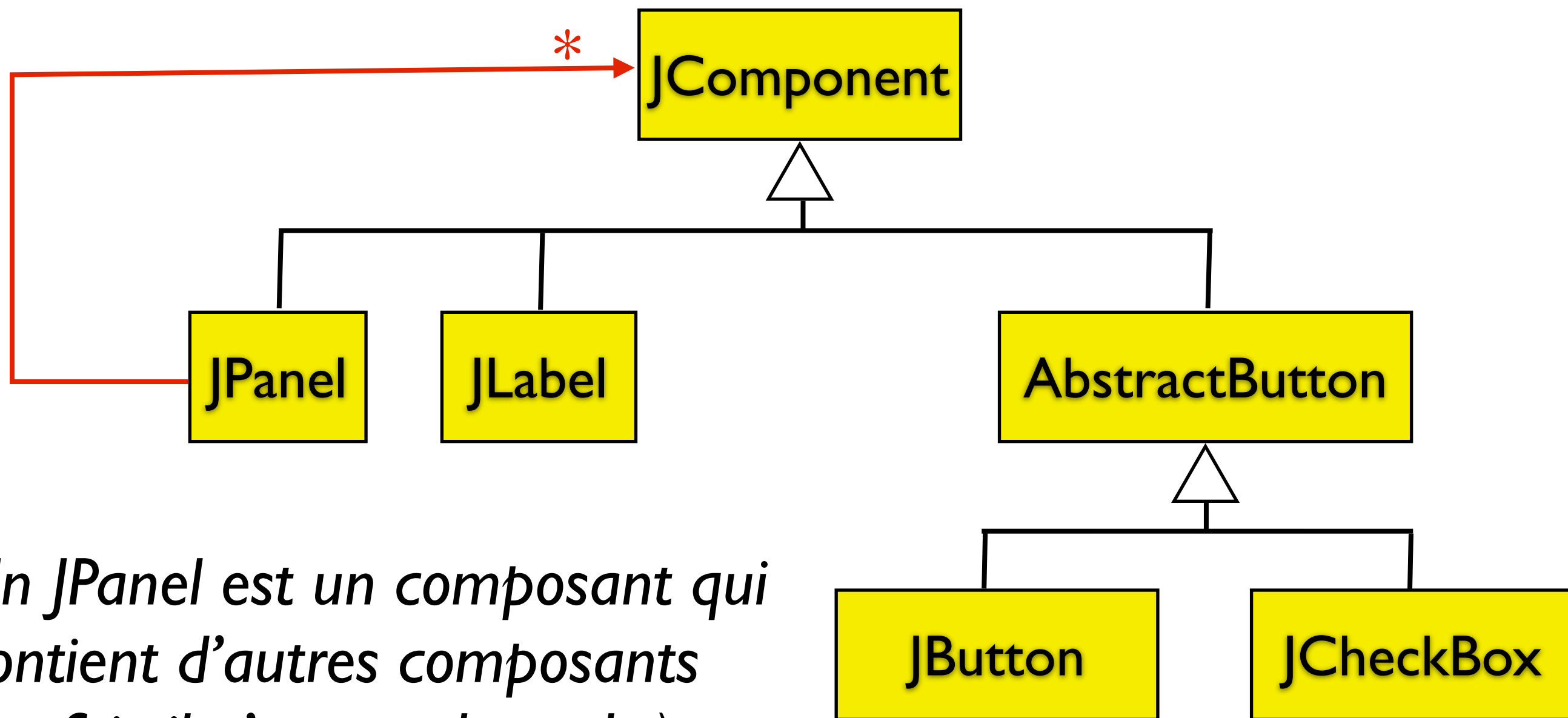


Composants...



Composite...

(attention : pattern !!!)



Un JPanel est un composant qui contient d'autres composants (en fait, il n'est pas le seul...)

Mise en page d'un JPanel

- le JPanel ne gère pas directement le positionnement des éléments qu'il contient
- il délègue ce travail à un «LayoutManager» (pattern **Stratégie**)
- Il existe des layouts pour beaucoup de mises en pages différentes : FlowLayout, BorderLayout, GridLayout... des layouts extérieurs : MigLayout. Netbeans simplifie le travail

Programmation événementielle

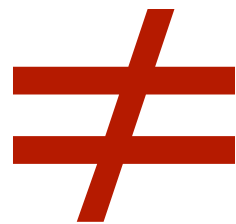
Programme séquentiel en mode texte:

demander a

demander b

$s = a + b$

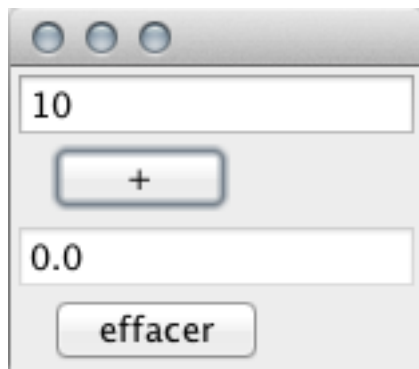
afficher s

A screenshot of a simple graphical user interface (GUI) window. It contains three input fields: 'Premier nombre' (First number), 'Second nombre' (Second number), and 'Résultat' (Result). Between the first and second input fields is a button labeled 'Ajouter' (Add). The window has a light gray background and a blue border.

