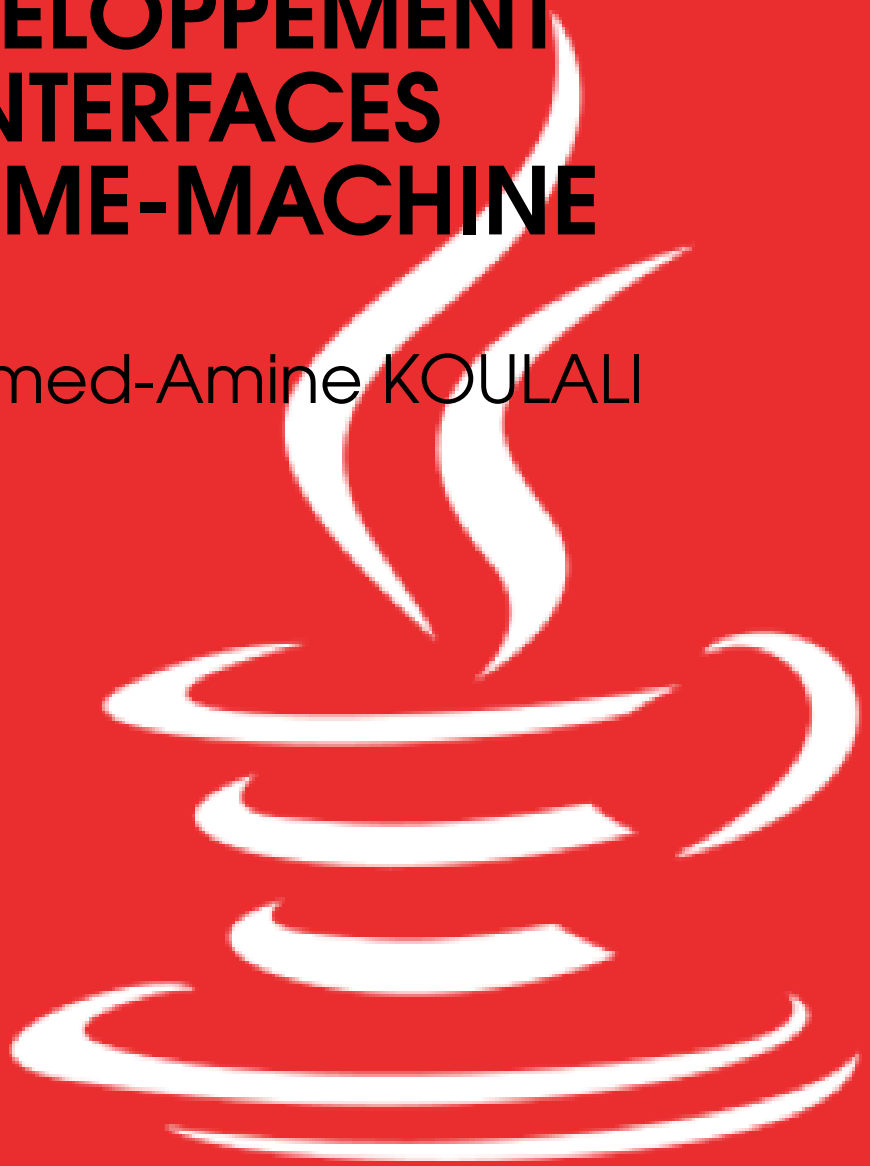


DEVELOPPEMENT INTERFACES HOMME-MACHINE

Mohammed-Amine KOULALI



ECOLE NATIONALE DES SCIENCES APPLIQUÉES D'OUJDA

Copyright © 2018 Mohammed-Amine KOULALI

Table des Matières

1	Introduction	4
1.	Motivation	4
2	Abstract Windowing Toolkit (AWT)	6
1.	Les APIs graphiques de JAVA	6
2.	Composition de AWT	6
3.	Container awt	7
4.	Container des haut niveau	7
5.	Container secondaire	8
6.	Components awt	8
a.	Label	9
b.	Button	10
c.	TextField	10
d.	TextArea	11
e.	CheckBox & boutons radio	11
f.	List	12
g.	Choice	13
h.	Exemple récapitulatif	14
3	Architecture du Framework Swing	16
1.	Introduction	16

