JAVA

Applications interactives

-

Programmation d'interfaces graphiques

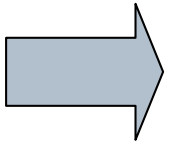
Stéphane HUOT

Dpt. Informatique

iUT ORSAY

Interfaces graphiques en java

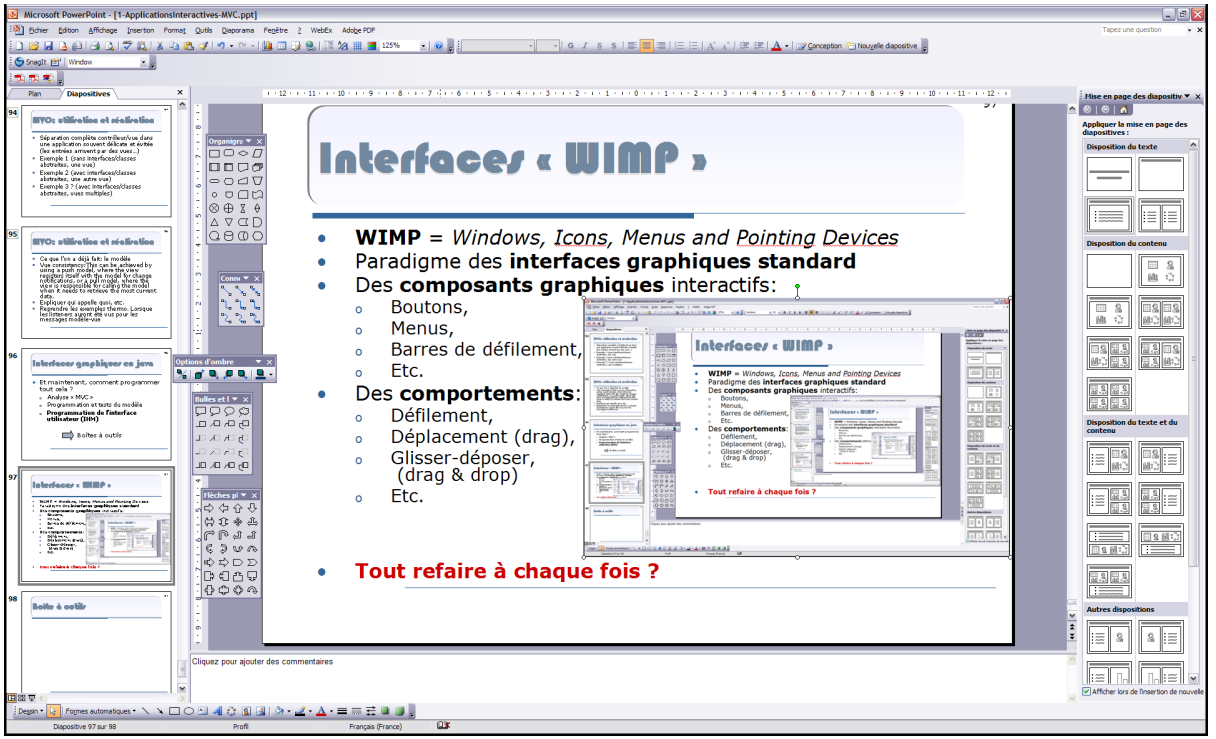
- Et maintenant, comment programmer tout cela ?
  - Analyse « MVC »
  - Programmation et tests du modèle
  - **Programmation de l'interface utilisateur (IHM)**



Boîtes à outils

Interfaces « WIMP »

- **WIMP** = *Windows, Icons, Menus and Pointing Devices*
- Paradigme des **interfaces graphiques standard**
- Des **composants graphiques** interactifs:
  - Boutons,
  - Menus,
  - Barres de défilement,
  - Etc.
- Des **comportements**:
  - Défilement,
  - Déplacement (drag),
  - Glisser-déposer, (drag & drop)
  - Etc.



- **Tout refaire à chaque fois ?**

« Boîtes à outils » d'IHM

- **GUI Toolkit**: Bibliothèques logicielles fournissant des composants et des mécanismes prédéfinis et adaptés à la programmation d'interfaces graphiques
- Composants atomiques:
  - La **'Frame'** (ou *canvas*): fenêtre assurant la liaison avec le système de fenêtrage hôte (MS Windows, Xwindows, …),
  - Le **'Widget'** (ou *control*): composant d'interface graphique (bouton, zone de texte, …),
  - Le **'Layout'**: définit le placement des contrôles,
  - Les **'Listeners'** (ou *reflexes*): mécanismes de gestion des événements et de déclenchement des actions des widgets

Boîtes à outils en Java

- 2 boîtes à outils dans l'API Java:
  - **AWT** (Abstract Window Toolkit):
    - La bibliothèque historique (1995)
    - Bibliothèque graphique de base de l'API Java
  - **Swing**:
    - La 'nouvelle' bibliothèque (1998)
    - Améliore les graphismes (*Java2D*) et les composants (plus complète)
    - MVC
- Autres BàO: SWT/JFace

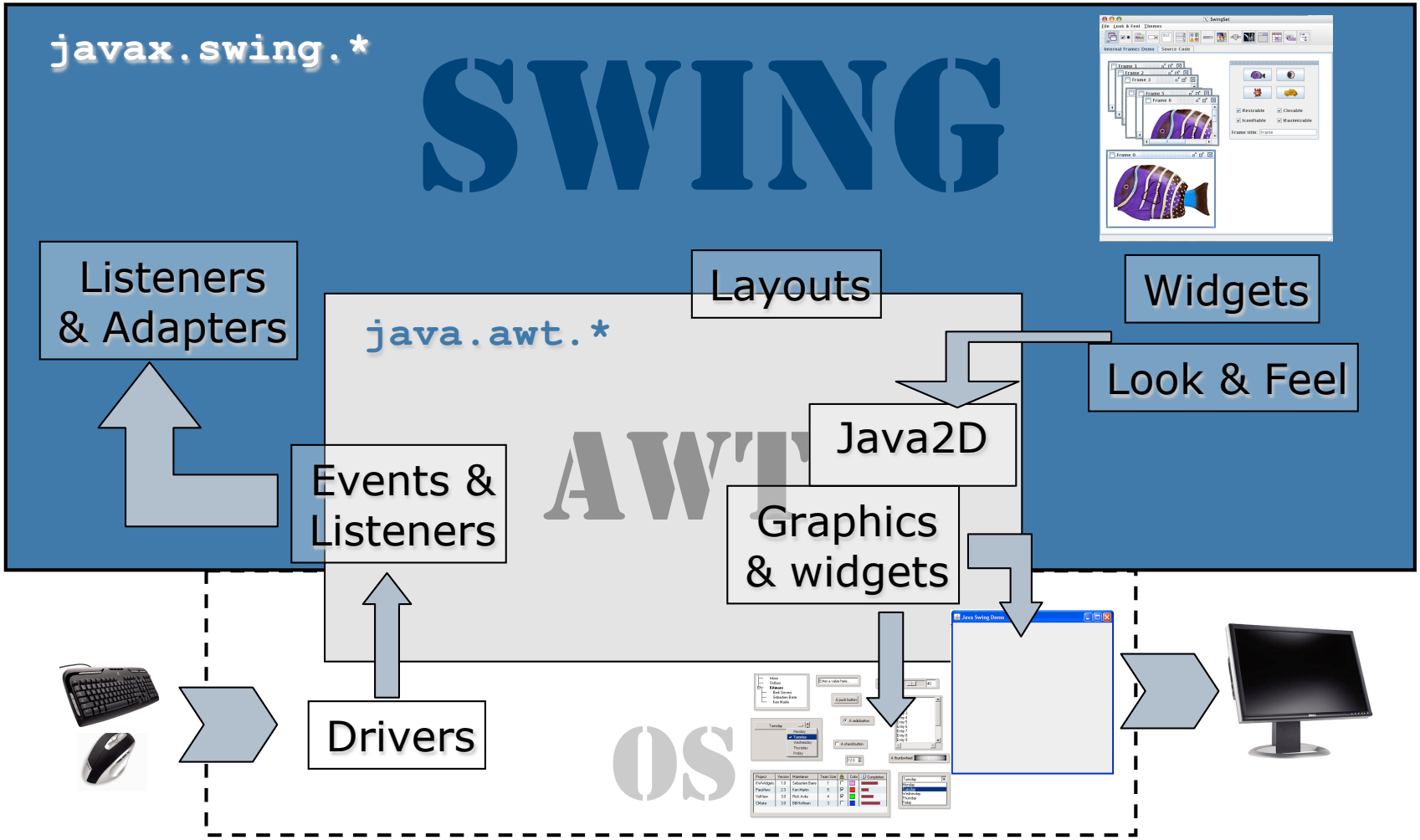
Java AWT

- Fonctionnalités graphiques de base
- Base du système d'événements et d'accès aux entrées de l'API Java
- 'Pont' avec les composants graphiques de la plateforme hôte (***heavyweight = composants lourds***)

Java Swing

- A permis d'améliorer le système graphique de Java (*Java2D* dans AWT)
- N'est plus liée aux composants graphique de la plateforme hôte (***lightweight = composants légers***)
- Implémentée et à utiliser en suivant **MVC**
- Introduit les 'look & feel' (aspects et comportements des widgets indépendants de leurs modèles)
- Fournit plus de composants, avec plus de possibilités

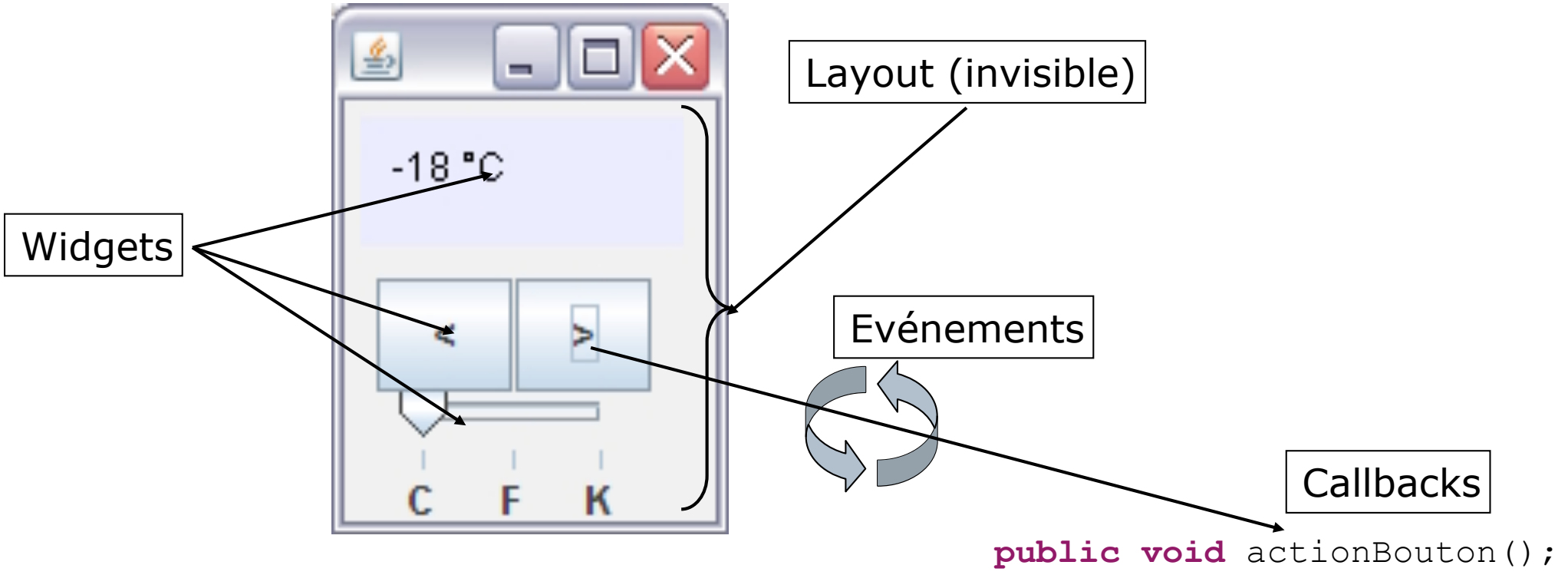
AWT, Swing, etc.



Ce que nous allons voir

- Beaucoup de Swing (`package(s) javax.swing.*`)
  - Les 'Widgets' de Swing
  - Les 'Adapters' et les 'Listeners' (gestion des événements)
  - Un peu de Java2D (graphique)
- Un peu de AWT (`package(s) java.awt.*`)
  - Les 'Layouts' (disposition des widgets à l'écran)
  - Les 'Listeners' (gestion des événements)

Lexique en 'image'



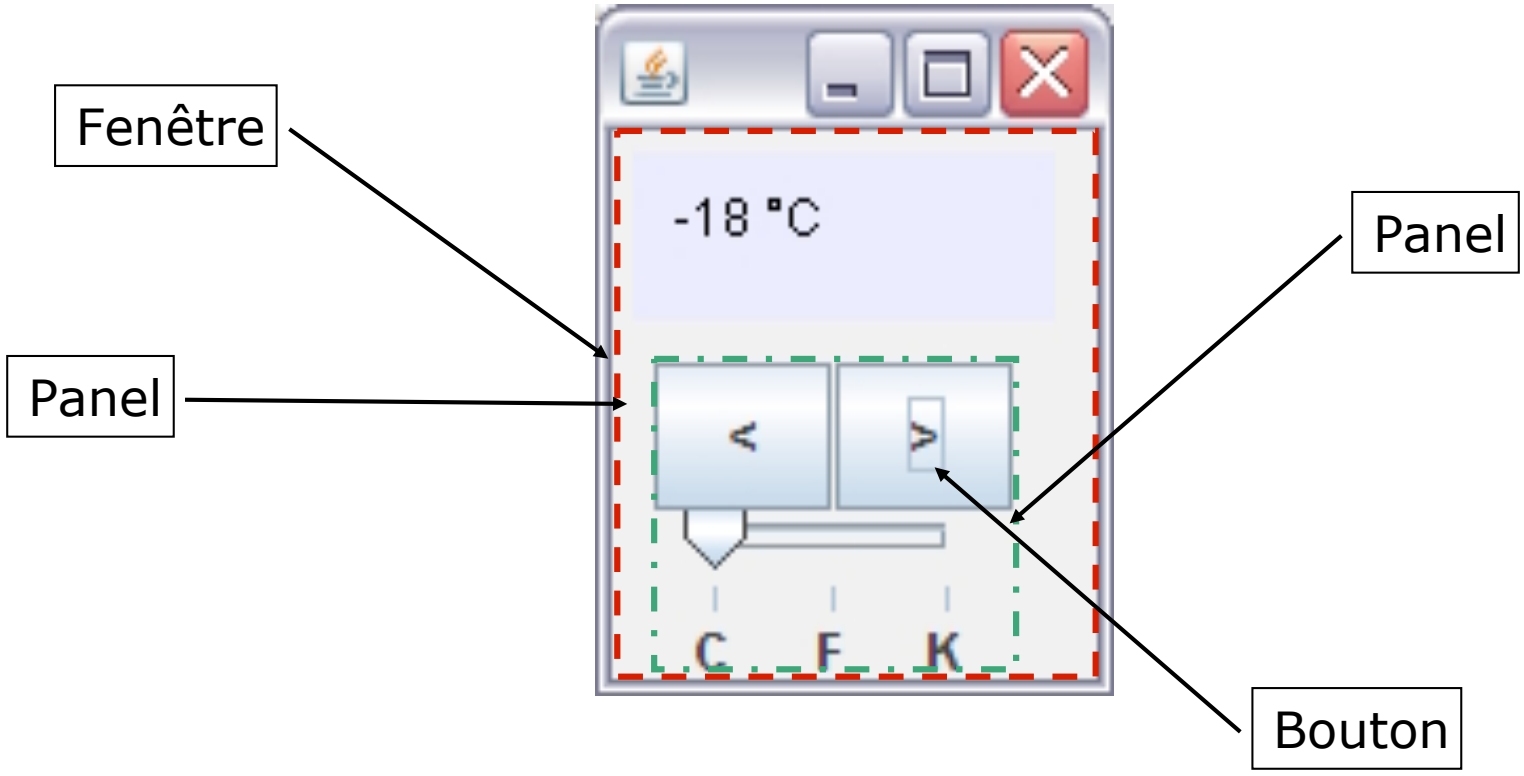
Swing: composants de base

- Les widgets de base:
  - Encapsulation et MVC
    - On ne s'intéresse qu'à ce que font les composants, pas comment c'est implanté
    - **Modèle**: le comportement abstrait du widget
    - **Vue** et **Contrôleur**: *Look & Feel* + *Listeners*
  - Nommés **'J...'**: `JButton`, `JPanel`, ...
  - Tout est `JComponent` : classe abstraite de base (issue de `Component` et `Container` de AWT pour compatibilité)
- Voir la Javadoc de l'API java…
- `Container`, `JFrame` et `JComponent`
- Exemples détaillés : `JPanel` et `JButton`

Notion de 'Container'

- Container = widget générique qui peut contenir d'autres widgets
- La classe `Container` dans AWT:
  - Structuration de l'interface graphique
  - Ordre et affichage des 'fils'
  - Gestion du transfert des événements (clicks souris, frappes clavier, etc.)
- Tous les widgets Swing sont des containers (`JComponent` hérite de `Container` qui hérite de `Component`)

Container: exemple



Méthodes de base de Container

- Fournit les méthodes de base pour la manipulation d'un ensemble de composants.
- Différentes méthodes d'ajout de composants:
  `container.add(child);`
- Différentes méthodes de retrait de composants:
  `container.remove(child);`
  `container.removeAll();`
- Obtenir les fils:
  `Component[] container.getComponents();`
- Voir la Javadoc de `Container`...