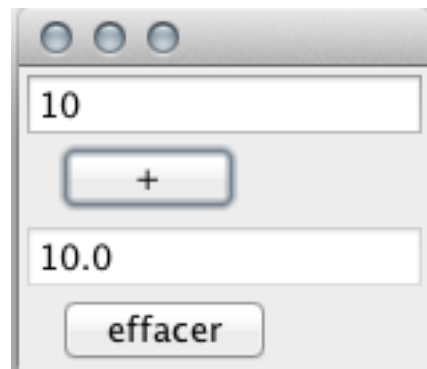
*un **événement**
(clic, etc) se produit
et le programme
réagit.*



Un exemple simple de programme graphique



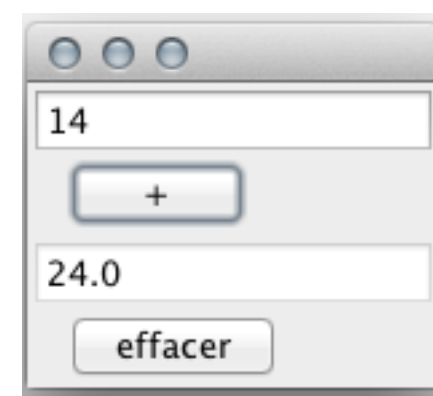
A calculator window with three standard window controls at the top. It contains two input fields: the top one has the value "10" and the bottom one has "0.0". Between the fields is a button with a "+" sign. Below the bottom field is a button labeled "effacer".



A calculator window with three standard window controls at the top. It contains two input fields: the top one has the value "10" and the bottom one has "10.0". Between the fields is a button with a "+" sign. Below the bottom field is a button labeled "effacer".



A calculator window with three standard window controls at the top. It contains two input fields: the top one has the value "14" and the bottom one has "10.0". Between the fields is a button with a "+" sign. Below the bottom field is a button labeled "effacer".

