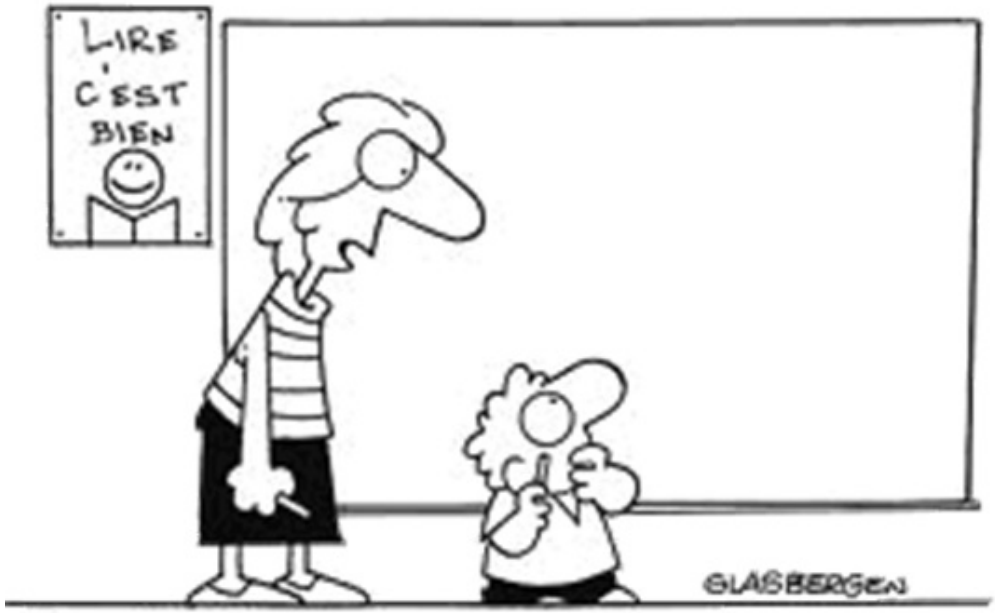
IHM



"Il n'y a pas d'icône à cliquer. C'est un tableau noir."

Au menu

1. Introduction
2. Fenêtres et composants
3. Gestionnaires de placement
4. Événements et écouteurs
5. PAC

Au menu

1 Introduction

2 Fenêtres et composants

3 Gestionnaires de placement

4 Événements et écouteurs

5 PAC

Interface Homme-Machine

- Regroupe tous les moyens et outils mis en œuvre afin qu'un humain puisse contrôler et communiquer avec une machine
- Mobilise des électroniciens et des mécaniciens (pour la partie matérielle), des informaticiens (pour la partie logicielle), mais aussi des psychologues, des sociologues ou encore des ergonomes
- Concevoir une IHM est donc compliquée !

Réalisation d'interfaces graphiques en Java

- Intitulé du cours abusif
- Se limite à la réalisation d'interfaces graphiques en Java sans périphérique atypique
  - Paradigme *WIMP* (*Windows, Icons, Menus and Pointing devices*)
- N'est pas question de la qualité des interfaces :
  - Ergonomie
  - Utilisabilité
  - Adaptabilité
  - Homogénéité
  - Charge cognitive
- Beaucoup de classes et d'interfaces utilisées pendant ce cours
  - Inutile d'apprendre par cœur toutes leurs caractéristiques
  - Prendre l'habitude de consulter la documentation Oracle

API Java pour les interfaces graphiques

- AWT (*Abstract Window Toolkit*)
  - `import java.awt.*;`
- Swing
  - `import javax.swing.*;`
- SWT (*Standard Widget Toolkit*)
  - `import org.eclipse.swt.widgets.*;`
  - Plus récente que les deux autres
  - S'appuie, comme AWT, sur les primitives graphiques natives avec un abandon partiel de la portabilité au profit de l'efficacité
  - Non montrée dans ce cours

AWT vs Swing

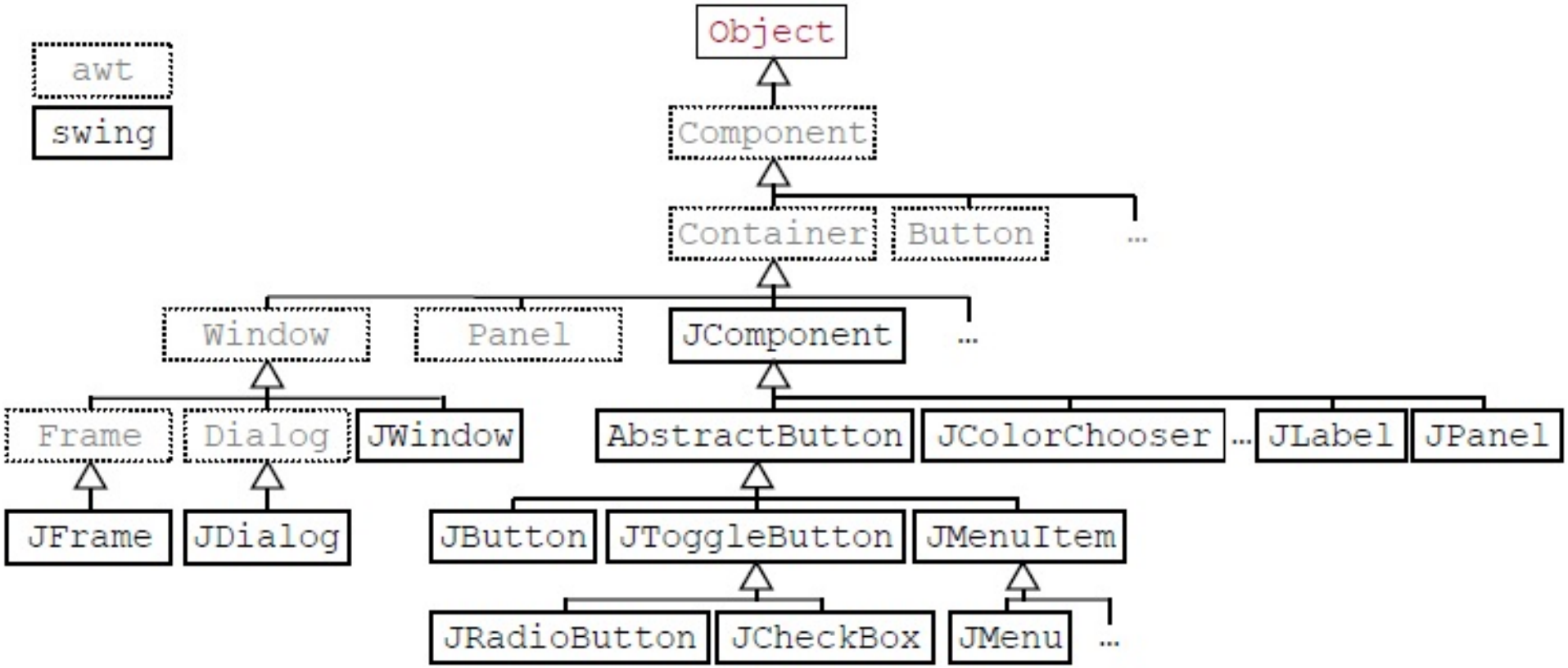
AWT

- Utilise au maximum les systèmes de Gestion d'Interface Utilisateur (GUI)
- Aspect des interfaces similaire à celui des autres interfaces du système
- Ne regroupe que les composants disponibles sur toutes les plateformes
- Difficile de garantir un comportement identique sur toutes les plateformes

Swing

- Minimise l'utilisation du système graphique sous-jacent
- Ne se base que sur des primitives basiques (*e.g.*, le dessin d'une ligne) et sur la gestion primitive des événements
- Aspect des interfaces peut être, selon les systèmes, légèrement différent des autres applications usuelles
- Garantit d'avoir le même comportement sur toutes les plateformes
- Tout nouveau composant est immédiatement disponible sur toutes les plateformes

Hiérarchie des classes



Mieux vaut ne pas mélanger au sein d'une même interface graphique des composants AWT et Swing

Au menu

1 Introduction

2 Fenêtres et composants

3 Gestionnaires de placement

4 Événements et écouteurs

5 PAC

Qu'est-ce qu'une interface graphique ?

Fenêtre

Des composants (*widgets*) qui réagissent *via* des écouteurs (*listeners*)

Qu'est-ce qu'une interface graphique ?