Container: règles

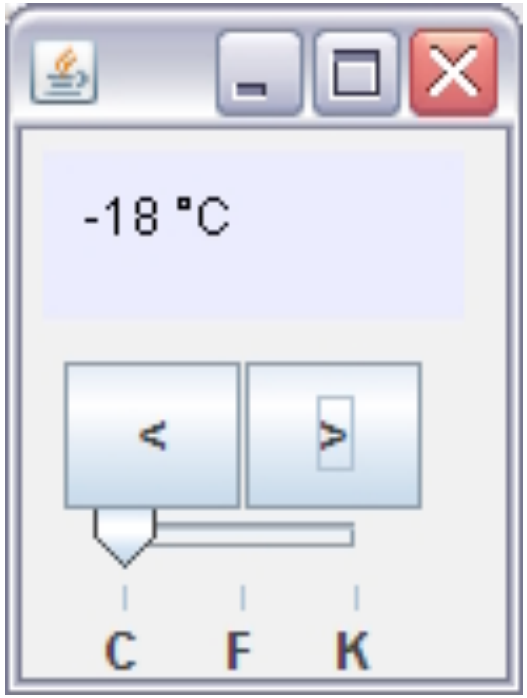
- Pour apparaître à l'écran, les composants doivent appartenir à une hiérarchie de containers
- Un composant ne peut appartenir qu'à un seul container
- La racine d'une hiérarchie de container est un container de haut-niveau:

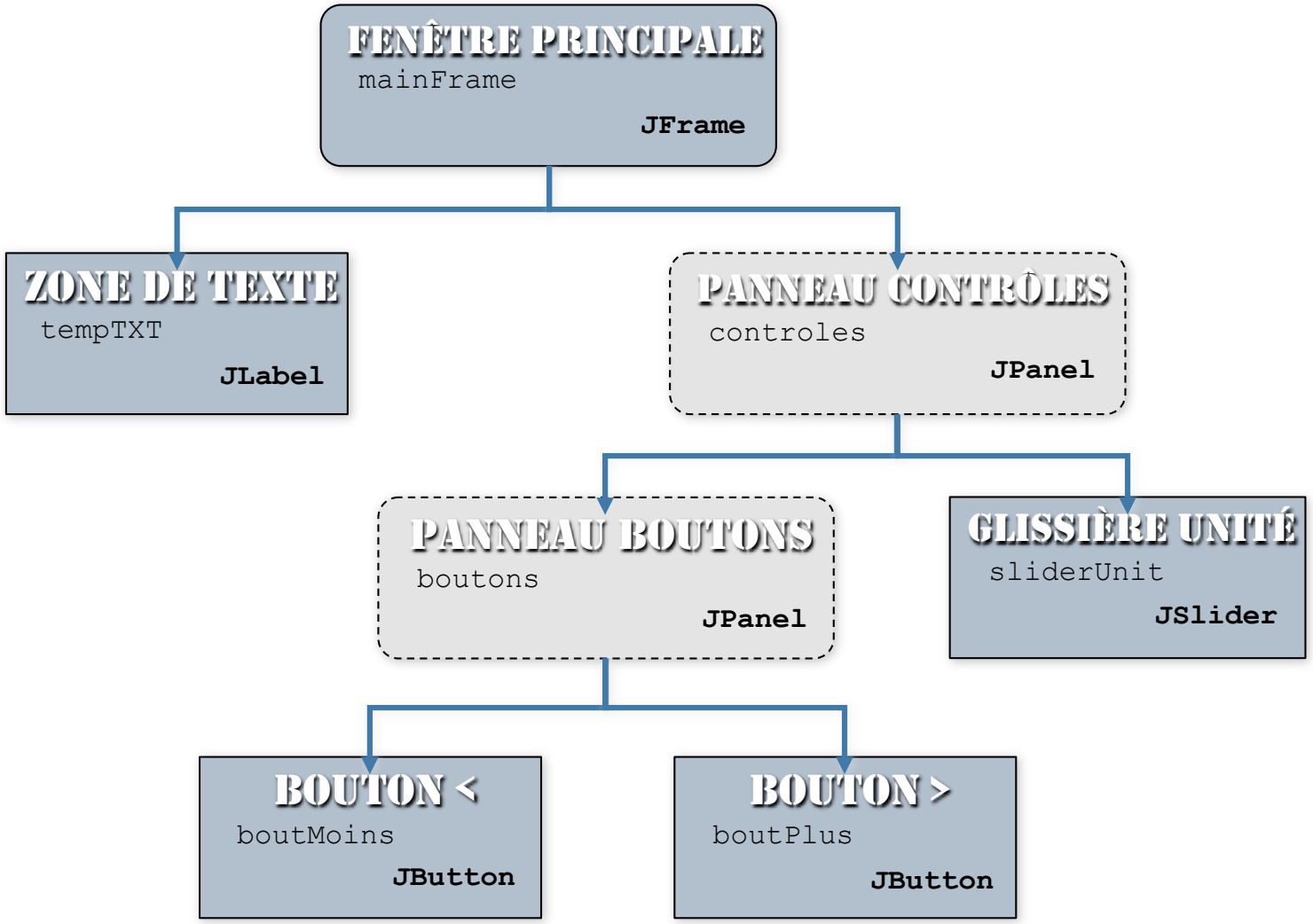
  Top-level container

Arbre de widgets

- Représentation de la structure des widgets de l'interface sous forme d'un arbre
  - Structure les objets de l'interfaces
  - Facilite l'analyse et la compréhension
  - Facilite l'implantation (reflète bien le code à produire)

Arbre de widgets: exemple





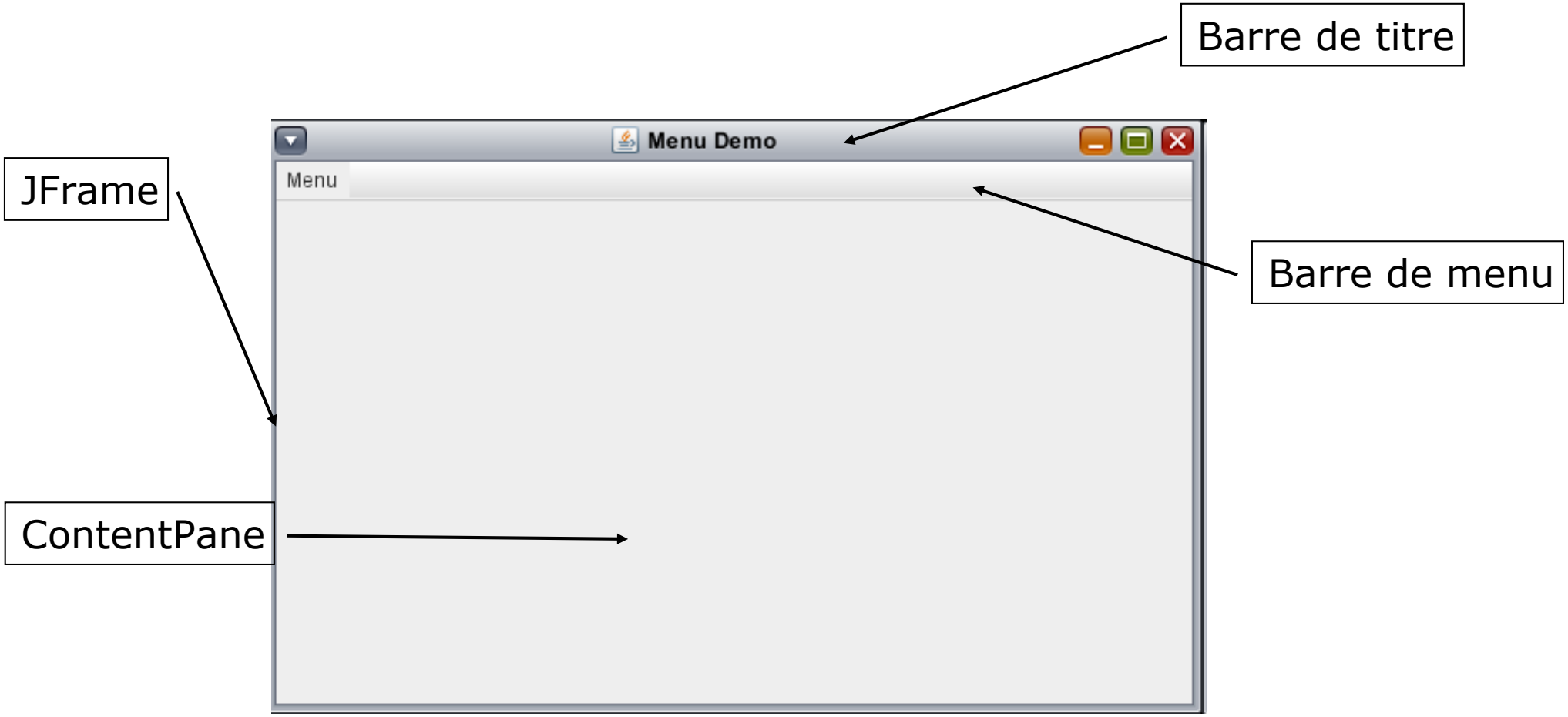
Fenêtre: JFrame

- Fenêtres des applications:
  - Crées à partir du système de fenêtrage natif (Windows, Xwindows, …)
  - En Swing: `JFrame` (hérite de `Frame` de AWT)
- Container de plus haut-niveau de la boîte à outils
- 'Racine' de l'interface graphique de l'application (créée dans la méthode `main` en général)

JFrame: structure

- Le **contour** et la **barre de titre**: système
- Le ***'ContentPane'***: partie qui va contenir les composants de l'interface (*Top-Level Container*)
- Possibilité d'ajouter une **barre de menu** (`JMenuBar`)

JFrame: structure



JFrame, bases

- Création d'une JFrame:
```
JFrame frame = new JFrame();
```
- Ajout d'un composant:
```
frame.add(child); //child est un Component
```
- Retrait d'un composant:
```
frame.remove(child); //child est un Component
```
- Affichage de la fenêtre:
```
frame.setVisible(true);
```

Retour sur le thermomètre v1

- L'application

```
package thermometre;
import java.awt.GridLayout;
import javax.swing.JFrame;
import thermometre.controleurs.ControleurThermometre;
import thermometre.modele.ModeleThermometre;
import thermometre.vues.VueThermometre;

public class AppliThermometreSimple {

        public static void main(String[] args) {
                //Creation d'une fenêtre pour l'application
                JFrame frame = new JFrame();

                //Creation d'un modèle de thermomètre
                ModeleThermometre modele = new ModeleThermometre(243.15);

                //Creation de la vue et du contrôleur
                VueThermometre vue = new VueThermometre (modele);
                final ControleurThermometre pt1 = new ControleurThermometre(modele, vue);

                //Ajout des panneau à la fenêtre
                frame.setLayout(new GridLayout(1, 2));
                frame.add(pt1);

                //Affichage de la fenêtre
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.pack();
                frame.setVisible(true);
        }
}
```

