JAVA

Applications interactives

-

Modèle Vue-Contrôleur (MVC)

Stéphane HUOT
Dpt. Informatique
iUT ORSAY

Introduction

- 6h00 de cours
  - Programmation d'applications interactives en Java
    - Structure d'une application interactive: le modèle **MVC** *(1-2h00)*
    - Les interfaces graphiques en Java: *AWT* et ***Swing*** *(4-5h00)*

Introduction

- Notions à retenir:

- Point important:

Introduction

- Pas de distribution des transparents du cours

Mais

- Mise en ligne du 'pdf' après le cours
- Prenez des notes, je dis des choses intéressantes et utiles parfois…

Plan du cours

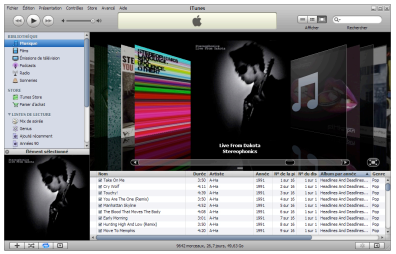
1. Structure d'une application interactive
   - Ce que l'on « fait » et que l'on « voit »
   - Ce qu'il se passe
2. Réalisation d'applications interactives: **Modèle-Vue-Contrôleur (MVC)**
   - Le modèle
   - La vue
   - Le contrôleur
3. Utilisation et réalisation de MVC
   - Analyse en terme de MVC
   - Réalisation en Java: 2 exemples pour démarrer

Plan du cours

1. Structure d'une application interactive
   - Ce que l'on « fait » et que l'on « voit »
   - Ce qu'il se passe
2. Réalisation d'applications interactives: **Modèle-Vue-Contrôleur (MVC)**
   - Le modèle
   - La vue
   - Le contrôleur
3. Utilisation et réalisation de MVC
   - Analyse en terme de MVC
   - Réalisation en Java: 2 exemples pour démarrer

Application interactive

- Une application avec laquelle l'utilisateur peut **interagir**:
  - L'application effectue des **opérations en réponse aux actions de l'utilisateur**
  - Coopération entre le programme et l'utilisateur, **commandée par l'utilisateur**



Ex: iTunes - ©Apple

  - Est-ce qu'un *shell* (ligne de commandes) est une application interactive ?

Programmer des applications interactives

- Tâche pouvant se révéler complexe:
  - Parce que la tâche que doit accomplir l'utilisateur peut être complexe
  - Parce qu'il faut prévoir les **scenarios d'interaction** (et donc les réactions de l'application)
  - Parce qu'il faut pouvoir **maintenir** et **réutiliser**

⇒ Passage à l'échelle (application importante)

Programmer des applications interactives

- Heureusement, il y a:
  - **Des concepts** (structures et modèles d'applications interactives)
    - MVC
    - PAC
    - Nombreux 'Design patterns' (patrons de conception)
    - …
  - **Des outils** (pour concrétiser ces concepts)
    - Des langages/environnements de programmation adaptés
    - Librairies et leurs APIs (Application Programming Interfaces): boîtes à outils
    - …

Structure d'une application interactive

- La partie 'visible' *(**front office**)*:
  ce que l'on fait et ce que l'on voit
  - Interface Homme-Machine (IHM)
- La partie 'invisible' *(**back office**):*
  ce qu'il se passe
  - Traitements
  - Données (stockage et accès)
  - Communications

Structure d'une application interactive:
Ce que l'on fait et ce que l'on « voit »

- La tâche de l'utilisateur et l'IHM de l'application
  - Interface non graphique:
    - *Ex:* Ligne de commande, tableau de commandes et indicateurs, …
  - Interface graphique (GUI):
    - *Ex:* Application standard Windows, page web, …
- L'utilisateur **'commande'** l'application (programmation événementielle)
  - L'IHM doit être adaptée à la tâche (cf PIG en S2)
  - L'application doit être **'réactive'** (pas de traitements trop longs ou alors les notifier à l'utilisateur)

Structure d'une application interactive:
Ce que l'on fait et ce que l'on « voit »

- Les **entrées** (ce que l'on fait)

et

- Les **sorties** (ce que l'on voit)

Les entrées






...

Les sorties

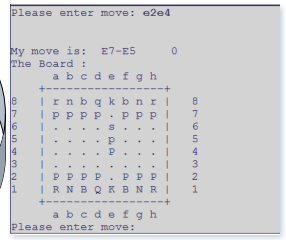




...

Exemple: Jeu d'échecs

- La tâche globale: jouer aux échecs!
  - Sous-tâche: déplacer les pièces
- Interfaces utilisateur:

```
Please enter move: e2e4

My move is:  E7-E5    0
The Board :
     a b c d e f g h
   +-----------------+
8  | r n b q k b n r |  8
7  | p p p p . p p p |  7
6  | . . . . s . . . |  6
5  | . . . . p . . . |  5
4  | . . . . P . . . |  4
3  | . . . . . . . . |  3
2  | P P P P . P P P |  2
1  | R N B Q K B N R |  1
   +-----------------+
     a b c d e f g h
Please enter move:
```

Ligne de commande

ou

Graphique

Structure d'une application interactive:
Ce qu'il se passe

- Le **'noyau'** de l'application:
  - Fonctionnalités
  - Accès aux données
  - Traitement des données
- Les données
- Produit les résultats aux actions de l'utilisateur

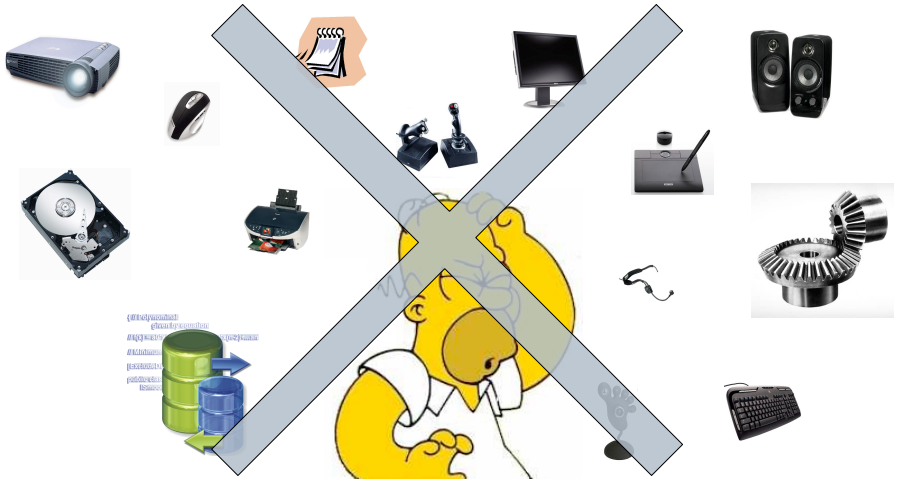
Exemple: Jeu d'échecs

- Fonctionnalités:
  - *Jouer* une partie
    - *Déplacer* des pièces
    - *Gérer* les tours de jeu
    - *Joueur virtuel*
    - …
  - *Enregistrer* une partie
  - *Charger* une partie
- Données:
  - État de la partie en cours
  - Parties sauvegardées
  - Catalogues d'ouvertures

Liaison entre ces 2 parties d'une application ?

Liaison entre ces 2 parties d'une application ? Programmation 'en vrac et comme on peut'…

Liaison entre ces 2 parties d'une application ? Programmation 'en vrac et comme on peut'…



Liaison entre ces 2 parties d'une application ?
… programmation selon un modèle organisé.

Ce qu'il faut retenir

- Application interactive ?
- L'utilisateur à le contrôle
- Développement complexe
- Séparation **Front Office / Back Office**

Plan du cours

1. Structure d'une application interactive
   - Ce que l'on « fait » et que l'on « voit »
   - Ce qu'il se passe
2. Réalisation d'applications interactives: **Modèle-Vue-Contrôleur (MVC)**
   - Le modèle
   - La vue
   - Le contrôleur
3. Utilisation et réalisation de MVC
   - Analyse en terme de MVC
   - Réalisation en Java: 2 exemples pour démarrer

Le modèle 'Modèle-Vue-Contrôleur' (MVC)

- **MVC** est:
  - Un *patron de conception (une solution standardisée à un problème, indépendante des langages de programmation)*,
  - Une *architecture logicielle (une manière de structurer une application ou un ensemble de logiciels)*.
- Introduit en 1979 par *Trygve Reenskaug*,
- Très fortement lié aux concepts de la programmation objet (*Smalltalk*).

MVC: structure du modèle

- Organiser, structurer une application interactive en séparant:
  - Les données et leurs traitements:
    Le Modèle
  - La représentation des données
    La Vue
  - Le comportement de l'application
    Le Contrôleur

Le modèle 'Modèle-Vue-Contrôleur' (MVC)

MODELE
- fonctionnalités de l'application
- accès aux données et traitements

VUE
- présentation des données à l'utilisateur

CONTROLEUR
- gestion des entrées de l'utilisateur
- définit le comportement de l'application

Le modèle 'Modèle-Vue-Contrôleur' (MVC)

MODELE
- fonctionnalités de l'application
- accès aux données et traitements

1-entrées utilisateur

VUE
- présentation des données à l'utilisateur

CONTROLEUR
- gestion des entrées de l'utilisateur
- définit le comportement de l'application

Le modèle 'Modèle-Vue-Contrôleur' (MVC)

MODELE
- fonctionnalités de l'application
- accès aux données et traitements

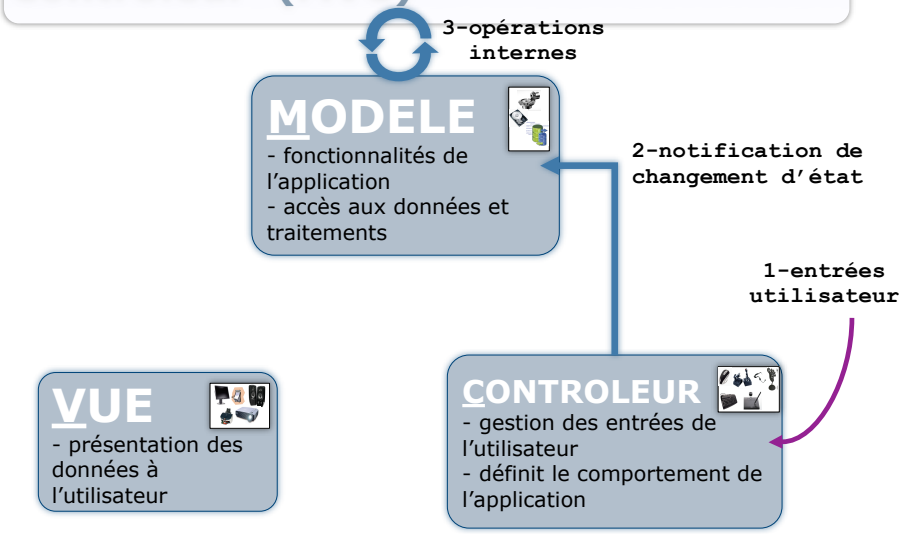
2-notification de changement d'état

1-entrées utilisateur

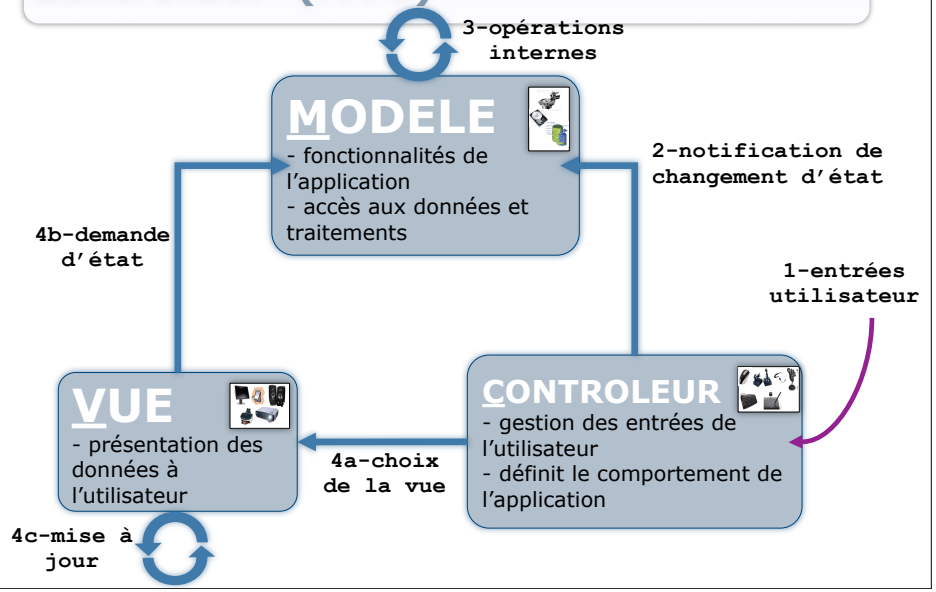
VUE
- présentation des données à l'utilisateur

CONTROLEUR
- gestion des entrées de l'utilisateur
- définit le comportement de l'application

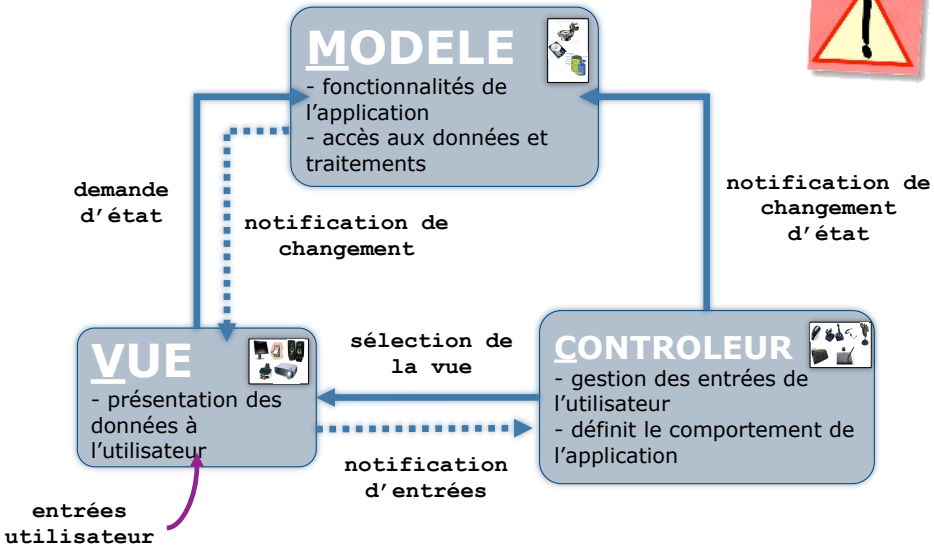
Le modèle 'Modèle-Vue-Contrôleur' (MVC)