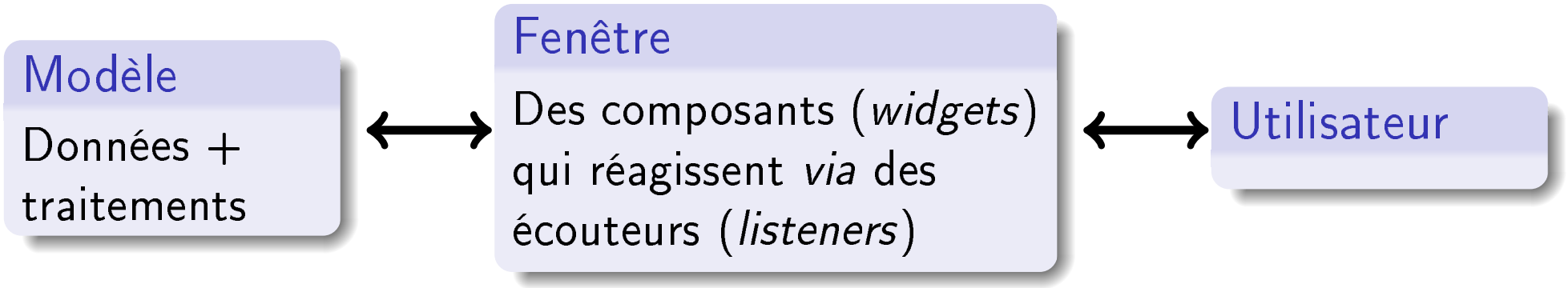
Modèle

Données + traitements

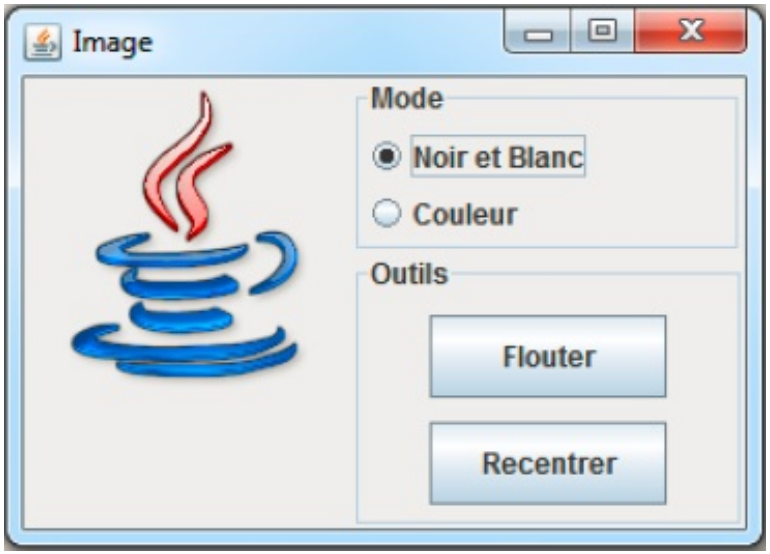
Fenêtre

Des composants (*widgets*) qui réagissent *via* des écouteurs (*listeners*)

Qu'est-ce qu'une interface graphique ?

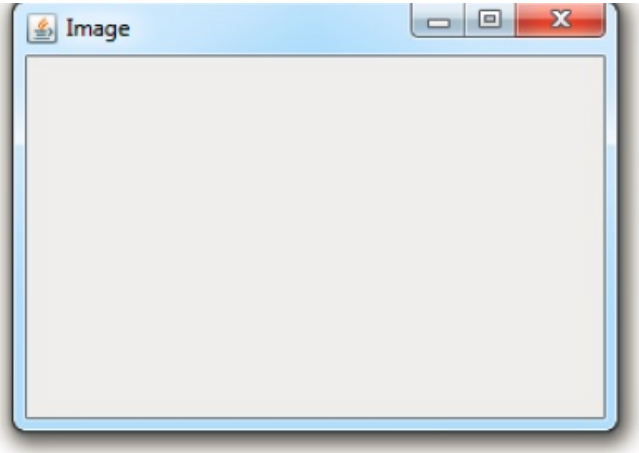


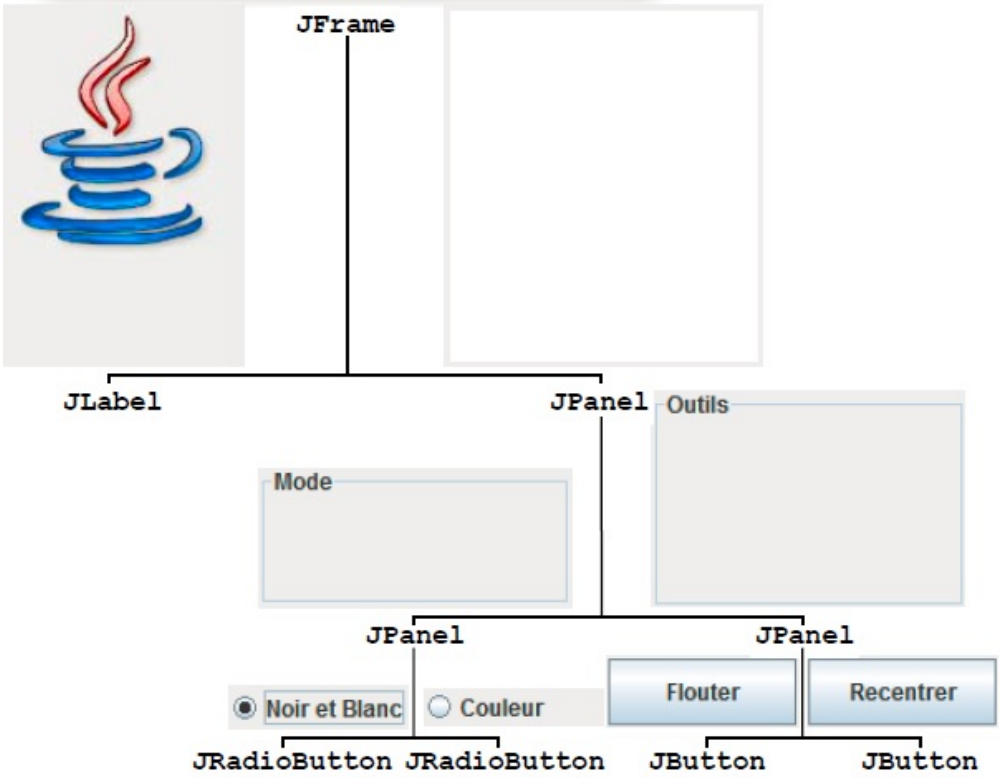
Fenêtre



- Bord (permettant de redimensionner)
- Barre de titre (avec les boutons usuels)
- Barre de menu (optionnelle)
- Conteneur principal (`getContentPane()`)
  - C'est dans celui-ci que les conteneurs intermédiaires et les composants sont ajoutés

Fenêtre





- Un conteneur racine
- Nœuds internes : des conteneurs
- Feuilles : des composants atomiques

Conteneurs racine

- `JApplet`
  - Applet : programme Java qui peut être exécuté par un navigateur internet
- `JWindow`
  - Fenêtre simple sans bordure ni décoration utilisées principalement pour les *splashscreens* affichées lors du lancement d'un programme (comme celle d'Eclipse)
- `JDialog`
  - Boîte de dialogue : fenêtre fille d'une application qui peut être modale, c'est-à-dire sans possibilité d'interagir avec la fenêtre mère tant que la boîte de dialogue est ouverte
- `JFrame`
  - Fenêtre principale d'une interface graphique Swing

Première fenêtre

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

public class FirstFrame extends JFrame {

   public FirstFrame() {
      super("Premiere fenetre");
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      Container contentPane = this.getContentPane();
      ...
   }