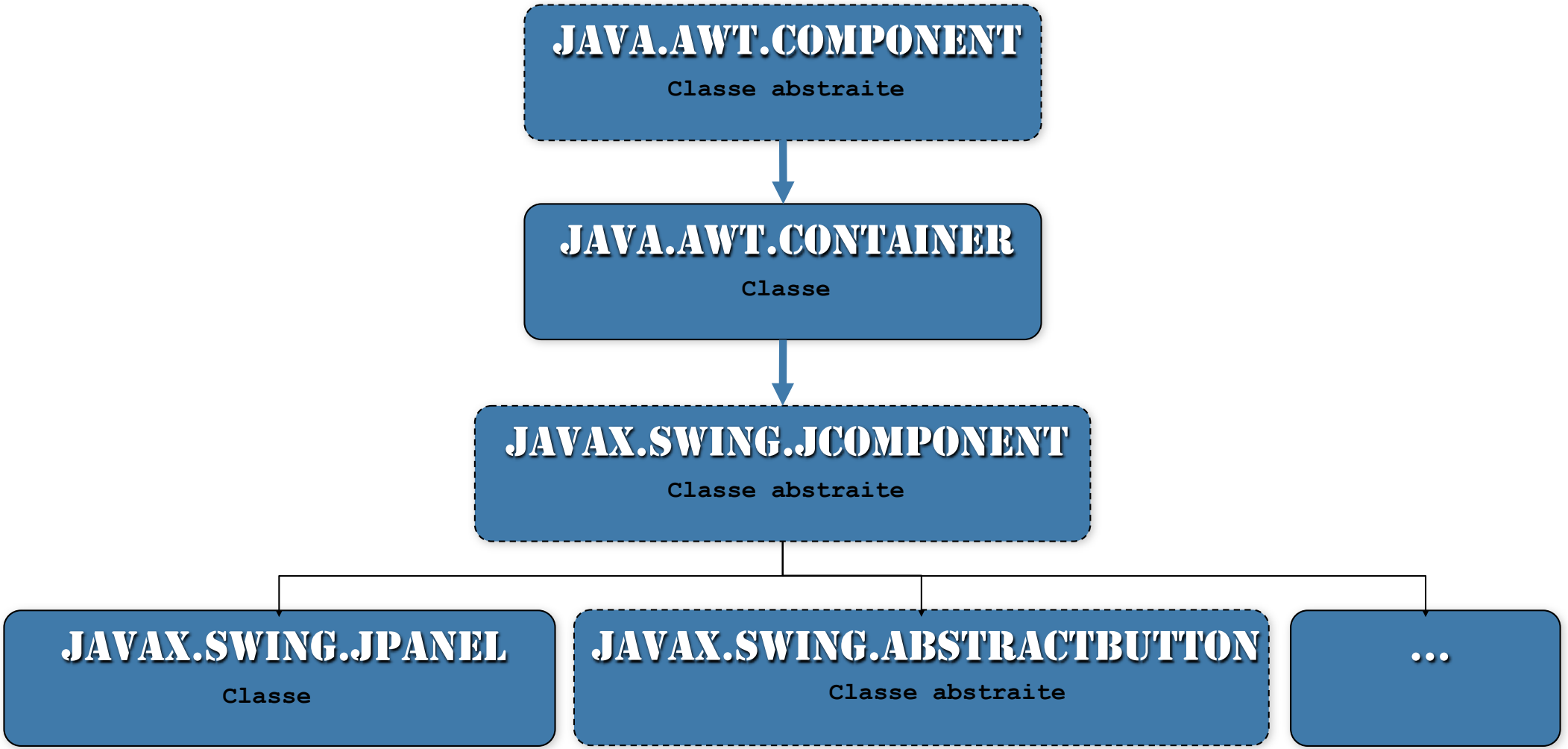
JFrame, un peu plus

- Changement du layout :
```
frame.setLayout(monLayout);
```
- Changement du titre (dans la barre):
```
frame.setTitle("Mon Appli");
```
- Comportement à le fermeture:
```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```
- 'Compactage':
```
frame.pack();
```
- Ajout d'une barre de menu:
```
frame.setJMenuBar(maMenuBar);
```
- Voir la Javadoc de `JFrame`…

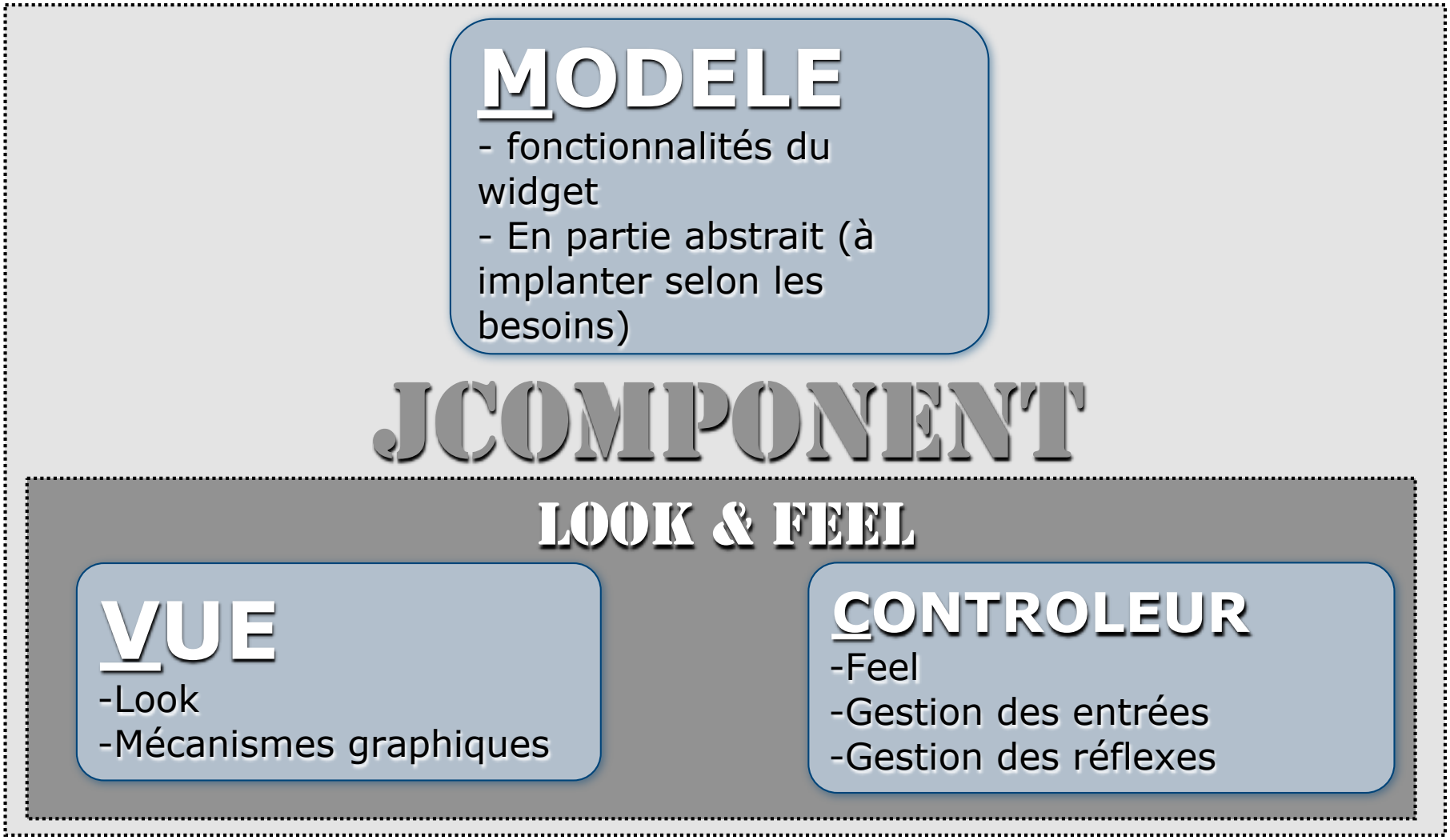
JComponent, bases

- Classe abstraite qui définit et fournit des méthodes de base pour tous les widgets
- Mécanismes de ***'Look & Feel'*** (apparence et comportement)
- **Entrées** souris et clavier
- ***Tooltips*** (messages contextuels)
- Mécanisme de **dessin** et d'**affichage** (*painting borders*, etc.)
- Gestion de la **position/orientation**, la **taille**, les **couleurs**, la **police de texte**, etc.

JComponent, ancêtres et descendance



JComponent: widgets SWING et MVC



JComponent, méthodes de base (1)

- Méthodes définies dans `JComponent` ou héritées de `java.awt.Component`
- Position et taille (peuvent dépendre du layout du container parent) :
  `Point getLocation()` ou `int getX()` et `int getY()`,
  `setLocation(int x, int y)` etc.
  `int getWidth()`, `int getHeight()` (largeur et hauteur)
  `Rectangle getBounds()` (rectangle englobant)
  `Dimension getSize()` et `setSize(Dimension d)`,
  `setPreferredSize(Dimension d)`, `setMaximumSize(Dimension d)`, `setMinimumSize(Dimension d)` (taille)

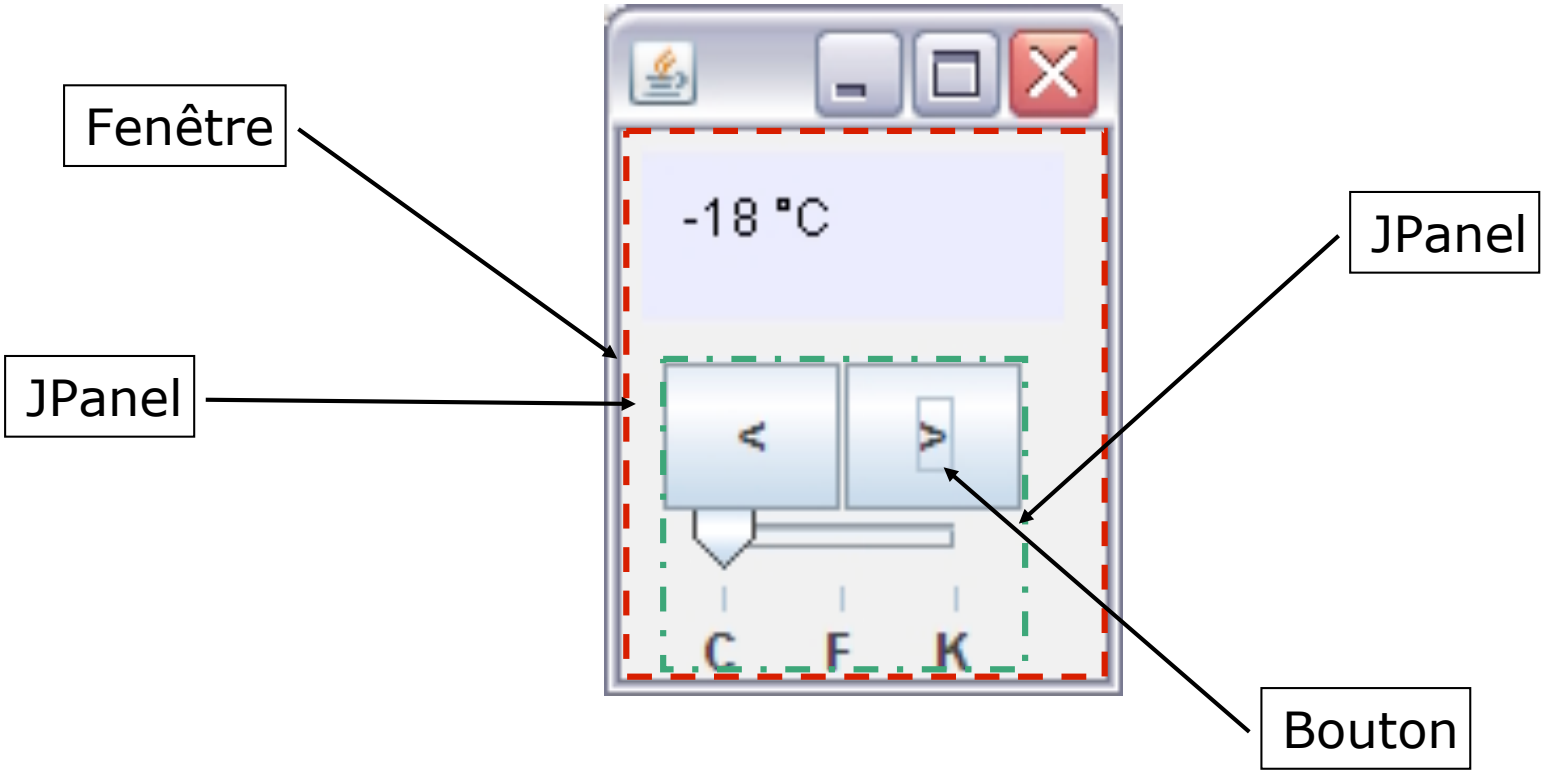
JComponent, méthodes de base (2)

- Couleur de fond:
  `setBackground(Color c)` et `Color getBackground()`
- Couleur de premier plan (texte):
  `setForeground(Color c)` et `Color getForeground()`
- Police du texte:
  `setFont(Font f)` et `Font getFont()`
- Méthodes d'affichage:
  `paint(Graphics2D g)` (appelée par Swing)
  `paintComponent(Graphics2D g)`, `paintBorder(Graphics2D g)` et `paintChildren(Graphics2D g)` (appelées par `paint`, celles que l'on surcharge en général)
- Voir la Javadoc de `JComponent`…

JPanel

- Container concret de base
- Permet de 'regrouper' des composants pour:
  - Structurer l'interface graphique
    - Tâches de l'utilisateur
    - Placements
    - Couleurs
    - …
  - Structurer le code
    - Sections de codes / Classes
    - Comportement (contrôleur)
    - …

JPanel: exemple



JPanel

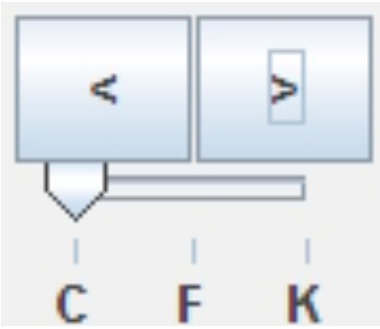
- Par défaut:
  - Ne dessine que son fond (*background*) et ses fils
  - N'a pas de bordure graphique
  - Est opaque
  - Adapte sa taille selon ses fils et son *'Layout'*
- Possibilités:
  - Changer le *'Layout'*
  - Changer les couleurs
  - Rendre transparent
  - Ajouter une bordure
  - …

JPanel, bases

- Création d'un JPanel:
```
JPanel panel = new JPanel();
```
- Ajout d'un composant:
```
panel.add(child); //child est un
Component
```
- Retrait d'un composant:
```
panel.remove(child); //child est un
Component
```
- Ajout à un autre container:
```
container.add(panel);
```

JPanel. Exemple

```
JPanel buttons = new JPanel();
//Ajout de boutons au panel…
buttons.add(mButton);
buttons.add(pButton);
buttons.add(mSlider);
//…
//Ajout du panel à un autre
   container…
```



JPanel, un peu plus

- Changement du layout :
```
panel.setLayout(monLayout);
```
- Ajout d'une bordure:
```
panel.setBorder(new LineBorder
(Color.BLACK));
```
- Changement de la couleur de fond:
```
panel.setBackground(Color.RED);
```
- Rendre le fond transparent:
```
panel.setOpaque(false);
```
- Etc.
- Voir la Javadoc de `JPanel`...

JButton

- Un widget… bouton!
- Etend `AbstractButton`
- Affiche un bouton avec:
  - Du texte
  - Une image
  - Du texte et une image
- Mécanisme de raccourcis clavier (*mnemonic*)
- Comportement programmé à l'aide
  - D'*`Action`*
  - De *`Listeners`*

JButton, bases

- Création d'un JButton:

```
//un bouton sans texte ni image
JButton button = new JButton();
//un bouton avec du texte
JButton button = new JButton(String text);
//un bouton avec une image
JButton button = new JButton(Icon icon);
```

- Activation/désactivation:

```
button.setEnabled(boolean b);
```

- Comportement:
  - Configuration de l'action:

```
button.setAction(Action a);
```

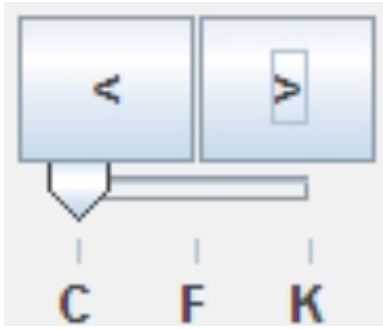
  - Ajout d'un ActionListener:

```
button.addActionListener(ActionListener l);
```

```
//L'action à réaliser est programmée dans une
//classe Action ou ActionListener
```

JButton. Exemple

```
JButton pButton = new JButton(">");
JButton mButton = new JButton("<");

//Ajout d'un 'contrôleur' sur le bouton "UP"
pButton.addActionListener(new
    ControleurThermometreButtons(modele, vue,
    BUTTONS.UP));
//Ajout d'un 'contrôleur' sur le bouton "DOWN"
mButton.addActionListener(new
    ControleurThermometreButtons(modele, vue,
    BUTTONS.DOWN));

//Ajout des boutons au panel
buttons.add(mButton);
buttons.add(pButton);
```



JButton, un peu plus

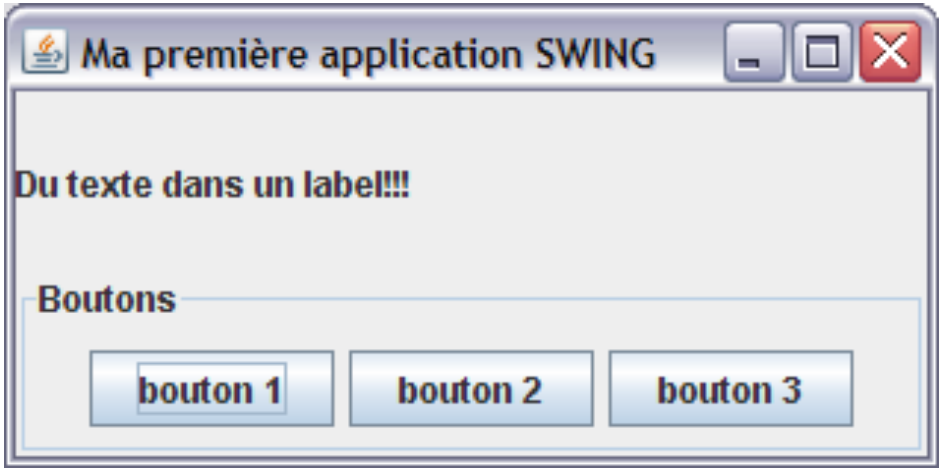
- Changement du texte :
```
button.setText(« Texte");
```
- 'Rollover':
```
button.setRolloverEnabled(true);
```
- Images:
```
button.setIcon(Icon i);
button.setPressedIcon(Icon i);
button.setRolloverIcon(Icon i);
button.setRolloverSelectedIcon(Icon i);
button.setDisabledIcon(Icon i);
```
- Etc.
- Voir la Javadoc de `JButton`...

Autres widgets...

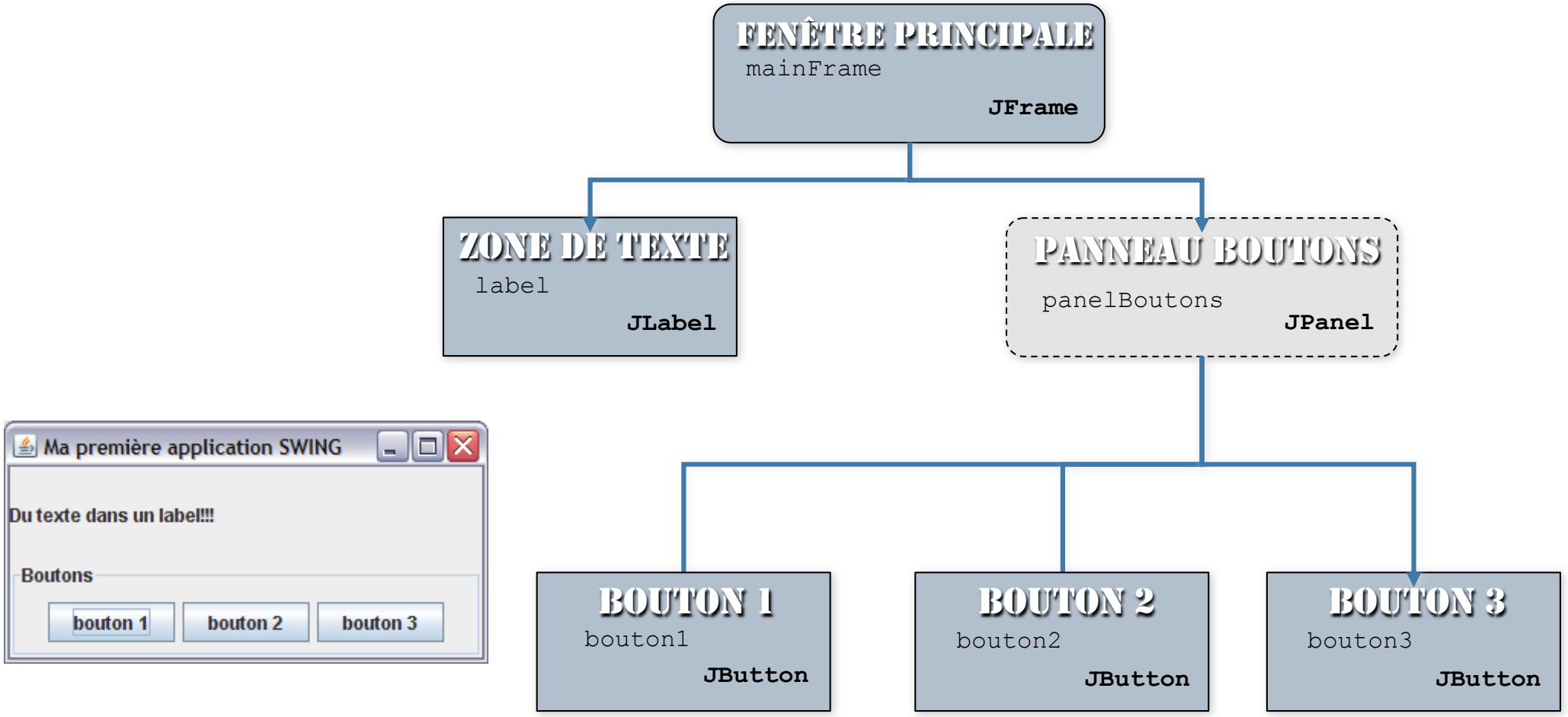
- Texte:
  - JLabel
  - JTextField
  - JTextArea
  - …
- Listes et arbres
  - JList
  - JTree
  - JComboBox
  - JMenu/JPopupMenu
- Choix
  - CheckBox
  - JRadioButton
- Dialogues
  - JDialog
  - JFileChooser
  - JColorChooser
  - …
- …

Une première interface simple

- Une application simple qui affiche dans sa fenêtre:
  - Un label contenant du texte
  - 3 boutons contenus dans un panel avec un titre



Arbre de widgets



Code 1

