I.H.M.

1. Sommaire

2. Introduction

2.1. Concevoir vs réaliser un IHM

L'intitulé de ce cours est abusif. L'« interaction homme-machine » au sens large regroupe tous les moyens et outils mis en œuvre afin qu'un humain puisse contrôler et communiquer avec une machine. Elles mobilisent des électroniciens et des mécaniciens pour les dispositifs matériels, des informaticiens pour la partie logicielle, mais aussi des psychologues, des sociologues, ou encore des ergonomes. Pour concevoir une bonne IHM mieux vaut donc avoir une équipe regroupant l'ensemble de ces spécialités.

Nous nous limiterons à la **réalisation d'une interface graphique en java** sans périphérique atypique.

Comme vous pourrez vous en rendre compte, un tas de classes et d'interfaces seront utilisées durant ce cours, mais Oracle fournit des tutoriels en plus de la classique documentation de l'API. **L'enjeu n'est donc pas d'apprendre par cœur toutes les caractéristiques de ces classes**, mais d'acquérir les notions fondamentales utiles à la conception d'une interface graphique en Java et à prendre l'habitude de consulter les documents d'Oracle lorsqu'on a besoin d'un complément d'information. Ce support ne se veut donc pas exhaustif, bien au contraire, il se focalise sur l'essentiel.

2.2. AWT vs Swing

Comme vous le savez, Java est multiplateforme, le bytecode généré peut être exécuté sur n'importe quelle machine disposant d'une machine virtuelle java. Dès les premières versions les concepteurs ont voulu fournir une API pour les interfaces graphiques qui soit elle aussi indépendante des plateformes. Mais les systèmes de Gestion d'Interface Utilisateur (GUI) sont très différents les uns des autres.

La première approche utilisée a été d'utiliser au maximum les systèmes graphiques, aboutissant à AWT (Abstract Window Toolkit). AWT a des avantages, en s'appuyant sur les GUI l'aspect des interfaces était celui des autres interfaces du système. Les concepteurs de AWT ont de plus pu s'appuyer sur des composants existant. Par contre, AWT ne regroupe que les composants disponibles sur toutes les plateformes et il est difficile de garantir un comportement identique sur toutes les plateformes.

A l'inverse, SWING qui a été intégrée au JDK dès la version 1.2 vise à minimiser l'utilisation du système graphique sous-jacent. Elle ne se base que sur des primitives très basiques comme le dessin d'une ligne ou d'un texte et sur la gestion primitive des évènements. Du coup les concepteurs de SWING ont du réimplémenter tous les composants à partir de ces primitives élémentaires et les interfaces graphiques réalisées avec SWING ont un aspect qui peut être, selon les systèmes d'exploitation, légèrement différent des autres applications qu'on trouve ordinairement. Mais le gain est énorme puisqu'on a alors l'assurance d'avoir le même comportement sur l'ensemble des plateformes et tout nouveau composant est immédiatement disponible sur l'ensemble des plateformes.

Mieux vaut ne pas mélanger au sein d'une même interface graphique des composants AWT et SWING. On n'utilisera pas conjointement des instances de `Button` et de `JButton`. Nous nous focaliserons sur SWING, mais certaines ressources introduites dans AWT n'ont pas eu à être modifiées pour SWING. Pour ne pas nous casser la tête à savoir quel package devra être ou non utilisé, nous importerons d'office tous les package de AWT et SWING en débutant nos codes avec les imports suivants.

```
import javax.swing.* ;  // le x n'est pas une erreur.
import java.awt.* ;
```

Une troisième bibliothèque plus récente, SWT, s'appuie comme AWT sur les primitives graphiques natives avec un abandon partiel de la portabilité au profit de l'efficacité. Contrairement aux autres codes java elle requière une désallocation explicite de la mémoire et il est moins aisé de réaliser un composant spécialisé multiplateforme. Nous ne l'utiliserons pas.

2.3. Ce qui vous attend

Avant de nous plonger dans les détails, précisons les grandes lignes. **Une interface graphique c'est une fenêtre qui contient des composants agencés à l'écran via des gestionnaires de placement et qui réagissent aux actions de l'utilisateur via des gestionnaires d'évènements.** C'est caricatural, un peu simpliste, mais l'essentiel est là. Une IHM ce sera avant tout une fenêtre (vide) à laquelle on aura ajouté les composants qui nous semblent pertinents. Pour disposer ces composants correctement entre eux nous préciserons les gestionnaires de placements à utiliser. Puis, pour que l'IHM prenne vie, nous préciserons des gestionnaires d'évènement qui réagiront aux actions de l'utilisateur.

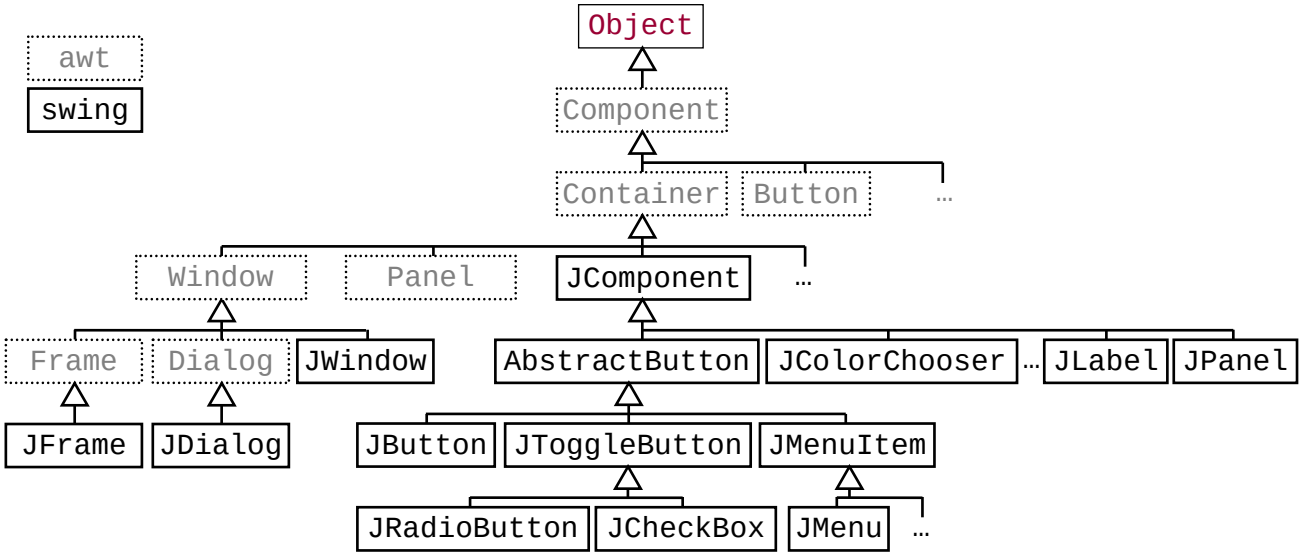
Rajouter une IHM dans une application peut-être très intrusif en mélangeant le code de l'IHM au sein des autres traitements. La lisibilité, la réutilisabilité, l'extensibilité et la maintenance en générale s'en trouvent affaiblies. Pour remédier à cela, nous introduirons dans le dernier chapitre l'un des modèles de conception visant à séparer l'IHM du reste de l'application.

Enfin, avant d'entrer dans le vif du sujet, sachez que vous allez changer de paradigme de programmation. Rassurez-vous, ce sera toujours du Java, mais tout comme vous avez pu être surpris d'aborder Java avec une approche récursive après avoir fait vos premiers codes de manière procédurale, vous risquez d'être surpris en passant ici à de la **programmation évènementielle**. L'idée n'est plus de dérouler un programme ligne à ligne depuis la première ligne du `main`, mais d'interagir avec l'utilisateur. Le déroulement du programme est guidé par l'utilisateur. Nous allons créer des objets permettant des affichages et/ou des saisies et nous devrons fournir des bouts de code à exécuter en réaction à chaque action possible de l'utilisateur. On prévoit les traitements à réaliser (les réactions) pour chaque évènement (action de l'utilisateur) possible.

3. Des fenêtres et des composants

Voici une représentation très partielle (Swing comprend plusieurs centaines de classes, sans compter les interfaces) de la hiérarchie des classes. La quasi-totalité des composants introduits par Swing héritent de `JComponent`.

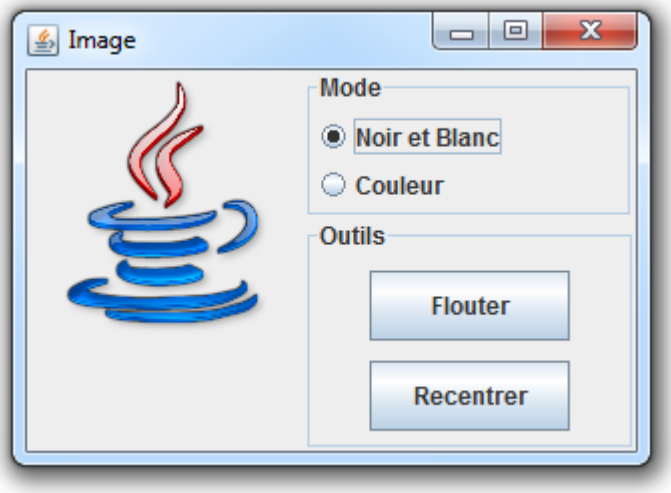
Une interface graphique sera composée d'un conteneur racine (`JFrame` en ce qui nous concerne) lequel contiendra potentiellement plusieurs conteneurs intermédiaires et des composants atomiques.

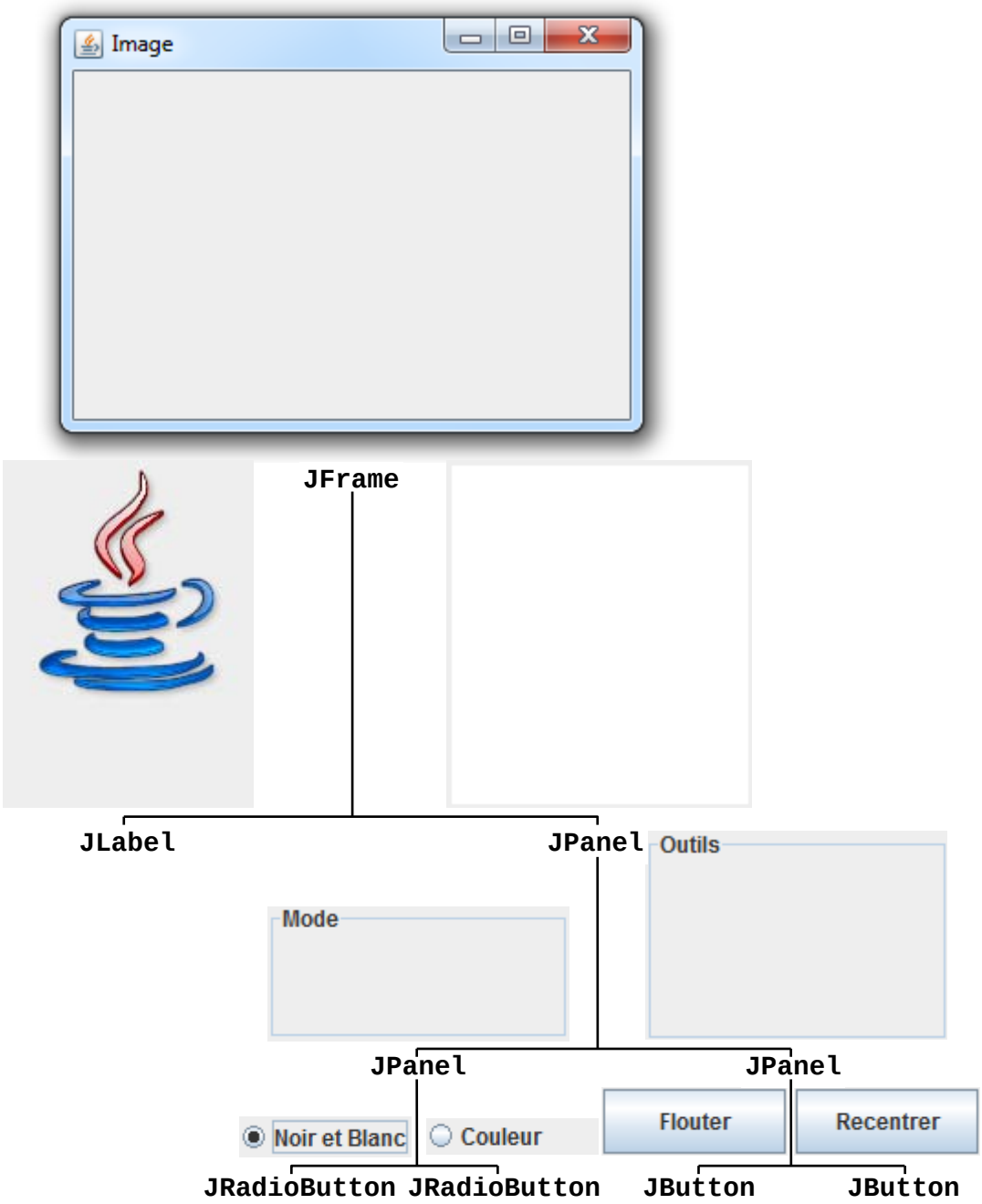


3.1. Conteneurs racine (*top-level container*)

Comme leur nom l'indique, les conteneurs sont destinés à contenir d'autres conteneurs ou composants. Cette organisation forme un arbre ayant un conteneur racine pour racine, d'autres conteneurs pour nœuds internes et des composants atomiques pour feuilles.

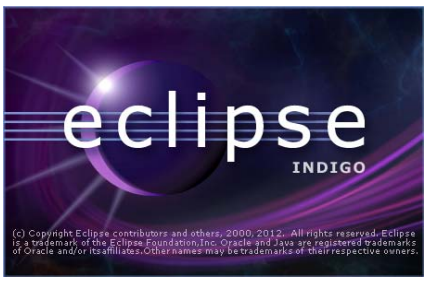
L'image ci-dessous vous montre une interface et l'organisation en arbre de ses composants. Elle est constituée d'une `JFrame` qui comporte un `JLabel` (la tasse de café) et un `JPanel` contenant toute la partie droite de l'interface. Ce dernier comporte deux `JPanel`, celui du haut incluant deux `JRadioButton` tandis que celui en bas comporte deux `JButton`.





Il existe 4 conteneurs racine :

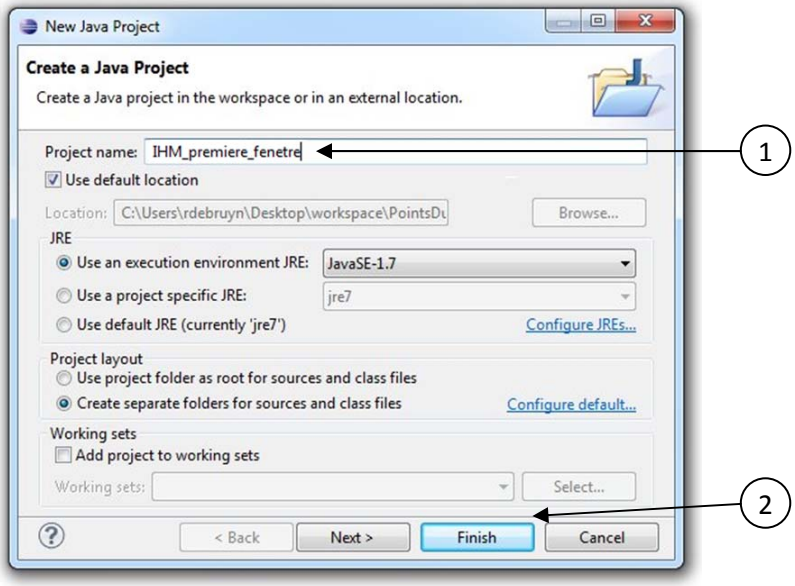
- `JApplet` : Une Applet est un programme Java qui peut-être exécuté par un navigateur internet. Elles ont contribué au succès de Java en montrant sa portabilité mais nous ne les étudierons pas.
- `JWindow` : Fenêtre simple sans bordure ni décoration utilisées principalement pour les « splashscreen » affichées lors du lancement d'un programme, comme celle d'eclipse ci-contre.