   public static void main(String[] args) {
      FirstFrame f = new FirstFrame();
      // f.setSize(600, 800);
      f.pack();
      f.setVisible(true);
   }
}
```

Que retenir ?

- Nos interfaces seront composées d'une instance de type `JFrame`, laquelle possède un conteneur principal accessible par la méthode `getContentPane()`
- Par défaut, une fenêtre n'est pas visible
  - Utiliser `setVisible` pour l'afficher
- Par défaut, la taille d'une fenêtre est nulle
  - Utiliser `setSize` pour la redimmensionner avec des valeurs données
  - Utiliser `pack` pour adapter ses dimensions à son contenu
- Un nouveau processus léger (*thread*) est associé à chaque fenêtre créée et affichée
  - Gère les événements liés à la fenêtre
- Par défaut, le bouton "croix" masque la fenêtre mais ne la ferme (détruit) pas, ce qui explique que le programme continue de s'exécuter même si le *thread* principal est terminé
  - Utiliser `setDefaultCloseOperation` pour changer ce comportement

Boîtes de dialogue

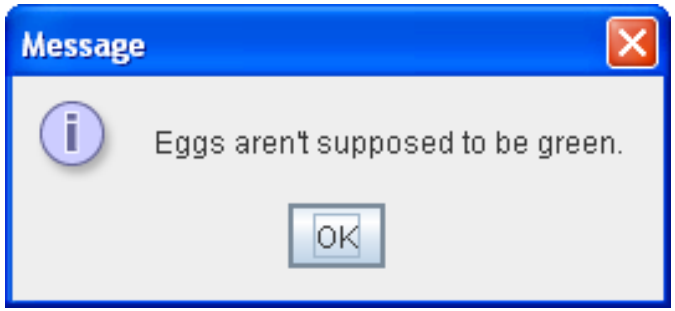
- Fenêtres secondaires affichées pour informer ou questionner
- Peuvent être modales
  - Impossibilité de passer à une autre fenêtre de l'application tant que la boîte de dialogue n'est pas fermée
- Exemples de boîtes de dialogue prédéfinies :
  - `JOptionPane`
  - `JFileChooser`
  - `JColorChooser`
- Créez vos propres boîtes de dialogue en spécialisant la classe `JDialog`

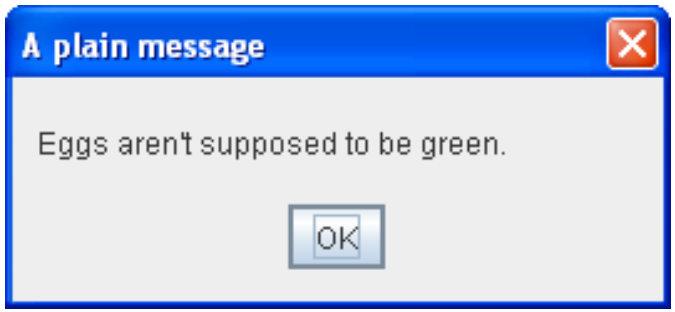
JOptionPane

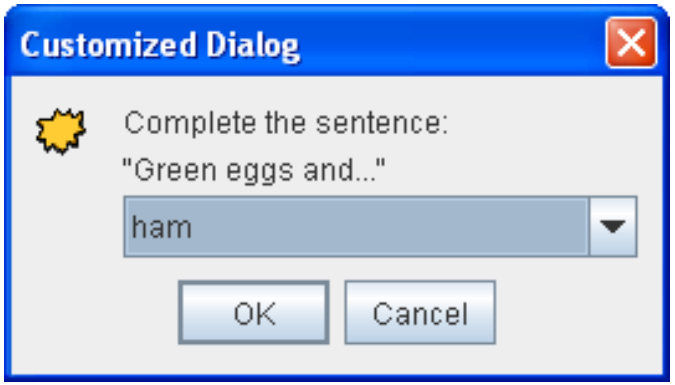
- Regroupe un ensemble de boîtes de dialogues prédéfinies pour informer, demander confirmation ou demander une valeur
- Ses méthodes :
  - `JOptionPane.showMessageDialog(...)` : informe
    - Pas de valeur de retour
  - `JOptionPane.showConfirmDialog(...)` : demande confirmation
    - Choix possibles (boutons présents) : `JOptionPane.YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`
    - Valeur retournée : `JOptionPane.YES_OPTION`, `OK_OPTION`, `NO_OPTION`, `CANCEL_OPTION`, `CLOSED_OPTION`
  - `JOptionPane.showInputDialog(...)` : demande une valeur
    - Soit avec un champ de saisie
    - Soit avec une liste déroulante
- Différents types de message :
  - `JOptionPane.ERROR_MESSAGE`, `WARNING_MESSAGE`, `INFORMATION_MESSAGE`,`PLAIN_MESSAGE`, `QUESTION_MESSAGE`

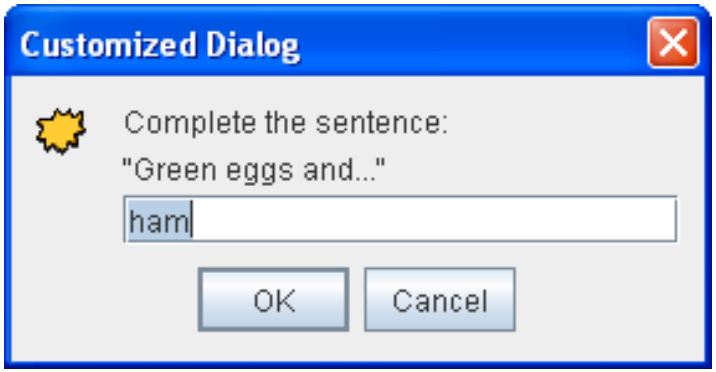
JOptionPane

Exemples







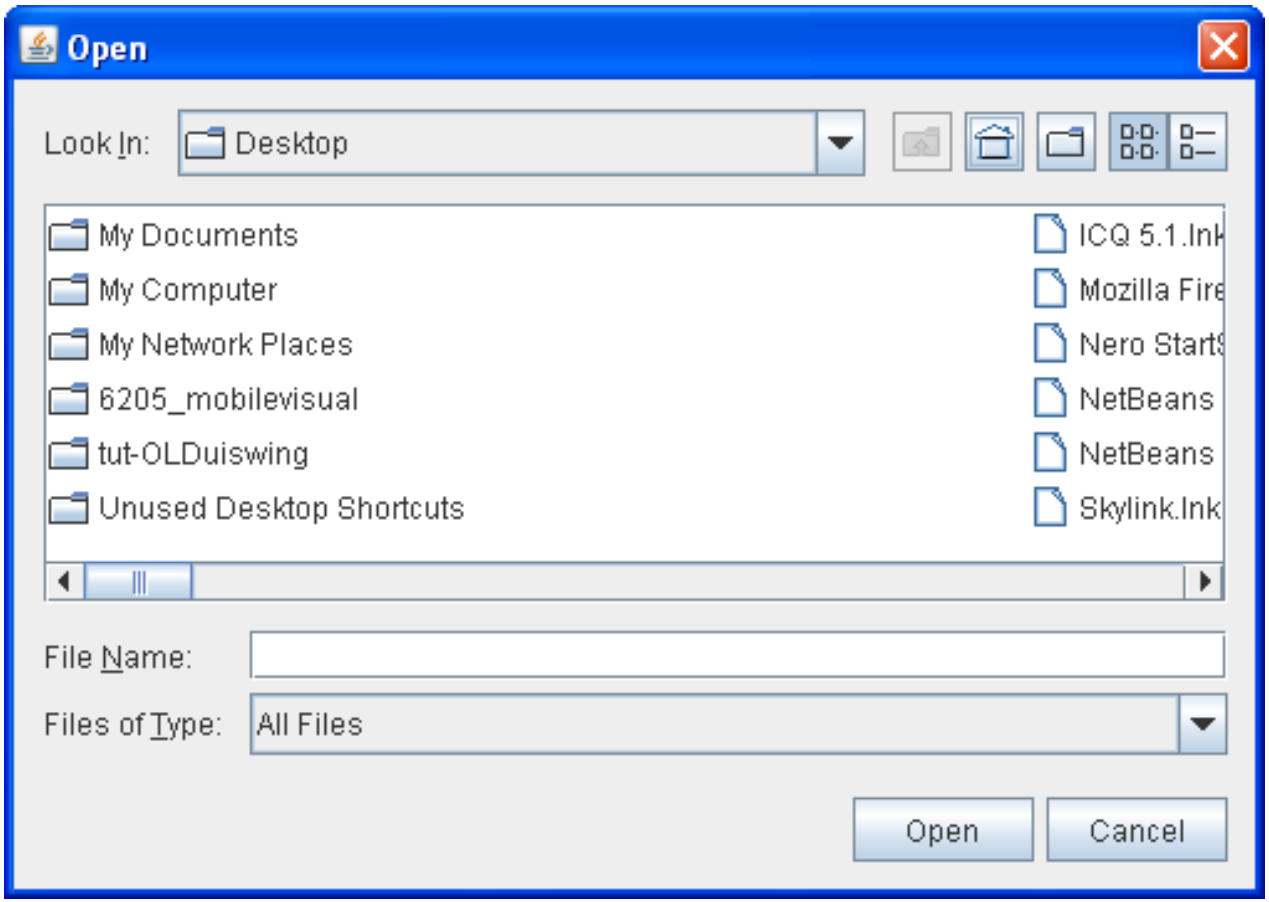


JFileChooser

- Permet à l'utilisateur de spécifier un nom de fichier ou de répertoire, en parcourant l'arborescence du système de fichier
- Ses méthodes (d'instance) :
  - `showOpenDialog` : demande le fichier à ouvrir
    - Valeur retournée : `JFileChooser.APPROVE_OPTION`, `CANCEL_OPTION`, `ERROR_OPTION`
  - `showSaveDialog` : demande sous quel nom enregistrer le fichier
  - `setFileSelectionMode` : définit à quoi doit correspondre la sélection
    - `JFileChooser.DIRECTORIES_ONLY`
    - `JFileChooser.FILES_AND_DIRECTORIES`
    - `JFileChooser.FILES_ONLY` (par défaut)
  - `addChoosableFileFilter` : ajoute un filtre permettant de n'afficher que certains types de fichier
    - En spécialisant la classe `FileFilter`
  - `setAcceptAllFileFilterUsed` : retire le filtre correspondant à "tous les types de fichier"
  - `getSelectedFile(s)` : retourne le(s) fichier(s) ouvert(s)/sauvé(s)

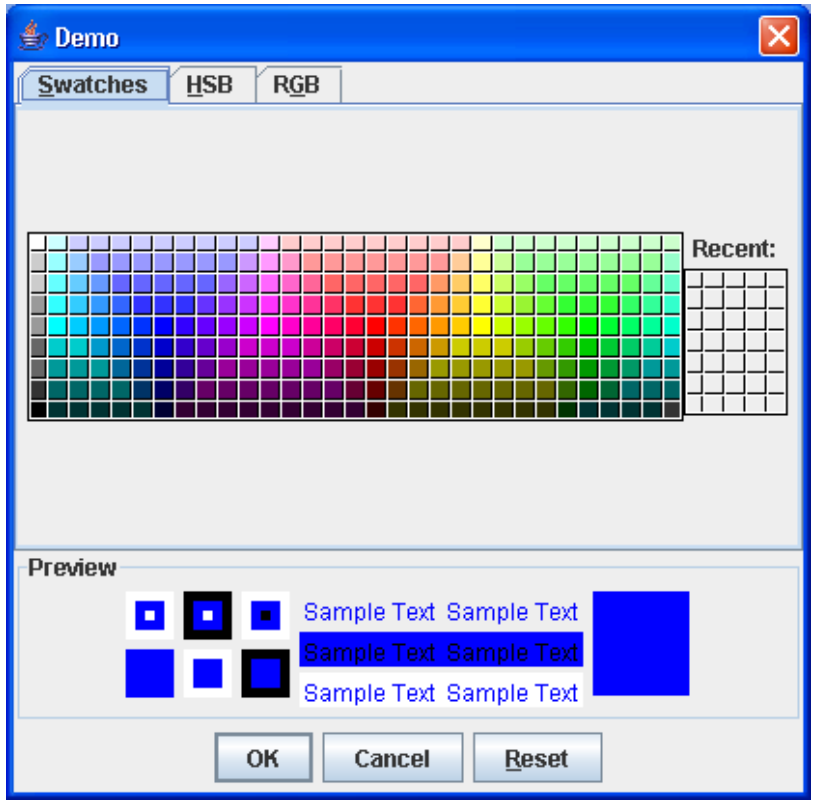
JFileChooser

Exemple



JColorChooser

- Permet à l'utilisateur de choisir une couleur
- Méthode à connaître :
  - `JColorChooser.showDialog(...)`
    - ★ Valeur retournée : la couleur choisie (objet de type `Color`) ou `null` si l'utilisateur clique sur "Annuler" ou le bouton "croix"



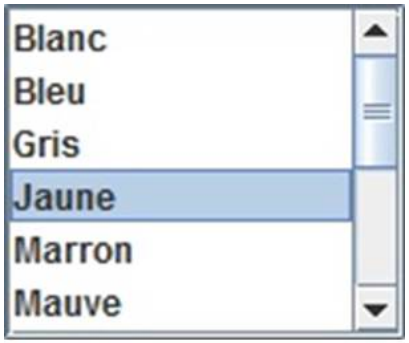
Composants

- Héritent de `JComponent`
- Quelques exemples :

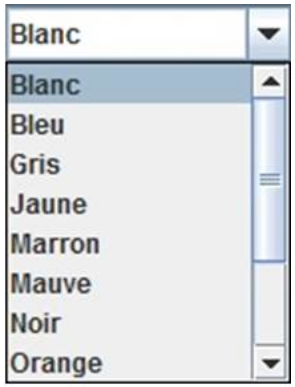
JButton


JToolBar

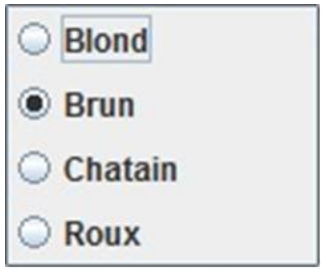
JSlider



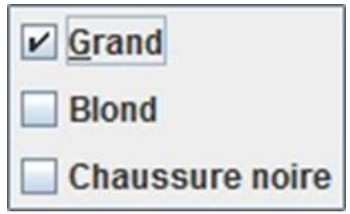
JList



JComboBox



JRadioButton



JCheckBox

Jlabel + JTextField

JProgressBar

JSpinner

- Et beaucoup d'autres...

Caractéristiques des composants

- Taille
  - `getWidth()`, `getHeight()`, `getSize()`, `setMaximumSize(Dimension)`, `setPreferredSize(Dimension)`, `setMinimumSize(Dimension)`, `setSize(Dimension)`
- Position
  - `getX()`, `getY()`, `setLocation(x,y)`, `setBounds(x,y,l,h)`
- Bordure (initialement `null`)
  - `getBorder()`, `setBorder(Border)`
- Disponibilité : le composant non disponible est grisé et l'utilisateur ne peut pas l'utiliser
  - `isEnabled()`, `setEnabled(boolean)`
- Visibilité : le composant peut être masqué
  - `isVisible()`, `setVisible(boolean)`
- Description (`ToolTip`) : ce qui s'affiche lorsque la souris survole le composant
  - `setToolTipText(String)`
- Alignement
  - `setAlignmentX(float)`, `setAlignmentY(float)`

Conteneurs intermédiaires

- Composants destinés à contenir d'autres composants afin d'aider à leur agencement
- Méthodes indispensables à connaître :
  - `add(Component)` : ajoute un composant
  - `setLayout(LayoutManager)` : définit le gestionnaire de placement à utiliser

Conteneurs intermédiaires

Classes

- `JPanel` : panneau graphique invisible permettant de grouper d'autres éléments et ainsi structurer votre interface graphique (important !)
- `JScrollPane`
  - Contient 1 composant et des barres de défilement (ascenseurs) permettant de déplacer la partie observable du composant
  - À utiliser lorsqu'un élément est trop grand pour être affiché dans sa totalité
- `JSplitPane` : permet de séparer l'espace en deux panneaux
  - `setDividerSize` : définit la taille de la séparation
  - `setOneTouchExpandable` : permet de masquer l'un des panneaux
- `JTabbedPane` : permet, *via* un système d'onglets, d'avoir plusieurs panneaux sur une même surface
  - `addTab`/`removeTabAt` : ajoute/supprime un onglet
  - `setMnemonicAt` : définit un raccourci clavier

Conteneurs intermédiaires

Classes

- `JLayeredPane` : permet de disposer les composants dans un espace tridimensionnel (profondeur/couche)
  - `add` : ajoute un composant dans une des couches
- `JInternalFrame` : permet de créer un environnement multi-fenêtré
  - Méthodes proches de celles de la classe `JFrame`
  - Mais n'est pas un conteneur racine
- `JToolBar` : barre d'outils qui regroupe un certain nombre de composants, généralement des boutons
  - `add` : ajoute un composant
  - `addSeparator` : ajoute un séparateur
  - `setFloatable` : rend la barre d'outils flottante (par défaut) ou non

Première fenêtre - suite

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

public class FirstFrame extends JFrame {

  public FirstFrame() {
    super("Premiere fenetre");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container contentPane = this.getContentPane();
    contentPane.add(new JLabel("Bonjour. C'est quoi ton nom ?"));
    contentPane.add(new JTextField());
  }

  public static void main(String[] args) {
    FirstFrame f = new FirstFrame();
    // f.setSize(600, 800);
    f.pack();
    f.setVisible(true);
  }
}
```