```
package gui;
import java.awt.FlowLayout;
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.TitledBorder;

public class ApplicationSimple {
	public static void main(String[] args) {
		//Création de la fenêtre de l'application
			JFrame mainFrame = new JFrame("Ma première application SWING");
			//Changement du layout de la fenêtre
			mainFrame.setLayout(new GridLayout(2, 1));

			//Création du label contenant le texte
			JLabel label = new JLabel("Du texte dans un label!!!");

			//Création du panel de boutons
			JPanel panelBoutons = new JPanel();
			//Changement du bord du panel
			panelBoutons.setBorder(new TitledBorder("Boutons"));

			//Suite au prochain transparent…
```

Code 2

```
        //Création des 3 boutons
        JButton bouton1 = new JButton("bouton 1");
        JButton bouton2 = new JButton("bouton 2");
        JButton bouton3 = new JButton("bouton 3");

        //Changement du layout du panel de boutons et ajout des boutons
        panelBoutons.setLayout(new FlowLayout());
        panelBoutons.add(bouton1);
        panelBoutons.add(bouton2);
        panelBoutons.add(bouton3);

        //Ajout du label à la fenêtre
        mainFrame.add(label);
        //Ajout du panel de boutons à la fenêtre
        mainFrame.add(panelBoutons);
        //'Compactage' de la fenêtre
        mainFrame.pack();
        //On quitte l'application quand la fenêtre est fermée
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Affichage de la fenêtre
        mainFrame.setVisible(true);
    }
}
```

Démonstration…

- Affichage de l'interface:
  - Placement des widgets
  - Redimensionnement

⇨ **LayoutManagers**

- Comportement des widgets
  - Presser un bouton
  - Réactions de l'application ?

⇨ **Actions et Listeners**

Ce qu'il faut retenir

- Lexique (*Widgets*, *Containers*, etc.)

- Notion de **Container**

- **Arbre de widgets**

- Méthodes de base communes aux widgets

- Séparation du **modèle** des widgets et de leurs **actions**

Layout

- Structurer une interface graphique:
  - Regrouper les contrôles de manière cohérente par tâches/fonctionnalités
  - S'assurer du maintien de la cohérence
    - Plateforme et résolution d'affichage
    - Redimensionnement par l'utilisateur
- Arrangement « semi »-automatique:
  - Les **LayoutManager**

LayoutManager

- Mécanisme de Swing pour:
  - Placer les widgets dans un container
  - Gérer les redimensionnements
- Concerne les Containers
  - Méthodes `add` spécialisées (paramètres de layout): `add (Component comp, Object constraints)`
- A une influence sur les widgets (propriétés `Size` et `Location`)
- Interface de AWT, implantée dans plusieurs classes de AWT ou Swing

LayoutManager

- Peut définir plusieurs propriétés:
  - Position des widgets dans le container
  - Taille des widgets
  - Espace entre les widgets
  - Comportement de ces propriétés en cas de redimensionnement ou de l'orientation du container
  - Etc.
- Les widgets peuvent avoir des propriétés qui vont influencer le LayoutManager:
  - `preferredSize`, `minimumSize` **et** `maximumSize`
  - `AlignmentX` **et** `AlignmentY`
  - `Insets` **(espace laissé entre le container et ses bords)**
  - Etc.

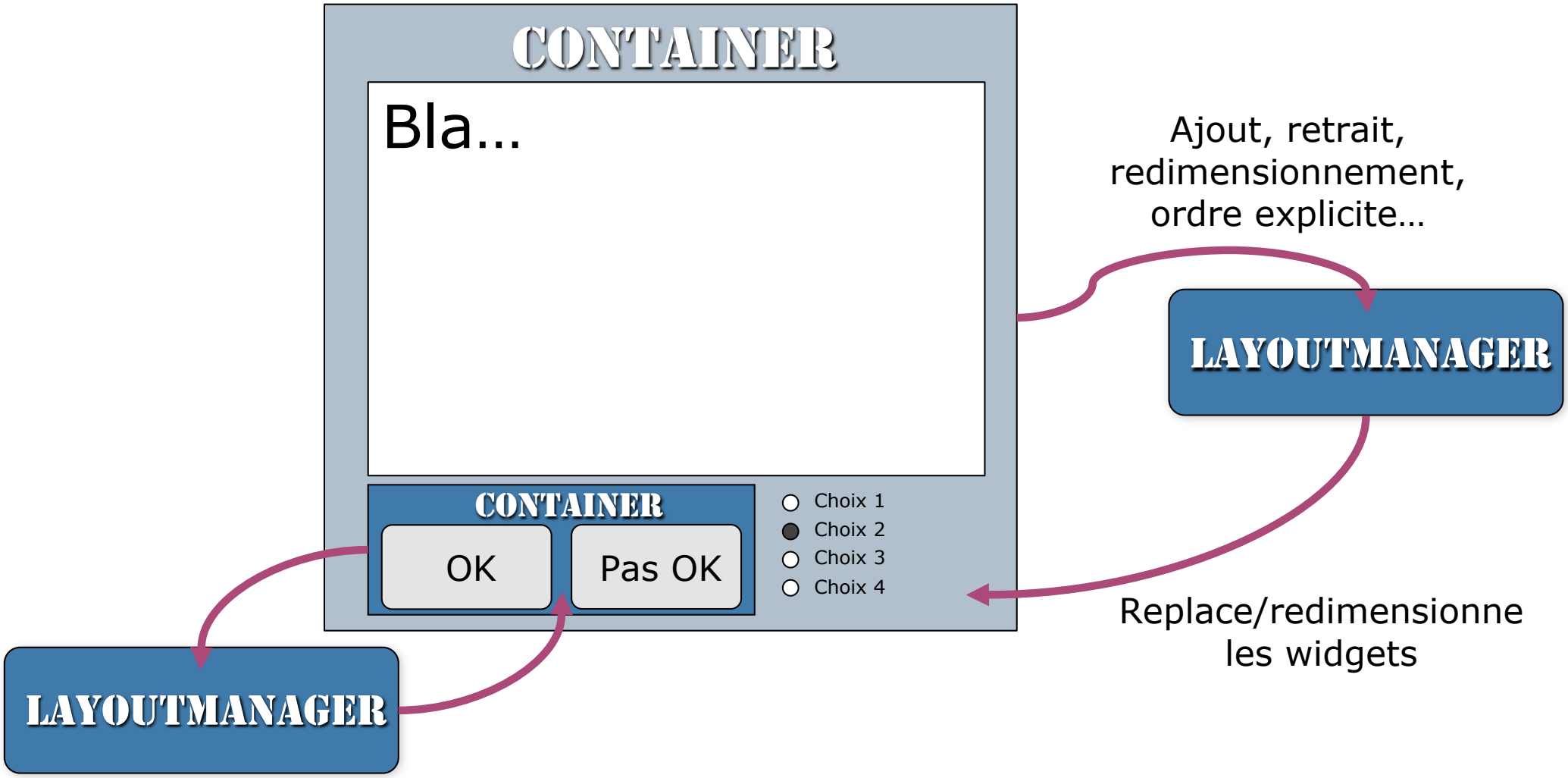
Un problème complexe

- Problème complexe:
  - Automatiser des comportements graphiques non triviaux
  - Prévoir des cas non génériques
  - Faciliter le travail du programmeur, mais lui laisser le contrôle
  - Encore des activités de recherche sur le placement des widgets!

Un problème complexe

- Résultat:
  - LayoutManager = 'Usines à gaz'
  - Intérêt des JPanel pour structurer l'interface:
    - Regroupements que les LayoutManagers ne permettent pas
    - LayoutManagers différents selon les groupes de contrôles
  - 'Détourner' et 'Jouer' avec les LayoutManagers pour arriver à ses fins
  - Essayer, expérimenter… pratiquer

LayoutManager: fonctionnement



LayoutManagers concrets

- **`BorderLayout (AWT)`**:
  Divise le container en 5 zones (Centre, Nord, Sud, Est et Ouest)

- **`BoxLayout (Swing)`**:
  Alignement des composants (axe X, axe Y, Line, Page)

- **`FlowLayout (AWT)`**:
  Positionnement en flux selon la place disponible (Centré, Gauche ou Droite)

- **`GridLayout (AWT)`**:
  Positionnement des composants dans une grille (avec tailles des cases égales)

- **`GridBagLayout(AWT)`**:
  Positionnement dans une grille où les composants peuvent prendre plusieurs cases (utilisation de contraintes)

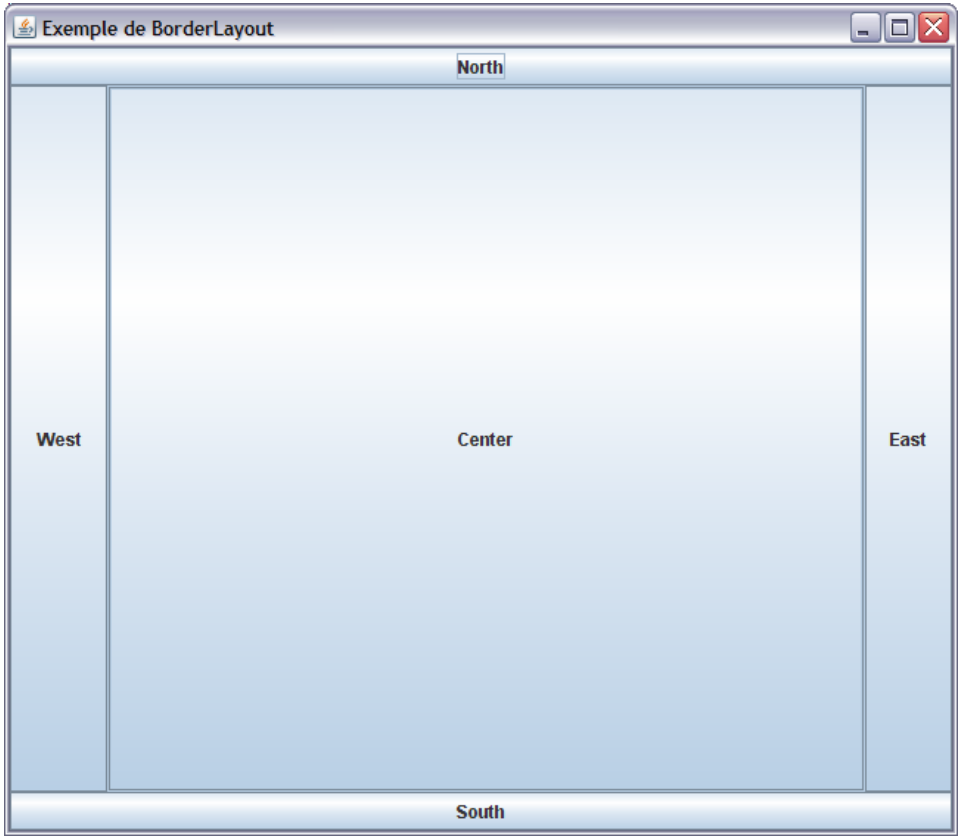
- **`Null!!!`**:
  Pas de LayoutManager (positionnement des composants 'à la main')

Etc… voir Javadoc…

BorderLayout

- Division du container en 5 régions:
  'CENTER', 'NORTH', 'SOUTH', 'EAST' et *'WEST'*
- Un composant par région
- Dimensionnement des widgets par rapport à leurs `'preferredSizes'` et redimensionnement proportionnel:
  - *NORTH* et *SOUTH* étirés horizontalement
  - *EAST* et *WEST* étirés verticalement
  - *CENTER* étiré rempli le reste de l'espace
- Utilisation de la méthode `container.add(child, Object constraints)` pour spécifier dans quelle région placer un composant (`container.add(child, BorderLayout.CENTER)`)
- Utilisation typique: fenêtres principales

BorderLayout. Exemple



Démonstration de redimensionnement

BorderLayout. Code

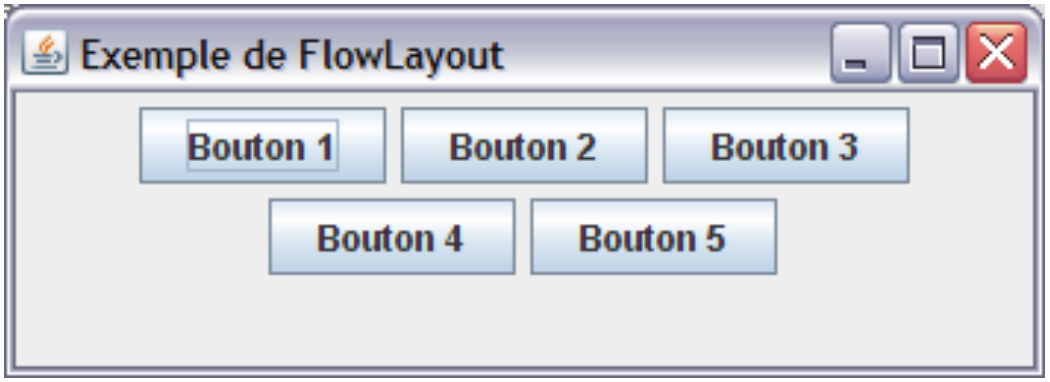
```
package gui;

import java.awt.BorderLayout;
import javax.swing.*;

public class ApplicationBorderLayout {

        public static void main(String[] args) {
                    //Création de la fenêtre de l'application
                    JFrame mainFrame = new JFrame("Exemple de BorderLayout");
                    //Changement du layout de la fenêtre
                    mainFrame.setLayout(new BorderLayout());

                    //Ajout des boutons
                    mainFrame.add(new JButton("North"), BorderLayout.NORTH);
                    mainFrame.add(new JButton("South"), BorderLayout.SOUTH);
                    mainFrame.add(new JButton("East"), BorderLayout.EAST);
                    mainFrame.add(new JButton("West"), BorderLayout.WEST);
                    mainFrame.add(new JButton("Center"), BorderLayout.CENTER);

                    //'Compactage' de la fenêtre
                    mainFrame.pack();
                    //On quitte l'application quand la fenêtre est fermée
                    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    //Affichage de la fenêtre
                    mainFrame.setVisible(true);
        }
}
```

FlowLayout

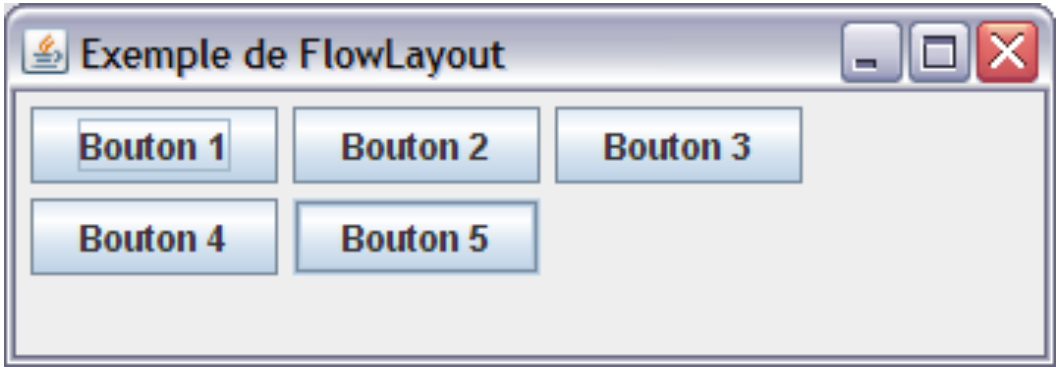
- Layout par défaut des JPanel
- Arrange les widgets horizontalement selon un flot directionnel
- Garde la taille définie des widgets et retourne à la ligne s'il n'y a pas assez de place
- L'alignement est déterminé par la propriété `Alignement` (`setAlignement` et `getAlignement`):
  - *CENTER*: lignes centrées (par défaut)
  - *LEFT*: lignes justifiées à gauche
  - *RIGHT*: lignes justifiées à droite
  - *LEADING* et *TRAILING*: justification en tête ou en queue selon l'orientation du container
- Utilisation typique: boutons dans des panels

FlowLayout. Exemple

CENTER



LEFT



Démonstration de redimensionnement

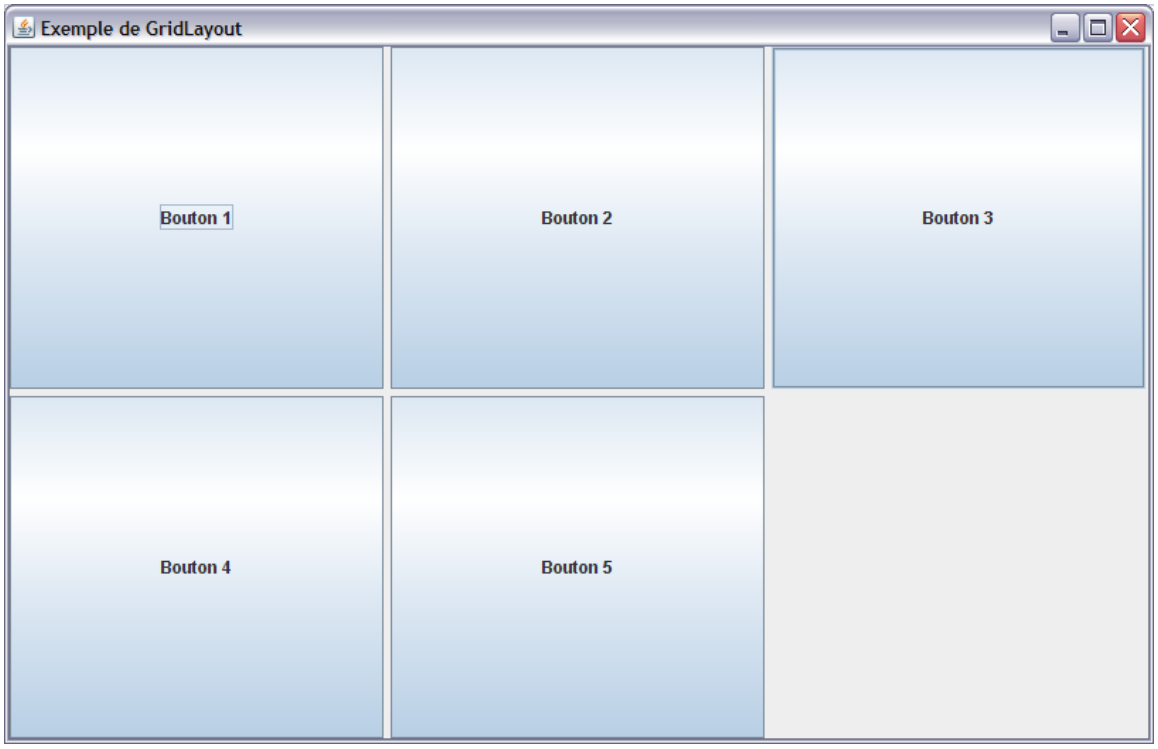
FlowLayout. Code