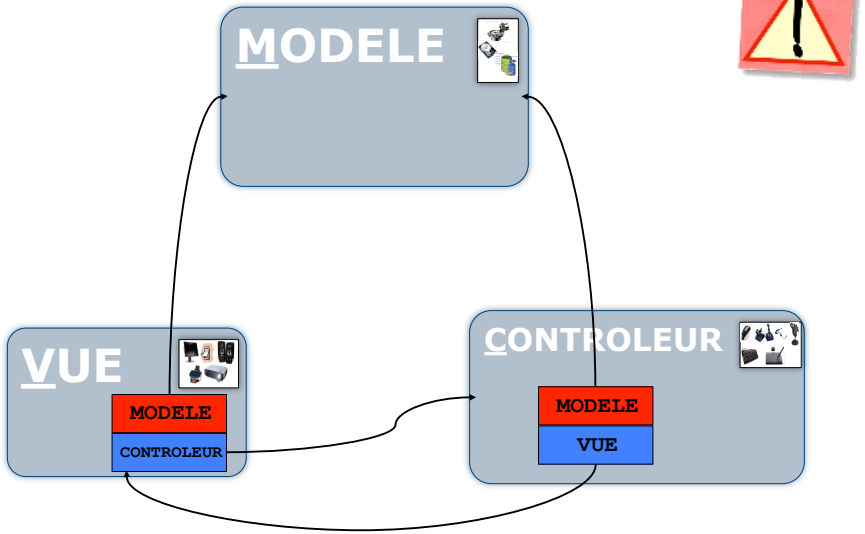
Le modèle 'Modèle-Vue-Contrôleur' (MVC)



Le modèle 'Modèle-Vue-Contrôleur' (MVC) Vision plus concrète



Le modèle 'Modèle-Vue-Contrôleur' (MVC) Références entre composants



MVC: le modèle

- Le modèle:
  - Représente les données
  - Fournit les accès aux données
  - Fournit les traitements applicables aux données
  - Expose les fonctionnalités de l'application

⇒ **Noyau Fonctionnel de l'application**

MVC: la vue

- La vue:
  - Représente la (ou une) représentation des données du modèle
  - Assure la consistance entre la représentation qu'elle donne et l'état du modèle/le contexte de l'application

⇒ **Sorties de l'application**

MVC: le contrôleur

- Le contrôleur:
  - Représente le comportement de l'application face aux actions de l'utilisateur
  - Fournit la traduction des actions de l'utilisateur en actions sur le modèle
  - Fournit la vue appropriée par rapport aux actions de l'utilisateur et des réactions du modèle

⇒ **Comportement et gestion des entrées de l'application**

Avantages de MVC

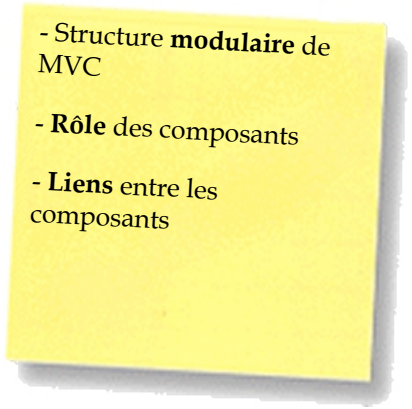
- Structure 'propre' de l'application
- **Indépendance** 'données' – 'représentation' – 'comportements'
- Adapté aux concepts de la programmation 0-0
- **Modulaire** et **réutilisable**
  - Vues interchangeables
  - Contrôleurs interchangeables
  - Changement de *'Look & Feel'*
- Facilite les vues et contrôleurs multiples
  - Synchronisation 'quasi-implicite'

Inconvénients de MVC

- Mise en place complexe dans le cas d'applications importantes
- Mises à jour potentiellement trop nombreuses
  - 'Spaghettis' dans le code
  - Temps d'exécution
- Contrôleur et Vue restent souvent fortement liés au Modèle

➡ **Adapter la réalisation au problème**

Ce qu'il faut retenir



Plan du cours

1. Structure d'une application interactive
   - Ce que l'on « fait » et que l'on « voit »
   - Ce qu'il se passe
2. Réalisation d'applications interactives: **Modèle-Vue-Contrôleur (MVC)**
   - Le modèle
   - La vue
   - Le contrôleur
3. Utilisation et réalisation de MVC
   - Analyse en terme de MVC
   - Réalisation en Java: 2 exemples pour démarrer

MVC: utilisation et réalisation

- Comment réaliser une application interactive selon le modèle MVC ?
- Ce qui a déjà été vu: **le modèle**
  - Implantation de modèles
  - Tests unitaires

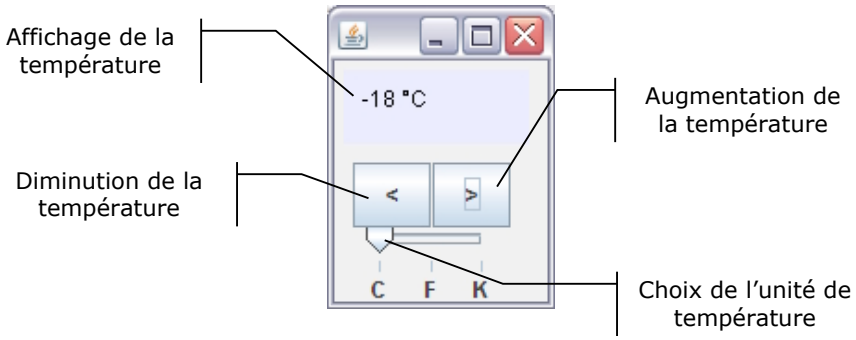
Conventions de nommage

- Paquetages:
  - Contrôleurs: `package application.controleurs;`
  - Vues: `package application.vues;`
  - Modèle: `package application.modele;`
- Classes:
  - Contrôleurs: `ControleurNomClasse.java`
  - Vues: `VueNomClasse.java`
  - Modèle: `ModeleNomClasse.java`

Exemple 1: le thermomètre v1

- Réaliser une application interactive simulant un *thermomètre*, sur laquelle l'utilisateur peut agir pour contrôler la température
- L'application fournira:
  - Un **affichage textuel** de la température courante mesurée par le thermomètre en °C ou en °K ou en F
  - Des **contrôles** permettant à l'utilisateur de **diminuer** et **augmenter** la température courante du thermomètre
  - Un **contrôle** permettant de **choisir l'unité d'affichage** de la température

Thermomètre v1

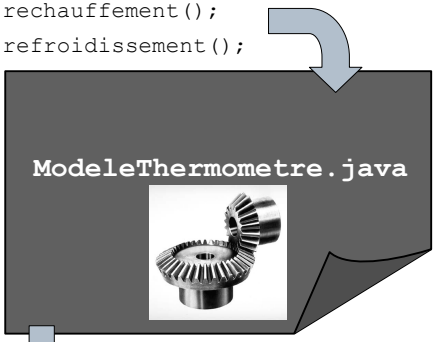


Thermomètre v1: analyse

- Le modèle (cf TP1b)
  - Données et traitements réalisés:
    - *Température courante*
    - *Maintient de l'état* de la température courante
    - *Conversions* de la température en différentes unités
  - Fonctionnalités exposées:
    - **Augmenter** la température de 1° (C ou K)
    - **Diminuer** la température de 1° (C ou K)
    - **Donner** la température en °C, °K ou F

Thermomètre v1: analyse

```
public void rechauffement();
public void refroidissement();
```



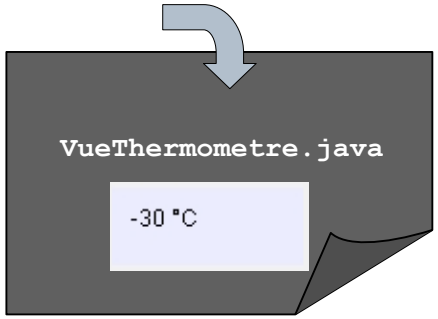

```
public double temperatureEnKelvin();
public double temperatureEnCelsius();
public double temperatureEnFarenheit();
```


Thermomètre v1: analyse

- La vue
  - **Affiche** la température courante sous forme de texte
  - **Adapte** son affichage à l'unité courante

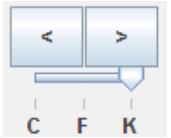
Thermomètre v1: analyse

```
public void redessiner();
public void reglerUnite(Unite unite);
```

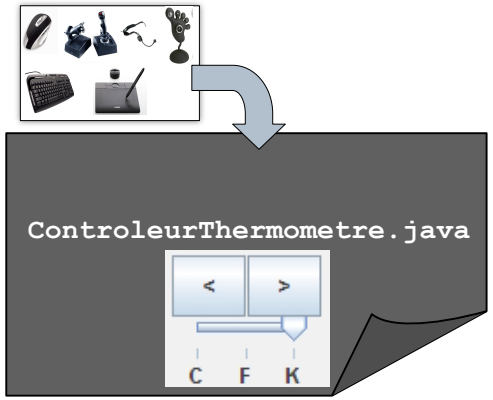


Thermomètre v1: analyse

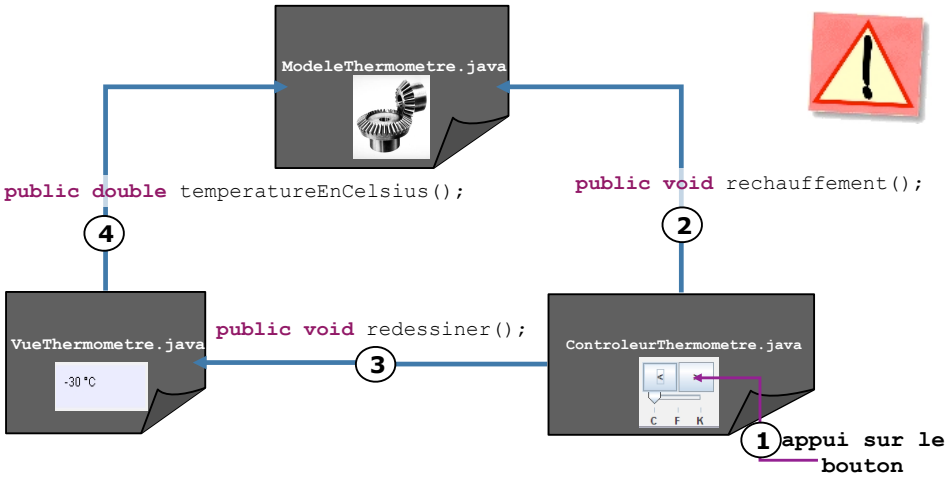
- Le contrôleur
  - **Fournit** à l'utilisateur les **contrôles** sur le modèle: **augmenter** ou **diminuer** la température
  - **Traduit** les actions de l'utilisateur en opération sur le modèle: déclenche les traitements par des appels de méthodes sur le modèle
  - **Sélectionne** et **mets à jour** la vue



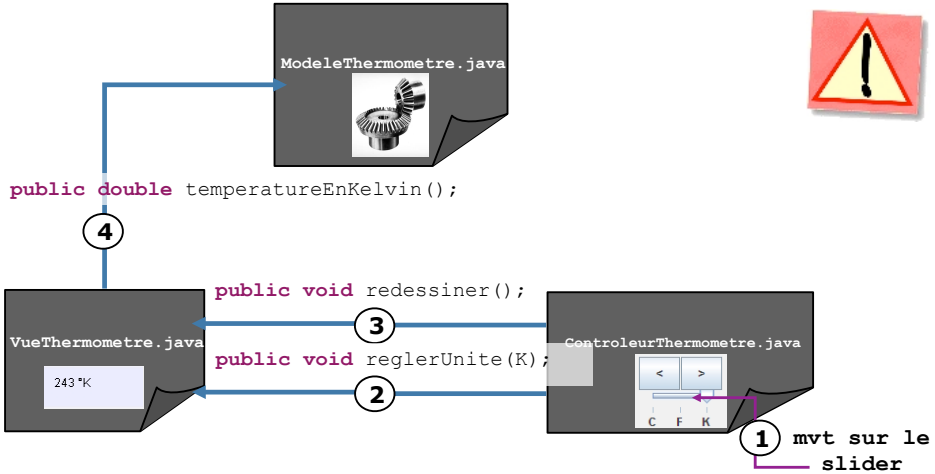
Thermomètre v1: analyse



Thermomètre v1: analyse



Thermomètre v1: analyse



Thermomètre v1: réalisation

- Le modèle (cf TP1b)

```
package thermometre.modele;
public class ModeleThermometre {
 private double _temperature;//la temperature en Kelvin
//… (constructeurs)
 public double temperatureEnKelvin(){
       return _temperature;
 }
  public double temperatureEnCelsius(){
       return _temperature - 273.15;
 }
 public double temperatureEnFarenheit(){
       return (9 / 5d) * _temperature - 459.67;
 }
 public void rechauffement(){
        _temperature++;
 }
 public void refroidissement(){
       if (_temperature > 1) {
                    _temperature--;
       }
 }
}
```

Thermomètre v1: réalisation

- La vue

```
package thermometre.vues;

public class VueThermometre {
 private ModeleThermometre _modele;//le modèle à représenter
 private Unite _unite; //l'unité d'affichage courante

 public VueThermometre (ModeleThermometre modele){
       _modele = modele;
 }

 public void reglerUnite (Unite unite){
     _unite = unite;
 }

//suite au prochain transparent...
```


Thermomètre v1: réalisation

- La vue (suite)

```
 public void redessiner (){
     double tempCourante = 0;
     switch (_unite) {
                 case CELSIUS: tempCourante = _modele.temperatureEnCelsius();
                         break;
                 case FAHRENHEIT: tempCourante = _modele.temperatureEnFarenheit();
                         break;
                 case KELVIN: tempCourante = _modele.temperatureEnKelvin();
     }

     //opérations de dessin de la vue avec la valeur tempCourante
     //...
 }

}
```