2.	Différences entre awt et Swing	16
3.	Création d'une IHM Swing et arbre des composants	17
a.	Création d'une application swing	17
b.	Arbre des composants	17
4.	Les différentes catégories de composants	18
5.	Architecture MVC des composants Swing	19
a.	Exemple:	19
6.	Pluggable look and feel	20
7.	Les conteneurs	21
a.	JFrame	21
b.	JPanel	22
8.	Les composants simples	22
a.	Les images	23
b.	Les boutons	23
c.	JCheckBox	25
d.	JRadioButton	26
e.	Les listes	27
f.	Les champs textuels	28
9.	Les composants swing élaborés	31
a.	Les arbres	31
b.	Les tables	35
4	Les gestionnaires de présentation	41
1.	LayoutManager sans contrainte :	41
a.	FlowLayout	41
b.	GridLayout	43
c.	BoxLayout	44
2.	LayoutManager avec contrainte :	45
a.	BorderLayout	46
b.	GridBagLayout	47
c.	GroupLayout	50

1. Introduction

1. Motivation

Une des problématiques majeure qu'il incombait aux développeurs de traiter était de gérer la portabilité de leurs applications entre différents systèmes d'exploitation et architectures matérielle et/ou logicielle. En effet, Les langages de programmation tels que C et C++ exigeait une recompilation pour chaque plateforme visée (Linux, Windows, Mac, ...). Les applications avec interfaces graphiques consistait un casse tête supplémentaire. Ainsi, les développeurs avait recours à des Application Programming Interface (API) spécifiques telles que MFC, Qt et la portabilité se voyait ainsi extrêmement réduite.

Lors du printemps 1995 la société Sun Microsystems dévoile Java pour résoudre ce dilemme. L'adage de Sun était: "*compile once-run everywhere!*". Il est devenu désormais possible de compiler son programme une seule fois en vue de générer du code intermédiaire qui était interprétable sur n'importe que architecture disposant d'une machine virtuelle java dédié. Sun offrait des JVM spécifiques à diverses architectures matérielles/logicielles et les développeurs pouvait se concentrer sur la production de logiciels. La portabilité couvrait aussi l'aspect visuel des interfaces graphiques qui avait désormais un look-and-feel transversal. Ce dernier point est particulièrement important pour permettre aux utilisateurs une familiarité avec l'application indépendamment de la plateforme utilisée pour son exécution.

Le présent support couvre le développement d'interfaces graphiques pour permettre l'interaction homme/machine en Java. Nous adopterons le timeline historique en introduisant d'abord le package awt puis swing. Les notions abordées supposent que les concepts

fondamentaux de JAVA abordés dans le cours de programmation objet en java sont maîtrisés par le lecteur.

2. Abstract Windowing Toolkit (AWT)

Le cours de POO & JAVA nous a permis de maîtriser les notions de base de la programmation objet en JAVA tels que l'encapsulation, le polymorphisme et l'héritage. Le langage JAVA comparé aux autres langages tels que C ou pascal, permet une grande abstraction moyennant des types appelés classes. Ces classes permettent de modéliser les concepts rencontrés dans la vie quotidienne (Personne, Vehicule, ...).

La JDK contient des classes dédiées à la création d'interfaces graphiques. Ces dernières sont organisées dans des packages spéciaux et se conforment à des motifs de conception évolués.

1. Les APIs graphiques de JAVA

Java comprend deux apis pour la programmation d'interfaces graphiques: AWT (Abstract Windowing Toolkit) et Swing. L'API awt a été introduit en java dès la version 1.0. Oracle (Société qui détient les droits sur le langage JAVA) considère que beaucoup de ses composants sont devenus obsolètes et recommande l'utilisation de Swing. Swing a été introduit quant à lui avec la version 1.1 de la jdk comme faisant partie de la Java Foundation Classes (JFC). La JFC comprends en plus de Swing JAVA2D, l'accessibilité, l'internationalisation et le look-and-feel adhoc.

Des éditeurs tiers proposent des api graphiques autres que awt et swing. C'est notamment le cas de Eclipse avec son framework swt et de Google avec son api GWT.