```
package gui;

import java.awt.FlowLayout;
import javax.swing.*;

public class ApplicationFlowLayout {

	public static void main(String[] args) {
			//Création de la fenêtre de l'application
			JFrame mainFrame = new JFrame("Exemple de FlowLayout");

			//Création d'un panel
			JPanel panelBoutons = new JPanel();
			//Changement du layout du panel
			panelBoutons.setLayout(new FlowLayout(FlowLayout.LEFT));
			//Ajout des boutons
			panelBoutons.add(new JButton("Bouton 1"));
			panelBoutons.add(new JButton("Bouton 2"));
			panelBoutons.add(new JButton("Bouton 3"));
			panelBoutons.add(new JButton("Bouton 4"));
			panelBoutons.add(new JButton("Bouton 5"));

			//Ajout du panel à la fenêtre
			mainFrame.add(panelBoutons);
			//'Compactage' de la fenêtre
			mainFrame.pack();
			//On quitte l'application quand la fenêtre est fermée
			mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
			//Affichage de la fenêtre
			mainFrame.setVisible(true);
	}
}
```

GridLayout

- Crée une grille dans le container, avec des cases de taille égale
- Un widget par case
- Redimensionne les widgets
- L'ordre d'ajout dans la grille dépend de la propriété `ComponentOrientation` du container
- Le nombre de lignes (rows) et de colonnes (columns) est spécifié par:
  - Le constructeur
    - `GridLayout()`: 1 colonne par composant et 1 ligne
    - `GridLayout(int rows, int cols)`:`rows` lignes et `cols` colonnes
    - `GridLayout(int rows, int cols, int hgap, int vgap)`:`rows` lignes et `cols` colonnes et écarts horizontaux et verticaux
  - Les méthodes `setRows` et `setColumns`
- Si il y a plus de widgets que de cases: le nombre de colonnes est ignoré (remplissage par ligne)
- Utilisation typique: boutons, checkboxes dans des panels

GridLayout. Exemple

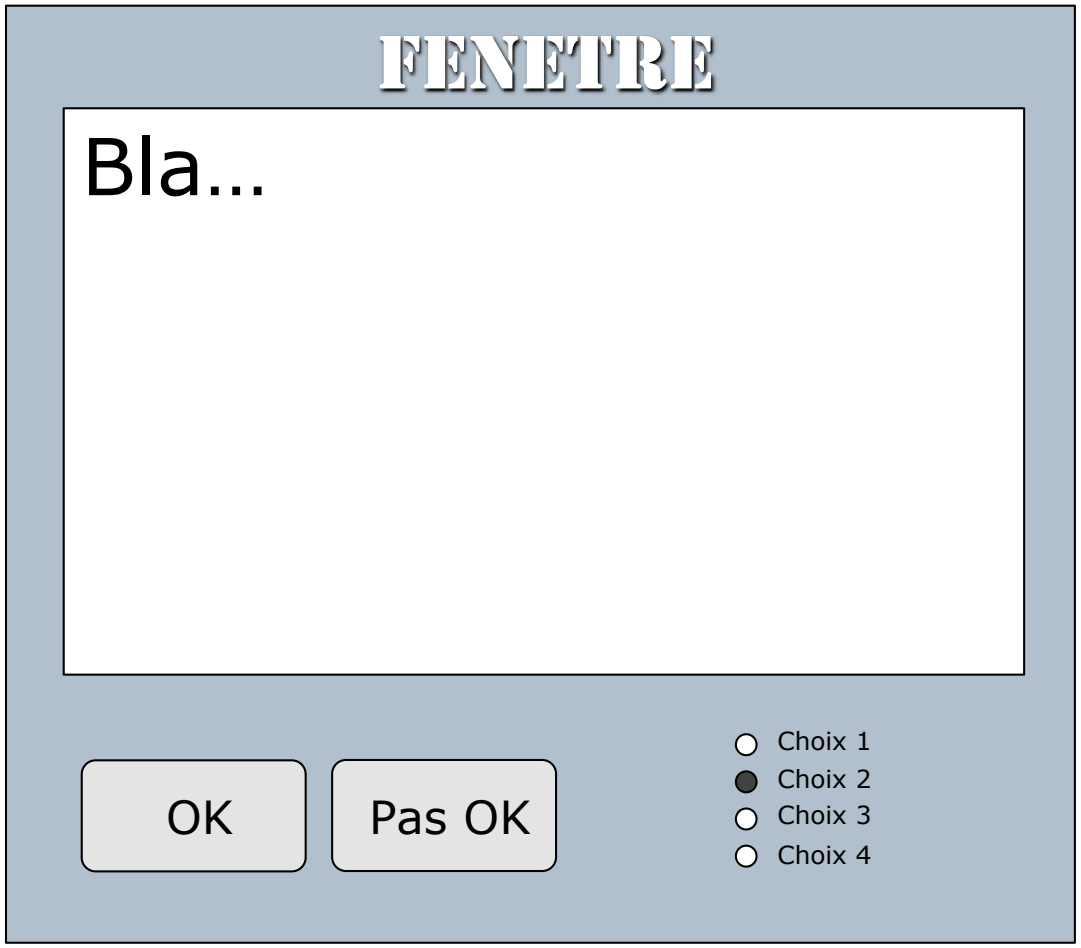


Démonstration de redimensionnement

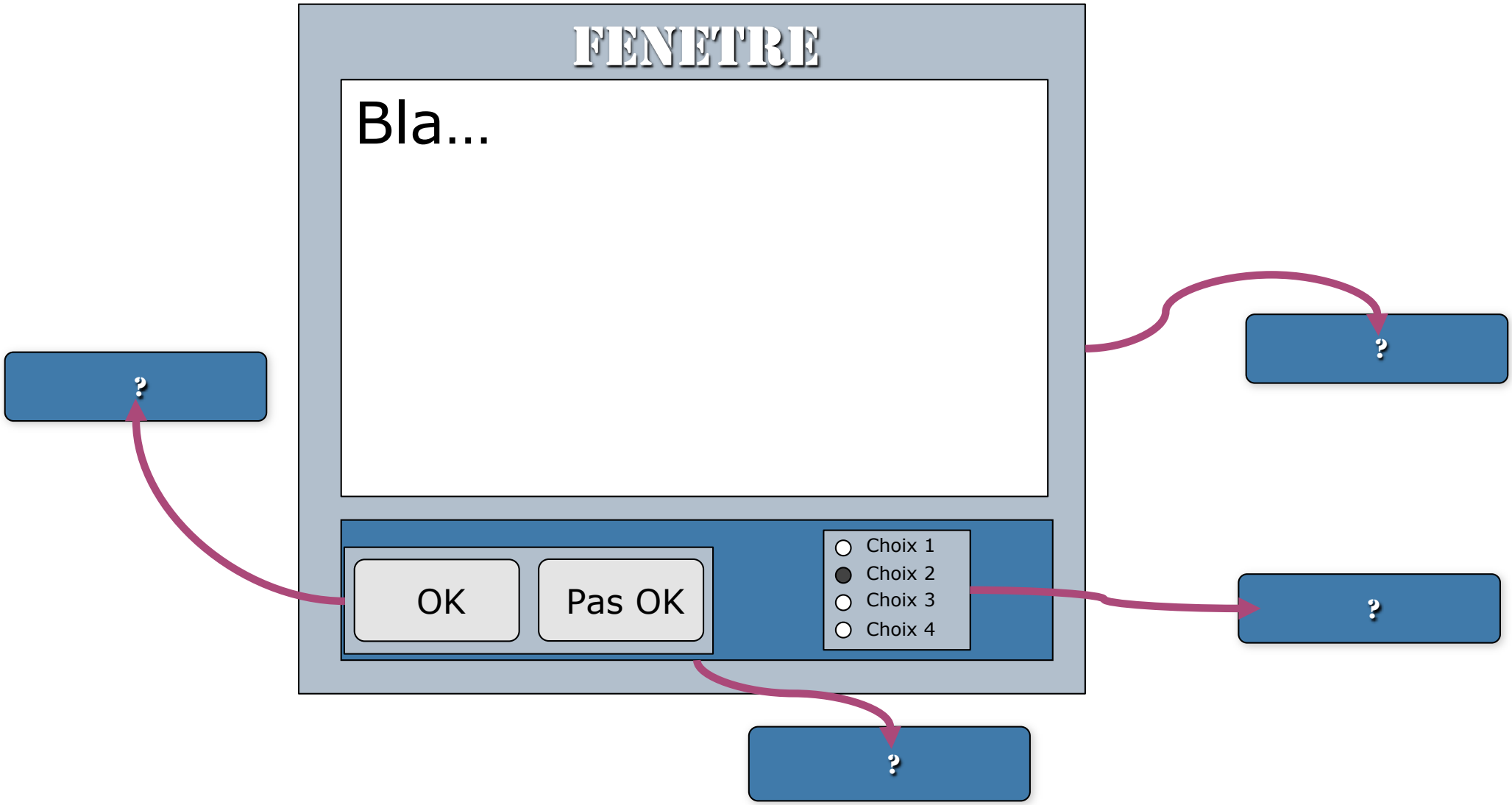
GridLayout. Code

```
package gui;

import java.awt.GridLayout;
import javax.swing.*;

public class ApplicationGridLayout {

	public static void main(String[] args) {
			//Création de la fenêtre de l'application
			JFrame mainFrame = new JFrame("Exemple de GridLayout");

			//Création d'un panel
			JPanel panelBoutons = new JPanel();
			//Changement du layout du panel
			panelBoutons.setLayout(new GridLayout(2, 2, 5, 5));
			//Ajout des boutons
			panelBoutons.add(new JButton("Bouton 1"));
			panelBoutons.add(new JButton("Bouton 2"));
			panelBoutons.add(new JButton("Bouton 3"));
			panelBoutons.add(new JButton("Bouton 4"));
			panelBoutons.add(new JButton("Bouton 5"));