Thermomètre v1: réalisation

- Le contrôleur

```
package thermometre.controleurs;
public class ControleurThermometre {
 private ModeleThermometre _modele;//le modèle à contrôler
 private VueThermometre _vue;//la vue pour représenter le modèle

 JButton _pButton = new JButton(">");//le bouton pour augmenter la température
 JButton _mButton = new JButton("<");//le bouton pour diminuer la température
 JSlider _mSlider = new JSlider(0, 20);//le slider pour choisir l'unité

 public ControleurThermometre (ModeleThermometre modele, VueThermometre vue){
     _modele = modele;
     _vue = vue;
     //placement des contrôles de l'IHM
     //...
 }

//Suite au prochain transparent...
```


Thermomètre v1: réalisation

- Le contrôleur (suite)

```
//Lors d'une action sur le bouton « UP » (les mécanismes d'événements seront détaillés plus tard
      dans ce cours)
 public void boutonUpActivé (){
     _modele.rechauffement();
     _vue.redessiner();
 }

//Lors d'une action sur le bouton « UP » (les mécanismes d'événements seront détaillés plus tard
      dans ce cours)
 public void boutonDownActivé (){
     _modele.refroidissement();
     _vue.redessiner();
 }

//Lors d'une action sur le bouton « UP » (les mécanismes d'événements seront détaillés plus tard
      dans ce cours)
 public void sliderActivé (){
     _vue.reglerUnite(uniteDuSlider);
     _vue.redessiner();
 }
}
```

Thermomètre v1: réalisation

- 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 ptl = new ControleurThermometre(modele, vue);

			//Ajout des panneaux à la fenêtre et affichage...
			//...
	}
}
```


Commentaires sur cette réalisation de MVC ?

- Validité par rapport au modèle
- Problèmes

Commentaires sur cette réalisation de MVC

- Mécanismes de gestion des événements => *seront traités dans la suite du cours*
- Modularité => *Oui*
- Indépendance Données-Vue-Interaction ? => *Oui*

Commentaires sur cette réalisation de MVC

- **Réutilisabilité** et **généricité** ?
  - Si l'on veut utiliser un autre modèle, une autre vue ou un autre contrôleur ?
- Par exemple, si l'on veut maintenant:
  - Utiliser un autre modèle de thermomètre avec cette vue et ce contrôleur ?
  - Utiliser une autre vue avec ce modèle et ce contrôleur ?
  - Utiliser un autre contrôleur avec ce modèle et cette vue ?
  - Utiliser une autre vue et un autre contrôleur simultanément avec cette vue, ce modèle et ce contrôleur ?
  - *Etc.*

Commentaires sur cette réalisation de MVC

- **Réutilisabilité** et **généricité** ? => *Non!*
  - Si l'on veut utiliser un autre modèle, une autre vue ou un autre contrôleur ?
- Par exemple, si l'on veut maintenant:
  - Utiliser un autre modèle de thermomètre avec cette vue et ce contrôleur ?
  - Utiliser une autre vue avec ce modèle et ce contrôleur ?
  - Utiliser un autre contrôleur avec ce modèle et cette vue ?
  - Utiliser une autre vue et un autre contrôleur simultanément avec cette vue, ce modèle et ce contrôleur ? (synchronisation)
  - *Etc.*
- Impossible en l'état car les références entre les objets de chaque module sont typées par leur classe concrète. **On doit modifier des parties du code de chaque module.**

Valide par rapport au modèle, mais n'en n'exploite pas le pouvoir d'abstraction (indépendance entre données/représentation/contrôle)

Réalisation 'propre' de MVC

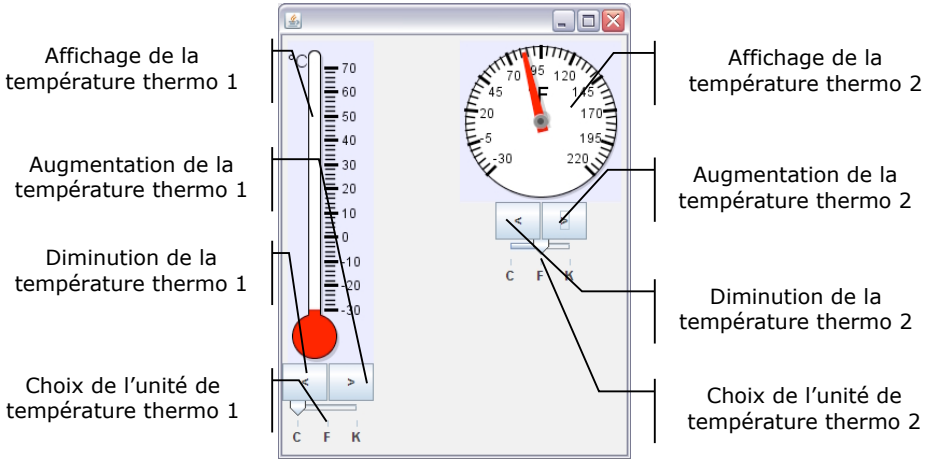
- Utiliser des *Interfaces* et des *classes abstraites*
  - **Abstraction des implantations** de chaque module
  - Interfaces: contrats que doivent respecter les modules pour assurer leur **interopérabilité**
- Remarque:

Interface Utilisateur ≠ Interface en O-O-P

Exemple 2: le thermomètre v2

- Réaliser une application interactive simulant **2 thermomètres** sur lesquels l'utilisateur peut agir pour contrôler la température
- L'application fournira:
  - Un affichage de la température courante mesurée par les thermomètre en °C ou en °K ou en F sous la forme:
    - D'un **thermomètre à mercure** pour le 1er thermomètre
    - D'un **cadran à aiguille** pour le 2nd thermomètre
  - Des contrôles permettant à l'utilisateur de diminuer et augmenter la température courante **de chaque** thermomètre
  - Un contrôle permettant de choisir l'unité d'affichage de la température pour **chaque vue**

Thermomètre v2



Thermomètre v2: analyse

- Le modèle
  - Identique au précédent
- Le contrôleur
  - Identique au précédent
  - MAIS doit opérer sur des vues potentiellement différentes
- La vue
  - Doit fournir différentes vue d'un même type de modèle (mais pas forcément la même instance)

Thermomètre v2: analyse

- Solutions pour la vue:
  1. Implanter des vues différentes dans la même classe `VueThermometre.java`

     Solution lourde et peu flexible
  2. Implanter des classes de vues différentes `VueThermometreMercure.java` et `VueThermometreCompteur.java`

     Il faut alors 2 contrôleurs (lourd et peu flexible)
  3. Fournir une **interface** `VueThermometre.java` spécifiant une vue de thérmomètre et spécifiant les prototypes de ces méthodes et l'implanter selon les besoins

     Bonne solution

Thermomètre v2: analyse

- Solutions pour la vue:
  1. Implanter des vues différentes dans la même classe `VueThermometre.java`

     ~~*Solution lourde et peu flexible*~~
  2. Implanter des classes de vues différentes `VueThermometreMercure.java` et `VueThermometreCompteur.java`

     Il faut alors 2 contrôleurs (lourd et peu flexible)
  3. Fournir une **interface** `VueThermometre.java` spécifiant une vue de thérmomètre et spécifiant les prototypes de ces méthodes et l'implanter selon les besoins

     Bonne solution

Thermomètre v2: analyse

- Solutions pour la vue:
  1. Implanter des vues différentes dans la même classe `VueThermometre.java`

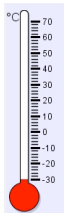
     ~~*Solution lourde et peu flexible*~~
  2. Implanter des classes de vues différentes `VueThermometreMercure.java` et `VueThermometreCompteur.java`

     ~~*Il faut alors 2 contrôleurs (lourd et peu flexible)*~~
  3. Fournir une **interface** `VueThermometre.java` spécifiant une vue de thermomètre et spécifiant les prototypes de ces méthodes et l'implanter selon les besoins

     Bonne solution

Thermomètre v2: analyse

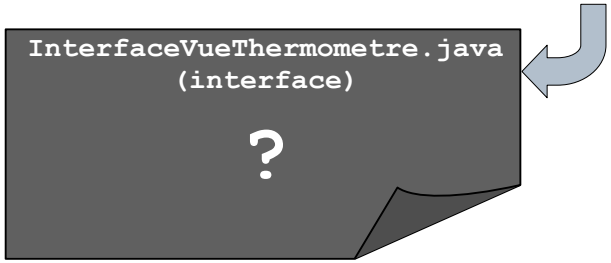
- Tous les types de vues
  - **Affichent** la température courante sous une forme indéterminée
  - **Adaptent** leur affichage à l'unité courante



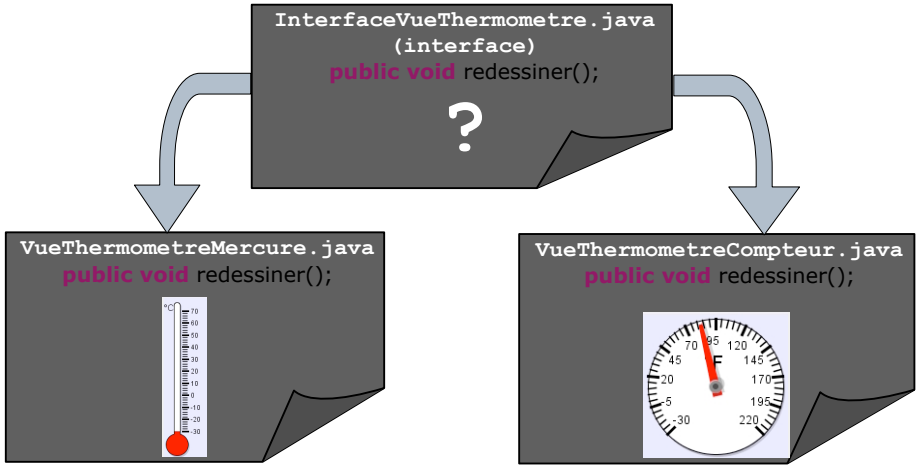
Thermomètre v2: analyse

Methodes requises:

```
public void redessiner();
public void reglerUnite(Unite unite);
```



Thermomètre v2: analyse



Thermomètre v2: réalisation

- Les vue(s)

```
package thermometre.vues;

public interface InterfaceVueThermometre {

 //...

 /***
 * Régle l'unité d'affichage du thermometre.
 *
 * @param unite l'unité désirée.
 */
 public void reglerUnite(Unite unite);

 /**
 * Methode pour notifier à la vue de se redessiner
 */
 public void redessiner();

}
```

Thermomètre v2: réalisation

- La vue 'Mercure' `VueThermometreMercure.java`

```
package thermometre.vues;
public class VueThermometreMercure implements InterfaceVueThermometre {
 //...
 protected ModeleThermometre _modele = null;
 protected Unite _unite = Unite.CELSIUS;

 public VueThermometreMercure(ModeleThermometre modele) {
  this._modele = modele;
 }

 public void reglerUnite(Unite unite) {
   this._unite = unite;
 }

 public void redessiner (){
        double tempCourante = 0;
        switch (_unite) {
                              case CELSIUS: tempCourante = _modele.temperatureEnCelsius();
                                            break;
                              case FAHRENHEIT: tempCourante = _modele.temperatureEnFarenheit();
                                            break;
                              case KELVIN: tempCourante = _modele.temperatureEnKelvin();
        }
        //Opérations spécifiques de dessin de la vue Mercure...
}
```


Thermomètre v2: réalisation

- La vue 'Compteur' `VueThermometreCompteur.java`

```
package thermometre.vues;
public class VueThermometreCompteur implements InterfaceVueThermometre {
 //...
 protected ModeleThermometre _modele = null;
 protected Unite _unite = Unite.CELSIUS;

 public VueThermometreCompteur(ModeleThermometre modele) {
  this._modele = modele;
 }

 public void reglerUnite(Unite unite) {
   this._unite = unite;
 }

 public void redessiner (){
        double tempCourante = 0;
        switch (_unite) {
                              case CELSIUS: tempCourante = _modele.temperatureEnCelsius();
                                            break;
                              case FAHRENHEIT: tempCourante = _modele.temperatureEnFarenheit();
                                            break;
                              case KELVIN: tempCourante = _modele.temperatureEnKelvin();
        }
        //Opérations spécifiques de dessin de la vue Compteur...
}
```


Thermomètre v2: réalisation

- Le modèle: identique au précédent
- Le contrôleur: identique au précédent mais référence vers `InterfaceVueThermometre`

Thermomètre v2: réalisation

- Le contrôleur

```
package thermometre.controleurs;

public class ControleurThermometre {
 private ModeleThermometre _modele;//le modèle à contrôler
 private InterfaceVueThermometre _vue;//la vue pour représenter le modèle

 JButton _pButton = new JButton(">");//le bouton pour augmenter la température
 JButton _mButton = new JButton("<");//le bouton pour diminuer la température
 JSlider _mSlider = new JSlider(0, 20);//le slider pour choisir l'unité

 public ControleurThermometre (ModeleThermometre modele, InterfaceVueThermometre vue){
     _modele = modele;
     _vue = vue;
     //placement des contrôles de l'IHM
     //...
 }

//Suite idem contrôleur précédent...
```

Thermomètre v2: réalisation

- L'application

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

public class AppliThermometreSimple {

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

                //Creation des modèles de thermomètre
                ModeleThermometre modele1 = new ModeleThermometre(243.15);
                ModeleThermometre modele2 = new ModeleThermometre();

                //Creation de la vue 'mercure' et de son contrôleur
                InterfaceVueThermometre vue1 = new VueThermometreMercure (modele1);
                ControleurThermometre pt1 = new ControleurThermometre(modele1, vue1);

                //Creation de la vue 'compteur' et de son contrôleur
                InterfaceVueThermometre vue2 = new VueThermometreCompteur (modele2);
                ControleurThermometre pt2 = new ControleurThermometre(modele2, vue2);

                //Ajout des panneaux à la fenêtre et affichage...
                //...
        }
}
```


Commentaires sur cette réalisation de MVC ?