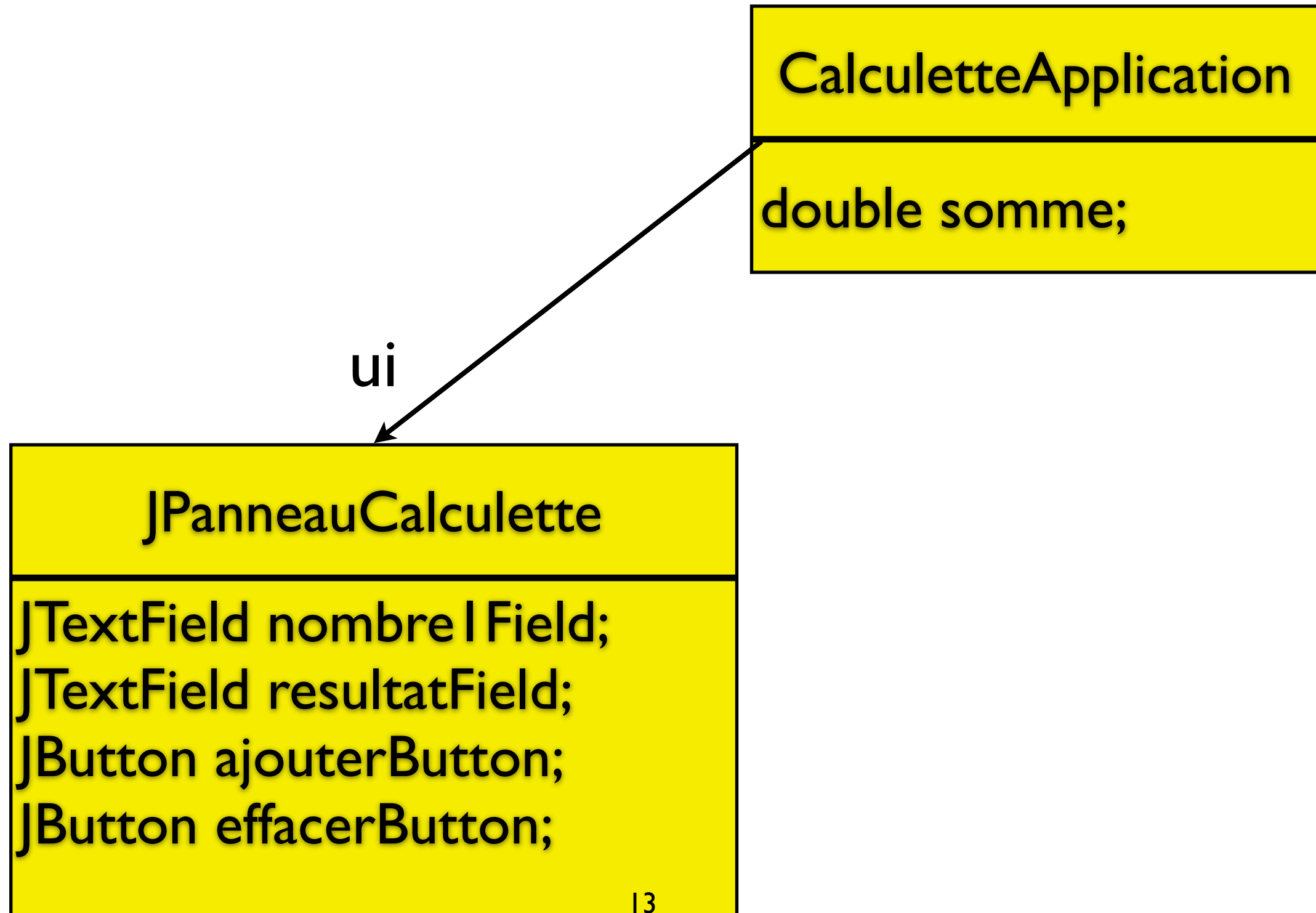


A calculator window with three standard window controls at the top. It contains two input fields: the top one has the value "14" and the bottom one has "24.0". Between the fields is a button with a "+" sign. Below the bottom field is a button labeled "effacer".

Architecture

- Une classe pour le main (lance le programme)
- une classe représentant la fenêtre graphique
- une classe représentant l'application, qui connaît le modèle et la fenêtre graphique.

Architecture (v1.0)



Fenêtre graphique

- On met le moins de logique possible dans cette classe. Plus simple à modifier, remplacer, etc. à long terme
- fenêtre contenant des composants
- fournit des accesseurs pour que l'application puisse modifier les composants.

Fenêtre graphique

```
public class JPanneauCalcullette extends JFrame {
    private JTextField nombre1Field, resultatField;
    private JButton ajouterButton;
    private JButton effacerButton;

    public JPanneauCalcullette() {
        nombre1Field= new JTextField(10);
        resultatField= new JTextField(10);
        resultatField.setEditable(false); // pas modifiable.
        ajouterButton= new JButton("+");
        effacerButton= new JButton("effacer");
        mettreEnPage();
    }

    public JTextField getNombre1Field() {
        return nombre1Field;
    }

    // et autres getters : getEffacerButton, getResultatField,
    // getAjouterButton
}
```

Fenêtre graphique

```
private void mettreEnPage() {  
    // On fixe le "layout" : on va empiler les composants  
    setLayout(new BorderLayout(panel, BorderLayout.PAGE_AXIS));  
    add(nombre1Field);  
    add(ajouterButton);  
    add(resultatField);  
    add(effacerButton);  
    // On fixe la taille de la frame :  
    this.pack();  
}  
}
```


Application...

```
public class CalculetteApplication {
    private JPanneauCalculette ui;
    private double somme; // modèle !

    public CalculetteApplication() {
        ui = new JPanneauCalculette();
        activer();
        ui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ui.setVisible(true);
    }

    void setSomme(double somme) {
        this.somme = somme;
        ui.getResultatField().setText(""+somme);
    }

    public void ajouter() {
        double val = Double.parseDouble(ui.getNombre1Field().getText());
        setSomme(somme + val);
    }

    public void effacer() {
        ui.getNombre1Field().setText("0");
        setSomme(0.0);
    }
}
```

Association d'une action à un bouton

- on écrit une classe qui implante **ActionListener**

```
class EffacerActionListener implements ActionListener {  
    private final CalculetteApplication calculetteApplication;
```

```
    EffacerActionListener(CalculetteApplication  
calculetteApplication) {  
        this.calculetteApplication = calculetteApplication;  
    }
```

```
@Override
```

```
public void actionPerformed(ActionEvent e) {  
    calculetteApplication.effacer();  
}
```

```
}
```

actionPerformed sera exécutée
quand on presse le bouton

Ajout d'un actionlistener à un bouton

- On appelle addActionListener sur le bouton, en passant l'actionlister comme argument:

```
ActionListener monActionLister=  
    new MonActionListener();  
monButton.addActionListener(monActionLister);
```

Java 8

- On peut utiliser les « lambda expression », et c'est bien plus rapide à coder:

```
public class Appli {  
    JFormulaire maFrame;  
    MonModele modele;  
    ...  
    private void activer() {  
        maFrame.getValiderButton().addActionListener(  
            e -> valider());  
    }  
    public void valider() {  
        ... code de la méthode ...  
    }  
}
```

Problème

- L'actionlistener doit avoir une référence sur le modèle (ou sur l'application) pour pouvoir agir.
- Exemple :