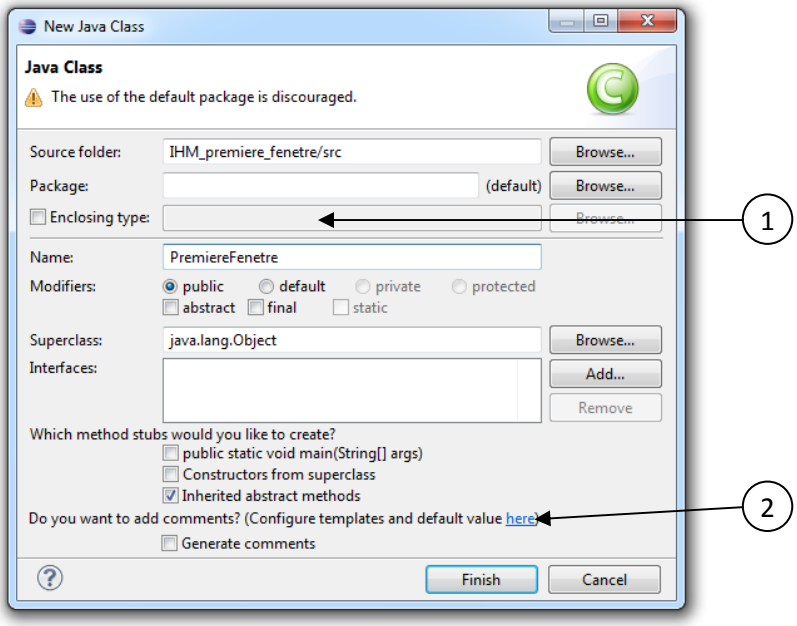
- `JDialog` : Boîte de dialogue. Fenêtre fille d'une application, qui peut-être éventuellement modale, c'est-à-dire sans possibilité d'interagir avec la fenêtre mère tant que la boîte de dialogue est ouverte. Nous les aborderons.
- `JFrame` : Fenêtre principale d'une IHM swing.

Comme vous l'avez compris, **nos interfaces graphiques utiliseront forcément une instance de** `JFrame` et nous ferons éventuellement appel à des `JDialog`. Les `JFrame` comportent un bord (permettant de redimensionner), une barre de titre avec bien sûr un titre et les boutons habituels pour iconifier, maximiser ou fermer la fenêtre. Les méthodes disponibles permettent de modifier tout cela. `setTitle("mon titre")` par exemple permet de redéfinir le titre, `setUndecorated(true)` rend la fenêtre dépourvue de bord, de barre de titre et d'icones tandis que `setResizable(false)` interdit son redimensionnement. La liste des méthodes est impressionnante et nous n'avons pas à les connaître toutes car (1) la javadoc est là pour nous éclairer sur les méthodes disponibles et leur spécifications et (2) la complétion d'éclipse permet de nous rappeler les méthodes accessibles. Plutôt qu'un long passage en revu construisons notre première fenêtre.

- Lancez Eclipse (double-cliquez sur son raccourci puis validez votre espace de travail).
- Dans le menu File sélectionnez New puis Java Project (File | New | Java Project).



- Dans la fenêtre qui s'ouvre, (1) indiquez un nom de projet (ici IHM_premiere_fenetre) puis (2) cliquez sur Finish.
- Faites un clic-droit sur votre nouveau projet et sélectionnez New | Class. (1) Indiquez `PremiereFenetre` pour nom de classe avant de (2) cliquer sur Finish.



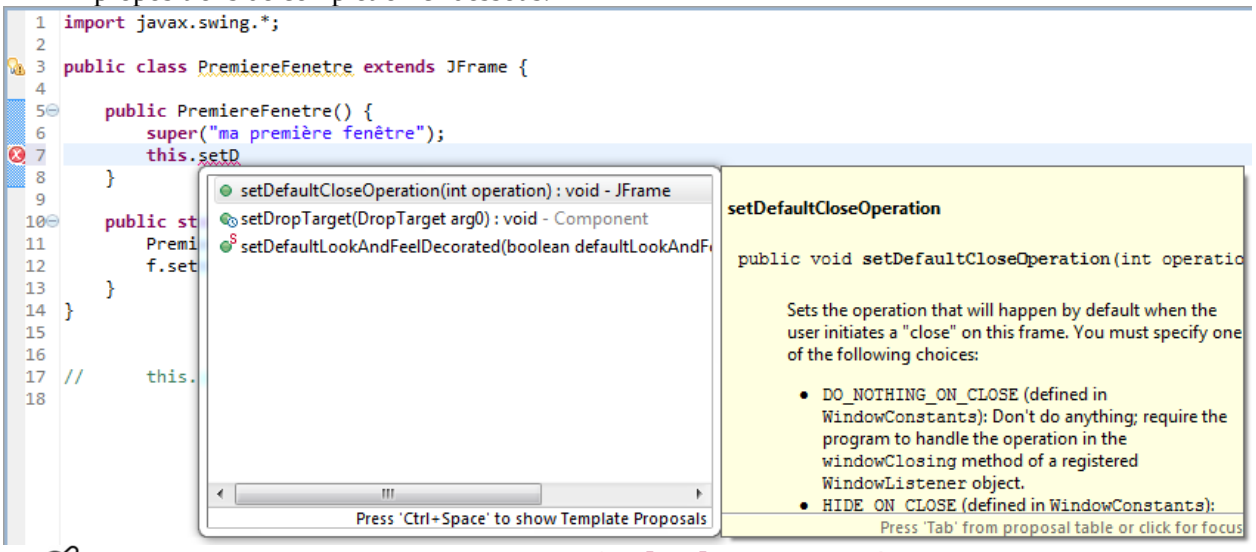
Précisez que `PremiereFenetre` hérite de `JFrame`, définissez un constructeur qui appel celui de la super-classe avec "`ma première fenêtre`" pour paramètre puis rajoutez une procédure principale qui crée une instance de `PremiereFenetre`. Vous devriez obtenir le code ci-dessous. Exécutez-le.

```
import javax.swing.*;

public class PremiereFenetre extends JFrame {
   public PremiereFenetre() {
      super("ma première fenêtre");
   }

   public static void main(String[] args) {
      PremiereFenetre f = new PremiereFenetre();
   }
}
```

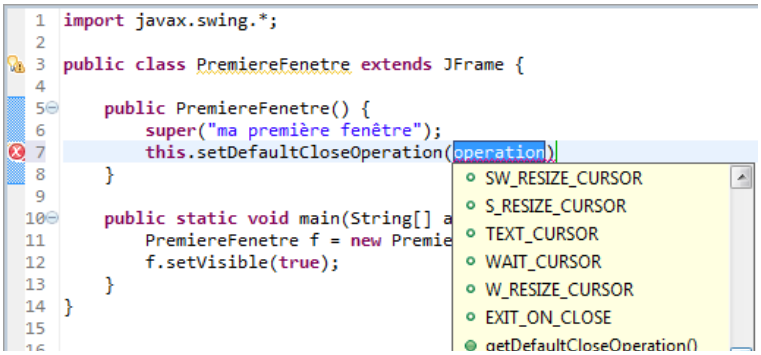
Vous ne voyez rien, et c'est normal. Nous avons créé une fenêtre mais nous ne l'avons pas rendu visible. Rajoutez `f.setVisible(true);` dans la procédure principale puis exécutez à nouveau. 

Une fenêtre apparaît mais elle est minuscule… et si vous essayez de la fermer en cliquant sur son icône en forme de croix elle disparait mais le programme ne se termine pas. Vous pouvez le constater sous éclipse par la présence d'un carré rouge juste au dessus du cadre de la console. Cliquez sur ce carré rouge pour interrompre le programme. 

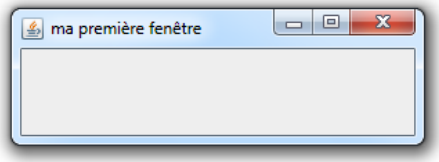
Tapez `this.setD` dans le constructeur et vous devriez voir eclipse vous indiquer les propositions de complétion ci-dessous.



Double-cliquez sur la proposition `setDefaultCloseOperation`. Une fenêtre vous propose divers paramètres. Sélectionnez `EXIT_ON_CLOSE`.



Cette fois le programme se ferme si on clique sur l'icône de fermeture de la fenêtre mais la fenêtre est toujours aussi petite. Rajoutez `this.setSize(300,100);` dans le constructeur.



Voici quelques leçons à retenir de cette première création de fenêtre :

- **A sa création une fenêtre est invisible** et il faut utiliser `setVisible` pour qu'elle s'affiche.
- **La taille d'une fenêtre est initialement nulle.** Nous pouvons la redimensionner avec des valeurs empiriques via `setSize` mais nous verrons que nous pourrons demander à ce que ses dimensions s'adaptent aux composants qu'elle utilise.
- **Lorsqu'on crée puis rend visible une fenêtre un nouveau thread** (processus léger) **se lance**. C'est lui qui va gérer les évènements liés à la fenêtre. Si le thread principal s'achève (parce qu'on a atteint la fin de `main`) l'application n'est pas terminée pour autant car le thread de la fenêtre subsiste. Par défaut l'icône en forme de croix masque la fenêtre mais ne la détruit pas ce qui explique que le programme continue de s'exécuter. On peut changer ce comportement par défaut au moyen de `setDefaultCloseOperation`.
- **La complétion d'Eclipse est très pratique** pour se souvenir des méthodes disponibles et pour éviter les fautes de frappe.

Avant de passer en revue les principaux composants de Swing, complétons notre code en rajoutant un texte et un champ de saisie.

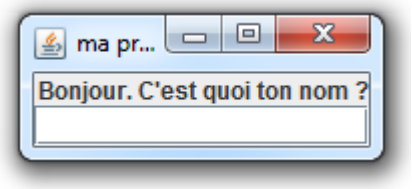
Modifiez votre code pour qu'il corresponde au code ci-dessous.

```
import java.awt.*;
import javax.swing.*;
public class PremiereFenetre extends JFrame {
  public PremiereFenetre() {
    super("ma première fenêtre");
    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) {
    PremiereFenetre f = new PremiereFenetre();
    f.pack();
    f.setVisible(true);
  }
}
```

Détaillons ce qui a été fait :

- L'intérieur d'une `JFrame` est en deux parties **: une barre de menu** (par défaut inexistante mais que nous pourrons spécifier) **et un conteneur principal** qui réunira tous les composants de la fenêtre. On accède à ce conteneur via la méthode `getContentPane()`.
- **On définit un gestionnaire de placement** (Layout manager) pour notre conteneur principale. On choisit ici un `BoxLayout` (nous en verrons d'autres) afin que les composants soient les uns par-dessus les autres.
- **On ajoute les composants**, ici un `JLabel` et un `JTextField`, dans le conteneur principal.
- Avant de rendre la fenêtre visible, on demande à la fenêtre **via sa méthode `pack()`** de recalculer les dimensions et le positionnement des éléments en tenant compte des layout manager.

Notez qu'il est possible d'utiliser la méthode `setLocationRelativeTo(c)` pour positionner la fenêtre à l'écran de telle sorte que le centre de la fenêtre coïncide avec le centre du composant `c` (avec le centre de l'écran si `c` vaut `null`).



Dans un premier temps nous allons nous attacher à savoir correctement agencer des composants pour bâtir une interface. Ce n'est qu'ensuite que nous les rendrons opérationnelles en définissant des réponses aux actions de l'utilisateur.

Résumé

- Nos interfaces seront composées d'une instance de `JFrame` laquelle possède un conteneur principal accessible via la méthode `getContentPane()`. C'est dans ce conteneur que sont ajoutés les conteneurs intermédiaires et les composants atomiques formant l'arbre d'affichage.
- Les boites de dialogue (`JDialog`) sont des fenêtres filles auxquelles nos IHM auront potentiellement recours.
- On peut spécifier manuellement les tailles et les positions des composants mais le résultat risque d'être inadéquat sur une autre configuration (matérielle et/ou logicielle) et inadapté si par exemple l'utilisateur modifie la taille de la fenêtre. Mieux vaut donc spécifier un gestionnaire de placement pour chaque conteneur afin d'agencer ses composants selon nos souhaites.
- Initialement une fenêtre a une taille nulle et n'est pas visible. On doit utiliser sa méthode `pack()` pour la forcer à recalculer les tailles et les positions de ses composants et sa méthode `setVisible(true)` pour la rendre visible.
- Lorsqu'on crée puis rend visible une fenêtre un nouveau thread se lance afin de gérer les évènements liés à cette nouvelle fenêtre. Tous les threads (le principal lié au lancement du programme et ceux liés aux fenêtres) doivent s'achever pour que l'application se termine.
- Par défaut l'icône en forme de croix d'une fenêtre ne fait que la masquer, elle ne la détruit pas. On peut changer ce comportement par défaut au moyen de `setDefaultCloseOperation`.
- La complétion d'Eclipse est très pratique pour se souvenir des méthodes disponibles et pour éviter les fautes de frappe.

3.2. Des composants

Les composants swing héritent de `JComponent`. Parmi les caractéristiques communes ont peut noter :

- Une **taille** : `getWidth()`, `getHeight()`, `getSize()`,`setMaximumSize(Dimension)`, `setPreferredSize(Dimension)`, `setMinimumSize(Dimension)`, `setSize(Dimension)`. Nous y reviendrons lorsque nous traiterons des gestionnaires de placement.
- Une **position** : `getX()`, `getY()`, `setLocation(x,y)`, `setBoundsocation(x,y,l,h)`.
- Une **bordure** (initialement **`null`**) : `getBorder()`, `setBorder(Border)`. A noter que les bordures et la barre de titre réduisent l'espace réellement utilisable (l'intérieur de la fenêtre). On peut consulter l'espace pris par la barre de titre et les bordure via la méthode `getInsets()`. La classe `BorderFactory` offre un tas de bordures possibles. Par exemple, les `JPanel` de droite de l'exemple utilisé pour illustrer l'arbre d'affichage utilisent `BorderFactory.createTitledBorder("Mode")`. Pour un aperçu des bordures consultez le how-to http://docs.oracle.com/javase/tutorial/uiswing/components/border.html .
- Une **disponibilité** : `isEnabled()`, `setEnabled(boolean)`. Un composant non disponible est grisé et l'utilisateur ne peut pas l'utiliser.
- Une **visibilité** : `isVisible()`, `setVisible(boolean)`. On peut masquer un composant.
- Une **description fugace** (ToolTip) qui s'affiche si on place la souris sur le composant : `setToolTipText(String)`.

Cette liste est loin d'être complète, vous pouvez vous reporter à la javadoc de `JComponent`.

Dans les sections qui suivent nous allons faire un tour rapide dès principaux composants en montrant un exemple ne montrant qu'une partie des possibilités offertes. N'hésitez pas à consulter les how-to (http://docs.oracle.com/javase/tutorial/uiswing/TOC.html) et la javadoc pour plus de détails.

Tout comme on ne lit pas un dictionnaire ou un guide de référence de la première page à la dernière, il est recommandé de ne consulter en première lecture que les éléments les plus utilisés à savoir `JPanel`, `JLabel`, `JTextField` et `JButton`.

3.2.1. Conteneurs intermédiaires

Il s'agit de composants destinés à contenir d'autres composants et d'aider en cela à leur agencement. Les méthodes indispensables à connaître sont :

- `add(Component)` pour rajouter un composant.
- `setLayout(LayoutManager)` afin de préciser le gestionnaire de placement à utiliser.

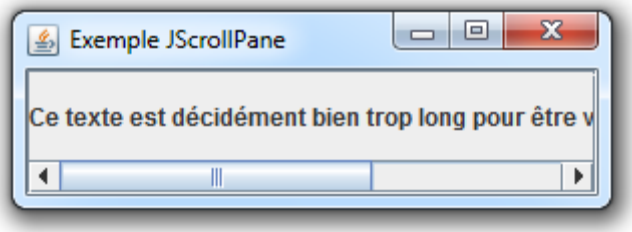
`JPanel`

S'il ne fallait en retenir qu'un ce serait celui là. Il s'agit d'un panneau graphique invisible qui permet de grouper d'autres éléments. Vous aurez bien souvent à imbriquer des `JPanel` dans d'autres `JPanel` afin de structurer votre interface.

Le `JPanel` peut également être vu comme un composant neutre. Cet aspect est utilisé pour bâtir ses propres composant par spécialisation de `JPanel`.

JScrollPane

Utilisé lorsqu'un élément est trop grand pour être affiché dans sa totalité. Un `JScrollPane` contient un composant et des barres de défilement/ascenseurs pour permettre de déplacer la partie observable du composant.

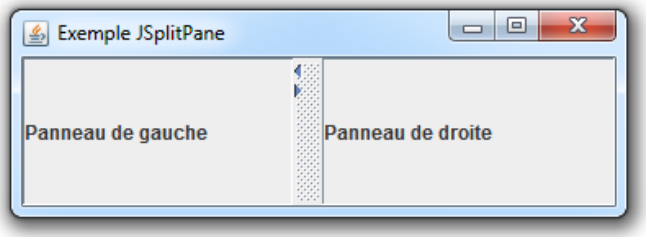


Exemple

```
public class ExempleJScrollPane extends JFrame {
   public ExempleJScrollPane() {
      super("Exemple JScrollPane");
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.setSize(300,100);
      JLabel texte = new JLabel("Ce texte est décidément bien trop long"
                          +" pour être visible dans une si petite fenetre");
      JScrollPane sp = new JScrollPane( texte );
      this.getContentPane().add( sp );
   }
   public static void main(String[] args) {
      ExempleJScrollPane maFenetre = new ExempleJScrollPane();
      maFenetre.setVisible(true);
   }
}
```

JSplitPane

Permet de séparer l'espace en deux zones. La barre de séparation (horizontale ou verticale) peut-être déplacée par l'utilisateur pour peu que les `MinimumSize` des composants l'autorisent.