- Validité par rapport au modèle
- Problèmes

Commentaires sur cette réalisation de MVC

- Mécanismes de gestion des événements => *seront traités dans la suite du cours*
- Mécanismes de dessin des vues => *seront traités dans la suite du cours*
- Modularité => *Oui*
- Indépendance Données-Vue-Interaction ? => *Oui*
- Code redondant dans les 2 types de vues => Utilisation d'une classe abstraite intermédiaire.

Commentaires sur cette 2nde réalisation de MVC

- **Réutilisabilité** et **généricité** ? => *Incomplète!*
  - Si l'on veut utiliser un autre modèle ou un autre contrôleur ?
- Par exemple, si l'on veut maintenant:
  - Utiliser un autre modèle de thermomètre avec ce(s) vue(s) et ce contrôleur ?
  - Utiliser une autre vue avec ce modèle et ce contrôleur ?
  - Utiliser un autre contrôleur avec ce modèle et ces vue ?
  - Utiliser une autre vue et un autre contrôleur simultanément avec cette vue, ce modèle et ce contrôleur ? (synchronisation)
  - *Etc.*
- Impossible en l'état car des références entre des objets de chaque module sont typées par leur classe concrète. **On doit encore modifier des parties du code de chaque module.**

⇒ **Valide par rapport au modèle, mais n'en n'exploite pas le pouvoir d'abstraction (indépendance entre données/représentation/contrôle)**

Réalisation de MVC

- Principes des interfaces à étendre à la réalisation du modèle et des contrôleurs:
  - Utiliser une interface `InterfaceModeleThermometre.java` (indépendance de la représentation et du contrôle avec l'implantation du modèle)
  - Idem pour le(s) contrôleur(s) (souvent moins utile car le contrôleur est relativement indépendant du modèle et de la vue)
- Chaque module référence les autres par leur **type apparent (Interface)** et les liens entre modules sont donc indépendants de leurs implantations

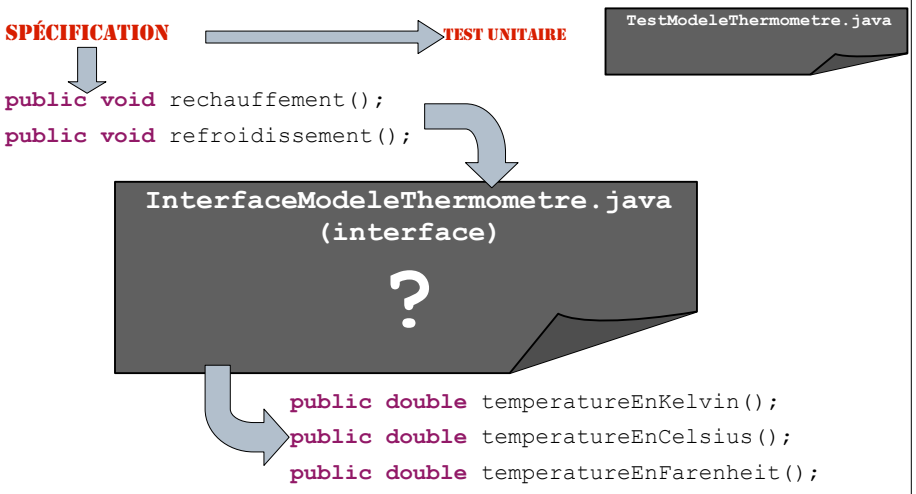
Exemple 3: le thermomètre MVC 'parfait'

- Réaliser une application interactive simulant **2 thermomètres** sur lesquels l'utilisateur peut agir pour contrôler la température
- Les thermomètres devront être de 2 types:
  - Le **thermomètre basique**, précis à 1°C près
  - Le **thermomètre « spatial »**, précis à 0.0001°C près
- L'application fournira:
  - Un affichage de la température courante mesurée par les thermomètre en °C ou en °K ou en F sous la forme:
    - D'un **thermomètre à mercure** pour le 1er thermomètre
    - D'un **cadran à aiguille** pour le 2nd thermomètre
  - Des contrôles permettant à l'utilisateur de diminuer et augmenter la température courante **de chaque** thermomètre
  - Un contrôle permettant de choisir l'unité d'affichage de la température pour **chaque vue**

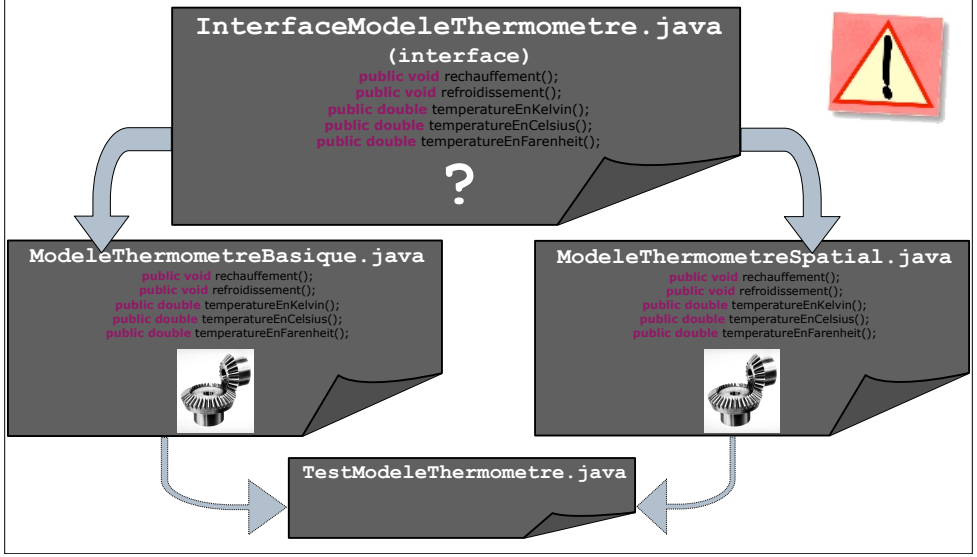
Thermomètre MVC: analyse

- Le modèle (cf TP1b)
  - Données et traitements réalisés:
    - *Température courante*
    - *Maintient de l'état* de la température courante
    - *Conversions* de la température en différentes unités
  - Fonctionnalités exposées:
    - **Augmenter** la température de 1° (C ou K)
    - **Diminuer** la température de 1° (C ou K)
    - **Donner** la température en °C, °K ou F

Thermomètre MVC: analyse



Thermomètre MVC: analyse



Thermomètre MVC: analyse

- Le contrôleur: identique au précédent mais référence vers `InterfaceModeleThermometre`

- L(es) vue(s): identique(s) au(x) précédente(s) mais référence vers `InterfaceModeleThermometre`

Thermomètre MVC: réalisation

- Le modèle

```
package thermometre.modele;
public interface InterfaceModeleThermometre {
  /**
  * indique la temperature en Kelvin
  * @return la temperature en Kelvin courante que le thermomètre indique
  */
  public double temperatureEnKelvin();
  /**
  * indique la temperature en Celsius
  * @return la temperature en Celsius courante que le thermomètre indique
  */
  public double temperatureEnCelsius();
  /**
  * indique la temperature en Farenheit
  * @return la temperature en Farenheit courante que le thermomètre indique
  */
  public double temperatureEnFarenheit();
  /**
  * permet au capteur d'envoyer l'information d'un rechauffement ambiant d'un degré de la température
  *
  */
  public void rechauffement();
  /**
  * permet au capteur d'envoyer l'information d'un refroidissement ambiant d'un degré de la température
  *
  */
  public void refroidissement();
}
```


Thermomètre MVC: réalisation

- Le modèle thermomètre basique

```
package thermometre.modele;

public class ModeleThermometreBasique implements InterfaceModeleThermometre {
  final static protected double zeroAbsoluCelcius=-273.15;
  private double _temperature;//la temperature en Kelvin


  /**
  * Construit un thermometre avec comme temperature initale 0 degre celcius
  */
  public ModeleThermometreBasique(){
     _temperature=0-zeroAbsoluCelcius;
  }

  public double temperatureEnKelvin(){
     //opérations propres à ce modèle...
  }
  //Implantation des autres méthodes définies dans l'interface...
}
```

Thermomètre MVC: réalisation

- Le modèle thermomètre spatial

```
package thermometre.modele;

public class ModeleThermometreSpatial implements InterfaceModeleThermometre {
  final static protected double zeroAbsoluCelcius=-273.15;
  private double _temperature;//la temperature en Kelvin


  /**
  * Construit un thermometre avec comme temperature initale 0 degre celcius
  */
  public ModeleThermometreSpatial(){
     _temperature=0-zeroAbsoluCelcius;
  }

  public double temperatureEnKelvin(){
     //opérations propres à ce modèle...
  }
  //Implantation des autres méthodes définies dans l'interface...
}
```


Thermomètre MVC: réalisation

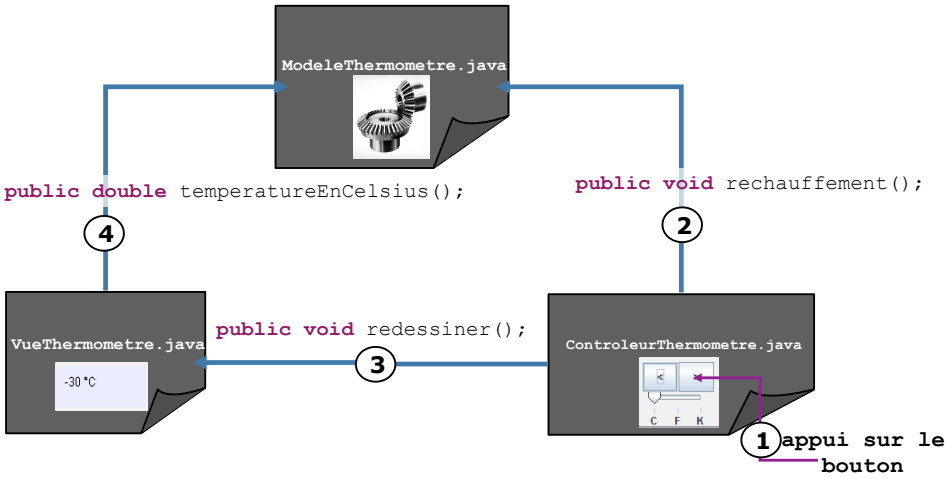
- L'application

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

public class AppliThermometreSimple {

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

                //Creation des modèles de thermomètre
                InterfaceModeleThermometre modele1 = new ModeleThermometreBasique(243.15);
                InterfaceModeleThermometre modele2 = new ModeleThermometreSpatial();

                //Creation de la vue 'mercure' et de son contrôleur
                InterfaceVueThermometre vue1 = new VueThermometreMercure (modele1);
                ControleurThermometre pt1 = new ControleurThermometre(modele1, vue1);

                //Creation de la vue 'compteur' et de son contrôleur
                InterfaceVueThermometre vue2 = new VueThermometreCompteur (modele2);
                ControleurThermometre pt2 = new ControleurThermometre(modele2, vue2);