			//Ajout du panel à la fenêtre
			mainFrame.add(panelBoutons);
			//'Compactage' de la fenêtre
			mainFrame.pack();
			//On quitte l'application quand la fenêtre est fermée
			mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
			//Affichage de la fenêtre
			mainFrame.setVisible(true);
	}
}
```

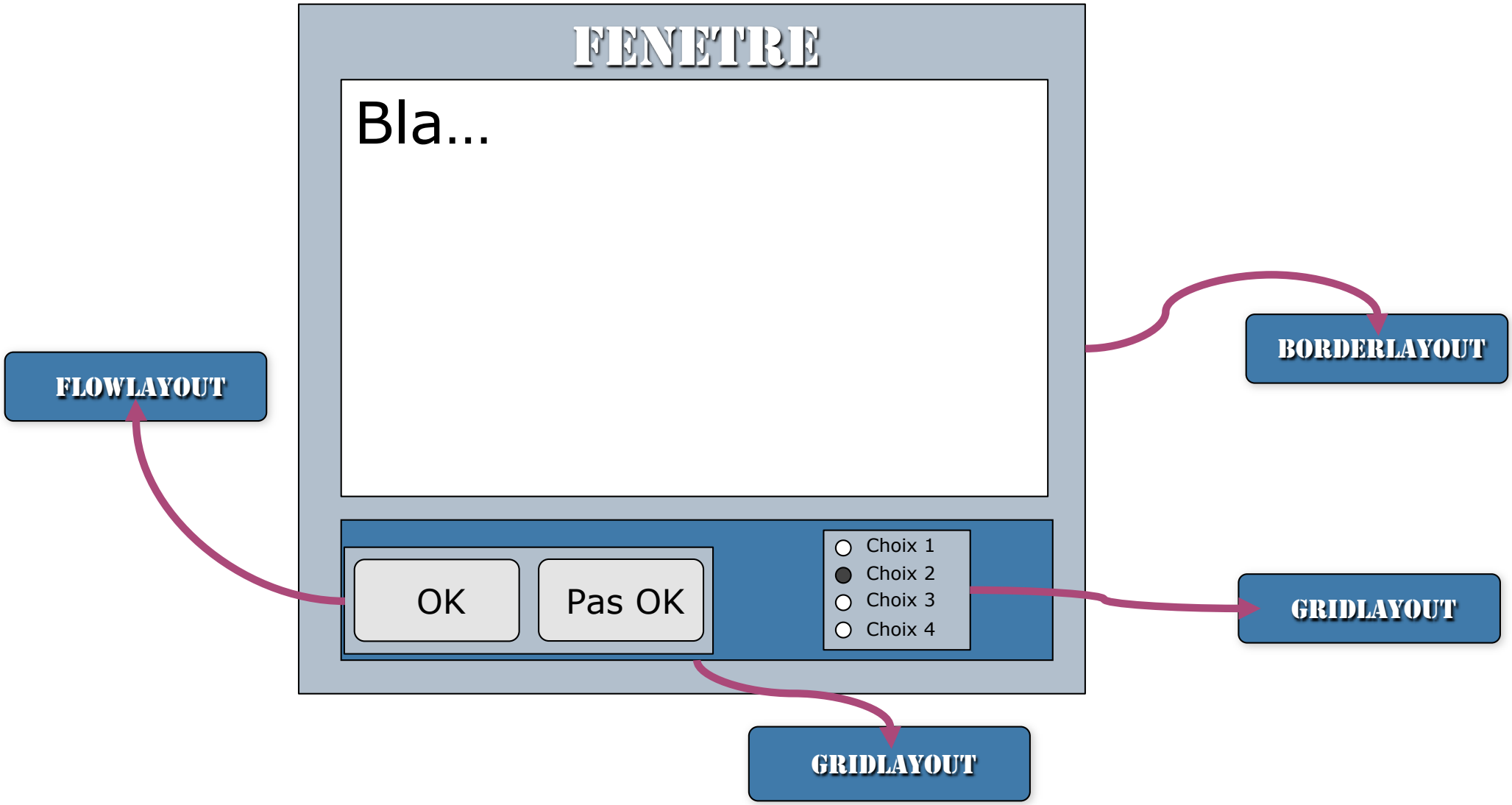
Application plus complète



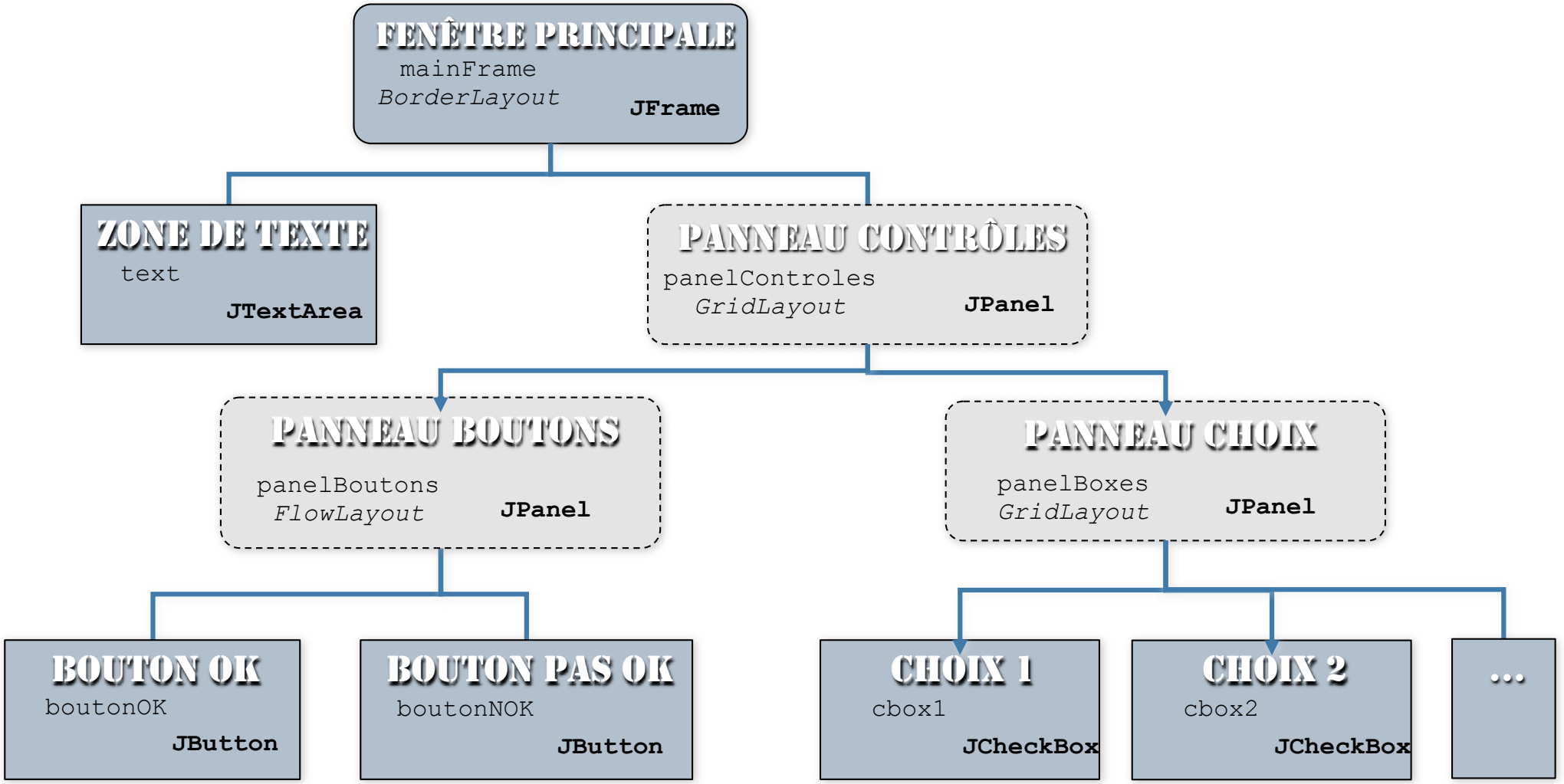
Application plus complète



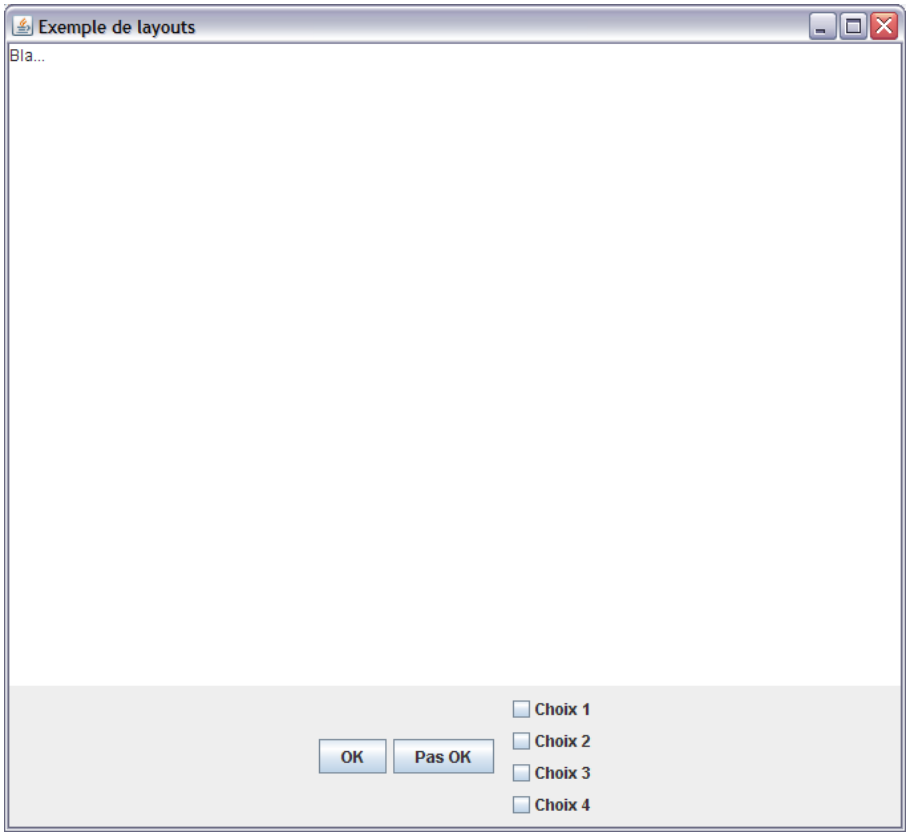
Application plus complète



Arbre de widgets

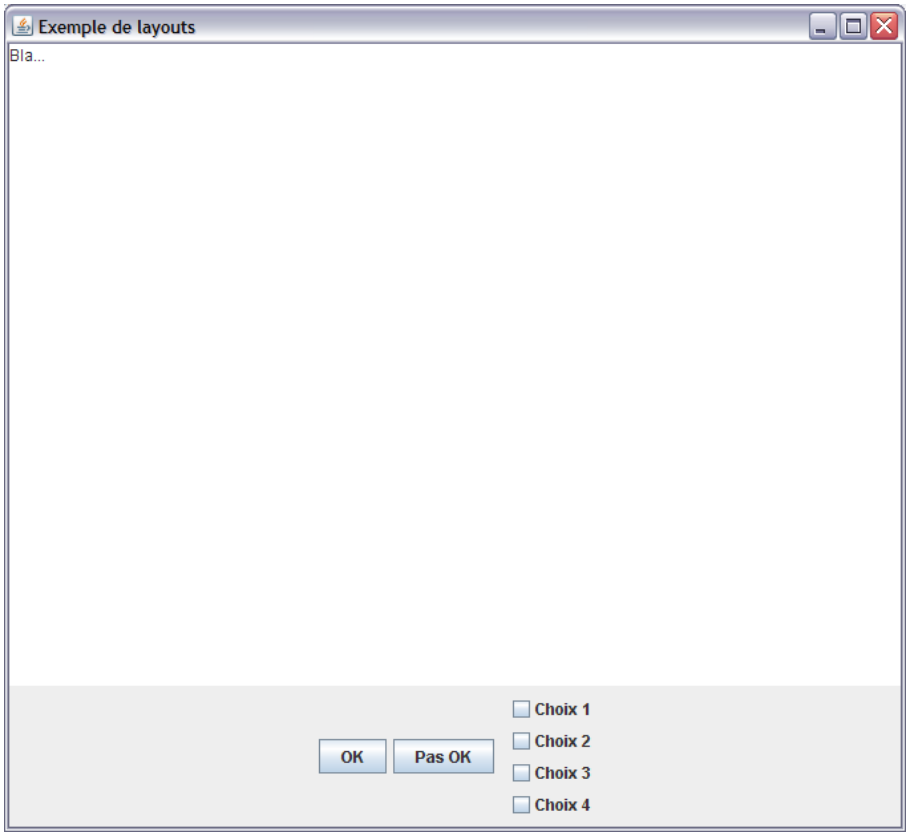


Application plus complète



Démonstration de redimensionnement

Application plus complète



Enlever les boutons de leur panel ?

Application plus complète. Code

```
public static void main(String[] args) {
      //Création de la fenêtre de l'application
      JFrame mainFrame = new JFrame("Exemple de layouts");

      //Création d'un panel pour les boutons
      JPanel panelBoutons = new JPanel();
      //Changement du layout du panel
      panelBoutons.setLayout(new FlowLayout());
      //Ajout des boutons
      panelBoutons.add(new JButton("OK"));
      panelBoutons.add(new JButton("Pas OK"));

      //Création d'un panel pour les checkBoxes
      JPanel panelBoxes = new JPanel();
      //Changement du layout du panel
      panelBoxes.setLayout(new GridLayout(4,1));
      //Ajout des checkboxes
      panelBoxes.add(new JCheckBox("Choix 1"));
      panelBoxes.add(new JCheckBox("Choix 2"));
      panelBoxes.add(new JCheckBox("Choix 3"));
      panelBoxes.add(new JCheckBox("Choix 4"));

//Suite au prochain transparent…
```

Application plus complète. Code

```
        //Création d'un panel pour les contrôles
        JPanel panelControles = new JPanel();
        //Changement du layout du panel
        panelControles.setLayout(new GridLayout(2,1));
        //Ajout des 2 panels précédents au panel contrôles
        panelControles.add(panelBoutons);
        panelControles.add(panelBoxes);

        //Changement du layout de la fenêtre
        mainFrame.setLayout(new BorderLayout());
        //Création de la zone de texte
        JTextArea text = new JTextArea("Bla...");
        //Ajout de la zone de texte à la fenêtre
        mainFrame.add(text, BorderLayout.CENTER);
        //Ajout du panel à la fenêtre
        mainFrame.add(panelControles, BorderLayout.SOUTH);

        //'Compactage' de la fenêtre
        mainFrame.pack();
        //On quitte l'application quand la fenêtre est fermée
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Affichage de la fenêtre
        mainFrame.setVisible(true);
}
```

LayoutManagers

- Le choix du LayoutManager dépend de ce que l'on veut faire… beaucoup de possibilités et besoin de pratique
- Il existe des constructeurs d'interfaces pour java (InterfaceBuilders) mais besoin de savoir ce qu'il se passe 'sous le capot' pour pouvoir ajuster, paramétrer et prévoir
- Construction dynamique: ajout de composants et changement des layouts à l 'exécution

Ce qu'il faut retenir

- Problème **compliqué**, parfois casse-tête

- Bien savoir **ce que l'on veut faire**

- **Plusieurs solutions** à un même problème

- Tout n'est pas encore résolu 'automatiquement'…

Programmer les 'interactions'

- Modèle(s)
- Vue(s)
  - Composants de l'interface: Widgets
  - Placements et gestion du layout: LayoutManagers
- Contrôleur(s)
  - Réagir aux entrées de l'utilisateur ?
  - Etablir les communications entre les M-V-C ?
  - Les **Listeners**

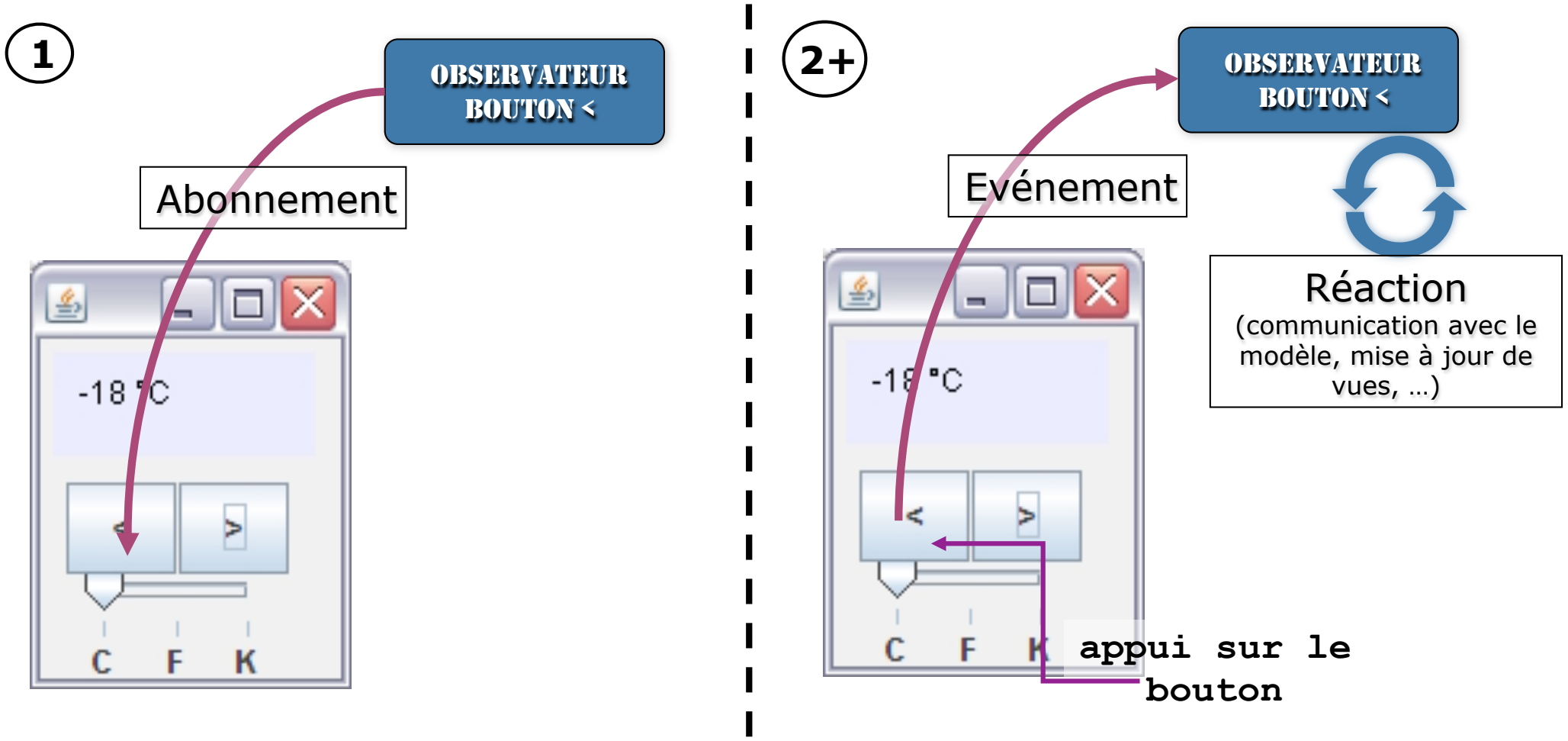
Listeners

- Littéralement: ‘écouteurs’
- Représentent le(s) contrôleur(s) de l’application
- Parties du code de l’application qui vont être exécutées en réaction à des événements dans le modèle MVC
  - Entrées utilisateur
  - Changements d’état d’un composant de MVC
  - …

Principes et mécanismes

- Patron de conception 'Observateur' (*Observer pattern*)
  - L'observé:
    - Maintient une liste de ses observateurs
    - Notifie ses observateurs des changements auxquels ils sont abonnés (envoie des événements)
  - L'observateur:
    - S'abonne à l'observé
    - Réagit lorsque l'observateur le notifie (reçoit des événements)
    - Peut se désabonner
  - Réduit les dépendances (interfaces/classes abstraites)

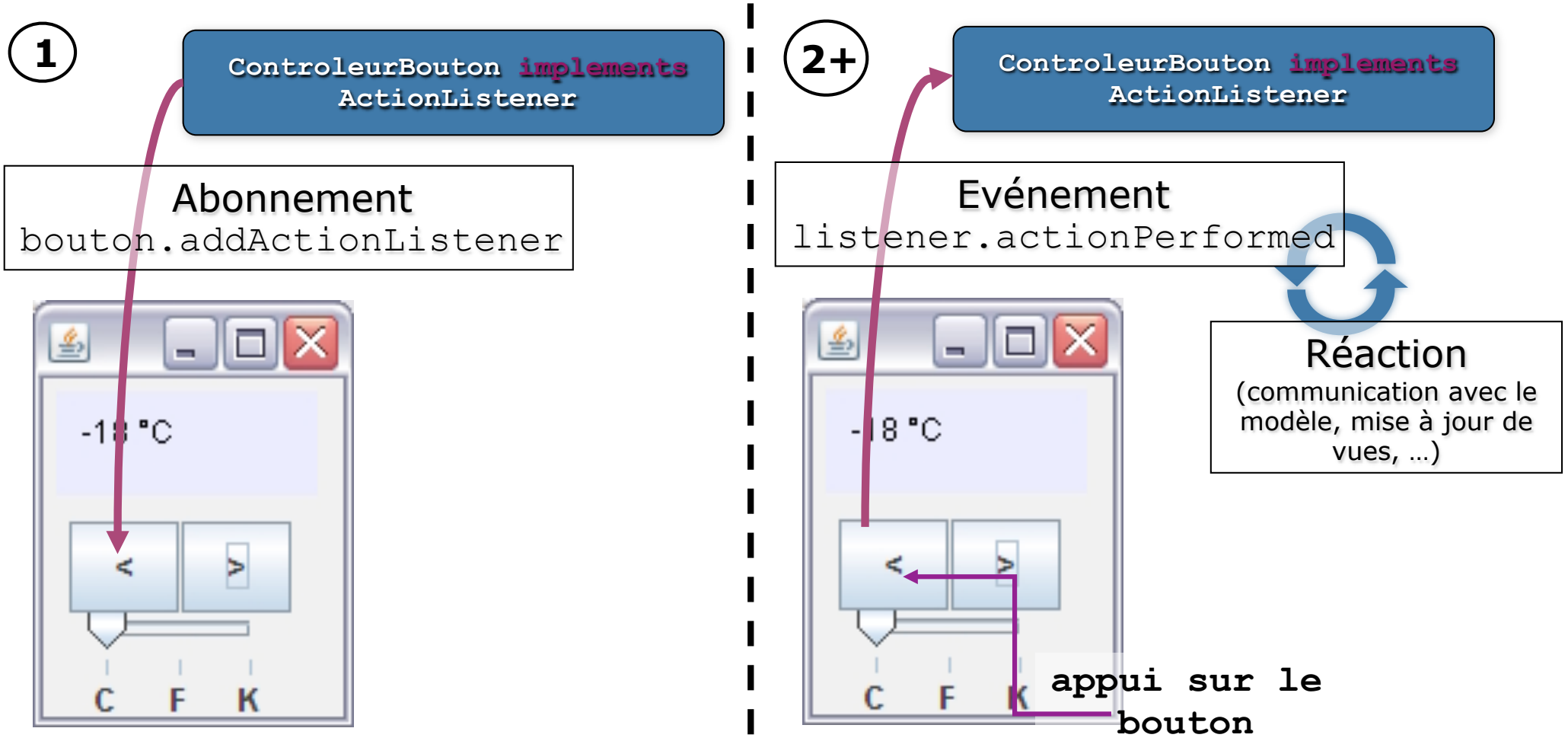
Exemple


1
OBSERVATEUR BOUTON <
Abonnement
-18 °C
<
>
C F K
2+
OBSERVATEUR BOUTON <
Evénement
Réaction
(communication avec le modèle, mise à jour de vues, …)
-18 °C
<
>
C F K
appui sur le bouton

En AWT/Swing: les listeners

- Observés = Widgets
  - Mécanismes d'abonnement
  - Mécanismes de notification
- Observateurs = Listeners
  - Interface *`EventListener`* de AWT et ses interfaces dérivées:
    - `ActionListener`, `ChangeListener`, `WindowListener`, `MouseListener`, `MouseMotionListener`, **etc.**
  - Implanter la (ou les) méthodes que doit appeler l'observé pour la notification
    - `actionPerformed`, `stateChanged`, **etc.**

Exemple 2



PROGRAMMATION ÉVÉNEMENTIELLE

Listeners: utilisation

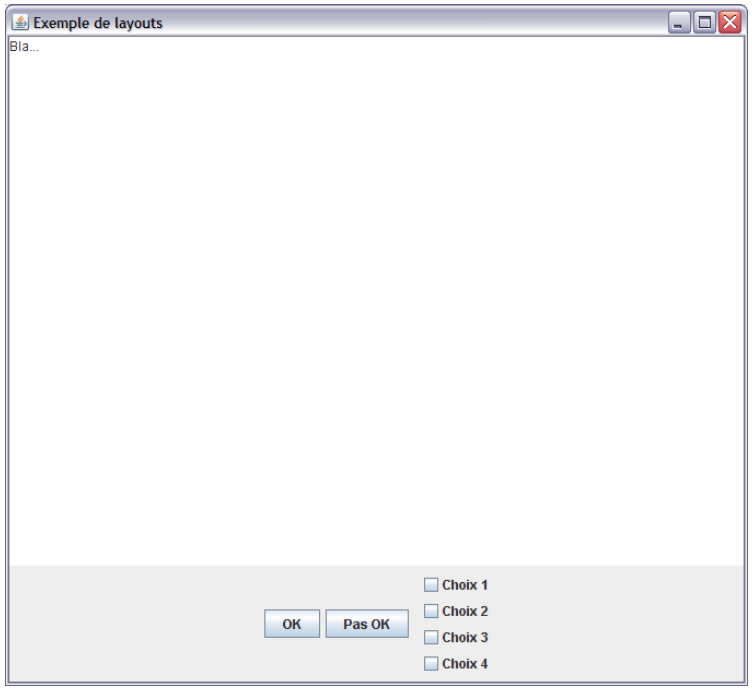
- Les interfaces décrivent des Listeners avec une 'sémantique' différente, selon les événements écoutés:
  - **`ActionListener`**: écouter des actions avec `actionPerformed (ActionEvent e)`
  - **`ChangeListener`**: écouter des changements d'état avec `stateChanged(ChangeEvent e)`
  - **`MouseMotionListener`**: écouter les mouvements de souris avec `mouseMoved(MouseEvent e)` **et** `mouseDragged(MouseEvent e)`
  - **`MouseListener`**: écouter les actions sur la souris avec `mouseClicked (MouseEvent e),mouseEntered(MouseEvent e)`, **etc.**
  - **`KeyListener`**: écouter les événements clavier avec `keyPressed (KeyEvent e),keyReleased (KeyEvent e)`, **etc.**
  - …

Etc… voir Javadoc…

Listeners: utilisation

- Les widgets permettent de s'abonner à certains types d'événements:
  - **`Component`**: `addKeyListener`, `addMouseListener`, `addMouseMotionListener`, **etc.**
  - **`JFrame`**: `addWindowListener`, **etc.**
  - **`JButton`**: `addActionListener`, `addChangeListener`
  - ...
- **Javadoc: décrit pour chaque widget quels Listeners peuvent être attachés et quels événements sont déclenchés à quels moments**

Exemple



Ajouter dans la zone de texte le texte du contrôle sur lequel on appuie

Listener: exemple 1

- Ecouter l'appui sur le bouton OK:
  - Créer une classe **`ControleurBoutonOK`** qui implante *`ActionListener`*
    - Écrire le code de la méthode `actionPerformed` qui sera appelée lorsque un événement sera notifié
  - Créer un bouton (`buttonOK = new JButton()`) et le placer dans un container
  - Créer une instance de **`ControleurBoutonOK`** (`ctrl = new ControleurBoutonOK()`) et l'abonner au bouton (`buttonOK.addActionListener(ctrl)`)
  - Rendre le container de haut-niveau visible
- La méthode `actionPerformed` de `ctrl` sera appelée à chaque appui sur `buttonOK` !

Exemple: code du listener pour le bouton OK

- **Action**: ajouter "OK" à la ligne dans la zone de texte => le listener doit 'connaître' la zone de texte