Que retenir ?

- Composants organisés en une arborescence de conteneurs (souvent des `JPanel`)
- Permet de réaliser des interfaces graphiques complexes
- Quelques conseils :
  - Faire un dessin de son interface sur papier pour en déduire l'arbre à réaliser
  - Regrouper les instructions correspondant à un même composant
  - S'il y a beaucoup de composants, mieux vaut diviser le constructeur en plusieurs méthodes

Au menu

1 Introduction

2 Fenêtres et composants

3 Gestionnaires de placement

4 Événements et écouteurs

5 PAC

Problématique

Spécifier les tailles et les positions des composants à la main est possible, mais :

- Laborieux
- Potentiellement inadaptées sur une autre configuration (matérielle et/ou logicielle)
- Besoin de prévoir le redimensionnement de la fenêtre (y compris avec des tailles trop réduites)

Solution : utiliser des gestionnaires de placement pour chaque conteneur

Gestionnaire de placement (*layout manager*)

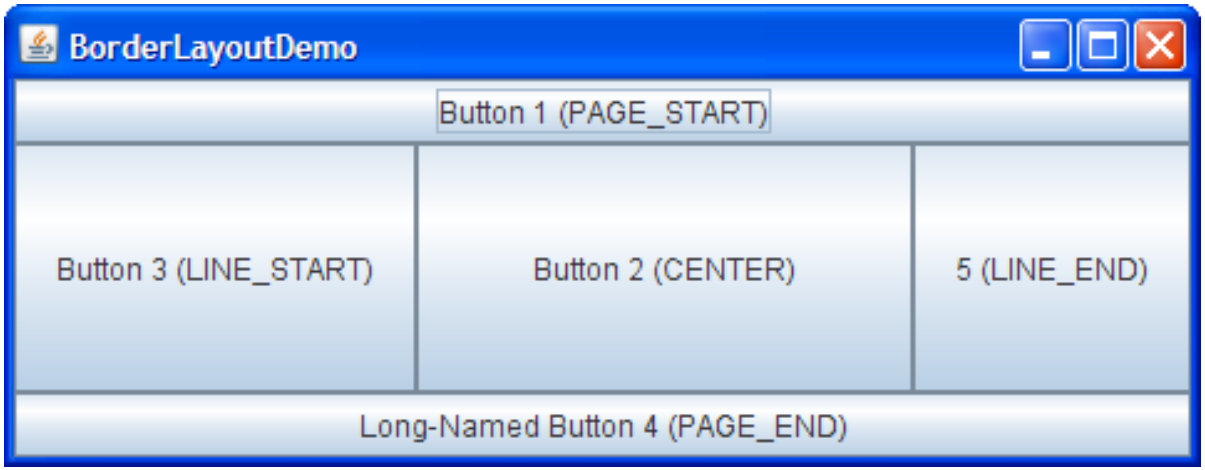
- Gère la disposition et les tailles des composants de façon portable (*i.e.*, multi-plateforme) en fonction :
  - De la taille du conteneur
  - De la politique de placement du gestionnaire
  - Des contraintes de disposition de chaque composant
  - Des caractéristiques communes à tous les composants
    - ★ `alignmentX`, `alignmentY`, `preferredSize`, `minimumSize` et `maximumSize`
- Agit de manière dynamique, en réalisant une mise à jour après chaque redimensionnement du conteneur
- Peut, ou non, respecter les alignements/dimensions désirés
- Exemples de gestionnaires de placement :
  - `BorderLayout`
  - `FlowLayout`
  - `GridLayout`
  - `GridBagLayout`
  - `BoxLayout`

BorderLayout

- Répartit les composants en cinq zones : `PAGE_START` (ou `NORTH`), `LINE_START` (ou `WEST`), `PAGE_END` (ou `SOUTH`), `LINE_END` (ou `EAST`) et `CENTER`
- Gestionnaire de placement par défaut des `contentPane`
- Politique de placement :
  - En `PAGE_START` et `PAGE_END`, les composants ont leur hauteur préférée (si possible) et s'étalent sur toute la largeur du conteneur
  - En `LINE_START` et `LINE_END`, les composants ont leur largeur préférée et s'étalent sur toute la hauteur restante entre `PAGE_START` et `PAGE_END`
  - Le composant central occupe tout l'espace restant
- Toutes les zones ne sont pas forcément occupées !
- Très pratique pour aligner des composants sur l'un des cotés et faire qu'un composant remplisse au mieux le reste de l'espace