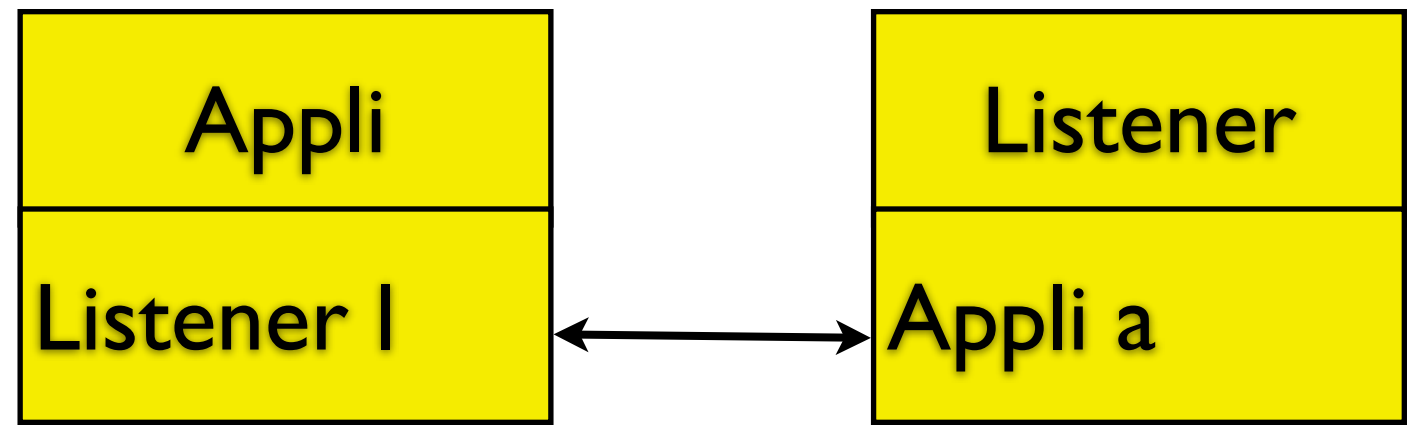
```
class AjouterActionListener implements ActionListener {  
    private final CalculetteApplication app;           var. d'instance  
  
    AjouterActionListener(CalculetteApplication app) {  
        this.app = app;                                   passée au constructeur  
    }  
    public void actionPerformed(ActionEvent e) {  
        this.app.ajouter();                               utilisée dans actionPerformed  
    }  
}
```

Lien bouton/ actionlistener

- Dans **CalculetteApplication** :

```
public class CalculetteApplication {  
    private JPanneauCalculette ui;  
    private double somme;  
  
    public CalculetteApplication() {  
        ui= new JPanneauCalculette();  
        activer();  
    }  
    private void activer() {  
        ui.getAjouterButton().addActionListener(  
            new AjouterActionListener(this));  
        ui.getEffacerButton().addActionListener(  
            new EffacerActionListener(this));  
    }  
}
```

Lien bidirectionnel entre objets



```
class Appli {  
    private Listener l;  
    public Appli() {  
        l = new Listener(this);  
    }  
}
```

```
class Listener {  
    private Appli a;  
    Listener(Appli a) {  
        this.a = a;  
    }  
}
```

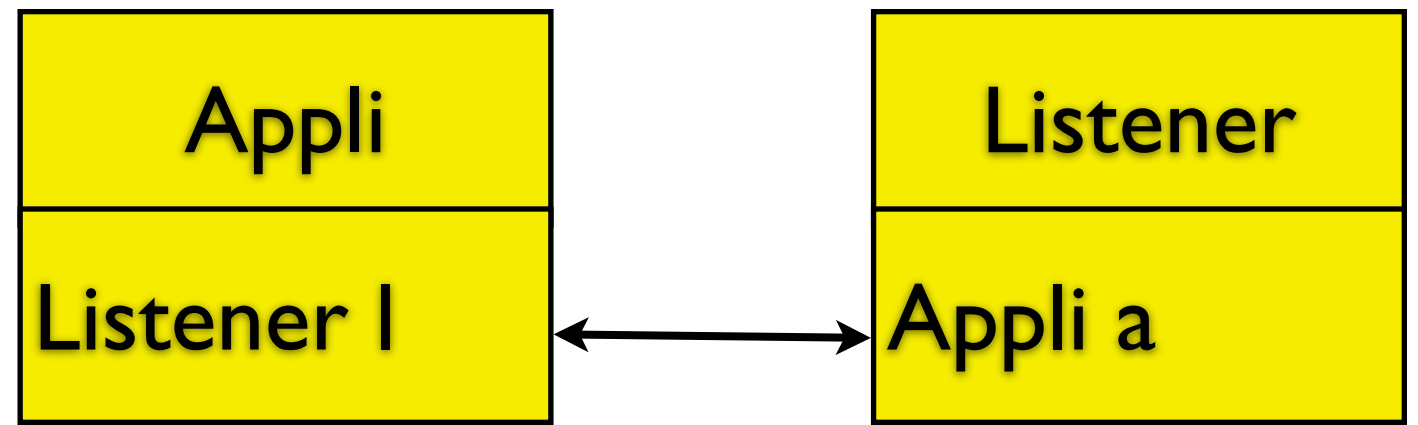
Récapitulation

- Dans notre architecture, c'est l'application qui met en place les actions (le *contrôle*)
- Quand on presse sur «+» :
AjouterActionListener est prévenu. la méthode actionPerformed appelle la méthode ajouter() de l'application.
- Celle-ci effectue l'opération d'ajout