2. Composition de AWT

L'API AWT est formé par 370 classes réparties sur 12 packages. Nous nous focaliserons sur les plus importants d'entre eux, à savoir, *java.awt* et *java.awt.event*. Le premier contient les classes graphiques formant le noyau de awt tels que: les composants (boutons,

labels, ...), les conteneurs (Frame et Panel), les gestionnaires de positionnement et des classes spéciales pour la gestion des fontes et des couleurs. Le second package sert à gérer les événements produit par l'interaction entre l'interface graphique et l'utilisateur: click de souris, saisie clavier,

La Figure 2.4 décrit les relations d'héritages entre les principales classes du package *java.awt*. Il existe deux catégories d'éléments graphiques: les composants (classe *Component* et ses sous-classes) et les conteneurs (classe *Container* et ses sous-classes). Un composant et un élément graphique élémentaire tel que *Button* (bouton), *Label* (label) ou *CheckBox* (case à cocher). Un conteneur a pour objectif de regrouper des composants selon un positionnement (Layout) bien défini. Notons qu'un conteneur est aussi un composant, ce qui rend possible à un conteneur de contenir d'autres conteneurs. Lier un objet à son conteneur se fait à travers la méthode *void add(Component c)* de ce dernier.

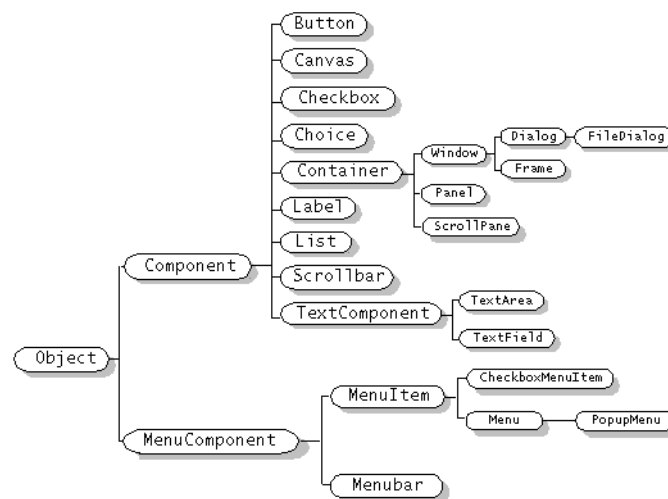


Figure 2.1: Hiérarchie des classes awt

3. Container awt

La Figure 2.2 illustre la hiérarchie des conteneurs awt. Ces conteneurs peuvent être classés en deux catégories selon les fonctionnalités offertes. On trouve ainsi, des conteneurs de haut niveau et des conteneurs secondaires.

4. Conteneur des haut niveau

Les classes *Frame*, *Dialog* et *Applet* représentent les trois conteneurs principaux de awt. On les appelle aussi conteneurs de haut niveau car tout programme graphique doit en contenir au moins une.

Les instances de la classe *Frame* représentent une fenêtre iconifiée munie d'une barre de titre, d'une barre de menu optionnelle et d'une zone d'affichage de contenu. Pour créer un

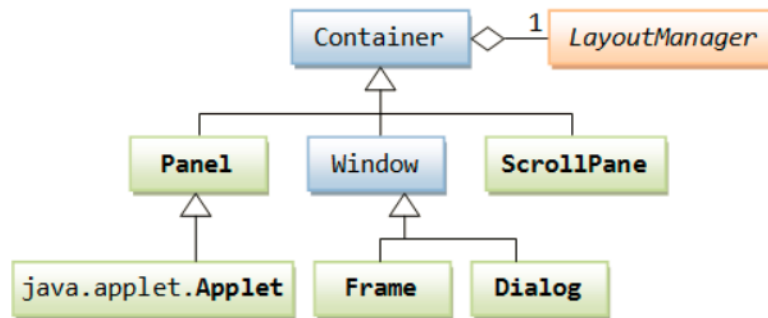


Figure 2.2: Hiérarchie des container awt.

programme avec une interface graphique on commence typiquement par hériter de la classe *java.awt.Frame*:

```

1 import java . awt . Frame ; // Using Frame class in package java . awt
2
3 public class MonProgramGraphique extends Frame {
4     // attributs
5     ...
6     // constructeur
7     public MonProgramGraphique () { ... }
8
9     public static void main (String [] args) {
10         MonProgramGraphique p = new MonProgramGraphique () ;
11     }
12 }

```

La classe *Dialog* représente une fenêtre "pop-up" et possède une barre de titre avec une icône, un bouton de fermeture et une zone d'affichage de contenu. Finalement, la classe *Applet* du package *java.applet* est un container de haut niveau qui permet d'exécuter des interfaces graphiques au sein d'un navigateur web.