BorderLayout

Exemple

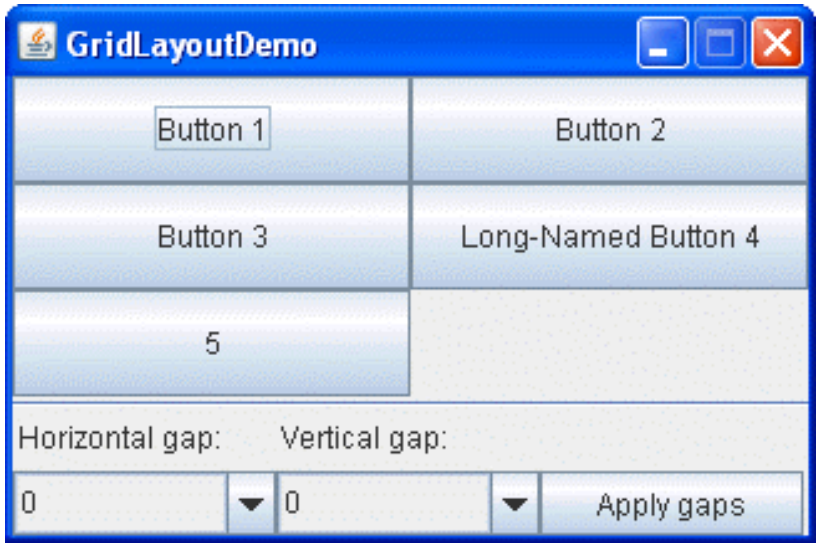


FlowLayout

- Place les composants de gauche à droite (en retournant à la ligne si besoin) en respectant les tailles préférées
- Possibilité de préciser un alignement en paramètre du constructeur
  - `FlowLayout.LEFT`, `RIGHT` ou `CENTER` (par défaut)

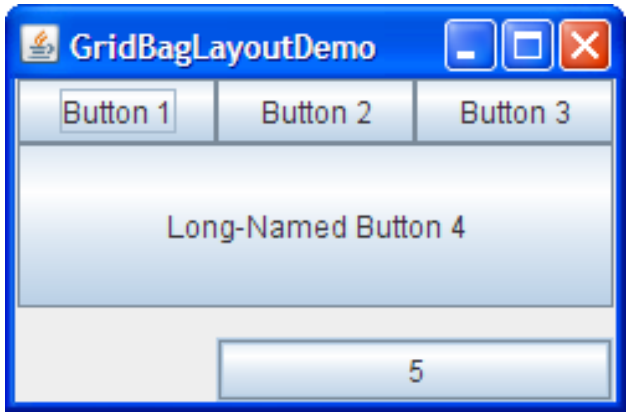
GridLayout

- Place les composants dans une grille dont les cases ont exactement la même taille
- Redimmensionne chaque composant pour qu'il occupe l'intégralité de la case (si possible)



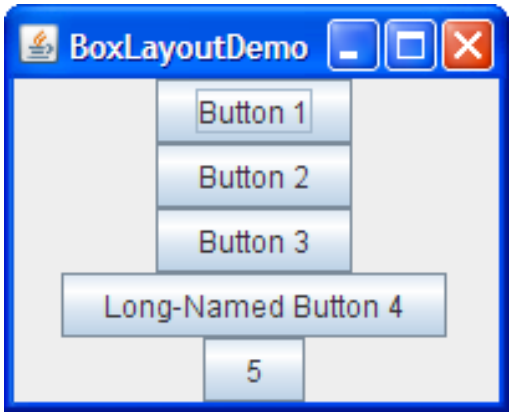
GridBagLayout

- Permet une disposition plus souple : un composant peut occuper plusieurs cases de la grille
  - Les composants peuvent donc ne pas avoir tous la même taille
- L'emplacement et la taille des composants repose sur :
  - La taille minimum des composants
  - La taille préférée du conteneur
  - Des contraintes
    - `gridx` et `gridy`
    - `gridwidth` et `gridheight`
    - `fill`
    - `ipadx` et `ipady`
    - `insets`
    - `anchor`
    - `weightx` et `weighty`



BoxLayout

- Permet d'aligner des composants
  - Verticalement (`BorderLayout.Y_AXIS`)
  - Horizontalement (`BorderLayout.X_AXIS`)
- Politique de placement vertical/horizontal :
  - Les composants ont leur hauteur/largeur préférée (mais étirés horizontalement/verticalement si possible)
  - Le gestionnaire tente de donner la largeur/hauteur du plus large/haut
  - Si le composant ne peut pas être étiré (`maximumSize` trop petit), le gestionnaire tient compte du `X_ALIGNMENT`/`Y_ALIGNMENT` pour le positionner



Box

- Permet de créer des composants invisibles très pratiques pour espacer les autres composants
- Ses méthodes :
  - `Box.createRigidArea()`
    - ★ Crée un composant invisible de dimension fixe
    - ★ Est utilisé pour créer un espace de dimension fixe entre deux composants
  - `Box.createHorizontalGlue()/Box.createVerticalGlue()`
    - ★ Crée un composant invisible "colle", élastique et extensible
    - ★ Est utilisé pour spécifier où l'espace en trop du conteneur devrait aller
  - `Box.Filler(...)`
    - ★ Composant invisible pour lequel on peut préciser des dimensions minimales, maximales et préférées

Première fenêtre - suite

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

public class FirstFrame extends JFrame {

  public FirstFrame() {
    super("Premiere fenetre");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container contentPane = this.getContentPane();
    contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.Y_AXIS
    contentPane.add(new JLabel("Bonjour. C'est quoi ton nom ?"));
    contentPane.add(new JTextField());
  }

  public static void main(String[] args) {
    FirstFrame f = new FirstFrame();
    // f.setSize(600, 800);
    f.pack();
    f.setVisible(true);
  }
}
```

Que retenir ?

- Chaque conteneur de l'interface graphique peut avoir son propre gestionnaire de placement
- `setLayout` : définit le gestionnaire de placement d'un conteneur

Au menu

1 Introduction

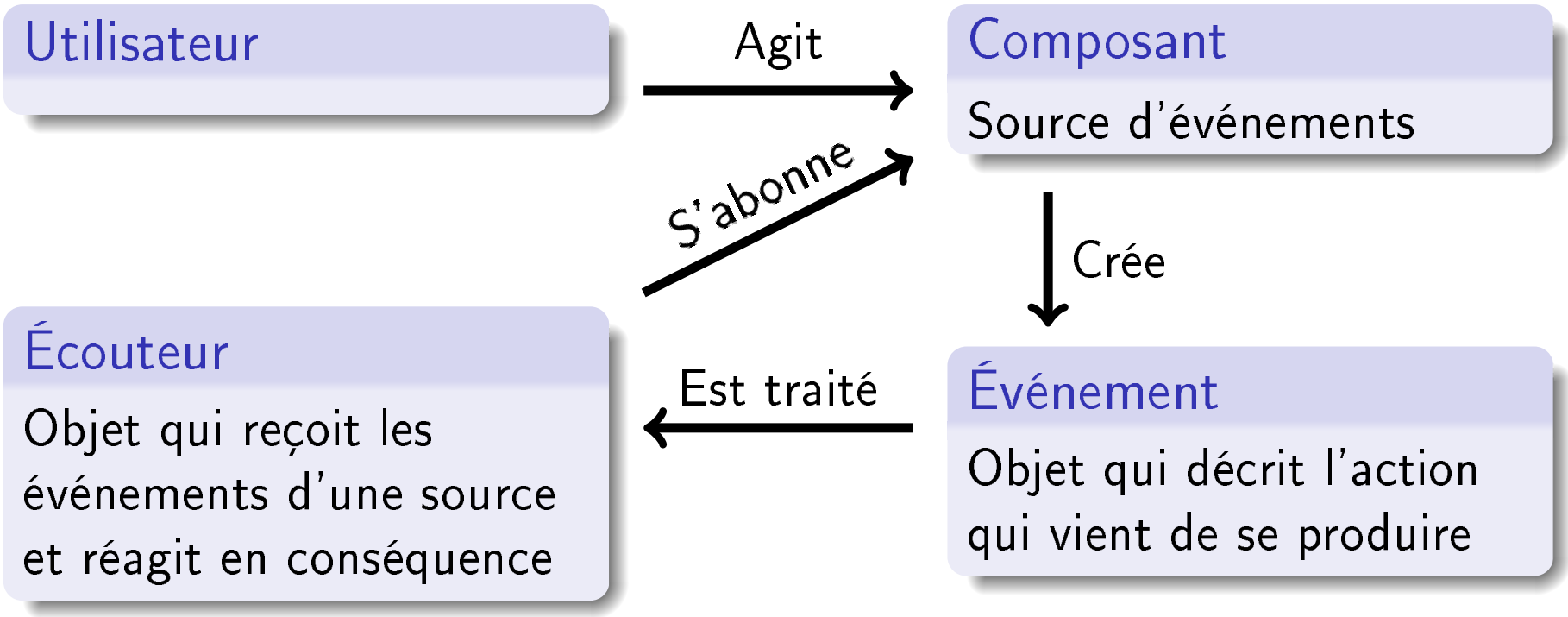
2 Fenêtres et composants

3 Gestionnaires de placement

4 Événements et écouteurs

5 PAC

Programmation événementielle



N.B. : plusieurs écouteurs peuvent s'abonner à la même source, qui enverra ses événements à chacun de ses abonnés

Écouteur (*listener*)

- Doit implémenter l'interface (*i.e.*, `XxxListener`) correspondant au type d'évènement (*i.e.*, `XxxEvent`)
  - Regroupe les signatures des méthodes à implémenter pour réagir aux événements
- Doit être abonné à la source d'événements (*via* `addXxxListener`)
- La classe de l'écouteur peut être :
  - Une classe externe
  - Le conteneur lui-même
  - Une classe interne
  - Une classe interne et anonyme
    - ★ Écrite à l'endroit de son instanciation

Écouteur - classe externe

Exemple