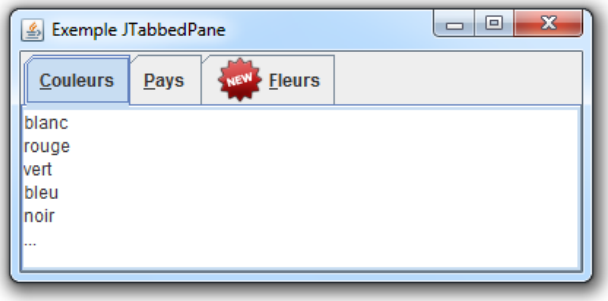


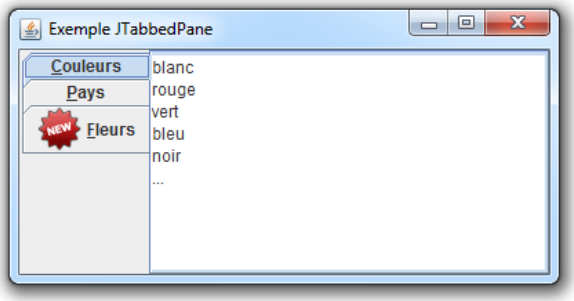
Exemple

```
public class ExempleJSplitPane extends JFrame {
  public ExempleJSplitPane() {
     super("Exemple JSplitPane");
     this.setDefaultCloseOperation(EXIT_ON_CLOSE);
     JLabel gauche = new JLabel("Panneau de gauche");
     JLabel droite = new JLabel("Panneau de droite");
     JSplitPane sp=new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,gauche,droite);
     sp.setDividerSize(20);// taille de la separation à 20
     sp.setOneTouchExpandable(true);// pour pouvoir masquer l'un des panneaux
     this.getContentPane().add( sp );
  }
  public static void main(String[] args) {
     ExempleJSplitPane maFenetre = new ExempleJSplitPane();
     maFenetre.pack();
     maFenetre.setVisible(true);
  }
}
```

JTabbedPane

Permet, via un système d'onglets, d'avoir plusieurs panneaux sur une même surface. On peut passer d'un onglet à l'autre en cliquant avec la souris, à l'aide des flèches gauche et droite du clavier ou au moyen de raccourcis clavier indiqués via la méthode `setMnemonicAt`. Dans l'exemple ci-dessous appuyer sur [Alt]+[P] affichera l'onglet « Pays ». Préciser de tels raccourcis n'a rien d'obligatoire, pas plus comme la précision d'une icône, l'exemple ci-dessous ne faisant qu'illustrer certaines possibilités offertes. Les titres des onglets sont par défaut au dessus mais on peut les positionner également à gauche, à droite ou en dessous au moyen de `setTabPlacement`. Vous remarquerez ci-dessous la méthode `addTab` pour ajouter des onglets mais on peut également retirer des onglets via `removeTabAt`.



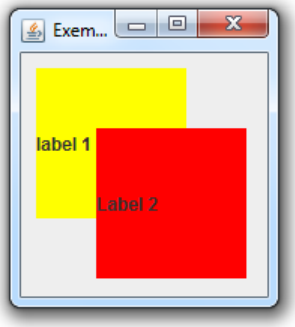


Exemple

```
public class ExempleJTabbedPane extends JFrame {

  public ExempleJTabbedPane() {
    super("Exemple JTabbedPane");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
     JTabbedPane tp = new JTabbedPane();
     JTextArea couleurs = new JTextArea("blanc\nrouge\nvert"+
                                        "\nbleu\nnoir\n...");
     tp.addTab("Couleurs", couleurs);// ajout d'un onglet sans icone
     tp.setMnemonicAt(0, 'C'); // définition d'un raccourci clavier
     JTextArea pays = new JTextArea("France\nBelgique\nEspagne\n...");
     tp.addTab("Pays", pays);
     tp.setMnemonicAt(1, 'P');
     JTextArea fleurs = new JTextArea("rose\ntulipe\npaquerette\n...");
     ImageIcon iconeNew = new ImageIcon("images"+File.separator+"new.png");
     tp.addTab("Fleurs", iconeNew, fleurs); // ajout d'un onglet avec icone
     tp.setMnemonicAt(2, 'F');
     tp.setTabPlacement(JTabbedPane.LEFT); // placer les onglets à gauche
     this.getContentPane().add( tp );
  }

  public static void main(String[] args) {
    ExempleJTabbedPane maFenetre = new ExempleJTabbedPane();
    maFenetre.setSize(400,200);
    maFenetre.setVisible(true);
  }
}
```

JLayeredPane

Permet de disposer les composants dans un espace tridimensionnel. En plus des positions 2D habituelles, on peut spécifier la profondeur (la couche) dans laquelle on ajoute le composant.

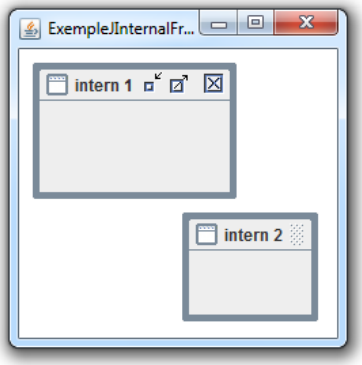


Exemple

```
public class ExempleJLayeredPane extends JFrame {
  public ExempleJLayeredPane() {
    super("Exemple JLayeredPane");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JLayeredPane lp = new JLayeredPane( );
    JLabel p1 = new JLabel("label 1");
    p1.setBackground(Color.YELLOW);
    p1.setOpaque(true);
    p1.setBounds(10,10,100,100);
    lp.add(p1, new Integer( 1));
    JLabel p2 = new JLabel("Label 2");
    p2.setBackground(Color.RED);
    p2.setOpaque(true);
    p2.setBounds(50,50,100,100);
    lp.add(p2, new Integer(2));
    this.getContentPane().add(lp);
  }
  public static void main(String[] args) {
    ExempleJLayeredPane fenetre = new ExempleJLayeredPane();
    fenetre.setSize(180,200);
    fenetre.setVisible(true);
  }
}
```

JInternalFrame

Permet de créer un environnement multi-fenêtré. Chaque `JInternalFrame` a un comportement et des méthodes proches de ceux des `JFrame` mais elles ne sont pas des conteneurs Top-Level.

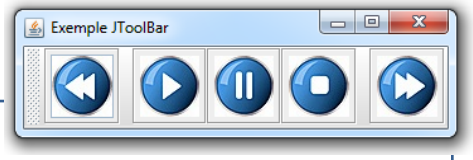


Exemple

```
public class ExempleJInternalFrame extends JFrame {
  public ExempleJInternalFrame() {
    super("ExempleJInternalFrame");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JDesktopPane dp = new JDesktopPane();
    JInternalFrame ifr = new JInternalFrame("intern 1");
    ifr.setBounds(10,10,150,100);
    ifr.setClosable(true);
    ifr.setMaximizable(true);
    ifr.setIconifiable(true);
    ifr.setVisible(true);
    dp.add(ifr);
    JInternalFrame ifr2 = new JInternalFrame("intern 2");
    ifr2.setBounds(120, 120, 100, 80);
    ifr2.setVisible(true);
    dp.add(ifr2);
    dp.setVisible(true);
    this.getContentPane().add(dp);
  }
  public static void main(String[] args) {
    ExempleJInternalFrame fenetre = new ExempleJInternalFrame();
    fenetre.setSize(250,250);
    fenetre.setVisible(true);
  }
}
```

JToolBar

Une barre d'outils qui regroupe un certain nombre de composants, généralement des boutons. Les barres d'outils ont par défaut la capacité de pouvoir être rendues flottantes si l'utilisateur les déplace, mais on peut écarter cette possibilité via un appel à `setFloatable(false)`. On peut espacer les composants en rajoutant un séparateur (`addSeparator()`).



Exemple

```
public class ExempleJToolBar extends JFrame {
  public ExempleJToolBar() {
    super("Exemple JToolBar");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JToolBar barre = new JToolBar();
    barre.add(new JButton(new ImageIcon("images"+File.separator+
                                                    "icon_rewind.jpg")));
    barre.addSeparator();
    barre.add(new JButton(new ImageIcon("images"+File.separator+"icon_play.jpg")));
    barre.add(new JButton(new ImageIcon("images"+File.separator+"icon_pause.jpg")));
    barre.add(new JButton(new ImageIcon("images"+File.separator+"icon_stop.jpg")));
    barre.addSeparator();
    barre.add(new JButton(new ImageIcon("images"+File.separator+
                                                    "icon_forward.jpg")));
    this.getContentPane().add(barre);
  }
  public static void main(String[] args) {
    ExempleJToolBar fenetre = new ExempleJToolBar();
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

3.2.2. Composants atomiques

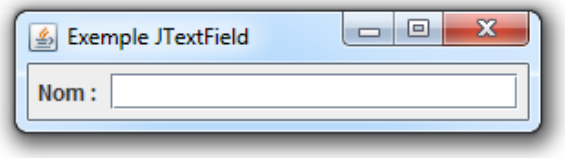
- *Composants textuels de saisie*

JTextField

Il s'agit d'un champ de saisie sur une seule ligne. Très utilisé dans les boîtes de dialogues, on l'utilise généralement en association avec un `JLabel` comme dans l'exemple ci-dessous fin de préciser ce qu'il convient de saisir. Il peut contenir un texte initial indiqué via la méthode `setText` ou en paramètre du constructeur (ex. `new JTextField("toto")`). Le constructeur prenant un entier (nombre de caractères) en paramètre s'appuie sur les caractéristiques de la fonte d'écriture courante pour dimensionner le `JTextField`.

La méthode indispensable à connaître est `getText()` qui permet de consulter la valeur indiquée dans le champ de saisie.

Notez que bien que par nature les `JTextField` sont éditable, on peut verrouiller la possibilité de modifier le texte via `setEditable(false)`.



Exemple

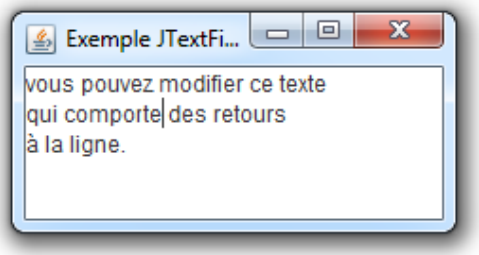
```
public class ExempleJTextField extends JFrame {
  public ExempleJTextField() {
    super("Exemple JTextField");
    Container conteneur = this.getContentPane();
    conteneur.setLayout(new FlowLayout());
    conteneur.add( new JLabel("Nom : "));
    conteneur.add( new JTextField(20)); // 20 caractères de large
    }
  public static void main(String[] args) {
    ExempleJTextField fenetre = new ExempleJTextField();
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

JPasswordField

Spécialisation de `JTextField` qui masque les caractères saisis par un caractère prédéfini (● par défaut).

JTextArea

Permet la saisie sur plusieurs lignes. Propose des fonctionnalités similaires à `JTextField` telles `void setText(String)`, `String getText()` ou encore `void setEditable(boolean)`.



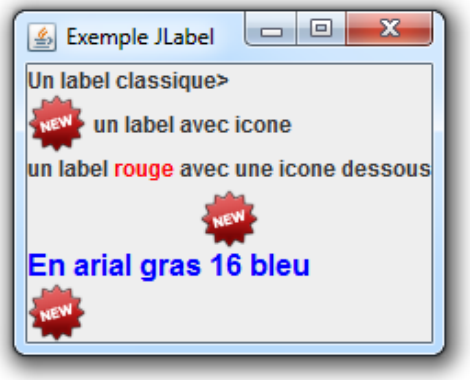
Exemple

```
public class ExempleJTextArea extends JFrame {
  public ExempleJTextArea() {
    super("Exemple JTextField");
    JTextArea texte = new JTextArea(5,20);
    // 5 lignes de 20 caracteres
    texte.setText("vous pouvez modifier ce texte "
      +"\nqui comporte des retours \nà la ligne.");
    this.getContentPane().add( texte );
  }
  public static void main(String[] args) {
    ExempleJTextArea fenetre = new ExempleJTextArea();
    fenetre.pack();
    fenetre.setVisible(true);
  }
```

- *Composants textuels d'affichage*

JLabel

Utilisé pour afficher du texte et/ou une image, le plus généralement en guise d'étiquette d'autres composants pour par exemple préciser ce qu'il convient d'indiquer dans un champ de saisie. Il n'est pas destiné à recevoir d'interactions de la part de l'utilisateur bien que comme tout autre composant il soit possible de le faire réagir aux actions de la souris et aux frappes du clavier. Les `JLabel` supportent le code html et il est donc possible d'afficher un texte à la mise en forme évoluée bien que les possibilités offertes par `setFont` soient généralement suffisantes en matière de mise en forme.

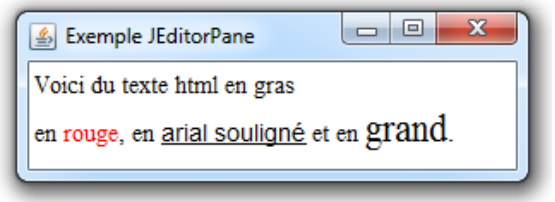


Exemple

```
public class ExempleJLabel extends JFrame {
  public ExempleJLabel() {
    super("Exemple JLabel");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container conteneur = this.getContentPane();
    conteneur.setLayout(new BoxLayout(conteneur, BoxLayout.Y_AXIS));

    JLabel etiquette = new JLabel("Un label classique>");
    conteneur.add(etiquette);

    JLabel avecIcone = new JLabel("un label avec icone");
    avecIcone.setIcon(new ImageIcon("images"+File.separator+"new.png"));
    conteneur.add(avecIcone);

    JLabel avecIconeDessous = new JLabel("<html>un label  <font color="
                  +"\"red\">rouge</font> avec une icone dessous</html>");
    avecIconeDessous.setIcon(new ImageIcon("images"+File.separator+new.png"));
    avecIconeDessous.setVerticalTextPosition(SwingConstants.TOP);
    avecIconeDessous.setHorizontalTextPosition(SwingConstants.CENTER);
    conteneur.add(avecIconeDessous);

    JLabel etiquetteArial = new JLabel("En arial gras 16 bleu");
    Font arialGras16 = new Font("Arial", Font.BOLD, 16);
    etiquetteArial.setFont(arialGras16);
    etiquetteArial.setForeground(Color.BLUE);
    conteneur.add(etiquetteArial);

    JLabel iconeSansTexte = new JLabel(new ImageIcon("images"+File.separator+
                                                     new.png"));
    conteneur.add(iconeSansTexte);
  }
  public static void main(String[] args) {
    ExempleJLabel fenetre = new ExempleJLabel();
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

JEditorPane

Il s'agit d'un éditeur riche acceptant plusieurs formats. Si vous ne voulez afficher/éditer que du texte (avec de la mise en forme comme RTF) vous pouvez utiliser la sous-classe `JTextPane`. On peut initialiser un `JEditorPane` en précisant le format et le texte comme ci-dessous mais aussi fournir l'url du document. C'est un moyen simple d'afficher une page web. Par défaut l'édition est possible et l'utilisateur a la possibilité de modifier le document mais on peut interdire l'édition comme dans l'exemple ci-dessous via la méthode `setEditable(false)`. La méthode `getText()` retourne le texte du document.



Exemple

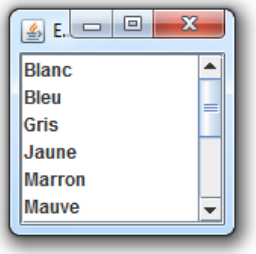
```
public class ExempleJEditorPane extends JFrame {
  public ExempleJEditorPane() {
    super("Exemple JEditorPane");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE)
    JEditorPane ep = new JEditorPane("text/html", "<html>Voici du texte html en"
          +" gras <br> en <font color=\"red\">rouge</font>, en <font face=\"Arial\""
          +"><u>arial soulign&eacute;</u></font>  et en <span style=\"font-size"
          +":22pt;\"> grand</span>.</html>");
    ep.setEditable(false); // l'utilisateur ne peut pas modifier
    System.out.println(ep.getText()); // affiche le texte sur la console
    this.getContentPane().add(ep);
  }
  public static void main(String[] args) {
    ExempleJEditorPane maFenetre = new ExempleJEditorPane();
    maFenetre.setSize(300,100);
    maFenetre.setVisible(true);
  }
}
```

- *Les listes et tableaux*

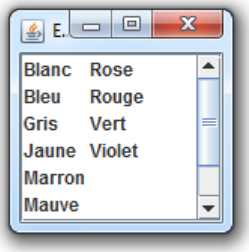
JList<E>

Initialisées avec un tableau d'objets, elles permettent à l'utilisateur de sélectionner un ou des valeurs dans une liste. Les listes pouvant être longues, on les utilise bien souvent dans un `JScrollPane` afin de pouvoir faire défiler la liste.

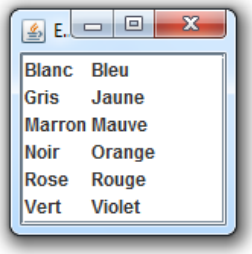
`setLoyaoutOrientation` permet de stipuler comment seront affichés les éléments de la liste. Par défaut elle est constituée d'une unique colonne (`JList.VERTICAL`) mais on peut les afficher sur plusieurs colonnes en remplissant la première colonne avant de passer à la seconde (`VERTICAL_WRAP`) ou en remplissant la première ligne avant de passer à la seconde (`HORIZONTAL_WRAP`).



VERTICAL

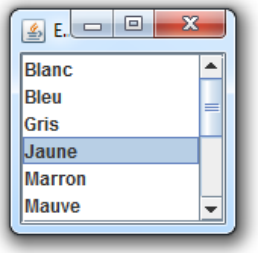


VERTICAL_WRAP

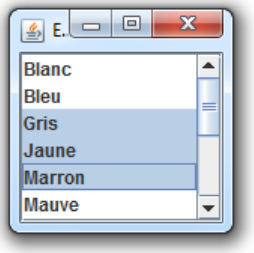


HORIZONTAL_WRAP

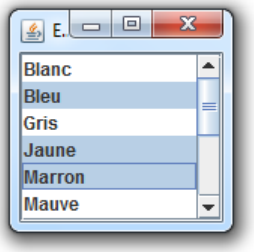
`setSelectionMode` permet de préciser le type de sélection désiré. Par défaut l'utilisateur peut sélectionner n'importe quel sous-ensemble d'items (`MULTIPLE_INTERVAL_SELECTION`) mais on peut également ne laisser choisir qu'une seule entrée (`ListSelectionModel.SINGLE_SELECTION`) ou ne laisser choisir qu'une plage continue de valeurs (`SINGLE_INTERVAL_SELECTION`).



SINGLE_SELECTION



SINGLE_INTERVAL_SELECTION



MULTIPLE_INTERVAL_SELECTION

On peut récupérer l'objet / les objets sélectionné(s) via les méthodes `E getSelectedValue()` / `List<E> getSelectedValuesList()`. On peut également effacer la sélection et tester si la sélection est vide via **`void`** `clearSelection()` et `boolean isSelectionEmpty()`.

Exemple