                //Ajout des panneaux à la fenêtre et affichage...
                //...
        }
}
```

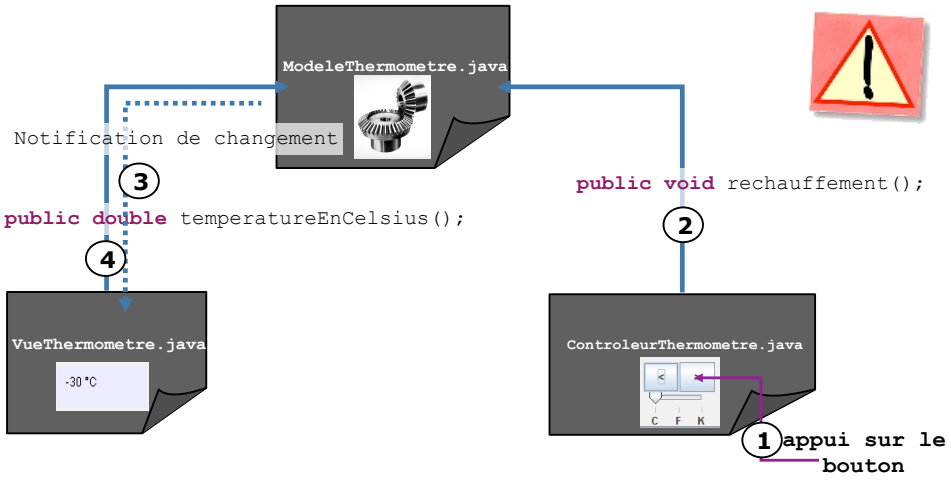

Thermomètre MVC: pour aller plus loin...

- Pour les vues:
  - Permettre à la vue d'être notifiée directement par le modèle de ses mises à jour (observateur)
    - Simplifie la réalisation de MVC
    - Réduit les messages et les réaffichages
    - Permet la synchronisation de plusieurs vues
  - Détails d'implantation dans la suite du cours (*listeners*)

Thermomètre v1: analyse (rappel)



Thermomètre



Ce qu'il faut retenir

- Importance de l'**analyse** du problème en « **pensant MVC** »

- **Abstraire** le schéma de l'application (interfaces et classes abstraites)

-**Indépendance** des modules

MVC: Conclusion et bilan

- Un modèle pour:
  - **Analyser** un « problème »
  - **Structurer** une application interactive
  - **Implanter** un système de manière modulaire, flexible et réutilisable
- Garantit et facilite:
  - L'indépendance **front-office** (IHM) – **back-office** (données et traitements)
  - La **maintenance** et la **réutilisation** de modules
- Mais ce n'est pas une solution universelle...

JAVA

Applications interactives

-

Programmation d'interfaces graphiques

Stéphane HUOT
Dpt. Informatique
iUT ORSAY

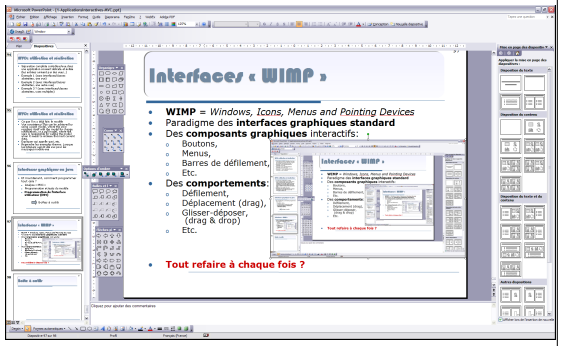
Interfaces graphiques en java

- 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

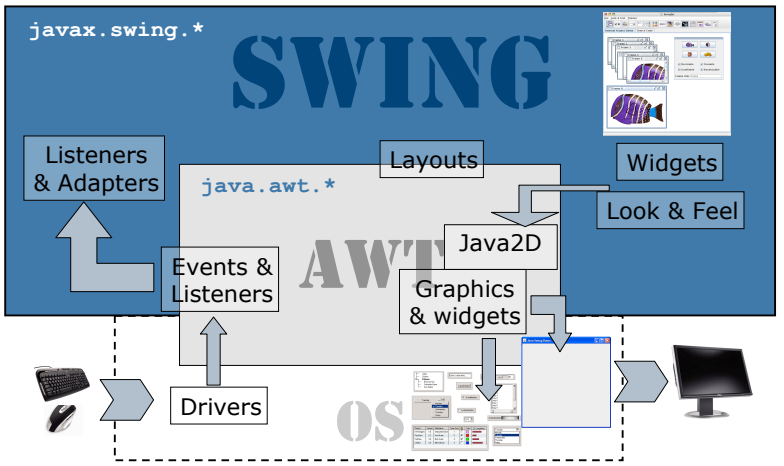
Java AWT

- 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

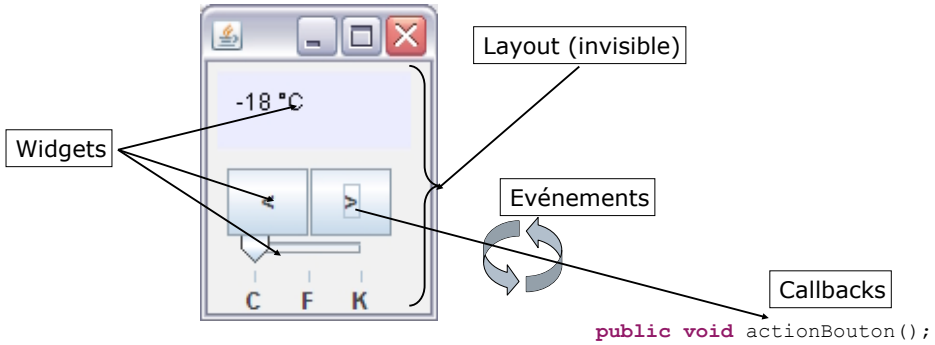
AWT, Swing, etc.



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'



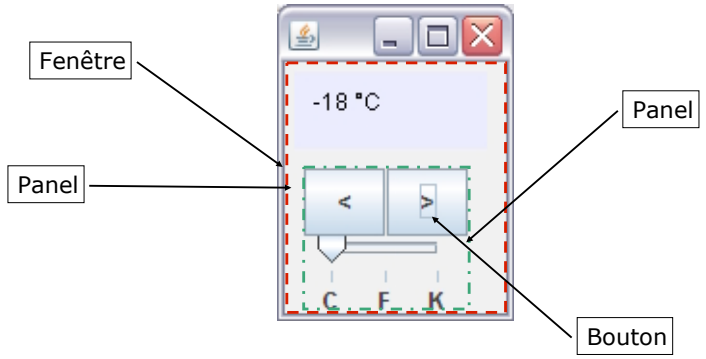
Swing: composants de base

- 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

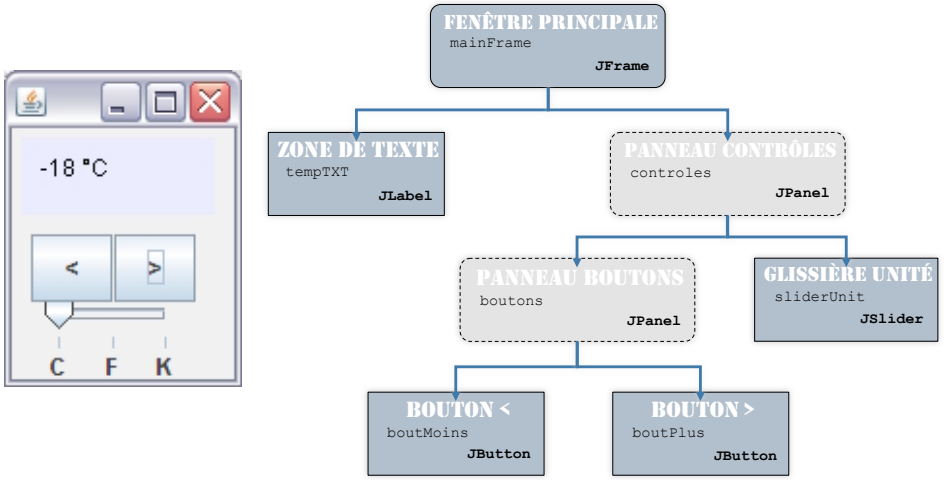
- Pour apparaître à l'écran, les composants doivent appartenir à une hiérarchie de containers
- Un composant ne peut appartenir qu'à un seul container
- La racine d'une hiérarchie de container est un container de haut-niveau:

  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



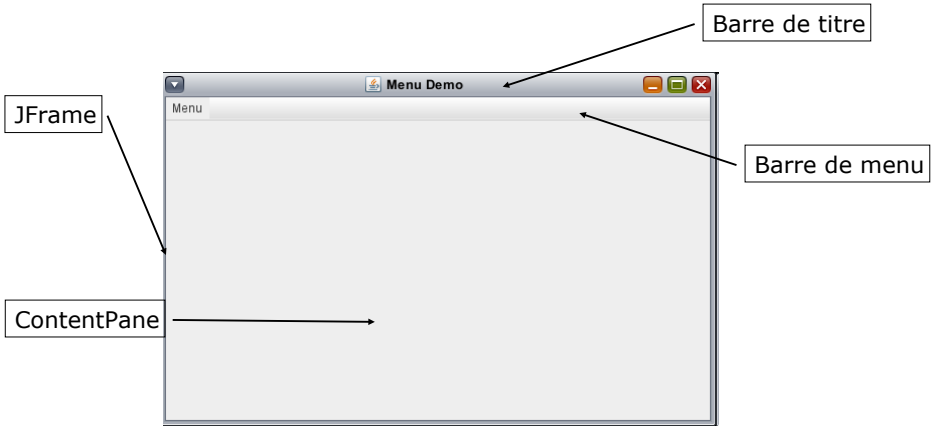
Fenêtre: JFrame

- 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 ptl = new ControleurThermometre(modele, vue);

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

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

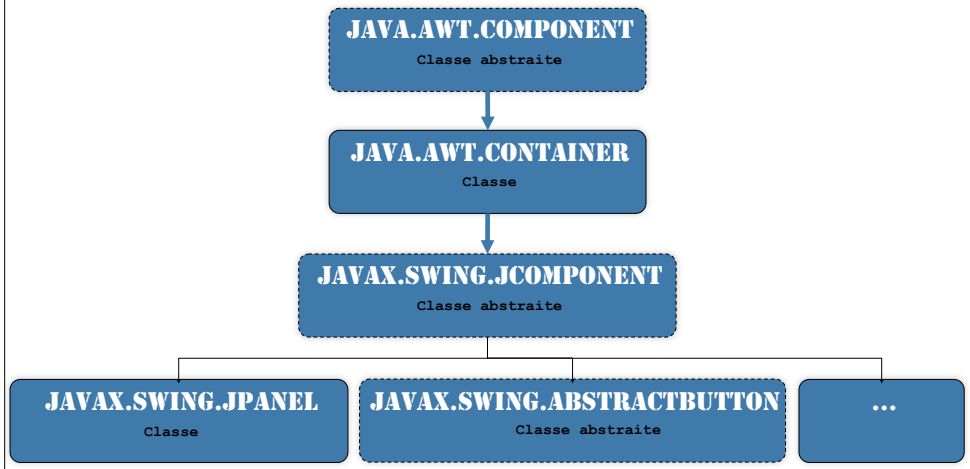

JFrame, un peu plus

- 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`…

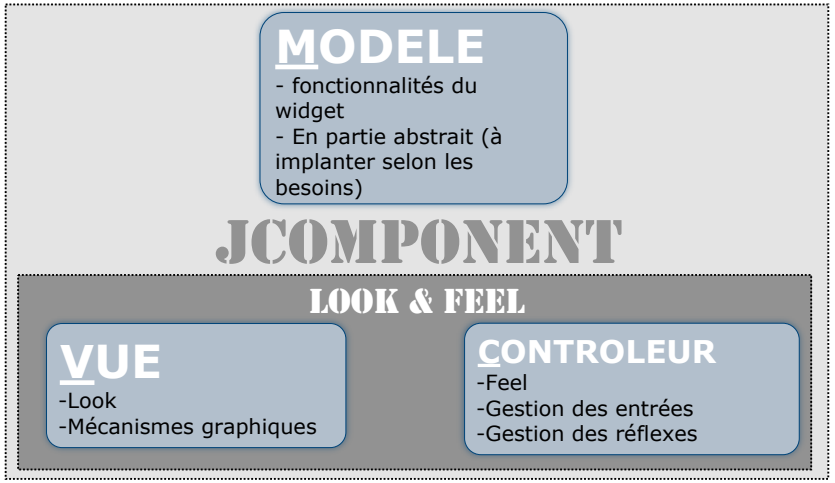
JComponent, bases

- 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)

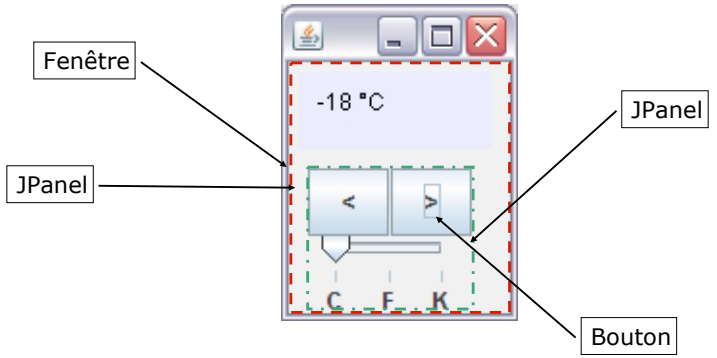
JComponent, méthodes de base (2)

- 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 géneral)
- 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

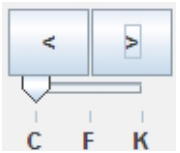
- Par défaut:
  - Ne dessine que son fond (*background*) et ses fils
  - N'a pas de bordure graphique
  - Est opaque
  - Adapte sa taille selon ses fils et son *'Layout'*
- Possibilités:
  - Changer le *'Layout'*
  - Changer les couleurs
  - Rendre transparent
  - Ajouter une bordure
  - …

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*

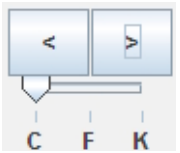
JButton, bases

- 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);
```

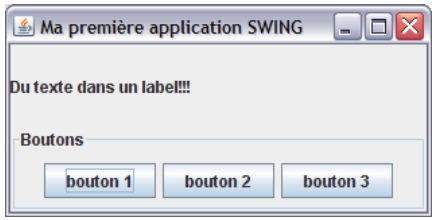
- Etc.
- Voir la Javadoc de `JButton`...

Autres widgets...

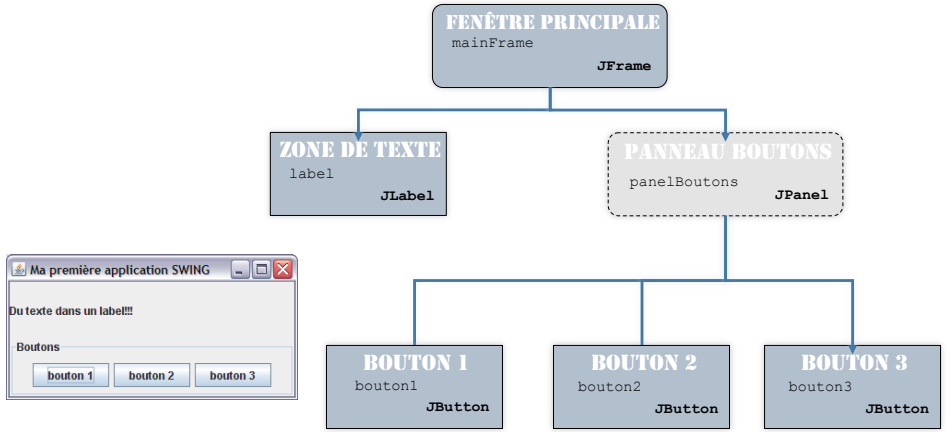
- Texte:
  - JLabel
  - JTextField
  - JTextArea
  - ...
- Listes et arbres
  - JList
  - JTree
  - JComboBox
  - JMenu/JPopupMenu
- Choix
  - CheckBox
  - JRadioButton
- Dialogues
  - JDialog
  - JFileChooser
  - JColorChooser
  - ...
- ...

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.