5. Container secondaire

Les containers secondaires sont placés au sein de leurs homologues de haut niveau ou secondaire. La classe *Panel* représente un container multi-usage rectangulaire sans caractéristiques visuelles (couleur, ...). La second container *ScrollPane* fournit un container avec une barre de défilement horizontale et/ou verticale pour un seul composant enfant.

6. Components awt

Les interfaces graphiques modernes sont battis sur la notion de composants. Ces derniers sont des block réutilisable possédant un rendu graphique et une logique métier. Awt propose

une collection de composants élémentaire tels que boutons, listes, Il est aussi possible de créer son propre composant en combinant les composants élémentaires. La liste des composants awt est résumée au sein de la Figure 2.3.



Figure 2.3: Les composants awt

a. Label

La classe `java.awt.Label` représente un texte descriptif utilisé pour labéliser un autre composant (champs textuel) ou fournir une description textuelle.

Pour créer une instance `Label` on dispose des constructeurs suivants:

```
1 public Label(String strLabel, int alignment);
2 public Label(String strLabel);
3 public Label();
```

Les paramètres `strLabel` et `alignment` décrivent respectivement le texte et l'alignement du label. Notez que Awt propose des valeurs pour le second paramètre définies comme des constantes statiques au sein de la classe `Label`, il s'agit de:

```
1 public static final LEFT; // Label.LEFT
2 public static final RIGHT; // Label.RIGHT
3 public static final CENTER; // Label.CENTER
```

Exemple 2.1:

```
1 Panel p=new Panel();
2 Label lblInput;
3
4 lblInput = new Label("Enter ID");
5
6 p.add(lblInput);
7 lblInput.setText("Enter password");
8
9 String ch= lblInput.getText();
```

b. Button

La classe `java.awt.Button` représente un bouton qui sert à déclencher une action suite à un click de l'utilisateur. Afin d'en construire des instances, awt propose d'utiliser les constructeurs suivants:

```
1 public Button(String btnLabel);  
2 public Button();
```

Il est aussi possible de récupérer/modifier le label d'un bouton et d'activer/désactiver un bouton moyennant les méthodes suivantes:

```
1 public String getLabel();  
2 public void setLabel(String btnLabel);  
3 public void setEnable(boolean enable);
```

Exemple 2.2:

```
1 Panel p = new Panel();  
2 Button btnColor = new Button("Red");  
3 p.add(btnColor);  
4 btnColor.setLabel("Green");  
5 String lb = btnColor.getLabel();  
6 p.add(new Button("Blue"));
```

c. TextField

Les zones de texte mono-ligne, instances de `java.awt.TextField`, sont destinées à recevoir le texte saisi par l'utilisateur. Pour en créer, awt propose un ensemble de son constructeurs surcharger pour contrôler le texte par défaut et/ou la taille du composant en termes de nombre de colonnes. Notons que le nombre de colonnes détermine la taille du `TextField` mais n'impose aucune contrainte sur la taille du texte saisie par l'utilisateur.

```
1 public TextField(String initialText, int columns);  
2 public TextField(String initialText);  
3 public TextField(int columns);
```

Les méthodes ci-dessous permettent de manipuler le texte contenu dans un `TextField` et de gérer son activation et désactivation.

```
1 public String getText(); instance  
2 public void setText(String strText);  
3 public void setEditable(boolean editable);
```