```
public class ExempleJList extends JFrame {
  public ExempleJList() {
    super("Exemple JList");
    String[] couleurs = {"Blanc", "Bleu", "Gris", "Jaune", "Marron",
            "Mauve", "Noir", "Orange", "Rose", "Rouge", "Vert", "Violet"};
    JList<String> listeCouleurs = new JList<String>(couleurs);
    listeCouleurs.setLayoutOrientation(JList.VERTICAL);
                        // ou VERTICAL_WRAP ou HORIZONTAL_WRAP
    listeCouleurs.setSelectionMode(
                          ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
                        //ou SINGLE_SELECTION ou SINGLE_INTERVAL_SELECTION
    JScrollPane scrollPane = new JScrollPane(listeCouleurs);
    this.getContentPane().add(scrollPane);
  }
}
```

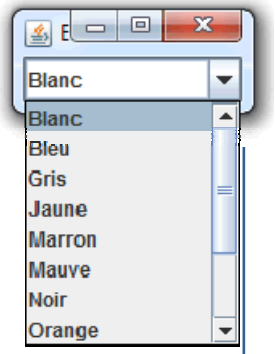
JComboBox<E>

Initialisées avec un tableau d'`Object`, elles permettent à l'utilisateur de sélectionner une valeur dans une liste verticale d'item. Elles offrent moins de souplesse que les `JList` mais sont compactes car seul l'item sélectionné est visibles. Pour consulter les autres items il faut ouvrir la liste au moyen de l'icône et éventuellement faire défiler.

On peut récupérer l'objet sélectionné via la méthode `Object getSelectedItem()`. On peut également au moyen de `setEditable(true)` (par défaut à false) laisser la possibilité à l'utilisateur de saisir une valeur plutôt que de simplement devoir en choisir une parmi celles proposées.

Exemple

```
public class ExempleJComboBox extends JFrame {
  public ExempleJComboBox() {
    super("Exemple JComboBox");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    String[] couleurs = {"Blanc", "Bleu", "Gris", "Jaune",
          "Marron","Mauve","Noir","Orange","Rose","Rouge",
          "Vert","Violet"};
    JComboBox<String> comboCouleurs=new JComboBox<String>(couleurs);
    comboCouleurs.setEditable(true);
    this.getContentPane().add( comboCouleurs);
  }
  …
}
```



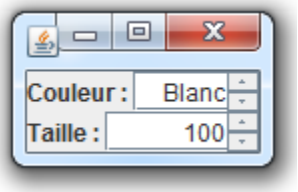
JSpinner

Tout comme les boîtes combo les `JSpinner` permettent de faire un choix parmi un ensemble fini de valeurs. Cependant, les `JSpinner` ne peuvent afficher que la valeur courante car elles n'empiètent pas sur les autres composants et ne peuvent afficher qu'une ligne. En conséquence, les `JSpinner` sont souvent utilisés lorsque la liste des valeurs est très grande et que la liste des valeurs potentielles est évidente. Comme avec les `JComboBox` il est possible de laisser à l'utilisateur la possibilité de taper directement la valeur.

Pour créer un `JSpinner` il faut tout d'abord créer un modèle comme ci-dessous (voire un `SPinnerDateModel`). Les boutons à droite permettent de se déplacer parmi les valeurs possibles.

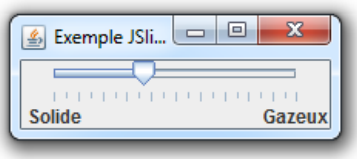
Exemple

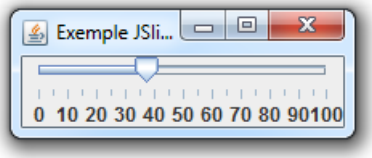
```
public class ExempleJSpinner extends JFrame {
  public ExempleJSpinner() {
    super("Exemple JComboBox");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container conteneur = this.getContentPane();
    conteneur.setLayout(new BoxLayout(conteneur, BoxLayout.Y_AXIS));

    JPanel couleur = new JPanel();
    couleur.setLayout(new BoxLayout(couleur, BoxLayout.X_AXIS));
    couleur.add(new JLabel("Couleur : "));
    String[] couleurs = {"Blanc", "Bleu", "Gris", "Jaune", "Marron", "Mauve",
                         "Noir", "Orange", "Rose", "Rouge", "Vert", "Violet"};
    SpinnerListModel modelCouleurs = new SpinnerListModel(couleurs);
    JSpinner spinnerCouleurs = new JSpinner( modelCouleurs );
    couleur.add(spinnerCouleurs);
    conteneur.add( couleur);

    JPanel taille = new JPanel();
    taille.setLayout(new BoxLayout(taille, BoxLayout.X_AXIS));
    taille.add(new JLabel("Taille : "));
    SpinnerNumberModel modelTaille = new SpinnerNumberModel(100, 0, 200, 1);
                                        //valeur courante, valeur min, valeur max et pas
    JSpinner spinnerTaille = new JSpinner( modelTaille );
    taille.add(spinnerTaille);
    conteneur.add( taille);
  }
  …
```



JSlider

Ils permettent la saisie graphique d'un nombre. On indique au constructeur la valeur minimale, la valeur maximale ainsi que la valeur initiale. Il est possible de modifier les graduations observables. Par défaut les étiquettes sont entières mais il est possible de préciser d'autres étiquettes comme ci-dessous. La fonction `getValue()` permet de connaître la valeur actuelle.





Exemple

```
public class ExempleJSlider extends JFrame {
  public ExempleJSlider() {
    super("Exemple JSlider");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 37);
    slider.setMajorTickSpacing(10); // on precise les graduations
    slider.setMinorTickSpacing(5);
    Hashtable etiquettes = new Hashtable();
    etiquettes.put( new Integer( 0 ), new JLabel("Solide") );
    etiquettes.put( new Integer( 100 ), new JLabel("Gazeux") );
    slider.setLabelTable( etiquettes ); // on précise les étiquettes
    slider.setPaintLabels(true); // on affiche les etiquettes
    slider.setPaintTicks(true); // on affiche les graduations
    this.getContentPane().add(slider);
  }
  …
```

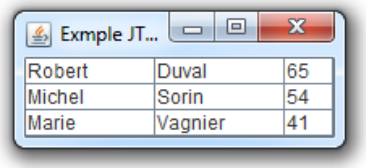
JTable

Destinés à afficher des composants sous forme de matrice, on les initialise en fournissant une matrice d'`Object` pour les données et un tableau d'`Object` pour les titres des colonnes. Par défaut l'affichage est purement textuel mais il est possible de modifier leur aspect en redéfinissant le renderer pour avoir tout type de composants dans les cases comme des boutons, des combobox ou encore des cases à cocher. On peut également redéfinir le model de données pour préciser finement quelles cellules sont éditables.

Par défaut les colonnes sont toutes de la même taille mais on peut spécifier des dimensions différentes comme dans l'exemple ci-dessous, les proportions sont alors respectées lors d'un redimensionnement.

Comme pour les `JList`, on peut préciser le type de sélections autorisées via `setSelectionMode` avec `ListSelectionModel.SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` ou `MULTIPLE_INTERVAL_SELECTION`.

`getSelectedRows()/getSelectedColumns()` retournent les index des lignes/colonnes sélectionnées.



Exemple

```
public class ExempleJTable extends JFrame {
  public ExempleJTable() {
    super ("Exmple JTable");
    Object[][] donnees = { {"Robert", "Duval", new Integer(65)},
                           {"Michel", "Sorin", new Integer(54)},
                           {"Marie", "Vagnier", new Integer(41)}};
    Object[] titres = {"Prénom", "Nom", "Age"};
    JTable table = new JTable(donnees, titres);
    table.getColumnModel().getColumn(0).setPreferredWidth(80);
    table.getColumnModel().getColumn(1).setPreferredWidth(80);
    table.getColumnModel().getColumn(2).setPreferredWidth(30);
    this.getContentPane().add(table);
  }
  public static void main(String[] args) {
    ExempleJTable fenetre = new ExempleJTable();
    fenetre.pack();
    fenetre.setVisible(true);
  …
```

- *Les boutons*

Les `JMenuItem`, `JCheckBoxMenuItem` et `JRadioButtonMenuItem` sont des boutons abordés plus loin dans la section `JMenu`.

JButton

Un bouton classique pouvant comporter du texte et/ou une image. Reportez vous à la section `JLabel` pour voir comment il est possible de positionner l'image par rapport au texte ou encore modifier la police d'écriture. On peut définir un raccourci clavier via `setMnemonic` comme dans l'exemple ci-dessous ou l'appui sur [Alt]+[O] équivaut à un clic sur le bouton « oui ». Un conteneur principal a la possibilité comme illustré ci-dessous de stipuler un bouton comme bouton par défaut. Ce dernier se distingue généralement par un cadre plus épais et l'appuie sur la touche entrée l'active.

Comme tout composant il peut-être disponible (enabled) ou non. Il est possible de définir une image pour les différents états du bouton via `setDisabledIcon`, `setPressedIcon`, `setSelectedIcon`, `setDisabledSelectedIcon` et `setRollOverIcon`.

Exemple

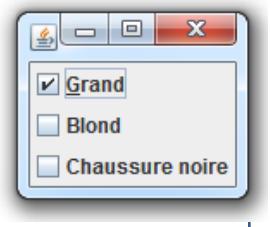

```
public class ExempleJButton extends JFrame{
  public ExempleJButton() {
    super("Exemple JButton");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container conteneur = this.getContentPane();
    conteneur.setLayout(new BoxLayout(conteneur, BoxLayout.X_AXIS));
    JButton oui = new JButton("Oui");
    JButton non = new JButton("Non");
    JButton peutEtre = new JButton("Peut-être",new ImageIcon("new.png"));
    oui.setMnemonic('o');
    non.setMnemonic('n');
    peutEtre.setMnemonic('p');
    JPanel pOui = new JPanel();
    JPanel pNon = new JPanel();
    JPanel pPeutEtre = new JPanel();
    pOui.add(oui);
    pNon.add(non);
    pPeutEtre.add(peutEtre);
    oui.setPreferredSize(new Dimension(80,45));
    non.setPreferredSize(new Dimension(80,45));
    peutEtre.setPreferredSize(new Dimension(200,45));
    conteneur.add(pOui);
    conteneur.add(pNon);
    conteneur.add(pPeutEtre);
    this.getRootPane().setDefaultButton(peutEtre);
  }
  public static void main(String[] args) {
    ExempleJButton fenetre = new ExempleJButton();
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

JToggleButton

Modélise un bouton à deux états. Utilisé lorsque l'on souhaite redéfinir un bouton à deux états à l'aspect personnalisé. On utilise alors `setIcon` et `setSelectedIcon` afin de préciser les images correspondant aux deux états. Ses classes filles, `JCheckBox` et `JRadioButton`, répondent cependant à la majorité des besoins et sont abordée ci-après.

JCheckBox

Bouton à deux états, ils correspondent à des cases à cocher. On retrouve les possibilités offertes aux `JButton` avec en plus la possibilité de préciser si la case est cochée ou non via `setSelected(boolean)`. `setSelected()` permet de consulter l'état.

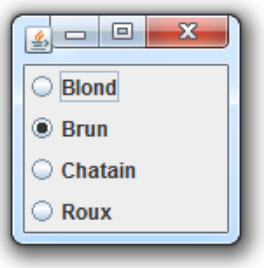


Exemple

```
public class ExempleJCheckBox extends JFrame {
  public ExempleJCheckBox() {
    super("Exemple JCheckBox");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container conteneur = this.getContentPane();
    conteneur.setLayout(new BoxLayout(conteneur, BoxLayout.Y_AXIS));
    JCheckBox grand = new JCheckBox("Grand");
    JCheckBox blond = new JCheckBox("Blond");
    JCheckBox chaussureNoire = new JCheckBox("Chaussure noire");
    grand.setMnemonic('g');
    grand.setSelected(true);
    conteneur.add(grand);
    conteneur.add(blond);
    conteneur.add(chaussureNoire);
  }
  public static void main(String[] args) {
    ExempleJCheckBox fenetre = new ExempleJCheckBox();
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

JRadioButton

Bouton à deux états généralement utilisé dans le cadre de choix exclusifs et donc en association avec une instance de `ButtonGroup` comme dans l'exemple ci-dessous. Ainsi, la sélection d'un item désélectionne les autres.

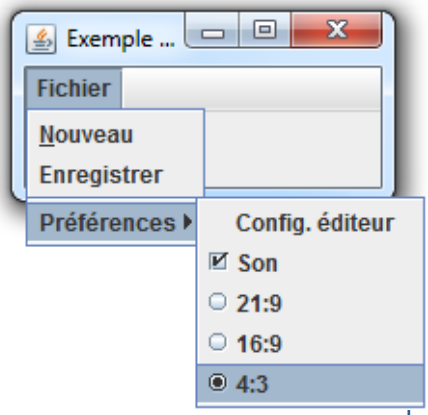


Exemple

```
public class ExempleJRadioButton extends JFrame {
  public ExempleJRadioButton() {
    super("Exemple JRadioButton");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container conteneur = this.getContentPane();
    conteneur.setLayout(new BoxLayout(conteneur,BoxLayout.Y_AXIS));
    ButtonGroup cheveux = new ButtonGroup();
    JRadioButton blond = new JRadioButton("Blond");
    JRadioButton brun = new JRadioButton("Brun");
    JRadioButton chatain = new JRadioButton("Chatain");
    JRadioButton roux = new JRadioButton("Roux");
    cheveux.add(blond);
    cheveux.add(brun);
    cheveux.add(chatain);
    cheveux.add(roux);
    cheveux.setSelected(brun.getModel(), true);
    conteneur.add(blond);
    conteneur.add(brun);
    conteneur.add(chatain);
    conteneur.add(roux);
  }
  public static void main(String[] args) {
    ExempleJRadioButton fenetre = new ExempleJRadioButton();
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

- *Les Menus*

Les applications ont souvent recours à une barre de menu afin de regrouper de façon compacte l'ensemble de leurs fonctionnalités regroupées sous forme d'arbre. A la racine on trouve une instance de `JMenuBar`, laquelle peut inclure des `JMenu`, `JPopupMenu`, `JSeparator`, `JMenuItem`, `JCheckBoxMenuItem` et `JRadioButtonItem`. Les trois derniers sont des boutons et doivent être gérés comme les autres boutons. Les `JMenu` peuvent à leur tour contenir d'autres composants de menu dont des `JMenu`.