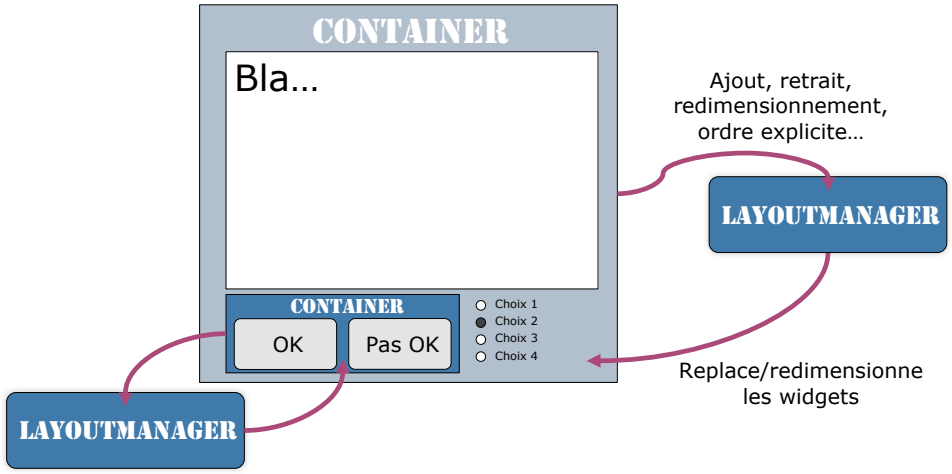
Un problème complexe

- 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

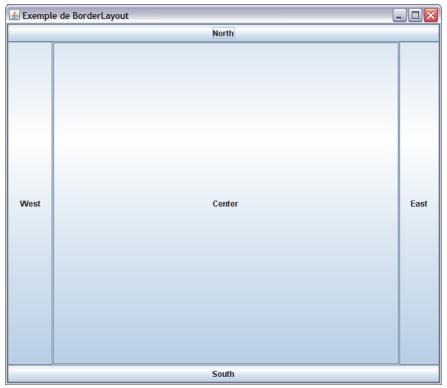
- `BorderLayout (AWT)`:
  Divise le container en 5 zones (Centre, Nord, Sud, Est et Ouest)
- `BoxLayout (Swing)`:
  Alignement des composants (axe X, axe Y, Line, Page)
- `FlowLayout (AWT)`:
  Positionnement en flux selon la place disponible (Centré, Gauche ou Droite)
- `GridLayout (AWT)`:
  Positionnement 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

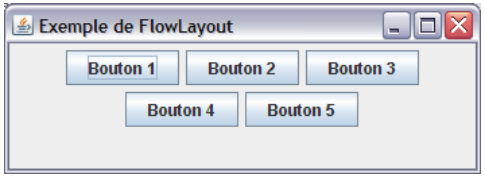
```
package gui;

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

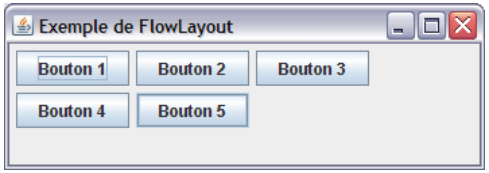
- Layout par défaut des JPanel
- Arrange les widgets horizontalement selon un flot directionnel
- Garde la taille définie des widgets et retourne à la ligne s'il n'y a pas assez de place
- L'alignement est déterminé par la propriété `Alignement` (`setAlignement` et `getAlignement`):
  - *CENTER*: lignes 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

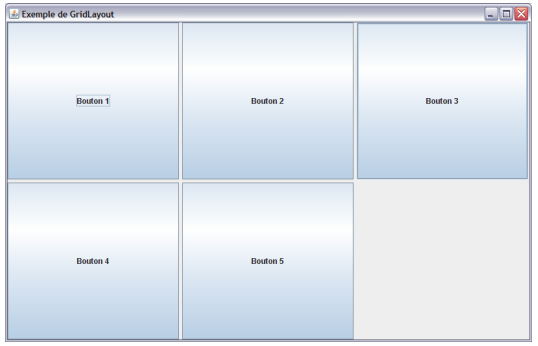
FlowLayout. Code

```
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

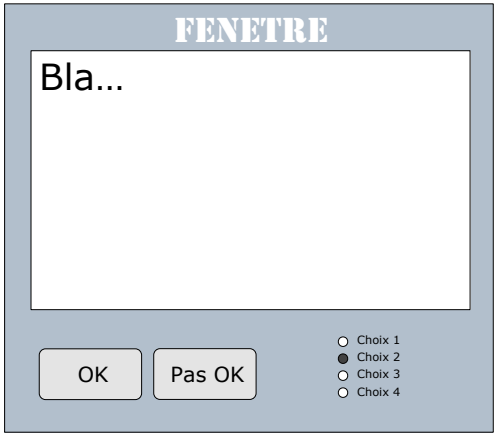


Démonstration de redimensionnement

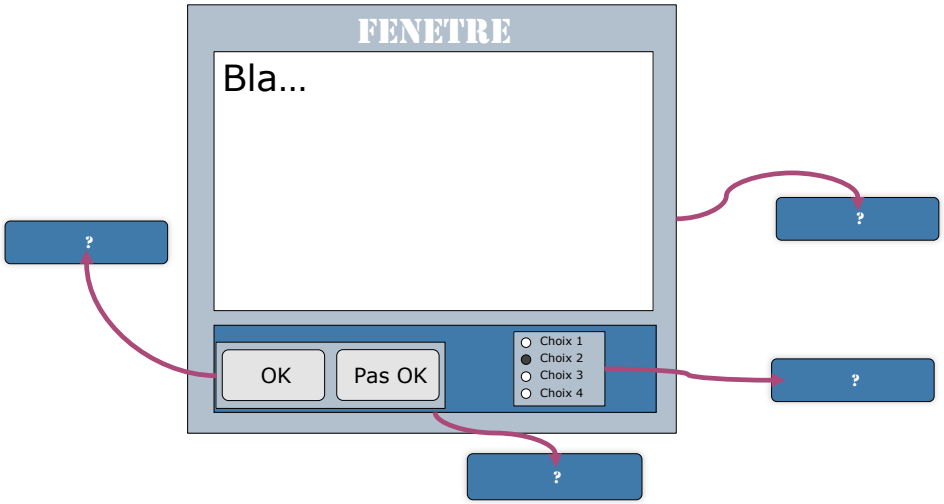
GridLayout. Code

```
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);
	}
}
```

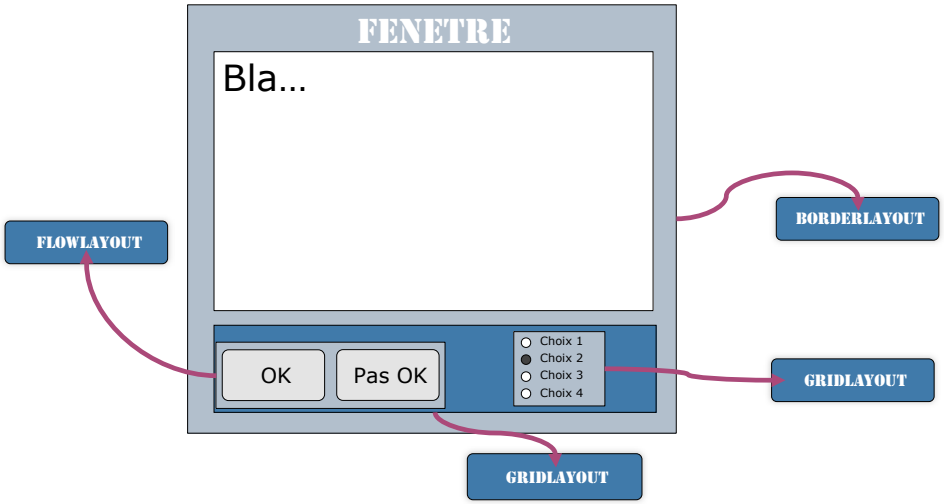

Application plus complète



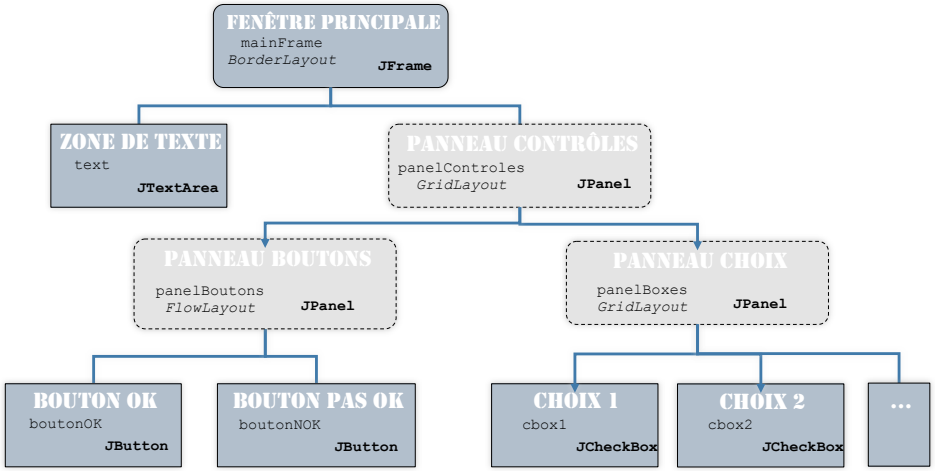
Application plus complète



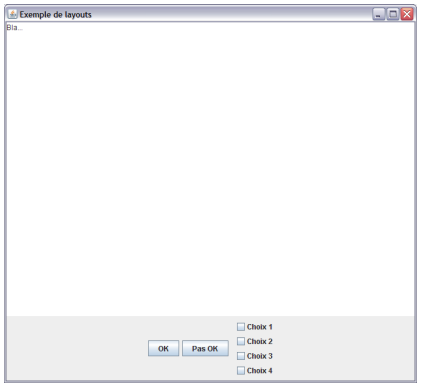
Application plus complète



Arbre de widgets

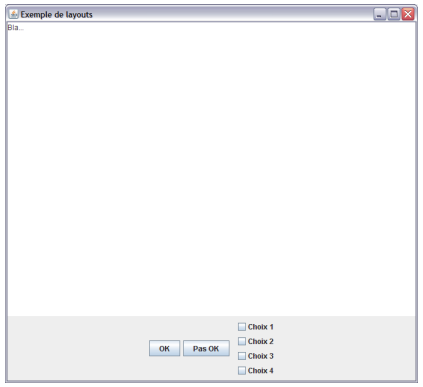


Application plus complète



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**

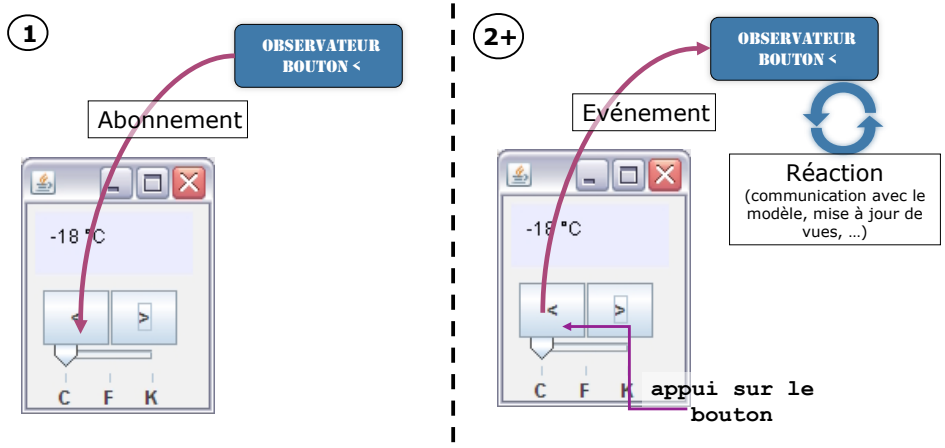
Listeners

- 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)

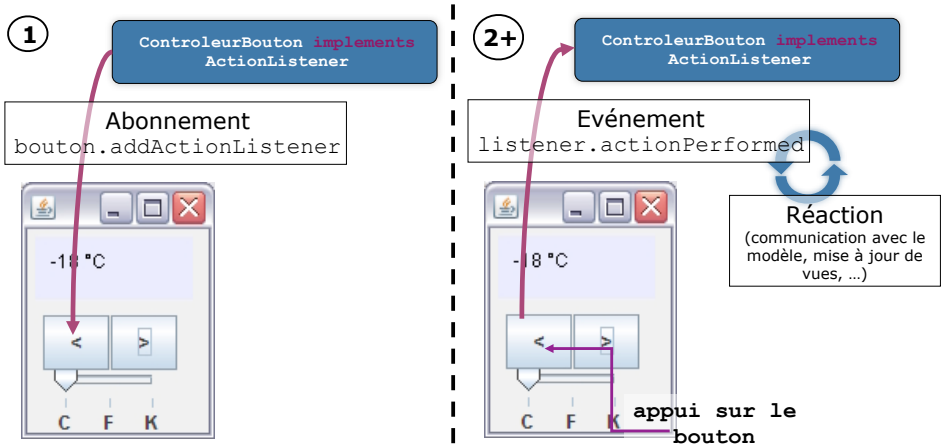
Exemple



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

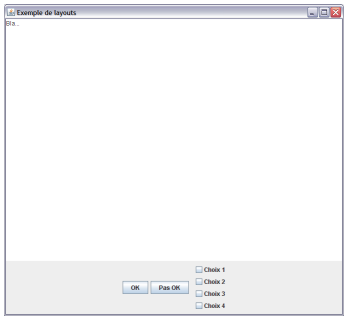
- Les interfaces décrivent des Listeners avec une 'sémantique' différente, selon les événements écoutés:
  - **`ActionListener`**: écouter des actions avec `actionPerformed(ActionEvent e)`
  - **`ChangeListener`**: écouter des changements d'état avec `stateChanged(ChangeEvent e)`
  - **`MouseMotionListener`**: 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

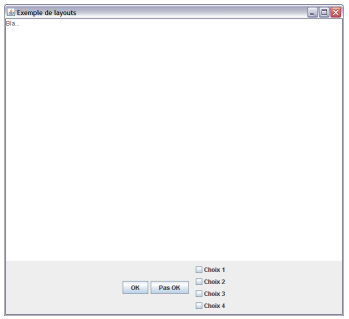
Code de l'action à réaliser

Exemple: dans le code du constructeur de la vue

- Ajout d'un Listener au bouton OK:

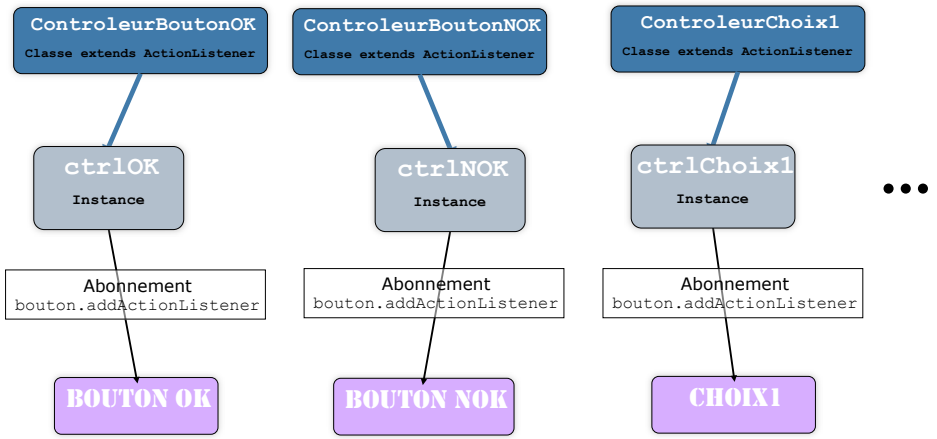
```
//Création de la zone de texte
JTextArea text = new JTextArea("Bla...");
//…
//Ajout des boutons
buttonOK = new JButton("OK");
panelBoutons.add(buttonOK);
//Ajout d'un listener au bouton OK
buttonOK.addActionListener(new ControleurBoutonOK
    (text));
//etc.
```