```
public class FirstFrame extends JFrame {
  public FirstFrame() {
    [...]
    JButton myButton = new JButton("ON");
    myButton.addActionListener(new FirstListener());
    this.getContentPane().add(myButton);
  }
}
public class FirstListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    ((JButton)e.getSource()).setText("OFF");
  }
}
```

- Avantage : peut être réutilisé
- Inconvénient : potentiellement beaucoup de classes

Écouteur - la fenêtre elle-même

Exemple

```
public class FirstFrame extends JFrame implements ActionListener {
  public FirstFrame() { [...]
    JButton myButton = new JButton("ON");
    myButton.addActionListener(this));
    this.getContentPane().add(myButton);
  }
  public void actionPerformed(ActionEvent e) {
    ((JButton)e.getSource()).setText("OFF");
  }}
```

- Avantage : évite de répartir le code dans un tas de classes
- Inconvénients :
  - Plusieurs boutons ⇒ consulter la source de l'événement (lisibilité ↘)
  - Plusieurs événements d'un même composant ⇒ éparpiller le code de traitement (maintenance ↘)
- Recommandable uniquement s'il y a très peu de composants et de types d'événements à traiter

Écouteur - classe interne

Exemple

```
public class FirstFrame extends Jframe {
  private JButton myButton;
  public FirstFrame() { [...]
    myButton = new JButton("ON");
    myButton.addActionListener(new FirstListener());
    this.getContentPane().add(myButton);
  }
  class FirstListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      myButton.setText("OFF"); }}}
```

- Est défini à l'intérieur de la classe de la fenêtre
- Étant interne, il peut accéder aux variables d'instance (plus utile de passer les composants utiles au constructeur de l'écouteur)
- Inconvénient : idem classe externe
- Recommandable s'il y a plusieurs instances du même écouteur et/ou si le code de traitement des événements est complexe

Écouteur - classe interne et anonyme

Exemple