Exemple

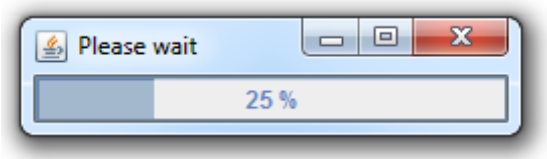
```
public class ExempleJMenu extends JFrame {
  public ExempleJMenu() {
    super("Exemple JMenu");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JMenuBar barreMenu = new JMenuBar();

    JMenu fichier = new JMenu("Fichier");
    JMenuItem nouveau = new JMenuItem("Nouveau");
    nouveau.setMnemonic('n');
    fichier.add(nouveau);
    JMenuItem enregistrer = new JMenuItem("Enregistrer");
    fichier.add(enregistrer);
    fichier.addSeparator();

    JMenu preferences = new JMenu("Préférences");
    preferences.add(new JMenuItem("Config. éditeur"));
    preferences.add(new JCheckBoxMenuItem("Son"));
    JRadioButtonMenuItem b219 = new JRadioButtonMenuItem("21:9");
    preferences.add(b219);
    JRadioButtonMenuItem b169 = new JRadioButtonMenuItem("16:9");
    preferences.add(b169);
    JRadioButtonMenuItem b43 = new JRadioButtonMenuItem("4:3");
    preferences.add(b43);
    ButtonGroup groupeFormat = new ButtonGroup();
    groupeFormat.add(b219);
    groupeFormat.add(b169);
    groupeFormat.add(b43);
    fichier.add(preferences);
    barreMenu.add(fichier);
    this.setJMenuBar(barreMenu);
  }
  public static void main(String[] args) {
    ExempleJMenu fenetre = new ExempleJMenu();
    fenetre.setSize(200,100);
    fenetre.setVisible(true);
  }
}
```

- *Barre de progression*

JProgressBar

Elles sont utilisées lors de traitements longs afin de donner à l'utilisateur une estimation du travail effectué. Sans elles l'utilisateur pourrait croire que l'ordinateur est victime d'une erreur ou qu'il ne fait rien. Le constructeur prend en paramètre deux valeurs entières, la première étant l'entier correspondant à un travail non débuté et la seconde correspondant à un travail achevé. La méthode à connaître est `setValue(int i)` qui permet de spécifier la valeur actuelle. De plus, `setStringPainted` précise si la valeur doit ou non être affichée.

Dans l'exemple ci-dessous on simule un travail important de 50 secondes par un `Timer` qui va tous les 500 millisecondes appeler la méthode `actionPerformed` laquelle incrémente de 1% le travail effectué et met à jour la barre de progression.



Exemple

```
public class ExempleJProgressBar extends JFrame implements ActionListener {
  private int travailEffectue=0;
  private JProgressBar avancement;
  private Timer t;

  public ExempleJProgressBar() {
    super("Please wait");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.t = new Timer(500, this);
    avancement = new JProgressBar(0,100);
    avancement.setStringPainted(true);
    this.getContentPane().add(avancement);
    this.t.start();
  }
  public void actionPerformed(ActionEvent arg0) {
    this.travailEffectue++;
    this.avancement.setValue(this.travailEffectue);
    if (this.travailEffectue==100) {
      this.t.stop();
    }
  }
  public static void main(String[] args) {
    ExempleJProgressBar fenetre = new ExempleJProgressBar();
    fenetre.setSize(250,60);
    fenetre.setVisible(true);
  }
}
```

3.3. Les boites de dialogue

Comme nous l'avons évoqué, les boîtes de dialogue sont des fenêtres secondaires affichées pour informer ou questionner. Elles peuvent être modales, et dans ce cas il est impossible de passer à une autre fenêtre de l'application tant qu'on n'a pas fermé la boîte de dialogue.

Si vos besoins vont au-delà des boîtes de dialogues prédéfinies listées ci-dessous vous pouvez créer vos propres boîtes de dialogue en spécialisant `JDialog`.

3.3.1. JOptionPane

La classe `JOptionPane` regroupe un ensemble de boîtes de dialogues prédéfinies pour informer, demander confirmation ou demander une valeur. :

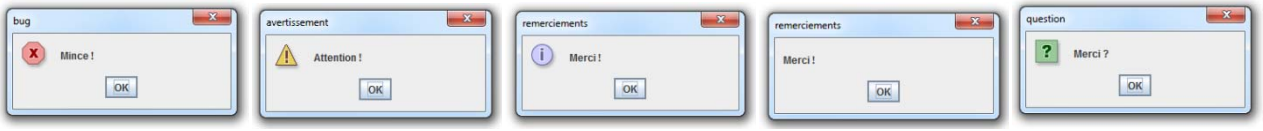
- Informer : `JOptionPane.showMessageDialog(…)`
- Demander confirmation : `JOptionPane.showConfirmDialog(…)`
- Demander une valeur : `JOptionPane.showInputDialog(…)`

• *Informer : JOptionPane.showMessageDialog(...)*

La signature est `static void showMessageDialog(Component mere, Object message, String titre, int typeMessage, Icon icone)` les deux derniers paramètres étant optionnels.

- `mere` doit être une référence désignant la fenêtre appelante.
- `message` correspond au message à afficher.
- `titre` est le titre de la boîte de dialogue.
- `typeMessage` peut prendre les valeurs `ERROR_MESSAGE` (), `WARNING_MESSAGE` (), `INFORMATION_MESSAGE` (), `PLAIN_MESSAGE` ou `QUESTION_MESSAGE` ().
- `icone` est l'image que l'on souhaite afficher (par défaut elle correspond à `typeMessage`).

Le but n'étant que d'informer, il n'y a pas de valeur retour.



Exemple

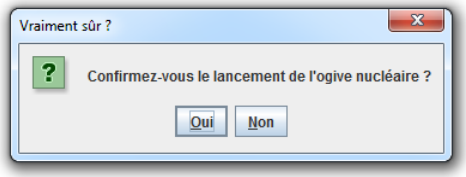
```
public class ExempleJOptionPane extends JFrame {
  public ExempleJOptionPane() {
    super("Exemple JOptionPane");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
 JOptionPane.showMessageDialog(this,"Mince !","bug", JOptionPane.ERROR_MESSAGE);
 JOptionPane.showMessageDialog(this,"Attention !","",JOptionPane.WARNING_MESSAGE);
 JOptionPane.showMessageDialog(this,"Merci","thanks",JOptionPane.INFORMATION_MESSAGE);
 JOptionPane.showMessageDialog(this,"Merci","thanks",JOptionPane.PLAIN_MESSAGE);
 JOptionPane.showMessageDialog(this,"Merci?","question",JOptionPane.QUESTION_MESSAGE);
  }
  public static void main(String[] args) {
    ExempleJOptionPane fenetre = new ExempleJOptionPane();
    fenetre.setSize(300,200);
    fenetre.setVisible(true);
  }
}
```

• *Demander confirmation : JOptionPane.showConfirmDialog(...)*

On peut ouvrir une telle boîte via la méthode **`static int showConfirmDialog(`** `Component mere, Object message, String titre, int options, int typeMessage, Icon icone)` les deux derniers paramètres étant optionnels.

- voir `showMessageDialog` pour `mere`, `message`, `titre`, `typeMessage` et `icone`.
- `options` indique les choix possibles (donc les boutons présents) et peut prendre les valeurs `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, or `OK_CANCEL_OPTION`.

La valeur retournée est `YES_OPTION`, `OK_OPTION`, `NO_OPTION`, `CANCEL_OPTION` ou `CLOSED_OPTION` selon l'action de l'utilisateur.



Exemple

```
public class ExempleJOptionPane extends JFrame {
  public ExempleJOptionPane() {
    super("Exemple JOptionPane");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    int choix = JOptionPane.showConfirmDialog(this, "Confirmez-vous le"
              +" lancement de l'ogive nucléaire ?", "Vraiment sûr ?",
              JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
    switch (choix) {
      case JOptionPane.YES_OPTION : System.out.println("bye bye...");break;
      case JOptionPane.NO_OPTION  : System.out.println("Ouf !");break;
      case JOptionPane.CANCEL_OPTION: System.out.println("J'annule");break;
      case JOptionPane.CLOSED_OPTION: System.out.println("pas de reponse");
    }
  }
  …
}
```

• *Demander une valeur : JOptionPane.showInputDialog (...)*

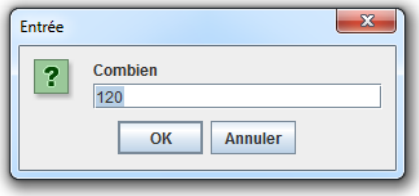
Avec champ de saisie :

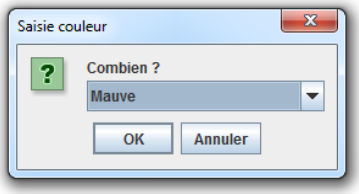
```
showConfirmDialog(                Object message)
showConfirmDialog(Component mere,Object message)
showConfirmDialog(                Object message,                    Object valInit)
showConfirmDialog(Component mere,Object message,String titre,int typeMessage)
```

Avec combobox :

```
showConfirmDialog(Component mere,Object message,String titre,int typeMessage,
                  Icon icone, Object[] vals, Object valInit)
```

- voir `showMessageDialog` pour `mere`, `message`, `titre`, `typeMessage` et `icone`.
- `vals` regroupe les valeurs possibles listées dans la combobox.
- `valInit` est la valeur initiale.

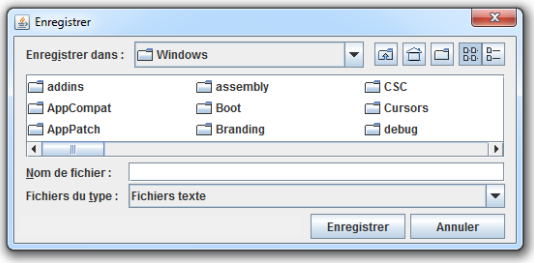




Exemple

```
public class ExempleJOptionPane extends JFrame {
  public ExempleJOptionPane() {
    super("Exemple JOptionPane");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    String rep = JOptionPane.showInputDialog("Combien", "120");
    Object[] couleurs = {"Blanc", "Bleu", "Gris", "Jaune", "Marron", "Mauve",
                         "Noir", "Orange", "Rose", "Rouge", "Vert", "Violet"};
    Object rep2 = JOptionPane.showInputDialog(this, "Laquelle ?", "Saisie couleur",
                           JOptionPane.QUESTION_MESSAGE, null, couleurs, "Mauve" );
    System.out.println(rep+" et "+rep2);
  }
  …
}
```

3.3.2. JFileChooser



Utiles pour que l'utilisateur puisse spécifier un nom de fichier ou de répertoire en parcourant l'arborescence du système de fichier. Le constructeur ne prend qu'un seul paramètre (optionnel qui plus est) correspondant au répertoire à afficher à l'ouverture de la boite de dialogue.

Les deux méthodes principales sont `showOpenDialog` et `showSaveDialog` qui affichent des boites de dialogues permettant respectivement d'indiquer un fichier à ouvrir et de préciser le nom sous lequel un fichier doit être enregistré.

Il est possible de préciser que la sélection doit correspondre uniquement à un répertoire comme ci-dessous en utilisant `setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY)` ou de laisser sélectionner à la fois des fichiers et des répertoires (`FILES_AND_DIRECTORIES`). Par défaut, seuls des fichiers peuvent être choisis (`FILES_ONLY`).

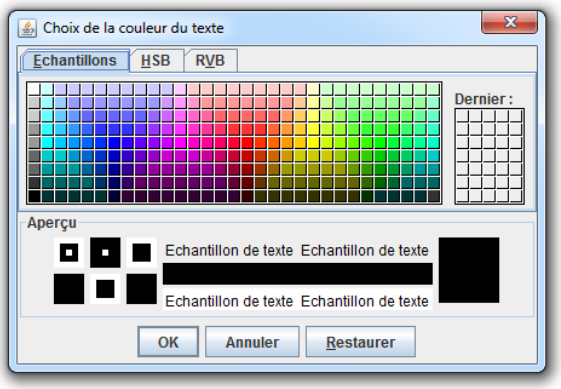
Enfin, par défaut tous les types de fichiers sont affichés mais on peut rajouter d'autres filtres pour n'afficher par exemple que les fichiers contenant du texte comme ci-dessous. Pour cela, il faut créer une spécialisation de la classe `FileFilter` puis faire appel à `addChoosableFileFilter` avec une instance du nouveau filtre en paramètre. On peut également retirer le filtre correspondant à « tous les types de fichiers » via `setAcceptAllFileFilterUsed(false)`.

La valeur retournée correspond au bouton utilisé pour fermer la boite de dialogue : `JFileChooser.APPROVE_OPTION` si l'utilisateur clique sur [Ouvrir]|[Enregistrer], `JFileChooser.CANCEL_OPTION` s'il clique sur [Annuler] ou ferme la boite avec l'icône en forme de croix et `JFileChooser.ERROR_OPTION` en cas d'erreur. On peut consulter la sélection faite par l'utilisateur via les méthodes `getSelectedFile` et `getSelectedFile` (pour les sélections multiples).

Exemple

```
public class ExempleJFileChooser extends JFrame {
  public ExempleJFileChooser() {
    super("Exemple JFileChooser");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JFileChooser choixNom = new JFileChooser("c:"+File.separator+"windows");
    choixNom.addChoosableFileFilter(new FiltreTexte());
    choixNom.setAcceptAllFileFilterUsed(false);
    int bouton = choixNom.showSaveDialog(this);
    if (bouton == JFileChooser.APPROVE_OPTION) {
      System.out.println("vous voulez enregistrer sous le nom "
                         +choixNom.getSelectedFile());
    }
    JFileChooser choixRep = new JFileChooser("c:"+File.separator+"windows");
    choixRep.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
    int bouton2 = choixRep.showOpenDialog(this);
    if (bouton2 == JFileChooser.APPROVE_OPTION) {
      System.out.println("vous voulez ouvrir "+choixRep.getSelectedFile());
    }
  }
  public static void main(String[] args) {
    ExempleJFileChooser fenetre = new ExempleJFileChooser();
    …
  }
}
class FiltreTexte extends FileFilter {
  public boolean accept(File f) {
    return f.isDirectory()
        || f.getName().toLowerCase().endsWith("txt")
        || f.getName().toLowerCase().endsWith(".rtf");
  }
  public String getDescription() {
    return "Fichiers texte";
  }
}
```

3.3.3. JColorChooser



Comme leur nom l'indique elles permettent à l'utilisateur de choisir une couleur. La méthode à connaître est `static Color showDialog( Component mere, String titre, Color couleurInitiale)` qui affiche la boite de dialogue avec `titre` pour titre et `couleurInitiale` pour couleur initiale. La valeur retournée est la couleur choisie (**`null`** si l'utilisateur clique sur [Annuler] ou ferme la boîte avec l'icône en forme de croix).

Exemple

```
public class ExempleJColorChooser extends JFrame {
  public ExempleJColorChooser() {
    super("Exemple JColorChooser");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JColorChooser choixCouleur = new JColorChooser();
    Color couleur = choixCouleur.showDialog(this, "Choix de la couleur"
                                          +" du texte", Color.BLACK);
    JLabel texte = new JLabel("Mon texte coloré");
    if (couleur!=null)  {
      texte.setForeground(couleur);
    }
    this.getContentPane().add(texte);
  }
  public static void main(String[] args) {
    ExempleJColorChooser fenetre = new ExempleJColorChooser();
    fenetre.setSize(300,50);
    fenetre.setVisible(true);
  }
}
```

4. Les gestionnaires de placement (Layout manager)

Il est possible de réaliser une interface en précisant les tailles et les positions exactes de chaque composant, mais n'oublions pas que Java est multiplateforme. Votre interface serait adaptée à votre système mais pourrait être totalement inutilisable sur un autre ordinateur. De plus, les positions et les tailles doivent être mises à jour en cas de redimensionnement de la fenêtre ce qui n'est pas simple si on gère soi même la disposition des composants.

Pour remédier à cela, nous nous appuierons sur des gestionnaires de placement (Layout Manager). C'est eux qui gèreront le positionnement relatif des composants de façon portable (multiplateforme). La disposition et les tailles seront ainsi gérées par ces gestionnaires qui agissent de façon dynamique, réalisant une mise à jour après chaque redimensionnement du conteneur. Le gestionnaire définit les positions est les tailles 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`.

4.1. Caractéristiques utilisées par les gestionnaires de placement

- **`public void setAlignmentX(float pos)`** permet de préciser l'**alignement horizontal** relatif que le composant aimerait avoir. On utilise généralement les constantes prédéfinies `Component.LEFT_ALIGNMENT`, `Component.CENTER_ALIGNMENT` et `Component.RIGHT_ALIGNMENT`.
- **`public void setAlignmentY(float pos)`** permet de préciser l'**alignement vertical** relatif que le composant aimerait avoir. On utilise généralement les constantes prédéfinies `Component.BOTTOM_ALIGNMENT` et `Component.TOP_ALIGNMENT`.
- **`public void setPreferredSize(new Dimension(x,y))`** permet de préciser la **taille préférée**, celle que le composant aimerait avoir. Le conditionnel est souligné car d'après la javadoc les gestionnaires de placement sont libres de respecter ou non ces préférences.
- **`public void setMinimumSize(new Dimension(x,y))`** permet de préciser la **taille minimale** que doit avoir le composant pour pouvoir être utilisable. Les gestionnaires respectent généralement davantage cette information que celle de la taille préférée.
- **`public void setMaximumSize(new Dimension(x,y))`** permet de préciser la **taille maximale** au-delà de laquelle le composant n'est plus utilisable. Lors d'un agrandissement du conteneur si la taille maximale d'un composant est atteinte le gestionnaire agrandira les autres ou interdira l'agrandissement si tous sont à leur taille maximale.