```
package gui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JTextArea;

public class ControleurBoutonOK implements ActionListener {

    JTextArea text;

    public ControleurBoutonOK(JTextArea text) {
                this.text = text;
    }

    public void actionPerformed(ActionEvent e) {
                text.setText(text.getText() + "\nOK");
    }
}
```

Zone de texte en paramètre du constructeur

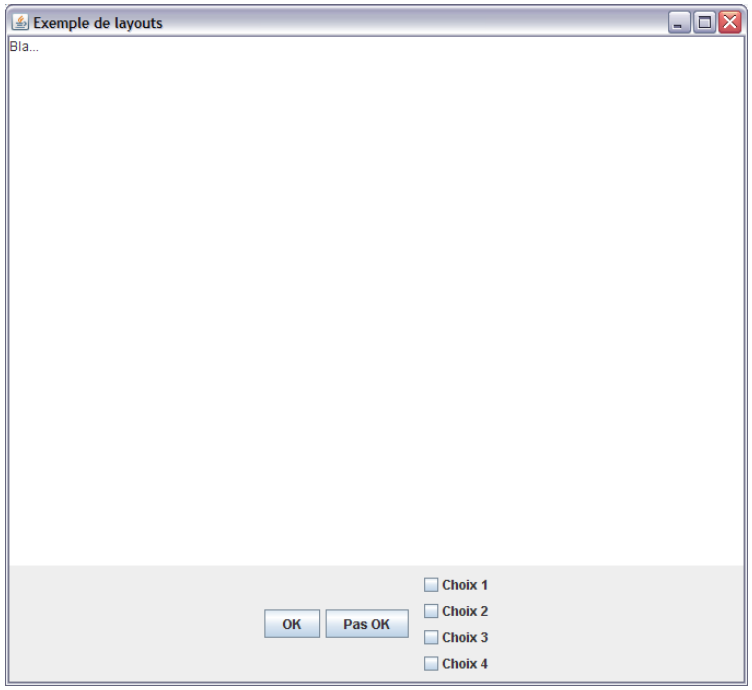
Code de l'action à réaliser

Exemple: dans le code du constructeur de la vue

- Ajout d'un Listener au bouton OK:

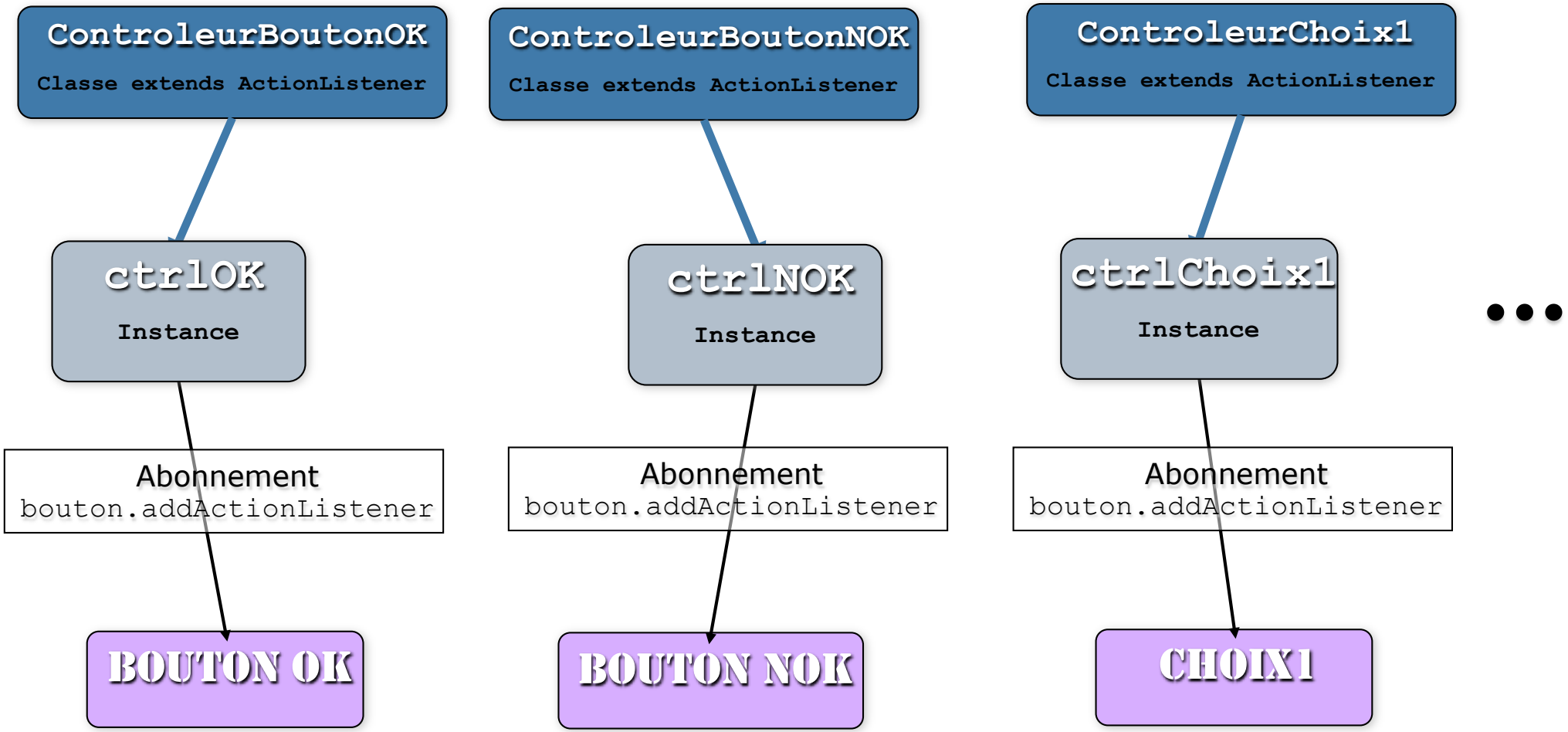
```
//Création de la zone de texte
JTextArea text = new JTextArea("Bla...");
//…
//Ajout des boutons
buttonOK = new JButton("OK");
panelBoutons.add(buttonOK);
//Ajout d'un listener au bouton OK
buttonOK.addActionListener(new ControleurBoutonOK
    (text));
//etc.
```

Exemple



Démonstration

Une classe et une instance de listener par widget...



Exemple: suite...

- Pour les autres contrôles:
  - Créer une nouvelle classe Listener par contrôle (controleurBoutonNOK, controleurChoix1, controleurChoix2, …)

⇒ LOURD (TOUS LES LISTENERS RÉALISENT LA MÊME TÂCHE)

  - Créer une seule classe de Listener qui effectue la même tâche, avec des widgets différents (avec une instance de Listener par widget)

PLUS LOGIQUE ET MOINS LOURD

  - Créer une seule classe de Listener qui effectue la même tâche, avec des widgets différents (avec une instance unique du Listener)

ENCORE MIEUX… MAIS UTILISER LES ÉVÉNEMENTS

Exemple: suite...

- Pour les autres contrôles:
  - Créer une nouvelle classe Listener par contrôle (controleurBoutonNOK, controleurChoix1, controleurChoix2, …)

    LOURD

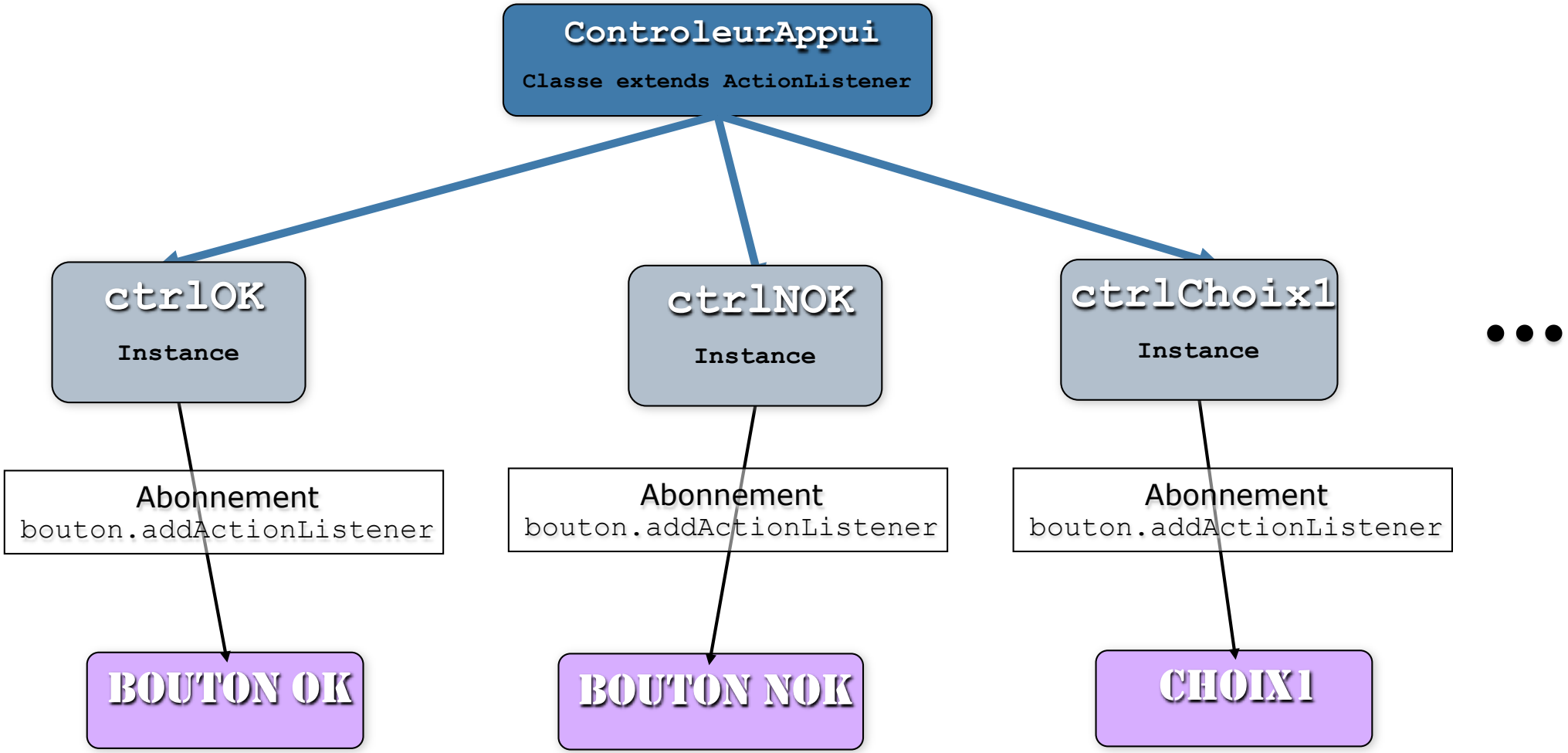
  - Créer une seule classe de Listener qui effectue la même tâche, avec des widgets différents (avec une instance de Listener par widget)

    ⇒ PLUS LOGIQUE ET MOINS LOURD

  - Créer une seule classe de Listener qui effectue la même tâche, avec des widgets différents (avec une instance unique du Listener)

    ENCORE MIEUX… MAIS UTILISER LES ÉVÉNEMENTS

Une seule classe de listener et une instance par widget...



Listener: exemple 2

- Ecouter l'appui sur un widget pour mettre à jour le texte:
  - Créer une classe **`ControleurAppui`** qui implante *`ActionListener`*
    - Les instances du Listener doivent connaitre le widget sur lequel ils opèrent (paramètre du constructeur)
    - Écrire le code de la méthode `actionPerformed` qui sera appelée lorsque un événement sera notifié
  - Créer les contrôles boutons, checkboxes, …(`buttonOK = new JButton()`) et les placer dans un container
  - Créer une instance de **`ControleurAppui`** pour chaque widget (`ctrl = new ControleurAppui(buttonOK)`) et l'abonner (`buttonOK.addActionListener(ctrl)`),…
  - Rendre le container de haut-niveau visible
- La méthode `actionPerformed` de `ctrl` sera appelée à chaque appui sur un widget !

Exemple 2: code du listener pour les widgets

- **Action**: ajouter le texte du widget à la ligne dans la zone de texte => le listener doit 'connaître' le widget qui lui est associé et la zone de texte