```
public class FirstFrame extends Jframe {
  public Firstframe() { [...]
    JButton myButton = new JButton("ON");
    myButton.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          myButton.setText("OFF");
        }
      });
    this.getContentPane().add(monBouton);
  }}
```

- Étant interne, il peut accéder aux variables d'instance
- Étant anonyme, son code peut être rapproché de la définition du composant qu'il traite (lisibilité ↗)
- Attention à la syntaxe ! (toujours impossible d'instancier une interface)
- Recommandable si le traitement est court et non réutilisé par ailleurs

Que retenir ?

- Les écouteurs peuvent avoir des variables d'instance pour désigner des composants de l'interface à consulter ou modifier, ou encore des éléments du modèle
- Les événements sont gérés dans un unique *thread*
  - Garantit que l'ensemble du traitement d'un évènement sera effectué avant de passer à l'événement suivant
  - Si un calcul lourd doit être effectué en réponse à un évènement, mieux vaut réaliser ce traitement dans un autre *thread* (l'exécuter dans le même *thread* bloquerait le traitement des autres évènements)

Événements basiques

Ceux que tout composant peut émettre :

- `ComponentListener` – `ComponentEvent`
  - `componentHidden` : appelée lorsqu'on masque le composant
  - `componentMoved` : appelée lorsqu'on modifie la position du composant
  - `componentResized` : appelée lorsqu'on modifie la taille du composant
  - `componentShown` : appelée lorsqu'on rend visible le composant
- `FocusListener` – `FocusEvent`
  - `focusGained` : appelée lorsque le composant obtient le focus (c'est lui qui reçoit les entrées clavier)
  - `focusLost` : appelée lorsque le composant perd le focus
- `KeyListener` – `KeyEvent`
  - `keyPressed` : appelée lorsqu'on appuie sur une touche du clavier et que le composant a le focus
  - `keyReleased` : appelée lorsqu'on relâche une touche du clavier et que le composant a le focus
  - `keyReleased` : appelée lorsqu'on a tapé (appuyé puis relâché) une touche du clavier et que le composant a le focus

Événements basiques

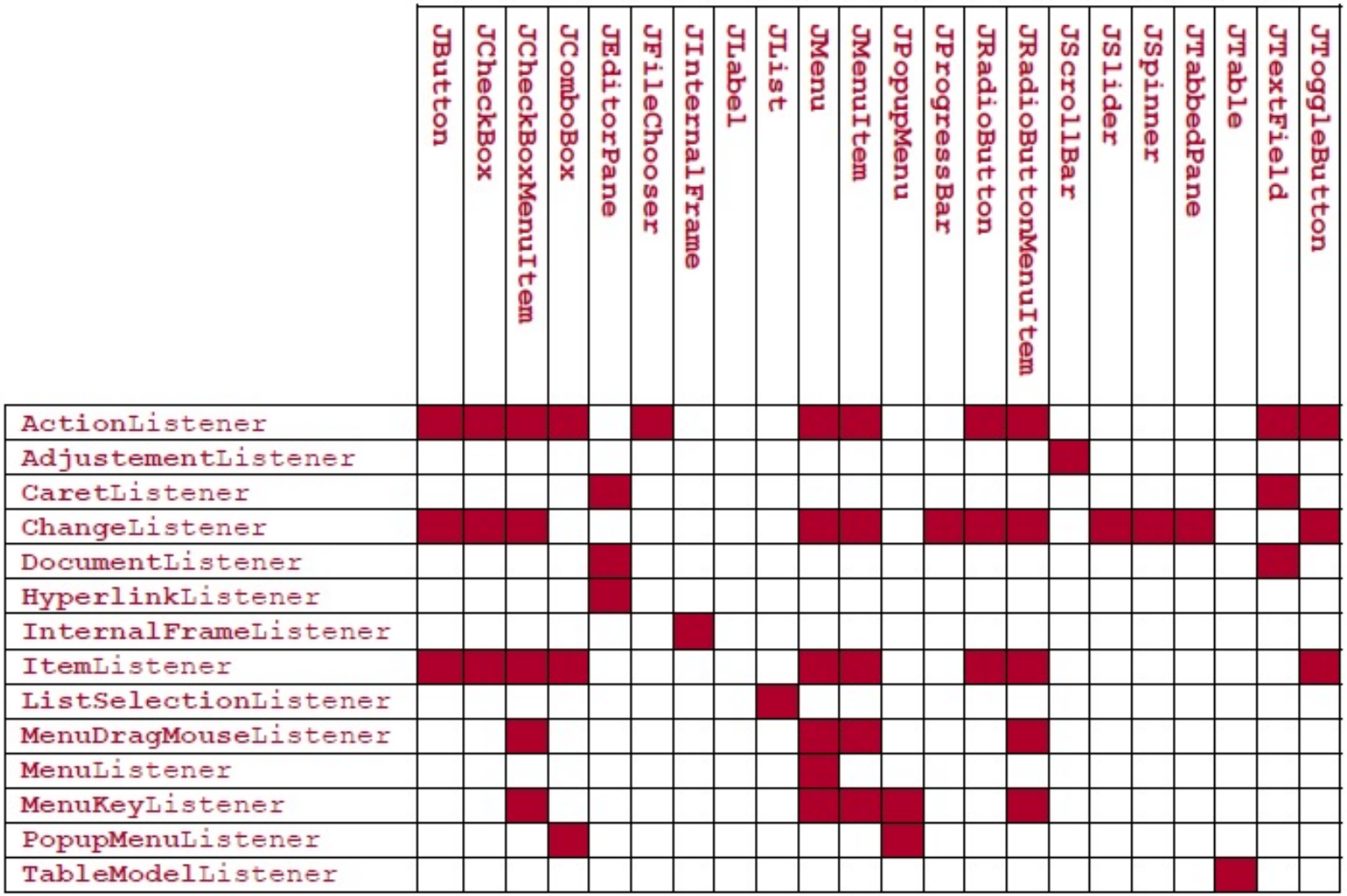
- `MouseListener` – `MouseEvent`
  - `mouseClicked` : appelée lorsqu'un bouton de souris a été cliqué (pressé puis relâché)
  - `mouseEntered` : appelée lorsque la souris rentre dans le composant
  - `mouseExited` : appelée lorsque la souris sort du composant
  - `mousePressed` : appelée lorsqu'un bouton de souris a été pressé
  - `mouseReleased` : appelée lorsqu'un bouton de souris a été relâché
- `MouseMotionListener` – `MouseEvent`
  - `mouseDragged` : appelée lorsque la souris est déplacée avec bouton enfoncé
  - `mouseMoved` : appelée lorsque la souris est déplacée sur le composant boutons relâchés
- `MouseWheelListener` – `MouseEvent`
  - `mouseWheelMoved` : appelée lorsqu'on tourne la molette de la souris

Événements basiques

- `WindowListener` – `WindowEvent`
  - `windowActivated` : appelée lorsque la fenêtre devient la fenêtre active
  - `windowClosed` : appelée lorsque la fenêtre a été fermée
  - `windowClosing` : appelée lorsqu'on tente de fermer la fenêtre
  - `windowDeactivated` : appelée lorsque la fenêtre perd son statut de fenêtre active
  - `windowDeiconified` : appelée lorsque la fenêtre redevient normale après avoir été iconisée
  - `windowIconified` : appelée lorsque la fenêtre est iconisée
  - `windowOpened` : appelée lorsque la fenêtre est rendue visible

D'autres événements

Ceux qui ne concernent qu'un sous-ensemble de composants :



Au menu

1 Introduction

2 Fenêtres et composants

3 Gestionnaires de placement

4 Événements et écouteurs

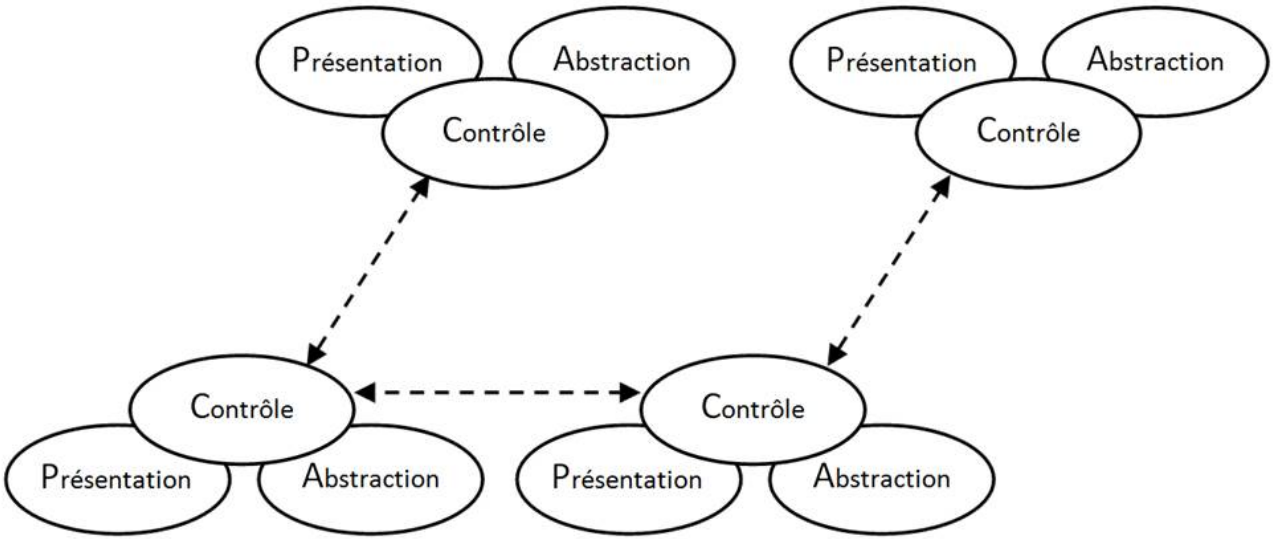
5 PAC

Problématique

- Ajouter une IHM dans une application peut être très intrusif
  - Mélange du code des interfaces graphiques avec celui du noyau applicatif (*i.e.*, le code qui modélise les données et fournit des traitements sur ces données)
  - Lisibilité, réutilisation, extensibilité et maintenance ↘
- On aimerait :
  - Séparer l'IHM du reste de l'application
  - Avoir un code modulaire (*i.e.*, la possibilité d'ajouter/retirer/substituer un module sans affecter les autres modules)

Présentation, Abstraction, Contrôle (PAC)

- Modèle abstrait d'architecture logicielle pour les IHM
- Similaire à MVC
- Organise le système interactif comme une hiérarchie de composants, chacun composé de trois types de facette



- Contrairement à MVC, la partie visualisation n'interagit pas directement avec le modèle !

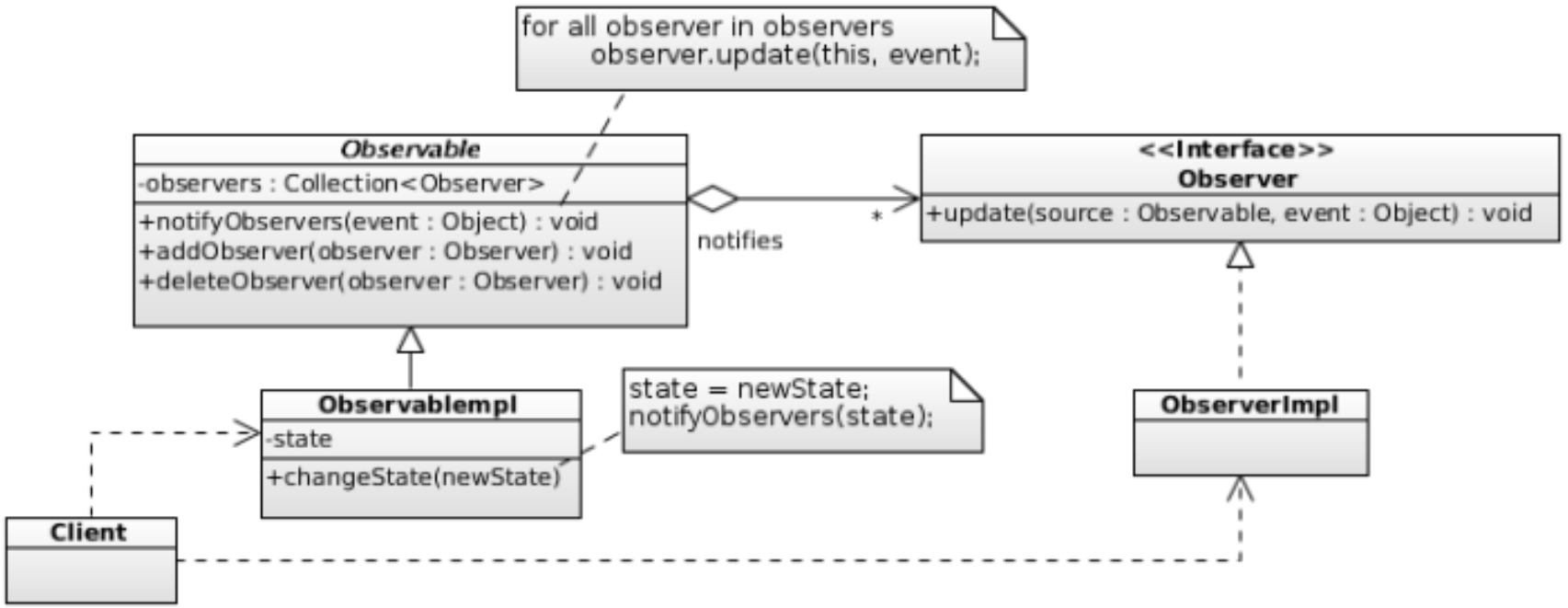
Trois types de facette

- Présentation
  - Purement IHM
  - Peut correspondre à un affichage ou à une saisie de données
- Abstraction
  - Noyau applicatif
  - Modélise la représentation et le traitement des données
- Contrôle
  - Lien entre les deux autres types de facette
  - Garantit la cohérence entre les données du modèle et leurs représentations
  - Traduit les actions de l'utilisateur en opérations sur le modèle

Utilisation du patron de conception *Observateur* de Java pour réaliser et connecter les différentes facettes

Patron de conception *Observateur*

- Définit une interdépendance de type un à plusieurs, de telle sorte que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour



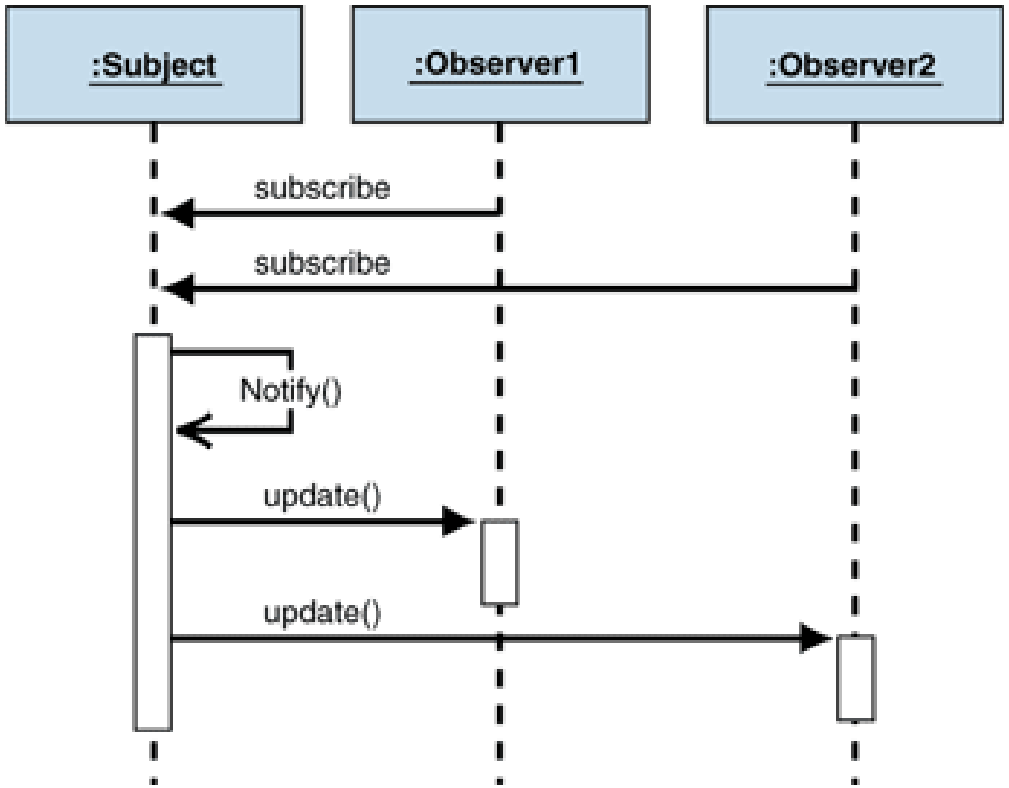
Classe abstraite `Observable`

- Fournit une interface pour ajouter/supprimer dynamiquement des observateurs
- Ses méthodes :
  - `addObserver/deleteObserver` : ajoute/supprime un observateur
  - `setChanged` : spécifie que l'objet a été modifié
  - `notifyObservers` : notifie les observateurs d'un changement
- Objet observé :
  - A sa classe qui étend la classe `Observable`
  - Connaît ses observateurs (un nombre quelconque d'observateurs peut l'observer)
  - Après chaque modification, appelle ses méthodes `setChanged` puis `notifyObservers` pour notifier ses observateurs lorsque son état change

Interface `Observer`

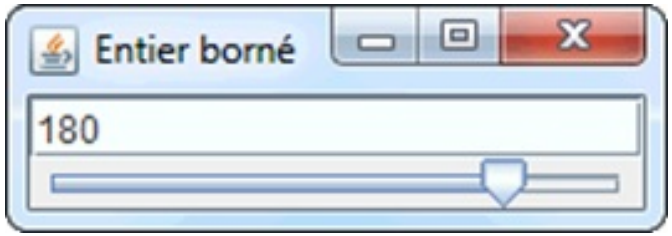
- Définit une interface de mise à jour pour les objets qui doivent être notifiés de changement dans un objet observé
- Sa méthode :
  - `update` : met à jour l'objet pour conserver la cohérence de son état avec celui de l'objet observé
- Objet observateur :
  - A sa classe qui implémente l'interface `Observer`
  - Doit être ajouté à la liste des observateurs de l'objet observé
  - Gère une référence sur l'objet qu'il observe
    - ★ Pour l'interroger sur son état afin d'adapter le sien
    - ★ Pour se supprimer de sa liste d'observateurs

Diagramme de séquence



Exemple

- Application manipulant une valeur entière bornée
  - Un `JTextField` qui affiche la valeur courante et peut permettre de saisir une autre valeur
  - Un `JSlider` qui permet de modifier l'entier en faisant glisser son curseur



- Interactions liées :
  - Si des traitements modifient la valeur de l'entier, les deux composants doivent être mis à jour
  - Si une valeur est saisie dans le `JTextField`, il faut vérifier qu'elle figure entre les bornes, et modifier le modèle et le `JSlider` en conséquence
  - Si le curseur du `JSlider` est déplacé, il faut mettre à jour le modèle et le `JTextField`

Problème

- Rien n'est indépendant : composants et modèle liés entre eux
- Risque d'oublier une mise à jour dans le traitement de l'un des événements
  - Introduction d'incohérence entre la vue et le modèle
- Remplacer le `JTextField` par un `JSpinner` $\Rightarrow$ revoir tout le code
- Solution : approche PAC

Abstraction