Classes internes

- alternative à l'utilisation de EventHandler
- plus compliquées
- pas d'obligation d'avoir des méthodes publiques
- Très utilisées en java

Rappel du problème : Lien bidirectionnel entre objets



```
class Appli {  
    private Listener l;  
    public Appli() {  
        l = new Listener(this);  
    }  
}
```

```
class Listener {  
    private Appli a;  
    Listener(Appli a) {  
        this.a = a;  
    }  
}
```

```
void setSomme(double somme) {  
    this.somme = somme;  
    ui.getResultatField().setText(""+somme);  
}
```

```
public void ajouter() {  
    double val= Double.parseDouble(ui.getNombre1Field().getText());  
    setSomme(somme + val);  
}
```

ActionListener et classe interne

- Classe interne: si on veut cacher le listener
- une classe *interne* est une classe définie dans une autre classe
- un objet de la classe interne **a un lien implicite** avec l'objet de la classe englobante qui l'a créé
- en pratique: *il peut appeler les méthodes de la classe englobante et accéder à ses variables d'instance*

Classes internes(2)

```
public class CalculetteApplication2 {  
    class AjouterInterne implements ActionListener {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            ajouter(); // Appelle la méthode ajouter de CalculetteApplication2  
        }  
    }  
  
    private void activer() {  
        ui.getAjouterButton().addActionListener(new AjouterInterne());  
        ui.getEffacerButton().addActionListener(new EffacerInterne());  
    }  
  
    public void ajouter() {...}
```

Pas de passage explicite de this...

Classes internes anonymes

- pas beau, mais pratique.
- quand la méthode à ajouter est très courte

```
private void activer() {  
    ui.getAjouterButton().addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            ajouter();  
        }  
    });  
}
```

) *Implantation de la classe*

...

On crée un nouvel objet `ActionListener`, en fournissant l'implantation de la classe, sans la nommer. Usage unique. Illisible. C'est une classe *interne* (`ajouter()` = méthode de `CalculetteApplication`).

la classe EventHandler

- Permet d'éviter l'écriture de listeners pour les cas simples
- Crée des objets « à la volée » pour implanter une interface donnée
- Seule limitation: les méthodes appelées doivent être publiques
- Bonne documentation : <http://ydisanto.developpez.com/tutoriels/java/eventhandler/>

EventHandler (exemple)

```
import java.beans.EventHandler;
```

```
public class Appli {  
    JFormulaire maFrame;  
    MonModele modele;
```

```
    ...  
    private void activer() {  
        maFrame.getValiderButton().addActionListener(  
            EventHandler.create(ActionListener.class,  
                               this, "valider"));  
    }  
    ...
```

On va créer un
ActionListener...

...qui appelle «this.valider()»

```
    public void valider() {  
        ... code de la méthode ...  
    }
```

attention à l'import : il existe
plusieurs classes nommées
EventHandler !!!

EventHandler

- La méthode statique `EventHandler.create(...)` prend trois arguments:
 - une interface à implanter (un objet de type `Class`) :
ici **ActionListener.class**
 - un objet cible **o**
 - un nom de méthode de l'objet cible **f** (une `String`)
- *Toutes les procédures de l'interface appelleront **o.f()***
- Ne fonctionne pas si l'interface contient des fonctions.

EventHandler

- Pratique : plus court à écrire qu'une classe interne.
- Utilise l'introspection (la capacité de java de «regarder» ce qu'il y a dans les classes).
- **Pas de vérification à la compilation :** moins sûr.
- Plus lent (introspection), mais ça n'est pas bien grave pour l'usage qui en est fait.