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

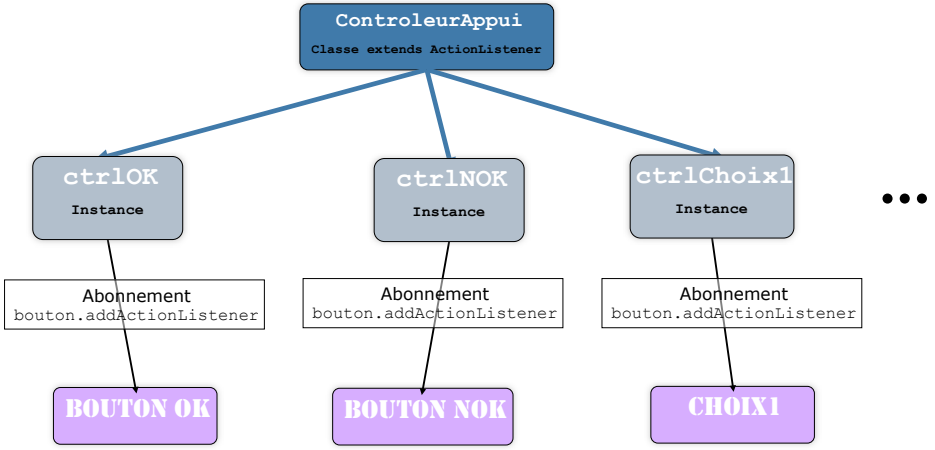
  - Créer une seule classe de Listener qui effectue la même tâche, avec des widgets différents (avec une instance de Listener par widget)

➡ 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, …)

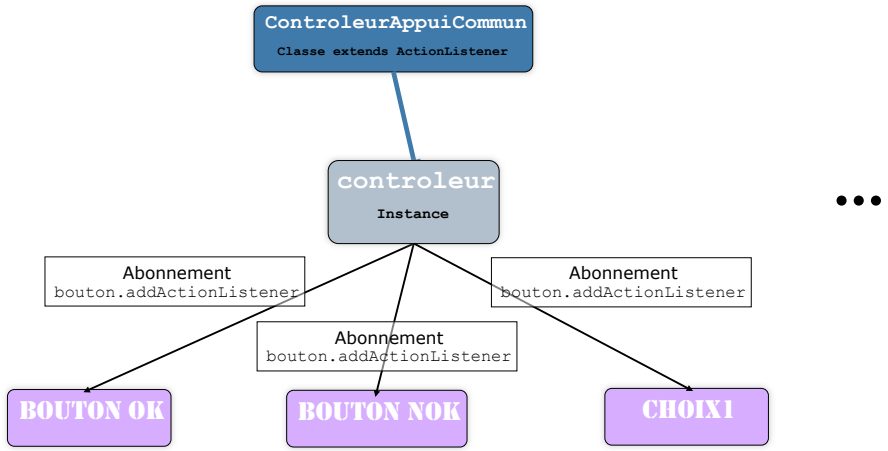
    LOURD
  - Créer une seule classe de Listener qui effectue la même tâche, avec des widgets différents (avec une instance de Listener par widget)

    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

Zone de texte en paramètres du constructeur

Code de l'action à réaliser

```
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());
    }
}
```


Exemple 3: dans le code du constructeur de la vue

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

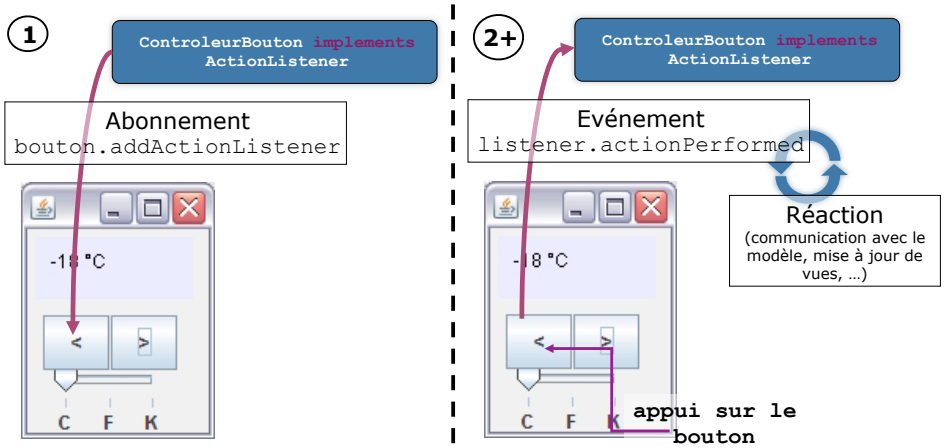
```
//Création de la zone de texte
JTextArea text = new JTextArea("Bla...");
//Création du listener
ControleurAppuiCommun ctrl = new ControleurAppuiCommun(text);
//Ajout des boutons
buttonOK = new JButton("OK");panelBoutons.add(buttonOK);
//Réglage de la propriété actionCommand
buttonOK.setActionCommand(buttonOK.getText());
//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...



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…

JAVA

Applications interactives

-

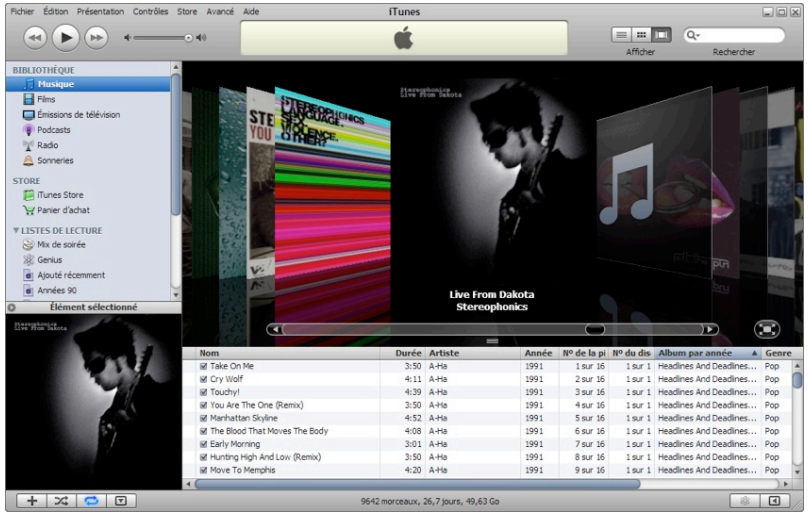
Programmer une application interactive

Stéphane HUOT
Dpt. Informatique
iUT ORSAY

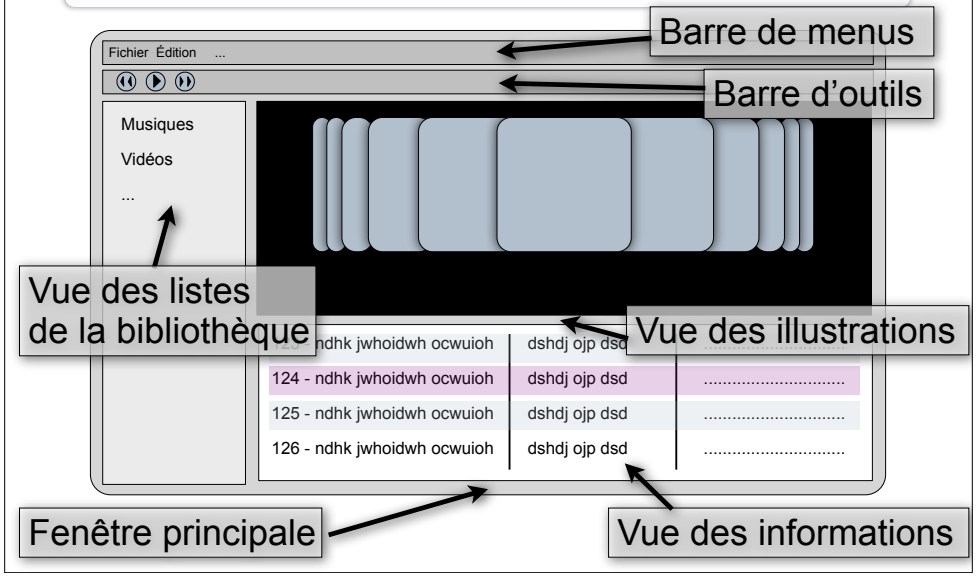
Un exemple: Lecteur multimédia 'tuTunes'

- 'à la iTunes'
- Fonctionnalités:
  - gérer une bibliothèque
    - ajouter
    - enlever
    - parcourir
    - ...
  - gérer les éléments de la bibliothèque
    - lire
    - déplacer
    - ...

Interface



Interface



Analyse, Modèles

- modèle bibliothèque
  `public class ModeleBibliotheque`
- modèle objet multimédia
  `public interface ModeleObjetMultimedia`
  - modèle objet audio
    `public class ModeleObjetAudio implements ModeleObjetMultimedia`
  - modèle objet video
    `public class ModeleObjetVideo implements ModeleObjetMultimedia`
- ...

ModeleBibliotheque (partiel)

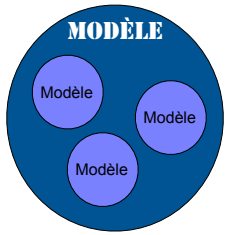
```
public class ModeleBibliotheque {

    private Collection<ModeleObjetMultimedia> objets;

     public ModeleBibliotheque() {
    }

    public boolean ajouterObjet(ModeleObjetMultimedia objet) throws ExceptionAjoutImpossible   {
       return true;
    }

    public boolean retirerObjet(ModeleObjetMultimedia objet) throws ExceptionRetraitImpossible {
        return true;
    }

    public void creerListe() {
    }

    public Collection<String> getNomsListes() {
       return null;
    }
}
```


ModeleObjetMultimedia (partiel)

```
public interface ModeleObjetMultimedia {

 public boolean charger(String chemin) throws ExceptionChargeObjet;

 public void lire() throws ExceptionLectureObjet;

 public void arreterLecture() throws ExceptionLectureObjet;

 public String getNom();

 public String getInfos();

 public String getDuree();

 public Image getImage();

}
```


Bilan Modèles

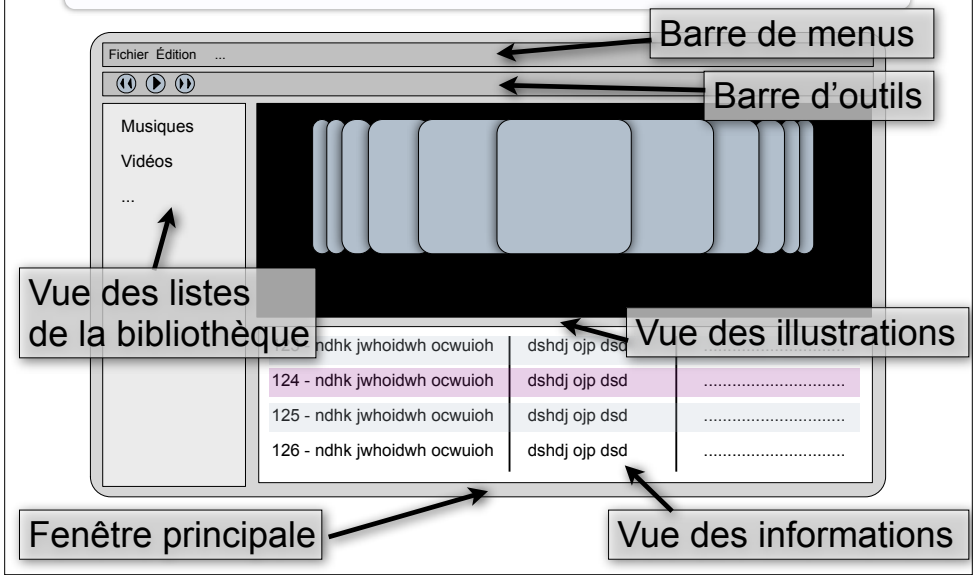
- Dans une 'vraie' application:
  - possibilité d'avoir plusieurs modèles
    - différents types/sources de données
    - différents ensembles de fonctionnalités
  - **LE** modèle = agrégation d'instances de différents modèles



Programmer le(s) modèle (s)

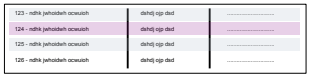
- **Besoin d'explications ???**

Analyse, Vues

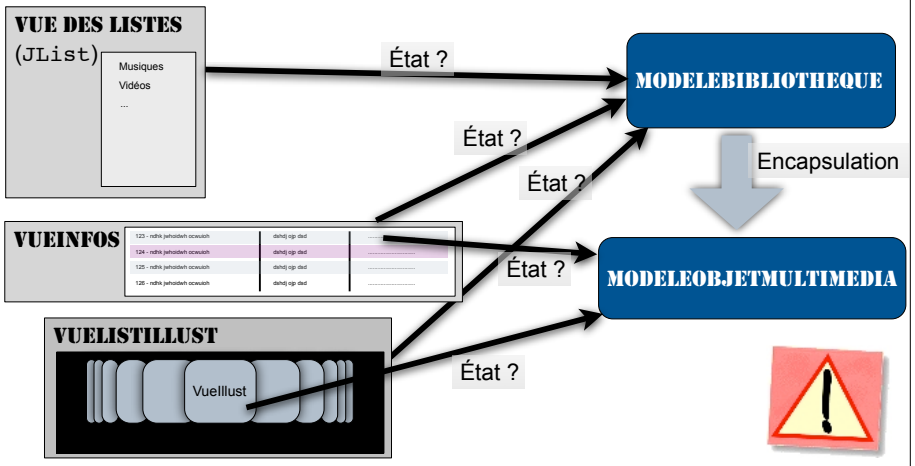


Analyse, Vues

- Vue de l'application
  `public class VueApplication`
- vue des listes de la bibliothèque
  `JList`
- vue des informations des objets
  `public class VueInfos extends JList`
- vue des illustrations des objets
  `public class VueListIllust extends JPanel`
  - vues de l'illustration d'un objet
    `public class VueIllust extends JPanel`
- `...`



Modèles et vues



Programmer la/les vues

- 2 méthodes:
  - ENCAPSULATION
    - agrégation de composants
  - SURCHARGE DE COMPOSANTS EXISTANTS
    - création de composants spécifiques à des besoins
    - utilisation comme des composants 'standard' (encapsulation)

Vue de l'application: encapsulation

- Vue de l'application:
  - regroupe les vues des différentes parties
  - une classe qui:
    - contient les widgets/vues des composants de l'application
    - fournit des méthodes utiles:
      - ajout de contrôleurs
      - visible ou non
      - re-affichage
      - accès à certains composants (widgets)
      - ...

VueApplication

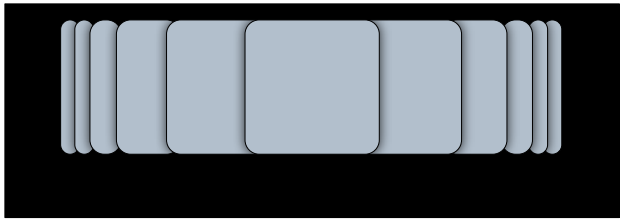
```
public class VueApplication {