```
import java.util.Observable;
public class EntierBorne extends Observable { // est observable
  private int valeur, min, max;
  public EntierBorne(int valeur, int min, int max) {
    this.valeur = valeur;
    this.min = min;
    this.max = max;
  }
  public int getValeur() {
    return this.valeur;
  }
  public int getMin() {
    return this.min;
  }
  public int getMax() {
    return this.max;
  }
  public void setValeur(int valeur) {
    this.valeur = Math.max( Math.min(valeur, this.max), this.min);
    this.setChanged();              // appelees a chaque
    this.notifyObservers(null); // modification
  }
}
```

Présentation

```
public class IHMEntierBorne extends JFrame {
  private EntierBorne abstraction;
  private JTextField textField; // facette de presentation
  private JSlider slider; // facette de presentation

  public IHMEntierBorne(EntierBorne entier) {
    super("Entier borne");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    abstraction = entier;
    int valeur = abstraction.getValeur();
    int min = abstraction.getMin();
    int max = abstraction.getMax();
    Container contentPane = this.getContentPane();
    contentPane.setLayout(new BoxLayout(contentPane,
                                        BoxLayout.Y_AXIS));
    textField = new JTextField(""+valeur);
    contentPane.add(textField);
    slider = new JSlider(JSlider.HORIZONTAL, min, max, valeur);
    contentPane.add(slider);
```

Notez que les deux composants n'ont pas de référence vers l'abstraction (*i.e.*, ils ne pourront pas communiquer directement avec le modèle)

Présentation - suite

```
        ControlJTextField cl = new ControlJTextField(abstraction, // (3)
                                                     textField); // (2)
        textField.addActionListener(cl); // (1)
        abstraction.addObserver(cl); // (4)
        ControlJSlider cs = new ControlJSlider(abstraction, // (3)
                                               slider); //(2)
        slider.addChangeListener(cs); // (1)
        abstraction.addObserver(cs); // (4)
    }
}
```

- Communication contrôle–présentation :
  1. Le contrôle est un écouteur du composant (*i.e.*, il réagira aux actions de l'utilisateur sur ce composant)
  2. Passage du composant en paramètre du constructeur du contrôle afin qu'il puisse agir sur le composant si les données du modèle changent
- Communication contrôle–modèle :
  3. Passage de l'abstraction en paramètre du constructeur du contrôle
  4. Ajout du contrôle à la liste des observateurs de l'abstraction

Contrôle - JSlider

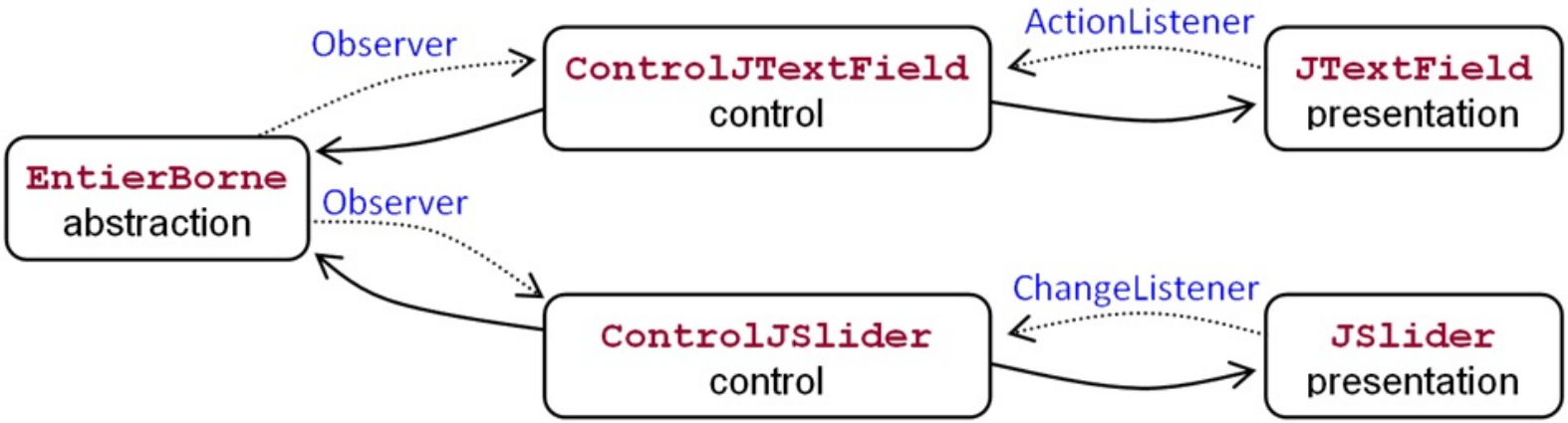
```
public class ControlJSlider implements Observer, ChangeListener {
  private EntierBorne entier;
  private JSlider slider;
  public ControlJSlider(EntierBorne entier, JSlider slider) {
    this.entier = entier;
    this.slider = slider;
  }
  public void stateChanged(ChangeEvent arg0) {
    entier.setValeur(this.slider.getValue());
  }
  public void update(Observable o, Object arg) {
    slider.setValue(this.entier.getValeur());
  }
}
```

- Constructeur : des références vers son abstraction et sa facette de présentation pour pouvoir y accéder
- Implémente `ChangeListener` pour pouvoir écouter le `JSlider`
- Implémente `Observer` pour pouvoir être ajouté à la liste des observateurs de l'abstraction

Contrôle - JTextField

```
public class ControlJTextField implements Observer, ActionListener {
  private JTextField textField;
  private EntierBorne entier;
  public ControlJTextField(EntierBorne entier, JTextField textField) {
    this.entier = entier;
    this.textField = textField;
  }
  public void update(Observable o, Object message) {
    textField.setText(""+entier.getValeur());
  }
  public void actionPerformed(ActionEvent arg0) {
    int valeur = Integer.MIN_VALUE;
    // Contrairement au JSlider, il est necessaire de verifier que
    // la valeur du JTextField est un entier compris entre les bornes
    try {
      valeur = Integer.parseInt(this.textField.getText());
    } catch(Exception e) { }
    if (valeur < this.entier.getMin() || valeur > entier.getMax()) {
      JOptionPane.showMessageDialog(null, "La valeur doit etre entre "
        + entier.getMin()+" et "+entier.getMax(), "Valeur hors bornes"
        JOptionPane.WARNING_MESSAGE);
      textField.setText(""+entier.getValeur());
    } else { entier.setValeur(valeur); }}}
```

En résumé



- Si on change la valeur du `JTextField` :
  - Le `ControlJTextField` vérifie si la valeur saisie est entre les bornes
  - Si ce n'est pas le cas, elle affiche une boîte de dialogue et met à jour le `JTextField`
  - Sinon, elle modifie l'entier borné
    - ★ L'entier borné notifie ses observateurs d'une modification
    - ★ La méthode `update` du `ControlJSlider` est appelée et celle-ci met à jour le `JSlider`
- Symétriquement, si on bouge le `JSlider` :
  - L'entier borné est mis à jour par le `ControlJSlider`, ce qui notifie le `ControlJTextField` qui actualise le `JTextField`

Conclusion

- Séparation nette entre la partie purement IHM et le noyau applicatif
- Composants indépendants à présent
  - `JSlider` n'a plus à se soucier du `JTextfield`
  - Modifications propagées *via* les contrôleurs
- Robustesse améliorée
  - Plus difficile d'oublier une mise à jour
- Meilleure modularité
  - Ajout/retrait d'une partie de l'interface sans impacter les autres parties