Quelques composants de base... et leur sélection de méthodes

voir le tutoriel d'oracle, en particulier *Using Swing Components*

JComponent

- classe abstraite
- méthodes intéressantes :
 - void setFont(Font font) : police du texte
 - void setForeground(Color fg) : couleur de dessin
 - void setBackground(Color bg) : couleur de fond
 - void setEnabled(boolean e) : active/désactive le composant
 - void setSize(Dimension d) : fixe la taille (une dimension a deux composantes, largeur et hauteur)

JButton

- Un bouton, qui affiche un texte et/ou une icone
- Constructeurs
 - JButton(String label)
 - JButton(Icon icone)
- Méthodes
 - addActionListener(ActionListener l) : fixe le comportement du bouton

JTextField

- Champ texte simple
- Constructeur :
 - `JTextField(int taille)` : taille en caractères *affichés*
 - `JTextField(String val)` : texte contenu au départ.
- Méthodes
 - `void setText(String val)`
 - `String getText()` : les deux méthodes pour modifier/récupérer le texte affiché dans le champ
 - `void addActionListener(ActionListener l)` : fixe le comportement du textfield quand on appuie sur «entrée»
 - `void setEditable(boolean e)` : autorise/interdit l'édition par l'utilisateur

JTextArea

- Champ texte sur plusieurs lignes
- Constructeur :
 - JTextField(int lignes, int colonnes) : taille en caractères *affichés*
- Méthodes
 - void setText(String val)
 - String getText() : les deux méthodes pour modifier/récupérer le texte affiché dans le champ

JLabel

- Pour afficher un texte ou une image non modifiable par l'utilisateur
- Constructeurs
 - `JLabel(String s)` et `JLabel(Icon icon)`

JScrollPane

- permet d'ajouter des « ascenseurs » à d'autres composants
- utilisation : on place le composant désiré dans le JScrollPane:

```
JTextArea text= new JTextArea();
```

```
JScrollPane scroll= new JScrollPane(text);
```