```
package gui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.AbstractButton;
import javax.swing.JTextArea;

public class ControleurAppui implements ActionListener {

        AbstractButton widget;
        JTextArea text;

        public ControleurAppui(JTextArea text, AbstractButton b) {
                        this.text = text;
                        widget = b;
        }

        public void actionPerformed(ActionEvent e) {
                        text.setText(text.getText() + "\n" + widget.getText());
        }
}
```

Zone de texte et widget en paramètres du constructeur

Code de l'action à réaliser

Exemple 2: dans le code du constructeur de la vue

- Ajout d'une instance du Listener par widget:

```
//Création de la zone de texte
JTextArea text = new JTextArea("Bla...");
//…
//Ajout des boutons
buttonOK = new JButton("OK");
panelBoutons.add(buttonOK);
//Ajout d'un listener au bouton OK
buttonOK.addActionListener(new ControleurAppui(text, buttonOK));
buttonNOK = new JButton("Pas OK");
panelBoutons.add(buttonNOK);
//Ajout d'un listener au bouton NOK
buttonNOK.addActionListener(new ControleurAppui(text, buttonNOK));

//idem pour les checkboxes (se sont des AbstractButtons…)
choix1 = new JCheckBox("Choix 1");
panelBoxes.add(choix1);
//Ajout d'un listener au choix 1
choix1.addActionListener(new ControleurAppui(text, choix1));
//etc.
```

Exemple: suite…

- Pour les autres contrôles:
  - Créer une nouvelle classe Listener par contrôle (controleurBoutonNOK, controleurChoix1, controleurChoix2, …)

    LOURD

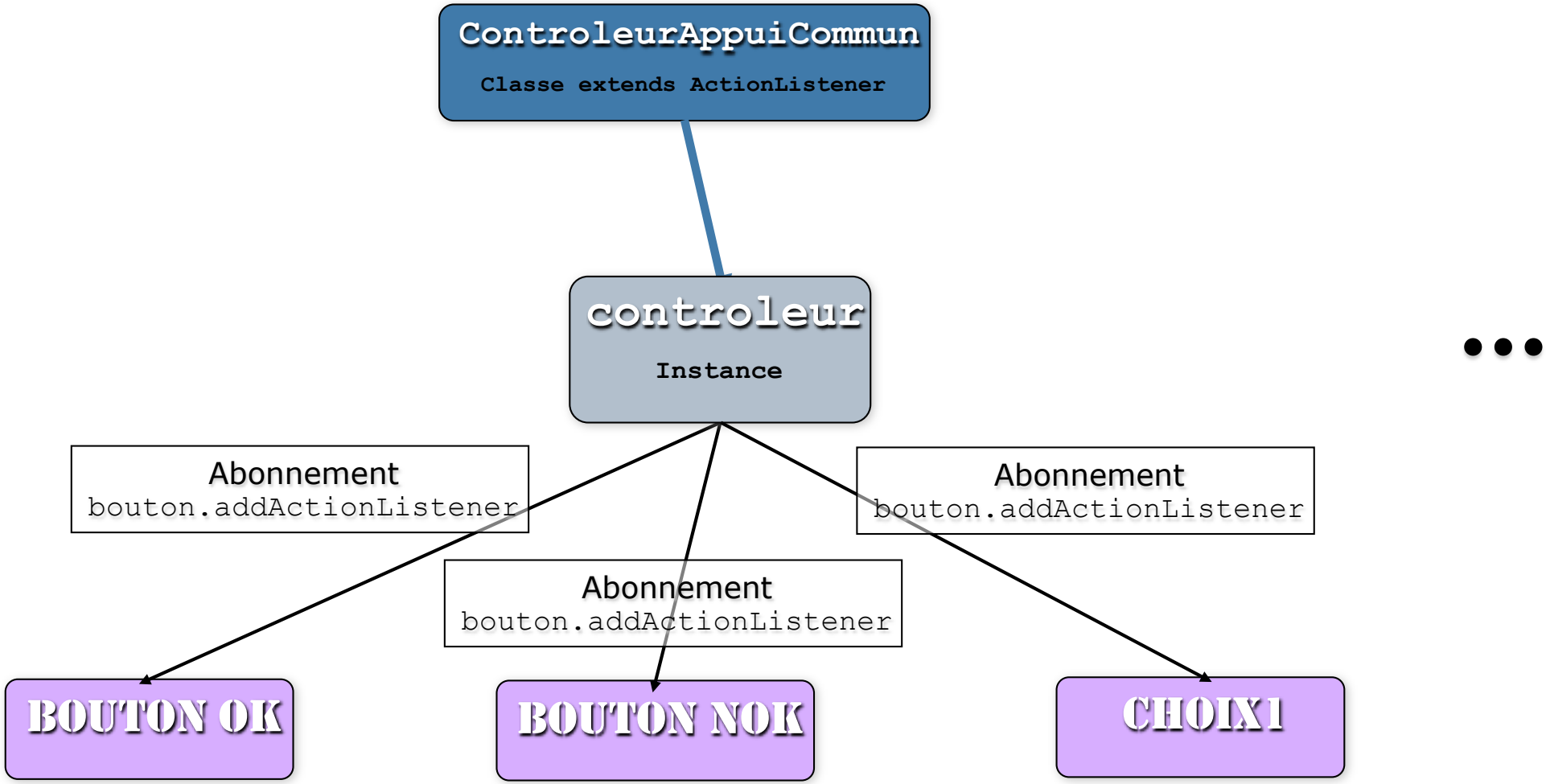
  - Créer une seule classe de Listener qui effectue la même tâche, avec des widgets différents (avec une instance de Listener par widget)

    PLUS LOGIQUE ET MOINS LOURD

  - Créer une seule classe de Listener qui effectue la même tâche, avec des widgets différents (avec une instance unique du Listener)

⇨ ENCORE MIEUX… MAIS UTILISER LES 'ÉVÉNEMENTS'

Une seule classe de listener et une seule instance...



...

Problème…

- Comment savoir quoi faire pour le Listener:
  - Widget qui a lancé l'action ?
  - Opérations / Actions à effectuer ?
  - …

⇨ **UTILISATION DES 'ÉVÈNEMENTS'**

Détails sur l'ActionListener

- La méthode `actionPerformed(ActionEvent e)`
  - Appelée lorsque une action est effectuée sur l'observé (bouton, checkbox, …, tout widget permettant d'ajouter un `actionListener`)
  - Le paramètre `ActionEvent e`:
    - Permet à l'observé de donner à l'observateur des informations sur l'événement à l'origine de la notification: la source, l'état (*'consumé'* ou non), etc.

Les événements

- Passés en paramètres des méthodes de notification des Listeners
- Héritent tous de la classe abstraite `java.awt.AWTEvent`
  - `ActionEvent` (pour `ActionListener`)
  - `MouseEvent` (pour `MouseListener` et `MouseMotionListener`)
  - `KeyEvent` (pour `KeyListener`)
  - …
- Générés par le composant source (observé)

Les événements

- Fournissent des informations à l'observateur
  - Tous: `public Object getSource()`
    Le composant source de l'événement
  - `ActionEvent:public String getActionCommand()`
    Une commande associée à l'action
  - `MouseEvent:public int getX(),public int getY(),public Point getPoint()`
    Les coordonnées du pointeur au moment de l'événement
  - …

Etc… voir Javadoc…

Listener: exemple 3

- Ecouter l'appui sur un widget pour mettre à jour le texte:
  - Créer une classe **`ControleurAppuiCommun`** qui implante *`ActionListener`*
    - Les instances obtiendront des informations sur l'action à réaliser par le paramètre ActionEvent
    - Écrire le code de la méthode `actionPerformed` qui sera appelée lorsque un événement sera notifié
  - Créer les contrôles boutons, checkboxes, …(`buttonOK = new JButton()`) et les placer dans un container
  - Créer une seule instance de **`ControleurAppuiCommun`** (`ctrl = new ControleurAppuiCommun(buttonOK)`) et l'abonner à tous les widgets (`buttonOK.addActionListener(ctrl)`),…
  - Rendre le container de haut-niveau visible
- La méthode `actionPerformed` de `ctrl` sera appelée à chaque appui sur un widget !

Exemple 3: code du listener pour les widgets

- **Action**: ajouter le texte du widget à la ligne dans la zone de texte => le listener doit 'connaître' la zone de texte et la propriété actionCommand') de chaque widget a été réglée

```
package gui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JTextArea;

public class ControleurAppuiCommun implements ActionListener {

	JTextArea text;

	public ControleurAppuiCommun(JTextArea text) {
			this.text = text;
	}

	public void actionPerformed(ActionEvent e) {
			text.setText(text.getText() + "\n" + e.getActionCommand());
	}
}
```

Zone de texte en paramètres du constructeur

Code de l'action à réaliser

Exemple 3: dans le code du constructeur de la vue

- Ajout de la même instance du Listener à chaque widget:

```
//Création de la zone de texte
JTextArea text = new JTextArea("Bla...");
//Création du listener
ControleurAppuiCommun ctrl = new ControleurAppuiCommun(text);
//Ajout des boutons
buttonOK = new JButton("OK");panelBoutons.add(buttonOK);
//Réglage de la propriété actionCommand
buttonOK.setActionCommand(buttonOK.getText());
//Ajout du listener au bouton OK
buttonOK.addActionListener(ctrl);
buttonNOK = new JButton("Pas OK");panelBoutons.add(buttonNOK);
//Réglage de la propriété actionCommand
buttonNOK.setActionCommand(buttonNOK.getText());
//Ajout du listener au bouton NOK
buttonNOK.addActionListener(ctrl);
//idem pour les checkboxes
choix1 = new JCheckBox("Choix 1");panelBoxes.add(choix1);
//Réglage de la propriété actionCommand
choix1.setActionCommand(choix1.getText());
//Ajout du listener au choix 1
choix1.addActionListener(ctrl);
//etc.
```

Exemple 3, détails

- Le Listener peut réaliser des actions différentes, selon le widget qui a déclenché l'événement

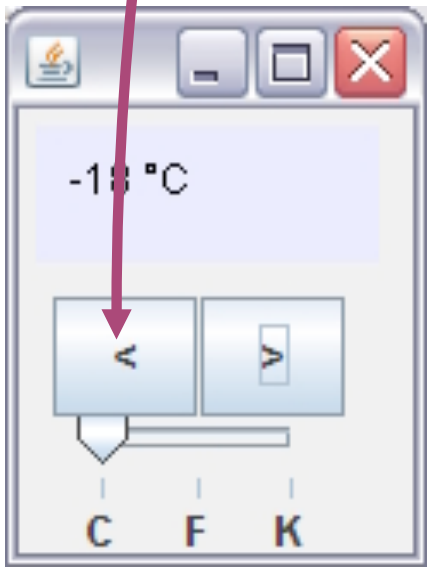
- Reconnaitre l'action à réaliser avec l'objet `Event` obtenu en paramètre:
  - En obtenant le widget source de l'événement (`getSource()`)
  - En utilisant des propriétés propres des événements (`getActionCommand()`,...)

Un dernier exemple...

1

ControleurBouton implements ActionListener

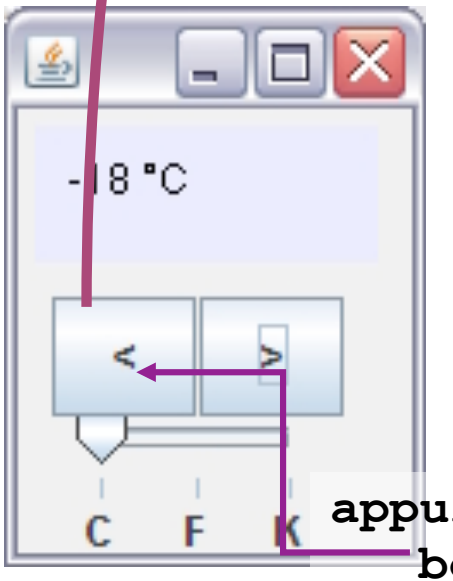
Abonnement
bouton.addActionListener



2+

ControleurBouton implements ActionListener

Evénement
listener.actionPerformed



appui sur le bouton

Réaction
(communication avec le modèle, mise à jour de vues, …)

L'ACTION EST SEMBLABLE POUR LES 2 BOUTONS

En utilisant getSource()...

- Le Listener doit connaître les boutons (paramètres du constructeur, accesseurs par l'objet `vue`, etc.)

```
private JButton boutonUP, boutonDOWN;
//...
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == boutonUP)
    modele.rechauffement();
    else if (e.getSource() == boutonDOWN)
    modele.refroidissement();
    vue.redessiner();
}
```

En utilisant getActionCommand()

- Spécifique aux `ActionListener`
- Il n'est plus nécessaire au Listener de connaître les widgets
- Plusieurs widgets peuvent lancer la même action
- Régler la propriété `actionCommand` des widgets qui sera transmise par l'objet `ActionEvent` en paramètre de la méthode `actionPerformed`

En utilisant getActionCommand()

- Définir des constantes pour les `actionCommand` (dans la classe du listener)

```
static public final String ACTION_RECHAUFFE =
    "RECHAUFFE";
static public final String ACTION_REFROIDIT =
    "REFROIDIT";
```

- Régler les propriétés `actionCommand` des widgets

```
pButton.setActionCommand
    (ControleurThermometreButtonsActionCommand.ACTION_
    RECHAUFFE);
mButton.setActionCommand
    (ControleurThermometreButtonsActionCommand.ACTION_
    REFROIDIT);
```

En utilisant getActionCommand()

- Implanter la méthode `actionPerformed`

```
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand() == ACTION_RECHAUFFE)
            modele.rechauffement();
    else if (e.getActionCommand() == ACTION_REFROIDIT)
            modele.refroidissement();
    vue.redessiner();
}
```

Quelques outils supplémentaires dans AWT et SWING

- 'Factorisation' de Listeners
  - L'interface `MouseInputListener` étend les interface `MouseMotionListener` (mouvements) et `MouseListener` (actions)
- Les '**Adapters**'
  - Classes abstraites qui implantent des interfaces 'Listener' avec des méthodes 'vides' (ne font rien)
  - Réduisent le code à écrire (on ne surcharge que les méthodes des événements auxquels on veut réagir)
  - Exemple: `MouseAdapter` qui implante `MouseListener`, `MouseMotionListener`, `MouseWheelListener` et `EventListener`.
- L'interface '**Action**'
  - Mécanisme qui simplifie et généralise l'utilisation des `ActionListener` sur les widgets de Swing (voir **Javadoc**)

Bilan sur l'utilisation des Listeners

- Implanter la ou les interfaces XListener selon les événements que l'on veut écouter
- 3 méthodes selon les besoins:
  - Implantation d'*une classe spécifique à un besoin* et *à un widget* (une classe / une instance)
  - Implantation d'*une classe spécifique à un besoin* et pouvant opérer sur *plusieurs widgets* (une classe / plusieurs instances)
  - Implantation d'une *classe prenant en compte plusieurs besoins* et pouvant opérer sur *plusieurs widgets* (une classe / une instances)
  - Mélange des méthodes…

⇨ **DÉPEND DES BESOINS, DES PROBLÈMES DES HABITUDES…**

Ce qu'il faut retenir

- **Observateur / Observé**

- Que **notifient** les widgets ?

- **Méthodes** pour implanter les Listeners

- Savoir s'**adapter** aux besoins…