4.2. Position et taille de la fenêtre principale

Chaque ajout d'un composant dans une fenêtre modifie sa `preferredSize`, généralement (l'impact dépend du layout manager) en agrandissant le conteneur au fur et à mesure des ajouts. Les dimensions initiales de la fenêtre varient selon que le dernier appel avant de la rendre visible soit `pack` ou `setSize`.

- La méthode `pack()` appliquée à la fenêtre va forcer à recalculer les dimensions et les positions en fonction des caractéristiques des composants et des layout. Généralement cela aboutit à obtenir pour la fenêtre les dimensions les plus compactes en s'appuyant sur les `preferredSize` des composants.
- `setSize(largeur, hauteur)` fixe quant à elle les dimensions de la fenêtre aux valeurs fournies en pixels. Cela rend votre application moins portable car notamment la taille des pixels diffère d'un écran à l'autre. Si on appelle `setSize` alors que la fenêtre est déjà affichée il convient d'appeler `validate()` afin de rafraichir le positionnement du contenu.

Bien que l'usage de `pack` soit recommandé pour une plus grande portabilité il arrive de devoir utiliser `setSize`. C'est le cas notamment si on souhaite occuper une portion déterminée de l'écran (tout l'écran, la moitié gauche, la moitié droite, …). L'exemple ci-dessous fait une utilisation conjointe de `setSize` et `setLocation` afin que la fenêtre occupe la moitié droite de l'écran.

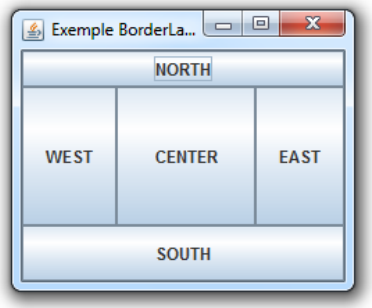
Exemple

```
public class ExempleSetSize {
  public static void main(String[] args) {
    JFrame fenetre = new JFrame();
    Dimension tailleEcran = Toolkit.getDefaultToolkit().getScreenSize();
    fenetre.setSize((int)(tailleEcran.getWidth()/2), (int)tailleEcran.getHeight());
    fenetre.setLocation((int)(tailleEcran.getWidth()/2), 0);
    fenetre.setVisible(true);
  }
}
```

4.3. Les principaux gestionnaires de placement

Une fois de plus nous ne seront pas exhaustifs, d'autant qu'on peut trouver sur internet des gestionnaires variés libres de droits et que nous avons la possibilité de définir nos propres gestionnaires. Nous nous focaliserons donc sur les gestionnaires de `awt` et `swing` les plus utilisés.

4.3.1. BorderLayout



Avec ce gestionnaire les composants du conteneur sont répartis en cinq zones appelées `NORTH`, `EAST`, `SOUTH`, `WEST` et `CENTER`. C'est le gestionnaire par défaut des `contentPane`. Lors de l'ajout d'un composant on précise pour deuxième paramètre la zone dans laquelle il doit être ajouté.

La politique de placement est la suivante :

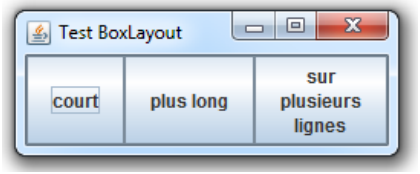
- Les composants des zones `NORTH` et `SOUTH` ont pour hauteur leur hauteur préférée si l'espace est suffisant. Ils s'étalent sur toute la largeur du conteneur.
- Les composants des zones `WEST` et `EAST` ont pour largeur leur largeur préférée si l'espace est suffisant. Ils s'étalent sur toute la hauteur restant entre les zones nord et sud.
- Le composant central occupe tout l'espace restant.

`BorderLayout` est très pratique pour aligner des composants sur l'un des cotés et faire qu'un composant remplisse au mieux le reste de l'espace. En plaçant des conteneurs aux gestionnaires appropriés dans ses différentes zones on peut parvenir à la réalisation d'interfaces complexes. Il est le gestionnaire par défaut du `contentPane` des `JFrame`.

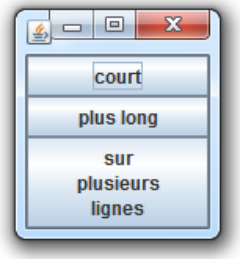
Exemple

```
public class ExempleBorderLayout {
  public static void main(String[] args) {
    JFrame fenetre = new JFrame("Exemple BorderLayout");
    Container conteneur = fenetre.getContentPane();
    // conteneur.setLayout(new BorderLayout()) (pas besoin, par defaut)
    JButton bouton = new JButton("NORTH");
    conteneur.add(bouton, BorderLayout.NORTH);// ou PAGE_START
    bouton = new JButton("EAST");
    conteneur.add(bouton, BorderLayout.EAST);// ou LINE_END
    bouton = new JButton("SOUTH");
    bouton.setPreferredSize(new Dimension(10,40));
    conteneur.add(bouton, BorderLayout.SOUTH);// ou PAGE_END
    bouton = new JButton("WEST");
    conteneur.add(bouton, BorderLayout.WEST);// ou LINE_START
    bouton = new JButton("CENTER");
    bouton.setPreferredSize(new Dimension(100,100));
    conteneur.add(bouton, BorderLayout.CENTER);
  …
```

4.3.2. BoxLayout

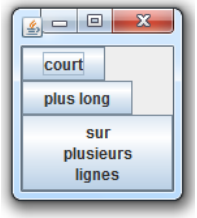


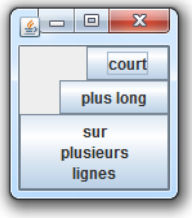
`BoxLayout` permet d'aligner des composants soit verticalement, soit horizontalement. Le constructeur prend deux paramètres, le premier étant le conteneur et le deuxième l'axe (`BoxLayout.X_AXIS` ou `BorderLayout.Y_AXIS`). Si par exemple on choisit `Y_AXIS` les composants seront disposés de haut en bas en respectant l'ordre d'insertion dans le conteneur.



Avec un axe vertical la politique de placement est la suivante :

- Les composants ont une hauteur correspondant à leur hauteur préférée.
- Le gestionnaire tente de donner aux composants la largeur du plus large des composants. Si le composant ne peut pas être étiré (`maximumSize` trop réduite) le gestionnaire positionne le composant horizontalement en tenant compte de son `X_ALIGNMENT`.



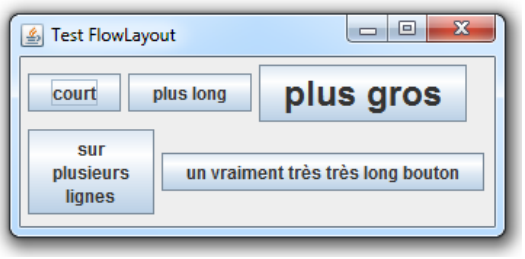


Les images ci-contre illustrent ce que donnerait l'exemple ci-dessous si on ne spécifiait pas des tailles maximales suffisantes. La différence entre les deux tient à la spécification de `Component.RIGHT_ALIGNMENT` pour alignement horizontal des composants pour l'image de droite.

Exemple

```
public class ExempleBoxLayout {
  public static void main(String[] args) {
    JFrame fenetre = new JFrame("Test BoxLayout");
    fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container conteneur = fenetre.getContentPane();
    conteneur.setLayout(new BoxLayout(conteneur, BoxLayout.Y_AXIS));

    JButton btn1 =new JButton("court");
    btn1.setAlignmentX(Component.RIGHT_ALIGNMENT);
    btn1.setMaximumSize(new Dimension(500,40));
    conteneur.add(btn1);

    JButton btn2 =new JButton("plus long");
    btn2.setMaximumSize(new Dimension(500,40));
    btn2.setAlignmentX(Component.RIGHT_ALIGNMENT);
    conteneur.add(btn2);

    JButton btn3 =new JButton("<html><p align=\"center\">sur<br>"
                              +"plusieurs<br>lignes</p></html>");
    btn3.setAlignmentX(Component.RIGHT_ALIGNMENT);
    conteneur.add(btn3);

    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

4.3.3. FlowLayout



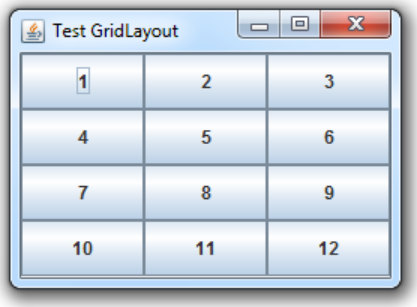
`FlowLayout` place les composants en ligne de gauche à droite et passe à une autre ligne une fois la ligne remplie. Avec ce gestionnaire les composants ont leur taille préférée.

Il est possible de préciser un alignement (`FlowLayout.LEFT`, `RIGHT` ou `CENTER`) en paramètre du constructeur. Par défaut, l'alignement est `CENTER`. Le layout manager par défaut du `JPanel` est un `FlowLayout`.

Exemple

```
public class ExempleFlowLayout {
  public static void main(String[] args) {
    JFrame fenetre = new JFrame("Test FlowLayout");
    fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container conteneur = fenetre.getContentPane();
    conteneur.setLayout(new FlowLayout(FlowLayout.LEFT));
    conteneur.add(new JButton("court"));
    conteneur.add(new JButton("plus long"));
    JButton btn3 =new JButton("plus gros");
    btn3.setFont(new Font("Arial",Font.BOLD,24));
    conteneur.add(btn3);
    conteneur.add(new JButton("<html><p align=\"center\">sur<br>"
                              +"plusieurs<br>lignes</p></html>"));
    conteneur.add(new JButton("un vraiment très très long bouton"));
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

4.3.4. GridLayout



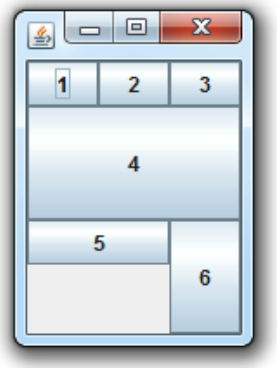
`FlowLayout` place les composants dans une grille. Toutes les cases ont exactement la même taille et le gestionnaire agrandit chaque composant de sorte à ce qu'il occuper l'intégralité de sa case.

Les cases ayant la même taille, les dimensions préférées du conteneur sont calculées en s'appuyant sur les composants les plus grands de la grille. Si le conteneur est redimensionné, la taille des cases est mise à jour pour occuper tout l'espace.

Exemple

```
public class ExempleGridLayout {
  public static void main(String[] args) {
    JFrame fenetre = new JFrame("Test GridLayout");
    fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container conteneur = fenetre.getContentPane();
    conteneur.setLayout(new GridLayout(4,3));// 4 lignes de 3 colonnes
    for (int i=1; i<=12; i++) {
      conteneur.add(new JButton(i+""));
    }
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

4.3.5. GridBagLayout



`GridBagLayout` permet une disposition plus souple car on peut disposer les composants sur une grille en ayant la possibilité de laisser un composant occuper plusieurs cases. 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, mais aussi la précision de contraintes. Les composants peuvent bien sûr être des conteneurs avec leur propre layout ce qui permet de réaliser des IHM très complexes.

L'exemple ci-dessous illustre l'utilisation des contraintes. Attention, on utilise ici une seule instance de contrainte qu'on réutilise en ne modifiant que ce qui est nécessaire. Cette utilisation d'une unique instance est sujette à erreur car on a vite fait d'oublier de mettre à jour l'une des propriétés. Les caractéristiques modifiables des contraintes sont :

- `gridx` et `gridy` précisent la colonne et la ligne à l'intersection desquelles le coin haut gauche du composant sera placé. Il ne s'agit pas de coordonnées en pixels mais d'index de matrice.
- `gridwidth` et `gridheight` indiquent le nombre de colonnes et de lignes que doit occuper le composant.
- `fill` indique comment il faut agrandir le composant quand la taille de celui-ci est inférieure à l'espace qui lui est alloué. Les valeurs possibles sont `NONE` (par défaut), `HORIZONTAL` pour que le composant s'étire horizontalement sans changer sa hauteur, `VERTICAL` pour qu'il s'étire verticalement et `BOTH` pour qu'il s'étire dans les deux directions.
- `ipadx` et `ipady` indiquent combien de pixels doivent être ajoutés à la taille du composant. La largeur du composant sera au minimum sa taille minimale plus `2*ipadx` pixels, et sa hauteur sera d'au moins sa hauteur minimal plus `2*ipady` pixels.
- `insets` indique la marge extérieure à ajouter au composant, donc l'espace qu'il faut laisser entre son contour et les bords de son emplacement. Par défaut il n'y a pas de marge.
- `anchor` précise comment est rattaché le composant à son emplacement. Les valeurs possibles sont `CENTER` (par défaut), `PAGE_START` (collé en haut), `PAGE_END` (collé en bas), `LINE_START` (collé à gauche), `LINE_END` (collé à droite), `FIRST_LINE_START` (collé par le coin supérieur gauche), `FIRST_LINE_END`, `LAST_LINE_START` et `LAST_LINE_END`.
- `weightx` et `weighty` indiquent comment l'espace supplémentaire est réparti entre les lignes et les colonnes. L'effet est visible surtout lors du redimensionnement du conteneur. Leur valeur est entre `0.0` (par défaut) et `1.0`. Si toutes les cellules ont `0.0` tous les composants seront agglutinés au centre car l'espace additionnel sera placé autour de la grille. Le poids d'une colonne est le plus grand poids parmi ceux de ses composants. Plus une colonne a un fort poids, plus elle bénéficiera d'espace.

Exemple

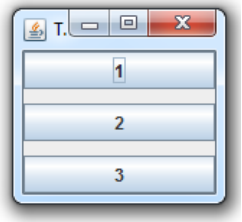
```
public class ExempleGridBagLayout {
  public static void main(String[] args){
    JFrame fen = new JFrame();
    Container cont=fen.getContentPane();
    cont.setLayout(new GridBagLayout());
    GridBagConstraints c = new
                    GridBagConstraints();
    c.fill=GridBagConstraints.HORIZONTAL;
    c.gridx = 0;
    c.gridy = 0;
    c.weightx=0.33;
    cont.add(new JButton("1"), c);
    c.gridx = 1;
    cont.add(new JButton("2"), c);
    c.gridx = 2;
    cont.add(new JButton("3"), c);
    c.gridx = 0;
    c.gridy = 1;
    c.gridwidth=3;
    c.ipady = 40;
    c.fill = GridBagConstraints.BOTH;
    cont.add(new JButton("4"), c);
    c.gridx = 0;
    c.gridy = 3;
    c.gridwidth=2;
    c.ipady = 0;
    cont.add(new JButton("5"), c);
    c.gridx = 2;
    c.gridy = 3;
    c.gridheight=2;
    c.ipady = 40;
    cont.add(new JButton("6"), c);
    fen.pack();
    fen.setVisible(true);
  }
```

4.4. Des composants invisibles

La classe `Box` permet de créer des composants invisibles très pratiques pour espacer les autres composants. Faisons le tour de ses méthodes.

4.4.1. Box.createRigidArea()

Cette méthode crée un composant invisible de dimension fixe. Il est utilisé pour créer un espace de dimension fixe entre deux composants. Dans l'exemple ci-dessous les trois boutons seraient collés sans l'ajout des deux boites rigides. En rajoutant des espacements de 10 pixels de haut on obtient l'interface ci-contre.



Exemple

```
public class ExempleBoxCreateRigidArea {
  public static void main(String[] args) {
    JFrame fenetre = new JFrame("Test Box");
    Container conteneur = fenetre.getContentPane();
    conteneur.setLayout(new BoxLayout(conteneur, BoxLayout.Y_AXIS));
    JButton btn1 =new JButton("1");
    btn1.setMargin(new Insets(2,60,2,60));// N, W, S, E
    conteneur.add(btn1);
    conteneur.add(Box.createRigidArea(new Dimension(0,10)));
    JButton btn2 =new JButton("2");
    btn2.setMargin(new Insets(2,60,2,60));
    conteneur.add(btn2);
    conteneur.add(Box.createRigidArea(new Dimension(0,10)));
    JButton btn3 =new JButton("3");
    btn3.setMargin(new Insets(2,60,2,60));
    conteneur.add(btn3);
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
```

4.4.2. Box.createHorizontalGlue() et Box.createVerticalGlue()

Il faut imaginer les composants créés comme un liquide transparent qui s'étend pour occuper toute la place qu'on lui offre. Ils sont utilisés pour indiquer l'endroit où l'on souhaite que l'espace additionnel soit placé. Si dans l'exemple ci-dessus on agrandit le conteneur en hauteur, l'espace additionnel s'ajoute après (donc ici en dessous) les composants rigides (image de gauche). Si on remplace les appels `Box.createRigidArea(new Dimension(0,10))` par `Box.createVerticalGlue()` l'espace additionnel se répartit entre les deux zones gluantes (image de droite).

4.4.3. Box.Filler

Il s'agit d'un composant transparent pour lequel on peut préciser des dimensions minimales, maximales et préférées comme dans l'exemple ci-dessous.