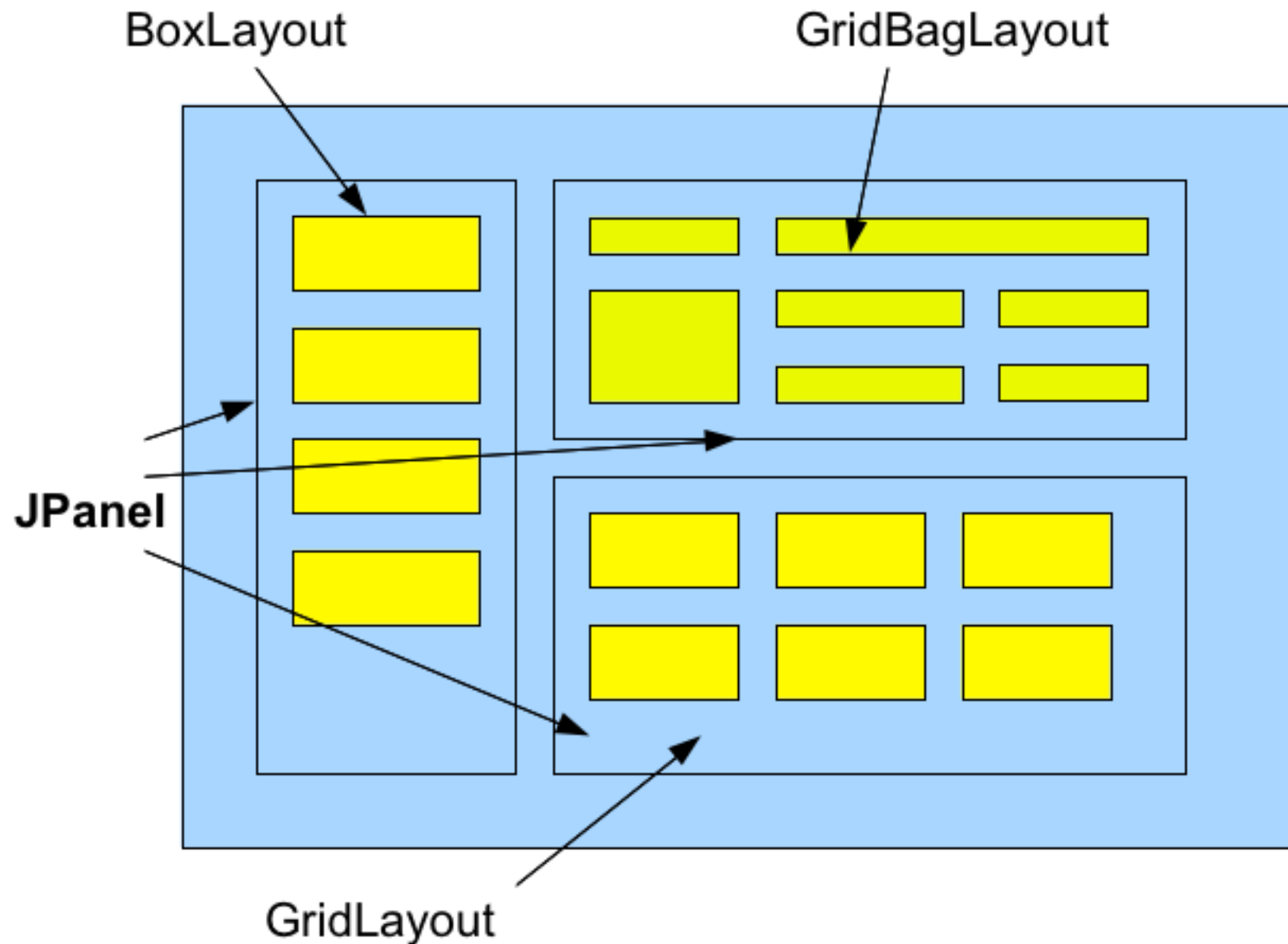
```
frame.add(scroll);
```

Layout

- Un peu compliqué pour démarrer
- En pratique :
 - l'éditeur graphique de netbeans est très bien
 - en enlevant la logique des composants graphiques, on peut programmer proprement avec netbeans
- autre solution : utiliser un layout amélioré comme **MigLayout**
- Ceci dit, il est utile d'en savoir un peu sur les layouts...
- *(remarque pratique : il n'y aura pas de question sur les détails des layout à l'examen).*

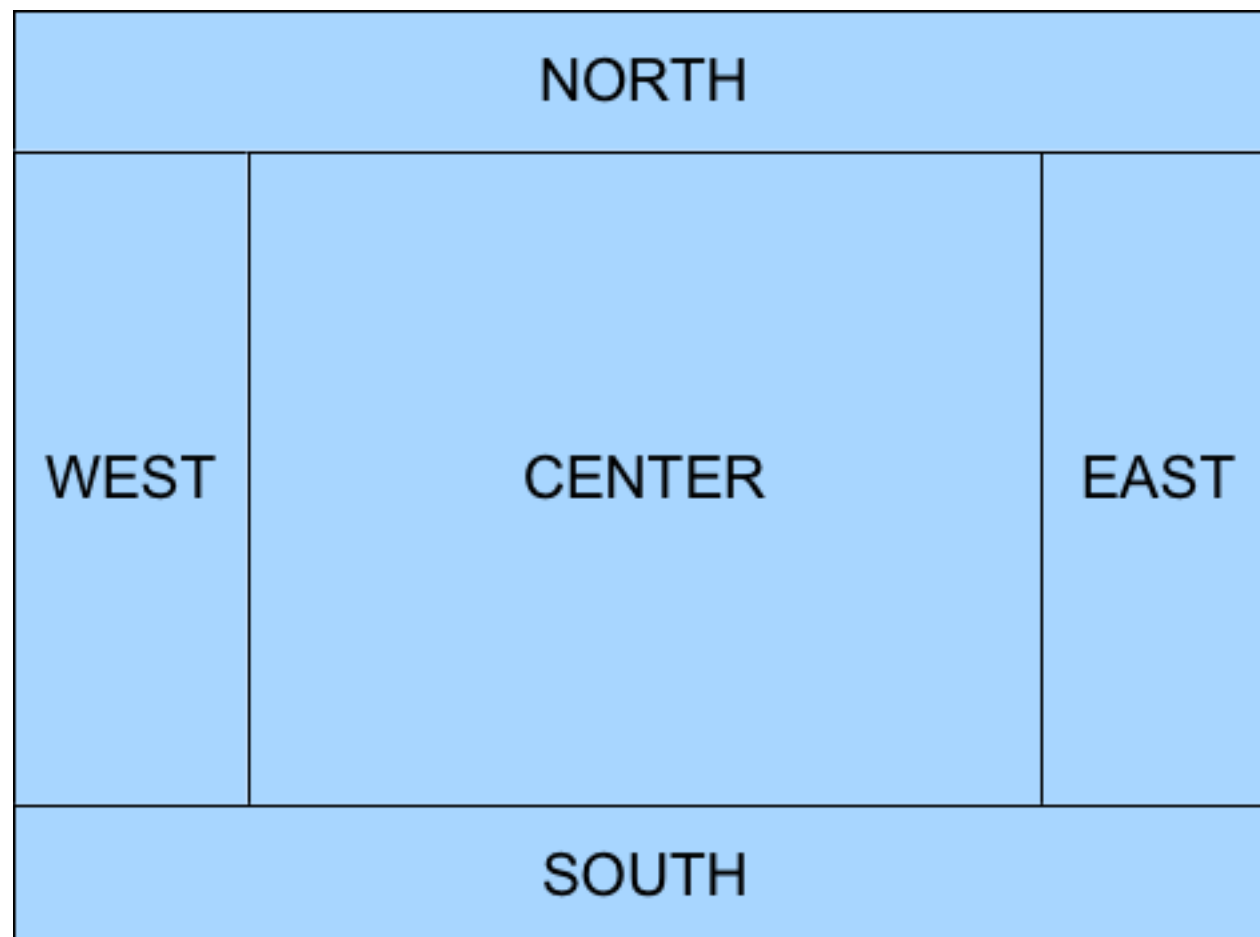
Utilisation de layouts et de JPanel

En imbriquant des JPanels les uns dans les autres, on peut obtenir des mises en pages complexes.