 private ModeleBibliotheque biblio;
 private JFrame mainFrame;
 private JList listesLecture;
 private VueListIllust vueListeIllustartions;

 public VueApplication(ModeleBibliotheque biblio) {
     this.biblio = biblio;

     mainFrame = new JFrame();
     //création et ajout des autres composants
     //etc.. voir cours précédents
     //instanciation et ajout des contrôleurs
 }
 //méthodes utiles...
 public void setVisible(boolean b) {
     mainFrame.setVisible(b);
 }
}
```


Vue: surcharge

- Lorsque les widgets par défaut ne conviennent pas ?
- Surcharge pour:
  - changer les comportements
  - changer l'apparence graphique
  - définir de nouveaux widgets

VueListIllust

- Existe en Swing ?



- Non...
- Création d'un nouveau widget...

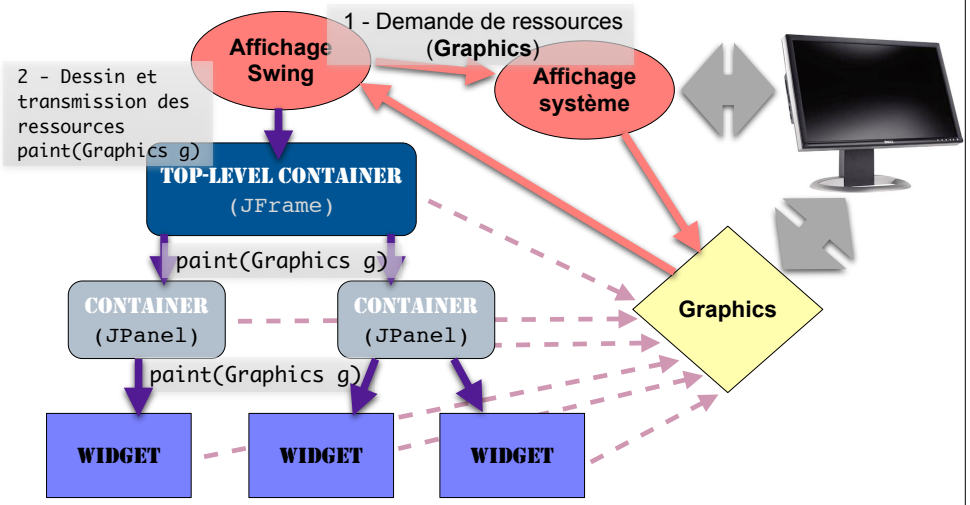
Créer un nouveau widget

- Un nouveau bouton ?
  - implanter `AbstractButton`
- Un widget non prévu:
  - en général, partir de `JPanel`
  - surcharge de la méthode `paint()` ou `paintComponent()`
  - surcharge et création de méthodes de comportement
  - gestion de listeners

les méthodes paint() et paintComponent() des widgets

- `public void paint(Graphics g)`
  - méthode principale de dessin du widget
  - appelée par le gestionnaire d'affichage de Swing
  - `Graphics g` : environnement graphique
  - appelle diverses méthodes de dessin du widget dont `public void paintComponent(Graphics g)`
    - chargé du dessin du widget proprement dit

Système d'affichage de Swing



paint ou paintComponent ?

- `public void paintComponent(Graphics g)`
  - ne redéfinir que le dessin du widget (fond, texte, intérieur)
  - les bords et les éventuels fils seront encore dessinés
- `public void paint(Graphics g)`
  - redéfinir tous les graphismes
  - gérer l'affichage des fils et des bords (si besoin)
- **Essayez...**

Environnement graphique

- Les classes `Graphics` et `Graphics2D`
- paramètre des méthodes de dessin des widgets (`paint`, etc.)
- Environnement graphique fournissant des primitives de dessin:
  - `drawLine`, `drawEllipse`, `drawRectangle`, `drawImage`, `fillEllipse`, ...
  - changement de couleurs, des propriétés de dessin (pens, brushes, ...)
  - **voir javadoc**
- Le dessin est fait en coordonnées locale du widget, dans sa zone
- Swing fournit des objets `Graphics2D` (plus complets)

Exemple

```
public class VueListIllust extends JPanel {
  //...

  public void paint(Graphics g) {
     Graphics2D g2 = (Graphics2D)g;
     //on peint le fond en noir
     g2.setBackground(Color.BLACK);
     g2.fillRect(0, 0, this.getWidth(), this.getHeight());
     //on dessine un rectangle arrondit pour chaque illustration
     //que l'on a obtenu à partir du modèle
     g2.setColor(Color.LIGHT_GRAY);
     for (int i = 0; i < illustrations.size(); i++) {
        Illustration ill = illustrations.get(i);
        g2.fillRoundRect(ill.x, ill.y, ill.width,
                         ill.height, ill.arcWidth, ill.arcHeight);
     }
  }
}
```


Suite et pratique...

- ... dans le TP7

Analyse, Contrôleurs

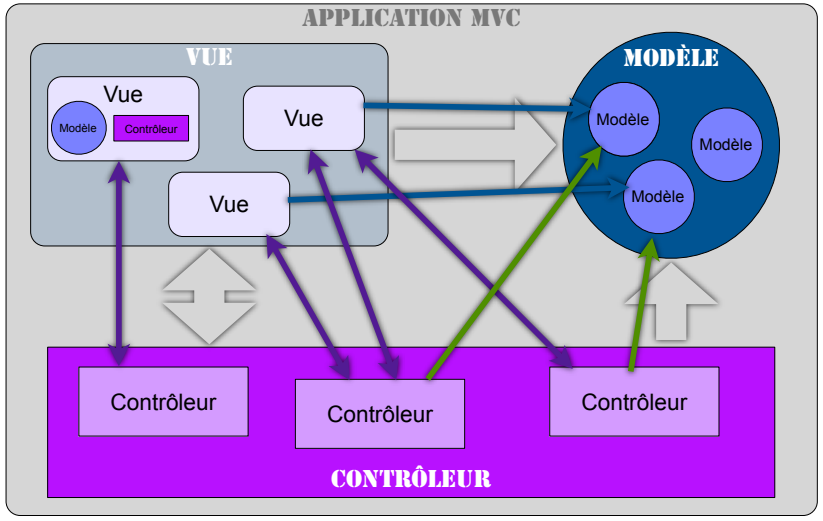
- Un contrôleur par tâche/groupe de tâche ou par widget/groupe de widget
- Ici, les tâches sont groupées par vue:
  - *barre de menus*: fonctionnalités de l'application
  - *barre d'outils*: fonctionnalités de lecture
  - *vue des listes*: gestion des listes de la bibliothèque
  - *vue des illustrations*: choix des objets
  - *vue des informations*: choix et gestion des objets
- Un contrôleur par vue...

Contrôleurs et modèles

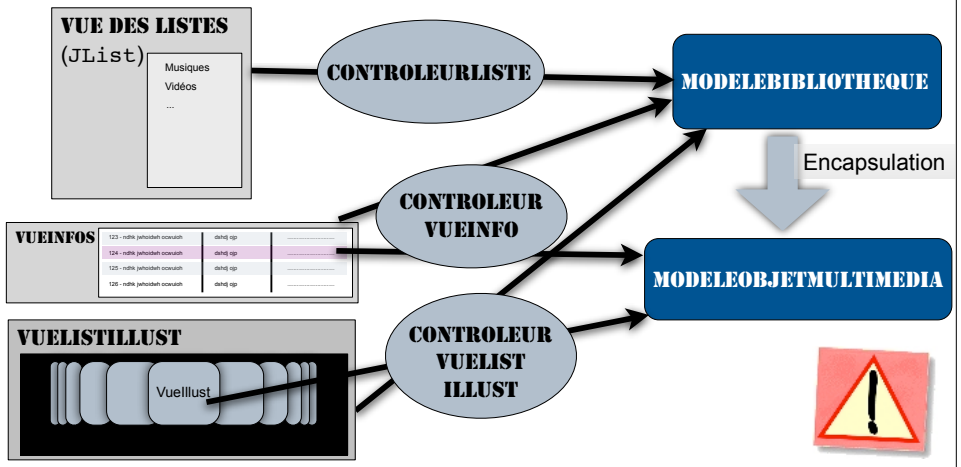
- Liens entre les contrôleur et les modèles ?
- Jusqu'à maintenant: contrôleur(s) liés à un seul modèle
- Ici:
  - contrôleurs liés à un modèle global (*bibliothèque*)
  - le modèle global donne accès à des modèles de plus bas-niveau (*objets multimédia*)

Imbrication/assemblage de composants MVC

Application MVC



Modèles, vues et contrôleurs



Programmer le(s) contrôleur(s)

- Cas simples:
  - action de base au niveau d'un modèle
  - actions directes sur des vues
- Cas plus compliqués:
  - actions sur plusieurs modèles
  - retours à l'utilisateur

Retours utilisateur

- Tâche du/des contrôleur(s)
- **Sélection de vue(s)**:
  - présenter les données
  - demander des entrées
  - afficher les actions/interactions en cours
  - notifier les résultats à l'utilisateur:
    - réussites
    - erreurs

Demander des entrées

- Exemple d'une ouverture de fichier de bibliothèque (contrôleur des menus de l'application)
  - avec la classe `JFileChooser` de Swing

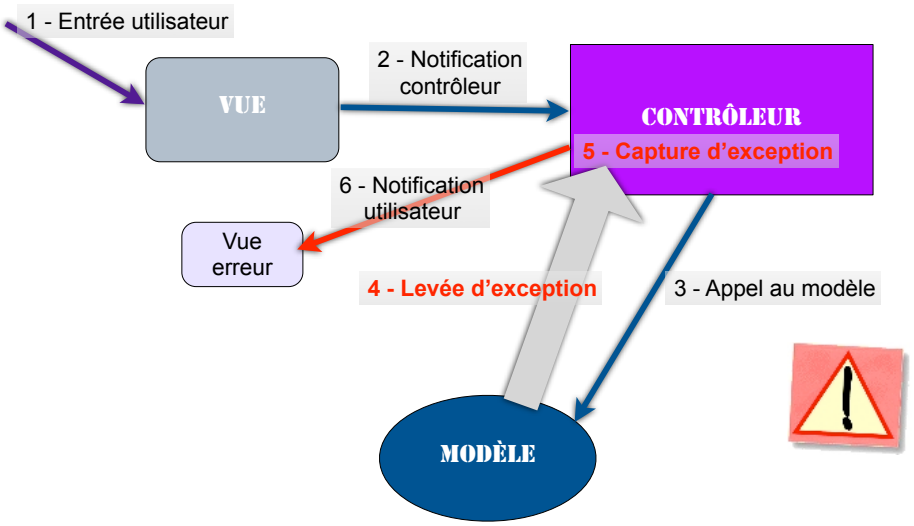
```
JFileChooser fc = new JFileChooser();
if (fc.showOpenDialog(vue.getMainFrame()) == JFileChooser.APPROVE_OPTION) {
   File f = fc.getSelectedFile();
   if (f != null) {
      if (biblio.chargerBibliotheque(f.getAbsolutePath())) {
         vueApplication.mettreAJour();
      }
   }
}
```

- Options: **voir Javadoc...**
- Autres: `JColorChooser` (couleurs), `JDialog` (générique)

Notifier des erreurs/ problèmes

- Faire remonter les informations à l'utilisateur avec:
  - les **valeurs de retour** de certaines méthodes des modèles
  - capture d'**exceptions**
- Analyse de la situation et retour à l'utilisateur si besoin:
  - `JDialog` et `JOptionPane`

Exceptions dans l'application



Capture d'exceptions

- Dans le contrôleur permettant d'ajouter un objet à la bibliothèque (`actionPerformed`) :

```
try {
   biblio.ajouterObjet(null);
  } catch (ExceptionAjoutImpossible e) {
     JOptionPane.showMessageDialog(null,"Impossible
d'ajouter cet objet","Erreur",JOptionPane.ERROR_MESSAGE);
  }
```

- `JOptionPane` fournit plusieurs types de dialogues pré-définis: `showConfirmDialog`, `showInputDialog`, `showMessageDialog`, `showOptionDialog`, avec des variations possibles: **voir Javadoc**

L'application principale

- Instancie un/des modèle(s):
  - ici, création/chargement d'un `ModeleBibliothèque`
- Instancie la/les vue(s) en leur fournissant le modèle
  - la vue instancie les contrôleurs (listeners) et les attache à ses composants
- Rend la vue principale visible

Application principale

```
public static void main(String[] args) {
   ModeleBibliotheque biblio = new ModeleBibliotheque();
   VueApplication vue = new VueApplication(biblio);
   vue.setVisible(true);
}
```

```
(peut être plus compliqué... plusieurs modèles et vues...)
```

```
...
```

Bilan

- L'application du modèle MVC n'est pas si triviale *MAIS*
- Toujours profitable
  - Automatismes
  - Structure
  - Modulaire
- TP7 en exemple

Ce qu'il faut retenir

- **Structure** réelle de l'application **plus complexe** que le modèle idéal

- **Encapsulation** (modèles et vues)

- Retours utilisateur => **gestion de la capture des exceptions**