```
…
Dimension min  = new Dimension(20, 40);
Dimension pref = new Dimension(20, 40);
Dimension max  = new Dimension(2000, 2000);
Conteneur.add(new Box.Filler(min, pref, max));
…
```

5. Evènements et Listeners

5.1. Le principe

La plupart des composants sont susceptibles de recevoir une action de la part de l'utilisateur. On pense surtout au bouton sur lequel on peut cliquer ou au champ de saisie dans lequel on peut écrire mais quasi tous sont sensibles aux mouvements de la souris et aux frappes clavier. Ces composants sont des ***sources*** d'évènements. Par ses actions sur ces composants l'utilisateur entraine la création d'***évènements*** qui ne sont rien d'autre que des objets qui décrivent l'action qui vient de se produire.

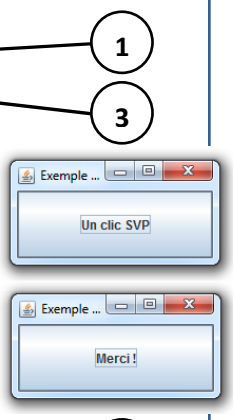
La plupart des évènements sont créés en pure perte car aucun code n'est prévu pour les traiter. C'est le cas pour tous les squelettes d'interfaces que nous avons créés jusqu'à présent. On a placé des boutons, l'utilisateur a pu cliquer sur ces boutons, des évènements correspondant à ces clic ont été créés, mais on n'a rien fait de ces évènements.

Pour que des évènements soient traités ont doit créer des ***écouteurs***, des objets qui se sont abonnés à une source d'évènements. Mais pour s'abonner à une source l'écouteur doit disposer d'un code pour traiter les différents types d'actions émises par la source. A chaque type d'évènement `xxxEvent` correspond une interface `xxxListener` regroupant les en-têtes des méthodes que l'écouteur doit s'engager à implémenter pour pouvoir s'abonner.

Prenons l'exemple d'un `JButton`. Il est une source d'évènements de type `ActionEvent`. Pour réagir aux clics sur ce bouton nous pouvons créer une classe implémentant `ActionListener`, et abonner une instance de cette classe à notre `JButton`. C'est ce qui est fait dans l'exemple ci-dessous.

Exemple

```
public class ExempleActionListener extends JFrame {
  public ExempleActionListener() {
    super("Exemple ActionListener");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JButton monBouton = new JButton("Un clic SVP");
    monBouton.addActionListener(new MonActionListener());
    this.getContentPane().add(monBouton);
  }
  public static void main(String[] args) {
    ExempleActionListener fenetre = new ExempleActionListener();
    fenetre.setSize(200,100);
    fenetre.setVisible(true);
  }
}
class MonActionListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    ((JButton)e.getSource()).setText("Merci !");
  }
}
```



Détaillons ce programme :

1- Le constructeur de `ExempleActionListener` crée les composants et les agence. L'interface est ici minimaliste puisque réduite à un unique bouton placé dans une `JFrame`.

2- La classe privé `MonActionListener` implémente l'interface `ActionListener` et s'engage donc à fournir une méthode d'en-tête `public void actionPerformed(ActionEvent e)` réagissant aux évènements de type `ActionEvent`. Le code utilise la méthode `getSource` de l'évènement pour obtenir une référence sur l'objet émetteur et changer son texte par `"Merci !"`.

3- On crée un objet receveur de type `MonActionListener` et on l'abonne à la source d'évènements de type `ActionEvent` de `monBouton` via la méthode `addActionListener`.

Un clic sur le bouton va générer un évènement de type `ActionEvent` qui sera passé en paramètre lors de l'appel de la `actionPerformed` de l'écouteur qui s'était abonné auprès du bouton.

Notez le `add` (et non le `set`) dans `addActionListener`. Plusieurs écouteurs peuvent s'abonner à la même source, bien que ce cas de figure soit rare. La source enverra alors l'évènement à chacun des écouteurs abonnés.

Regardons un deuxième exemple basé sur une interface comportant plusieurs méthodes et qui montre comment l'action sur un composant peut en modifier un autre. Le but ici est d'afficher une image dans un `JLabel` et de réagir à chaque mouvement de souris sur cette image afin d'afficher dans un autre `JLabel` la position du curseur. C'est un comportement observable dans presque tous les logiciels de traitement d'image afin de connaitre précisément les coordonnées du pixel désigné par la souris.

Les méthodes à implémenter pour réagir à un déplacement de souris sont précisées dans l'interface `MouseMotionListener`. L'écouteur aura accès au `JLabel` de l'image via la méthode `getSource` de l'évènement, mais il lui faut accéder à l'autre `JLabel`, celui dans lequel il va préciser les coordonnées de la souris. C'est pour cette raison que dans le code ci-dessous la classe `MonListener` dispose d'une variable d'instance de type `JLabel` initialisée via son constructeur. Ainsi, lorsqu'on crée l'écouteur on lui fournit une référence sur le `JLabel` à modifier, référence qu'il conservera via sa variable d'instance.

Notez qu'ici nous ne voulons réagir qu'aux déplacements de souris mais l'interface précise également une méthode réagissant au fait que l'on dragde (déplacer avec le bouton enfoncé) la souris. Nous sommes donc obligés de fournir une implémentation (vide) de cette deuxième méthode.

Exemple

```
public class ExempleMouseMotionListener extends JFrame {
  public ExempleMouseMotionListener() {
    super("Exemple MouseMotionListener");
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    JLabel coordonnees = new JLabel("x: 0 y: 0");
    this.getContentPane().add(coordonnees, BorderLayout.NORTH);
    JLabel departements = new JLabel( new ImageIcon("images"+
                                      File.separator+"departements.png"));
    departements.addMouseMotionListener(new MonListener(coordonnees));
    this.getContentPane().add(departements, BorderLayout.CENTER);
  }
  public static void main(String[] args) {
    ExempleMouseMotionListener fenetre = new ExempleMouseMotionListener();
    fenetre.pack();
    fenetre.setVisible(true);
  }
}
class MonListener implements MouseMotionListener {
  private JLabel etiquette; // l'etiquette a mettre a jour

  public MonListener( JLabel l) {
    this.etiquette = l;
  }
  public void mouseMoved(MouseEvent e) {
    this.etiquette.setText("x: "+e.getX()+" y: "+e.getY());
  }
  public void mouseDragged(MouseEvent e) {
  }
}
```

Notez que tous les évènements de votre interface sont gérés dans un unique thread. Ceci garantit que l'ensemble du traitement d'un évènement sera effectué avant de passer à l'évènement suivant. En conséquence, si un calcul lourd doit être effectué en réponse à un évènement il faut réaliser ce traitement dans un autre thread. L'exécuter dans le même thread bloquerait le traitement des autres évènements.

5.2. Les évènements basiques

Les évènements basiques sont ceux que tout composant peut émettre. La liste ci-dessous passe sous silence les interfaces `AncestorListener`, `ContainerListener` et `HierarchyListener` moins utiles en première approche.

ComponentListener - *ComponentEvent*	
componentHidden	Appelée lorsqu'on masque le composant.
componentMoved	Appelée lorsqu'on a modifié la position du composant.
componentResized	Appelée lorsqu'on a modifié la taille du composant.
componenteShown	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 (*).
keyReleased	Appelée lorsqu'on relache une touche du clavier (*).
keyTyped	Appelée lorsqu'on a tapé (appuyé puis relâché) une touche du clavier (*).

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 draguée (déplacée avec bouton enfoncé).
mouseMoved	Appelée lorsque la souris est déplacée sur le composant boutons relâchés.

MouseWheelListener - *MouseEvent*	
mouseWheelMoved	Appelée lorsqu'on tourne la roue de la souris.

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.

(*) et que le composant a le focus.

5.3. Les autres évènements

D'autres évènements ne concernent qu'un sous-ensemble des composants. En voici une liste non-exhaustive.

ActionListener - *ActionEvent*	
actionPerformed	- Un bouton est actionné (`JButton`, `JCheckBox`, `JCheckBoxMenuItem`, `JMenu`, `JMenuItem`, `JRadioButton`, `JRadioButtonMenuItem`, `JToggleButton`) - Un élément est sélectionné (`JComboBox`) - La touche [Entrée] est pressée (`JTextField`) - Un fichier est sélectionné (`JFileChooser`)

AdjustementListener - *AdjustmentEvent*	
adjustmentValueChanged	- Modification de la valeur (`JScrollBar`)

CaretListener - *CaretEvent*	
caretUpdate	- Modification de la position du curseur dans le texte (`JEditorPane`, `JTextField`)

ChangeListener - *ChangeEvent*	
stateChanged	- Changement d'état (`JButton`, `JCheckBox`, `JCheckBoxMenuItem`, `JMenu`, `JMenuItem`, `JRadioButton`, `JRadioButtonMenuItem`) - Sélection d'un onglet (`JTabbedPane`) - Déplacement du curseur (`JProgressBar`, `JSlider`, `JSpinner`)

DocumentListener - *DocumentEvent*	
changeUpdate	- Modification du texte (`Document` le modèle de `JEditorPane` et `JTextField`)
insertUpdate	- Insertion de texte (`Document`).
removeUpdate	- Suppression de texte (`Document`).

HyperlinkListener - *HyperlinkEvent*	
hyperlinkUpdate	- Activation d'un hyperlien (`JEditorPane`)

InternalFrameListener - *InternalFrameEvent*	
internalFrameActived	- Activation (`JInternalFrame`).
internalFrameClosed	- Appel après fermeture (`JInternalFrame`).
internalFrameClosing	- Tentative de fermeture (`JInternalFrame`).
internalFrameDeactivated	- Perte de l'activation (`JInternalFrame`).
internalFrameDeiconified	- Passage de iconisée à normale (`JInternalFrame`).
internalFrameIconified	- Passage de normale à iconisée (`JInternalFrame`).
internalFrameOpened	- Après ouverture (`JInternalFrame`).

ItemListener - *ItemEvent*	
itemStateChanged	- Un item a été sélectionné/désélectionné (`JComboBox`) - Un bouton est actionné (`JButton`, `JCheckBox`, `JCheckBoxMenuItem`, `JMenu`, `JMenuItem`, `JRadioButton`, `JRadioButtonMenuItem`, `JToggleButton`)

ListSelectionListener - *ListeSelectionEvent*	
valueChanged	- Modification de la sélection (`JList`)

MenuDragMouseListener - *MenuDragMouseEvent*	
menuDragMouseDragged	- La souris est draguée dans le menu (`JCheckBoxMenuItem`, `JMenu`, `JMenuItem`, `JRadioButtonMenuItem`)
menuDragMouseEntered	- La souris draguée entre dans le menu.
menuDragMouseExited	- La souris draguée sort du menu.
menuDragMouseReleased	- La souris draguée a son bouton relâché.

MenuListener - *MenuEvent*	
menuCanceled	- Annulation du menu (`JMenu`)
menuDeselected	- Désélection du menu (`JMenu`)
menuSelected	- Sélection du menu (`JMenu`)

MenuKeyListener - *MenuKeyEvent*	
menuKeyPressed	- Touche enfoncée dans un menu (`JCheckBoxMenuItem`, `JMenu`, `JMenuItem`, `JRadioButtonMenuItem`, `JPopupMenu`)
menuKeyReleased	- Touche relâchée dans un menu.
menuKeyTyped	- Touche tapée dans un menu.

PopupMenuListener - *PopupMenuEvent*	
popupMenuCanceled	- Popup menu annulé (`JPopupMenu`)
popupMenuWillBecomeInvisible	- Popup menu va être masqué (`JPopupMenu`)
popupMenuWilleBecomeVisible	- Popup menu va être rendu visible (`JPopupMenu`)

TableModelListener - *TableModelEvent*	
tableChanged	- Modification d'une case (modèle d'une `JTable`)

Voici un récapitulatif des composants concernés par ces évènements.

	JButton	JCheckBox	JCheckBoxMenuItem	JComboBox	JEditorPane	JFileChooser	JInternalFrame	JLabel	JList	JMenu	JMenuItem	JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem	JScrollBar	JSlider	JSpinner	JTabbedPane	JTable	JTextField	JToggleButton
ActionListener	■	■	■	■		■				■	■			■	■						■	■
AdjustementListener																■						
CaretListener					■																■	
ChangeListener	■	■	■							■	■		■	■	■		■	■	■			■
DocumentListener					■																■	
HyperlinkListener					■																	
InternalFrameListener							■															
ItemListener	■	■	■	■						■	■			■	■							■
ListSelectionListener									■													
MenuDragMouseListener			■							■	■				■							
MenuListener										■												
MenuKeyListener			■							■	■	■			■							
PopupMenuListener				■								■										
TableModelListener																				■		

5.4. Classes internes et classes anonymes

Une interface peut comporter beaucoup de composants et il serait très lourd de devoir créer une classe écouteur pour chaque évènement possible, d'autant que ces classes écouteur son souvent très spécifiques et donc rarement réutilisables. Un autre inconvénient de cette approche est que certains écouteurs doivent accéder à un sous-ensemble des composants comme nous l'avons vu dans notre second exemple. La solution de fournir en paramètre du constructeur les composants utile peut là encore devenir très lourd lorsque le nombre de composants augmente. Voyons différentes possibilités pour alléger le code.

5.4.1. Classes externes

C'est la solution qui a été retenue dans les deux exemples d'écouteurs que nous avons vu. On utilise cette solution lorsque l'écouteur peut-être réutilisé, soit uniquement au sein de notre IHM et dans ce cas mieux vaut en faire une classe privée, soit en dehors de notre IHM et on opte alors pour une classe publique (dans son propre fichier).

5.4.2. Le conteneur est l'écouteur

La classe définissant notre fenêtre nous pouvons implémenter l'interface désirée et devenir notre propre écouteur. Comme illustré ci-dessous les changements se limitent à (1) déclarer l'implémentation des interfaces, à (2) ramener le code des méthodes au sein de la classe définissant la fenêtre et à (3) déclarer la fenêtre comme écouteur de son propre bouton.

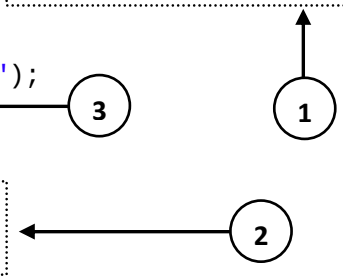
L'avantage est qu'on évite de répartir le code dans un tas de classes, les liens entre toutes les classes pouvant rendre le code plus difficile à appréhender.

Il y a cependant des inconvénients. Ici nous n'avons qu'un seul bouton. Si nous en avions plusieurs nous devrions au sein de la méthode `actionPerformed` consulter la source de l'évènement afin d'identifier le bouton à l'origine de l'évènement et effectuer le traitement adapté. Mélanger au sein d'une même méthode les traitements à effectuer pour tous les boutons peut grandement réduire la lisibilité. Dans la continuité de ce constat, nous pourrions réagir à plusieurs évènements d'un même composant. Pour consulter le comportement d'un composant nous devrions parcourir des méthodes distantes en recherchant à l'intérieur d'elles la partie concernant ce composant. Nous aimerions au contraire que le code des traitements attachés à un composant soient proches les uns des autres pour facilité la maintenance.

Cette solution n'est donc recommandable que pour des interfaces comportant peu de composants et au nombre limité de types d'évènements traités.

Exemple

```
public class ExempleActionListener2 extends JFrame implements ActionListener{
   public ExempleActionListener2() {
      super("Exemple ActionListener2");
      JButton monBouton = new JButton("Un clic SVP");
      monBouton.addActionListener( this );
      this.getContentPane().add(monBouton);
   }
   public void actionPerformed(ActionEvent e) {
      ((JButton)e.getSource()).setText("Merci !");
   }
   public static void main(String[] args) {
      ExempleActionListener2 fenetre = new ExempleActionListener2();
      fenetre.setSize(200,100);
      fenetre.setVisible(true);
   }
}
```



5.4.3. Classes internes

Il est possible de déclarer une classe au sein d'une autre classe. Cette ***classe interne*** a alors accès aux variables d'instance de la classe principale. On évite ainsi de devoir fournir en paramètre du constructeur de l'écouteur les composants de l'IHM auxquels il doit accéder.

Le code ci-dessous illustre cette possibilité sur l'exemple vu pour `MouseMotionListener`. Les changements sont que (1) le composant utile à l'écouteur (`coordonnees`) n'est plus local au constructeur mais mis en variable d'instance, ainsi (2) le code de l'écouteur placé en tant que classe interne peut accéder à `coordonnees` et (3) il n'est plus utile de passer `coordonnees` au constructeur de l'écouteur.

On conserve toutefois l'inconvénient d'un grand nombre de classes si il y a de nombreux composants et types d'évènements à traiter. Cette méthode demeure enviable lorsqu'un écouteur peut-être réutilisé au sein de la même IHM et lorsque le traitement de l'évènement requiert un code non trivial.

Exemple