BorderLayout

- Fixe 5 zones dans le panel.



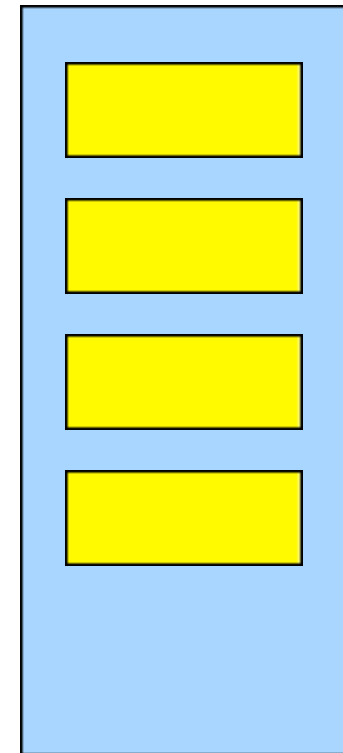
BorderLayout

```
public class MonPanel extends JPanel {  
    ....  
    private void mettreEnPage() {  
        setLayout(new BorderLayout());  
        // barre d'outils à gauche de l'écran  
        add(toolBar, BorderLayout.WEST);  
        // On ajoute le champ de l'éditeur,  
        // avec en plus des ascenseurs.  
        add(new JScrollPane(texteField),  
            BorderLayout.CENTER);  
    }  
}
```

BoxLayout

- Empile les composants selon l'axe horizontal ou vertical

```
JPanel panel= ...;  
BoxLayout b= new  
BoxLayout(panel,BoxLayout.PAGE_AXIS);  
panel.setLayout(b);  
panel.add(objet1);  
panel.add(objet2); ...
```



- Valeurs possibles pour l'argument du constructeur : `LINE_AXIS` et `PAGE_AXIS`

GridLayout

- Une grille d'éléments de la même taille
- Constructeur:
- `GridLayout(int rows, int cols)`
- si cols est nul, le nombre de colonnes n'est pas limité (idem pour rows).
- les objets sont ajoutés ligne par ligne.

GridBagLayout

- Le plus flexible des layouts de base
- grille à cases irrégulières
- On utilise un object auxiliaire, de type **GridBagConstraints**, pour spécifier comment ajouter chaque élément.
- on peut placer un élément dans une case précise, réunir plusieurs cases, etc.

GridbagLayout

```
JPanel p= new JPanel();  
p.setLayout(new GridBagLayout());  
JTextField nomField = new JTextField(30);  
JTextField prenomField = new JTextField(20);  
GridBagConstraints cc = new GridBagConstraints();  
p.add(new JLabel("Nom"), cc); // ajoute en pos. 0,0  
cc.gridx = 1;  
p.add(nomField, cc); // ajoute en pos. 1,0  
cc.gridx = 0; cc.gridy = 1;  
p.add(new JLabel("Prenom"), cc); // pos. 0,1  
cc.gridx = 1;  
p.add(prenomField, cc); // ajoute en pos 1,1
```

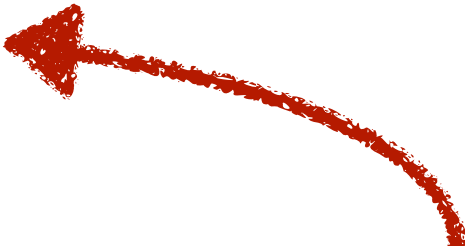
Attributs de GridBagConstraints

- gridx, gridy : position de l'objet à ajouter
- fill : comment l'objet occupe-t-il l'espace disponible ?
- weightx, weighty: répartition de l'espace disponible entre les objets
- gridwidth: nombre de colonnes occupées par l'objet

MigLayout

- bibliothèque à récupérer sur le web. Très puissant pour les formulaires, etc...

```
JPanel p= new JPanel();  
p.setLayout(new MigLayout());  
p.add(new JLabel("nom"));  
p.add(nomTextField, "wrap");  
p.add(new JLabel("prenom"));  
p.add(prenomTextField);
```



Informations de mise en page
passées sous forme de Strings.
Ici, on demande de passer à la
ligne.

Un layout particulier: CardLayout

- Layout qui permet de changer le contenu affiché d'un panneau. En fait, un CardLayout gère plusieurs objets (typiquement des panneaux), et n'en montre qu'un seul.

```
JPanel parent= new JPanel();  
CardLayout layout= new CardLayout();  
parent.setLayout(layout);  
layout.addLayoutComponent(panel1, "p1");  
layout.addLayoutComponent(panel2, "p2");  
layout.show(parent, "p1");
```

Bibliographie

- tutoriel java officiel : <http://docs.oracle.com/javase/tutorial/uiswing/>
- en particulier «Using Swing Components»
- un livre avancé sur la création d'une application graphique complexe :
 - E. Puybaret, *Les cahiers du programmeur swing*, Eyrolles, 2006