```
public class ExempleMouseMotionListener2 extends JFrame {
   private JLabel coordonnees;                                          <-- 1

   class MonListener implements MouseMotionListener {
      public void mouseMoved(MouseEvent e) {
         coordonnees.setText("x: "+e.getX()+" y: "+e.getY());           <-- 2
      }
      public void mouseDragged(MouseEvent e) {
      }
   }
   public ExempleMouseMotionListener2() {
      super("Exemple MouseMotionListener");
      coordonnees = new JLabel("x: 0 y: 0");
      this.getContentPane().add(coordonnees, BorderLayout.NORTH);
      JLabel departements = new JLabel( new ImageIcon("images"+
                                        File.separator+"departements.png"));
      departements.addMouseMotionListener(new MonListener());           <-- 3
      this.getContentPane().add(departements, BorderLayout.CENTER);
   }
   public static void main(String[] args) {
      ExempleMouseMotionListener2 fenetre =new ExempleMouseMotionListener2();
      fenetre.pack();
      fenetre.setVisible(true);
   }
}
```

5.4.4. Classes internes anonymes

Dans cette solution la classe de l'écouteur est écrite à l'endroit de son instanciation. En d'autres termes, c'est là où l'on crée l'écouteur qu'on trouve son code. D'un point de vue lisibilité on y gagne énormément car pour chaque composant on regroupe au même endroit du code sa définition, sa disposition et la gestion de ses évènements.

Attention à la syntaxe ! Il demeure toujours impossible d'instancier une interface. Nous ne pourrions pas avoir dans notre code « `new MouseMotionListener();` ». L'encadré ci-dessous exhibe la syntaxe pour créer une instance d'une classe anonyme implémentant `MouseMotionListener`.

S'agissant d'une classe interne elle a accès aux variables d'instances de la classe principale (ici `coordonnees`) et étant anonyme son code peut-être rapproché de la définition du composant qu'elle traite. C'est la solution recommandable lorsque le code du traitement est court et qu'il n'est pas réutilisé par ailleurs.

Exemple

```
public class ExempleMouseMotionListener3 extends JFrame {
   private JLabel coordonnees;

   public ExempleMouseMotionListener3() {
      super("Exemple MouseMotionListener3");
      coordonnees = new JLabel("x: 0 y: 0");
      this.getContentPane().add(coordonnees, BorderLayout.NORTH);
      JLabel departements = new JLabel( new ImageIcon("images"+File.separator
                                                      +"departements.png"));
      departements.addMouseMotionListener(
         new MouseMotionListener() {
            public void mouseMoved(MouseEvent e) {
               coordonnees.setText("x: "+e.getX()+" y: "+e.getY());
            }
            public void mouseDragged(MouseEvent e) {
            }
         });
      this.getContentPane().add(departements, BorderLayout.CENTER);
   }
   public static void main(String[] args) {
      ExempleMouseMotionListener3 fenetre =new ExempleMouseMotionListener3();
      fenetre.pack();
      fenetre.setVisible(true);
   }
}
```

6. PAC

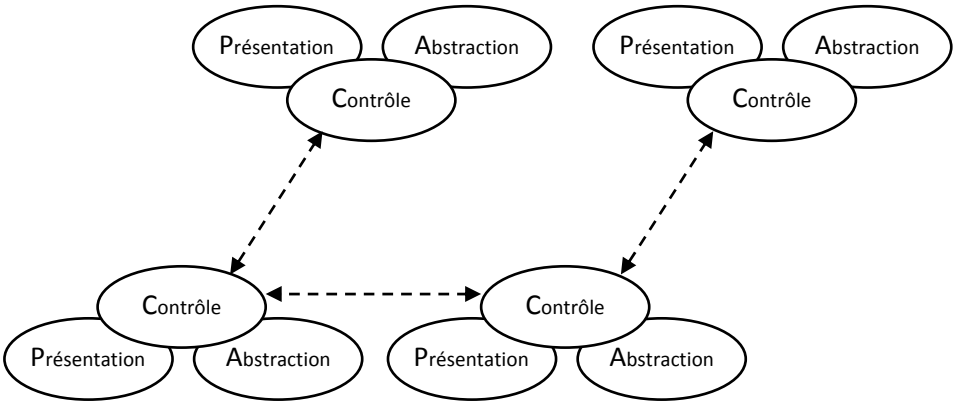
6.1. Le principe

Pour de petites interfaces le code est relativement concis et le faible nombre de composants (et d'interactions entre eux) rend l'application facile à prendre en main. Mais le passage à l'échelle pose problème. De nombreuses méthodologies ont par conséquent été proposées dans le but d'améliorer les qualités (Robustesse, extensibilité, réutilisabilité, lisibilité) du code. Nous allons aborder l'une de ces méthodologies nommée PAC pour Présentation-Abstraction-Modèle.

On aimerait :
- Pouvoir découpler le code lié à l'interface du noyau applicatif (le code fonctionnel, c'est-à-dire le reste du programme qui modélise les données et fourni des traitements sur ces données).
- Avoir un code modulaire avec la possibilité d'ajouter/retirer un moduler ou de le substituer par un autre sans que cela affecte les autres modules.

Comme son nom le laisse deviner, PAC s'appuie sur une séparation en trois types de facettes :
- Les facettes de Présentation sont des parties purement IHM. Une telle facette peut correspondre à un affichage ou a une saisie. La réunion de toutes les facettes présentations forme l'IHM du programme.
- Les facettes d'Abstraction qui modélisent une partie de l'application et de ses traitements. La réunion des facettes abstraction constitue le noyau applicatif, le code chargé de représenter les données et de les traiter.
- Les facettes de Contrôle qui font le lien entre les deux autres types de facettes. Elles permettent d'assurer la cohérence entre les données du modèle et les représentations de ces données. Elles traduisent également les actions de l'utilisateur en termes d'impact sur le modèle.

Dans cette architecture une facette de présentation ne communique qu'avec sa facette de contrôle. C'est là une différence fondamentale avec le modèle MVC (Modèle-Vue-Controleur) souvent confondu avec PAC. De même, les facettes d'abstraction ne communiquent qu'avec les facettes de contrôle. On retrouve l'idée que les contrôles font le pont entre l'abstraction et la présentation. Les facettes de contrôle peuvent également communiquer entre elles. On peut schématiser les interactions par le dessin ci-dessous sachant que les échanges entre les facettes de contrôle sont possibles mais n'ont rien de systématique :



PAC demeure un modèle abstrait, un schéma général qu'on peut transcrire de différentes façons. Nous allons l'aborder en nous appuyant sur le patron de conception Observer.

6.2. Le patron Observer

On trouve des schémas récurrents de code. Il a du d'ailleurs vous arriver de reproduire plus ou moins la même approche durant différents TP. Ces schémas classiques forment des patrons de conception (Pattern).

Celui qui nous intéresse devrait vous faire penser à certains réseaux sociaux. Dans ce schéma, une classe étend `Observable`. Cela lui permet d'avoir une liste d'objets qui l'observent (« qui la suivent » dans l'analogie) et d'avertir ces objets quand ses données ont changées. Les classes qui veulent observer doivent implémenter l'interface `Observer` et les instances doivent s'ajouter à la liste des observateurs (« préciser qu'elles veulent suivre ») de l'objet observé.

Les méthodes indispensables à connaître de la classe `Observable` sont :
- `void addObserver(Observer o)` : ajoute `o` à la liste des objets à informer en cas de modification.
- `void setChanged()` : marque que l'objet a été modifié.
- `void notifyObservers(Object arg)` : si l'objet a été modifié (cf. `setChanged`), notifie (appelle la méthode `update` avec `arg` pour argument) tous les observateurs puis passe l'état à « non modifié ».

L'interface `Observer` quant à elle requiert une seule méthode :
- `void update(Observable o, Object arg)` : méthode appelée lorsque l'objet observé notifie un changement. `o` est une référence vers l'objet observé et `arg` est l'information utile qu'il nous communique, généralement utilisé pour déterminé quel type de changement a eu lieu.

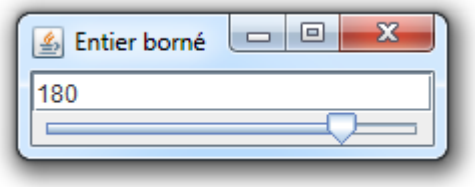
On le voit, ce patron permet d'informer une partie du code d'un changement d'une partie des données. Pour le mettre en place, l'objet observé doit :
- Déclarer étendre `Observable`.
- Après chaque modification de valeur, par exemple dans les setters, faire un appel à `setChanged` suivi d'un appel à `notifyObservers`.

L'objet observateur doit lui :
- Implémenter l'interface `Observer` et donc fournir une méthode `update(Observable o, Object arg)` .
- Etre ajouté à la liste des observateurs de l'objet observé via un appel à `addObserver` sur ce dernier.

6.3. PAC par l'exemple

Il arrive bien souvent qu'une application manipule plusieurs valeurs bornées. Commençons petit ! Regardons le cas d'une application manipulant une valeur entière bornée. On pourra supposer qu'il y a d'autres données et des traitements liés, mais nous considèrerons dans notre cas que l'interaction porte sur cet unique entier. On désire offrir de la souplesse en laissant la possibilité de manipuler l'entier via :

- 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.



Les interactions sont liées. Si les traitements modifient la valeur de l'entier les deux composants doivent être mis à jour, si on saisit une valeur dans le `JTextField` il faut vérifier qu'elle figure entre les bornes et affecter le modèle et le `JSlider` en conséquence, et si on bouge le curseur du `JSlider` il faut mettre à jour le modèle et le `JTextField`. A première vue, rien n'est indépendant, les composantes et le modèle sont tous liés entre eux. Le risque est grand pour qu'on oublie une mise à jour dans le traitement de l'un des évènements avec à la clef de l'incohérence. De plus, si on décide de remplacer le `JTextField` par un `JSpinner` il faut revoir tout le code. C'est un tout petit exemple mais pour lequel on sent déjà l'intérêt d'utiliser l'approche PAC.

Notre noyau fonctionnel est très petit puisque réduit à un entier borné. Nous n'aurons donc qu'une seule facette d'abstraction modélisée par la classe ci-dessous. Notez que pour pouvoir être observée par les facettes de contrôle elle étend `Observable` et fait appel à `setChanged` et `notifyObservers` lorsque ses données sont modifiées.

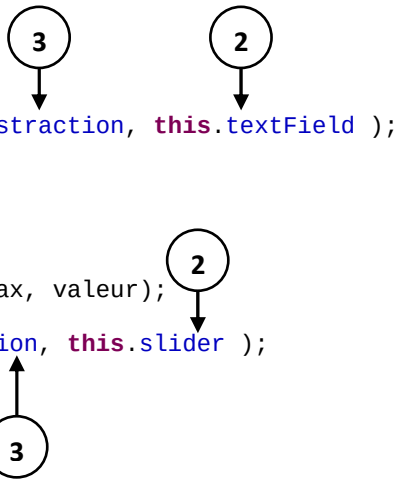
Exemple

```
package abstraction;
import java.util.Observable;

public class EntierBorne extends 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();
		this.notifyObservers(null);
	}
}
```

Les facettes de contrôle seront implémentées via les classes `ControlJSlider` et `ContrlJTextField` mais avant de nous pencher sur eux observons comment l'IHM crée les facettes. L'une des facettes de présentation correspond au `JTextField` (nous aurions pu éventuellement bâtir une classe fille) l'autre au `JSlider`. Ces deux composants sont initialisés avec la valeur courante de l'entier mais n'ont aucune référence vers l'abstraction et ils ne communiqueront pas directement avec le modèle. Tous les échanges passent par les contrôles. (1) Le contrôle est un écouteur du composant et réagira par conséquent aux actions effectuées par l'utilisateur sur ce composant, et (2) on fournit une référence vers le composant au constructeur du contrôle afin que ce dernier puisse agir sur le composant si les données du modèle sont modifiées. Ces deux actions ont donc rendu les échanges possibles dans les deux sens entre le contrôle et sa facette de présentation. On rend les échanges possibles entre le contrôle et son abstraction en (3) passant l'abstraction en paramètre du constructeur du contrôle et en (4) ajoutant le contrôle à la liste des observateurs de l'abstraction.

Exemple

```
package presentation;
import java.awt.Container;
import javax.swing.*;
import controle.ControlJSlider;
import controle.ControlJTextField;
import abstraction.EntierBorne;

public class IHMEntierBorne extends JFrame {
   private EntierBorne abstraction;
   private JTextField textField;
   private JSlider slider;

   public IHMEntierBorne(EntierBorne abstraction) {
      super("Entier borné");
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.abstraction=abstraction;
      int valeur = this.abstraction.getValeur();
      int min = this.abstraction.getMin();
      int max = this.abstraction.getMax();

      Container conteneur = this.getContentPane();
      conteneur.setLayout(new BoxLayout(conteneur, BoxLayout.Y_AXIS));


      // ===    T E X T    F I E L D   ===
      this.textField = new JTextField(""+valeur);
      conteneur.add(this.textField);
      ControlJTextField cl = new ControlJTextField(this.abstraction, this.textField );
      this.textField.addActionListener(cl);
      this.abstraction.addObserver(cl);


      // ===    S L I D E R   ===
      this.slider = new JSlider(JSlider.HORIZONTAL, min, max, valeur);
      conteneur.add(this.slider);
      ControlJSlider cs = new ControlJSlider(this.abstraction, this.slider );
      this.slider.addChangeListener(cs);
      this.abstraction.addObserver(cs);
   }

   public static void main(String[] args) {
      EntierBorne temperature = new EntierBorne(19, -273, 300);
      IHMEntierBorne fen = new IHMEntierBorne(temperature);
      fen.pack();
      fen.setVisible(true);
   }
}
```



Il reste à voir les contrôles. Regardons d'abord celui du `JSlider`. On retrouve qu'il prend en paramètre de son constructeur des références vers son abstraction et sa facette de présentation afin qu'il puisse accéder à eux. De plus, il implémente `ChangeListener` pour pouvoir écouter le `JSlider` et implémenter `Observer` pour pouvoir être ajouté à la liste des observateurs de l'abstraction.

Le composant a été initialisé dans l'interface avec les bornes de l'entier borné. Par conséquent, toutes les valeurs que peut prendre le `JSlider` sont entre ces bornes. C'est pourquoi la méthode `statChanged` met à jour l'abstraction sans même avoir à vérifier quoi que ce soit.

Exemple

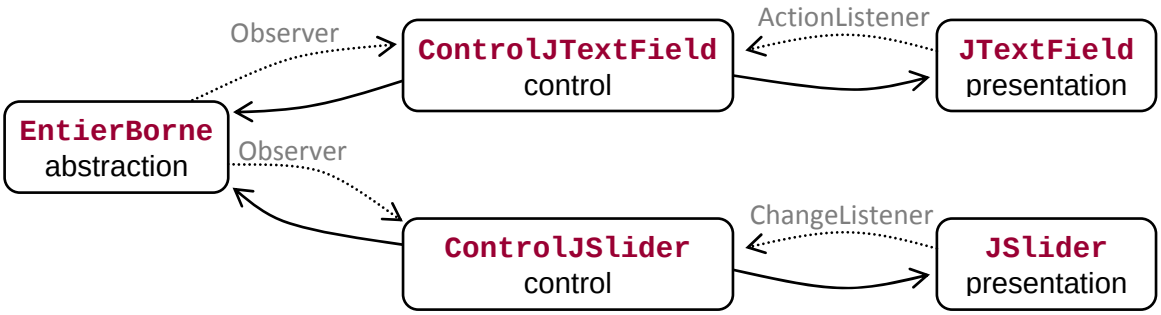
```
…
public class ControlJSlider implements Observer, ChangeListener {
   private EntierBorne entier;
   private JSlider slider;

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

L'utilisateur est en revanche libre de taper n'importe quoi dans le `JTextField`. C'est pourquoi son contrôleur doit vérifier que la valeur du `JTextField` est bien un entier compris entre les bornes avant de modifier l'abstraction. Si ce n'est pas le cas, il affiche une boîte de dialogue pour rappeler à l'utilisateur les bornes entre lesquelles l'entier doit demeurer et met à jour le `JTextField` avec la valeur courante de l'entier borné.

Exemple

```
…
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) {
      this.textField.setText(""+this.entier.getValeur());
   }
   public void actionPerformed(ActionEvent arg0) {
      int valeur = Integer.MIN_VALUE;
      try {
         valeur = Integer.parseInt(this.textField.getText());
      }
      catch(Exception e) {
      }
      if (valeur<this.entier.getMin() || valeur>this.entier.getMax()) {
         JOptionPane.showMessageDialog(null, "La valeur doit être entre "+
                                       this.entier.getMin()+" et "+this.entier.getMax(),
                                       "Valeur hors bornes", JOptionPane.WARNING_MESSAGE);
         this.textField.setText(""+this.entier.getValeur());
      }
      else {
         this.entier.setValeur(valeur);
      }
   }
}
```

On peut schématiser les échanges entre les classes par :



Les objectifs sont atteints puisqu'il y a une séparation nette entre la partie purement IHM et le noyau fonctionnel. De plus les composants sont à présents indépendants. Le `JSlider` n'a plus à se soucier du `JTextfield`. Les modifications seront propagées via les contrôleurs. On a gagné en robustesse car il est beaucoup moins facile d'oublier une mise à jour, et en modularité.