Exemple 2.3:

```
1 Panel p = new Panel();
2 TextField tfIn = new TextField(30);
3 p.add(tfIn);
4 TextField tfRes = new TextField();
5 tfRes.setEditable(false);
6 p.add(tfRes);
7 try {
8     int number = Integer.parseInt(tfIn.getText());
9     number *= number;
10    tfRes.setText(number + "");
11 } catch (NumberFormatException e) {
12     System.out.println(e.getMessage());
13 }
```

d. TextArea

L'API awt supporte aussi les zones de texte multi-ligne à travers la classe *java.awt.TextArea*. Le développeur peut contrôler le rendu visuel en spécifiant un nombre de colonnes et de lignes pour fixer la taille de la zone de saisie. On dispose des constructeurs suivants pour créer des instances *TextArea*:

```
1 TextArea(int rows, int columns)
2 TextArea(String text)
3 TextArea(String text, int rows, int columns)
4 TextArea(String text, int rows, int columns, int scrollbars)
```

Pour manipuler le texte saisi, le modifier et ou le remplacer on dispose des méthodes:

```
1 void setText(String)
2 String getText()
3 void append(String str)
4 void insert(String str, int pos)
```

Exemple 2.4:

```
1 Panel p = new Panel();
2 TextArea ta = new TextArea("Zone de texte multi-ligne", 5, 40);
3 p.add(ta);
```

e. CheckBox & boutons radio

Les cases à cocher sont des composants graphique à deux état (on/off) et qui basculent d'un état à un autre suite à un click utilisateur.

Awt n'offre pas de classe dédiée pour créer des boutons radio. Toutefois on peut contourner cette limitation en regroupant des cases à cocher avec une instance de la classe

CheckboxGroup. En effet, ces derniers deviennent immédiatement à sélection exclusive simulant ainsi le comportement des boutons radio.

```

1 Checkbox( String label )
2 Checkbox( String label , boolean state )
3 Checkbox( String label , boolean state , CheckboxGroup group )
4 Checkbox( String label , CheckboxGroup group , boolean state )

```

```

1 String  getLabel()
2 void    setLabel( String label )
3 boolean getState()
4 void    setState( boolean state )

```

Exemple 2.5:

```

1 add( new Checkbox( "one" , null , true ) );
2 add( new Checkbox( "two" ) );
3 add( new Checkbox( "three" ) );

```

Exemple 2.6:

```

1 Pnael p = new Panel();
2 CheckboxGroup cbg = new CheckboxGroup();
3 p.add( new Checkbox( "one" , cbg , true ) );
4 p.add( new Checkbox( "two" , cbg , false ) );
5 p.add( new Checkbox( "three" , cbg , false ) );

```

f. List

La classe *java.awt.List* permet de choisir sélectionner un élément ou plusieurs éléments au sein d'une liste fixe de choix. On distingue les listes à choix unique de ceux à choix multiples. Les constructeurs de la classe *List* sont:

```

1 List()
2 List( int rows )
3 List( int rows , boolean multipleMode )

```

Pour manipuler les instances de *List* on dispose des méthodes:

```

1 void    setMultipleMode( boolean b )
2 void    add( String item )
3 void    add( String item , int index )
4 String  getItem( int index )
5 int     getItemCount()
6 String []  getItems()
7 int     getRows()

```

```

8  int  getSelectedIndex ()
9  int []  getSelectedIndexes ()
10 String  getSelectedItem ()
11 String []  getSelectedItems ()
12 Object []  getSelectedObjects ()
13 isIndexSelected (int) .

```

Exemple 2.7:

```

1  Panel cnt = new Panel ();
2  List lst = new List (4, false);
3  lst.setMultipleMode (true); // essayez avec false
4  lst.add ("Mercury");
5  lst.add ("Venus");
6  lst.add ("Earth");
7  lst.add ("JavaSoft");
8  lst.add ("Mars");
9  lst.add ("Jupiter");
10 lst.add ("Saturn");
11 lst.add ("Uranus");
12 lst.add ("Neptune");
13 lst.add ("Pluto");
14 cnt.add (lst);

```

g. Choice

Les listes déroulantes sont des instances de la classe *java.awt.Choice* et permettent à l'utilisateur de choisir un élément parmi une liste. Pour on construire des instances on utilise:

```

1  public Choice ()

```

Pour manipuler les liste déroulantes on utilise les méthodes:

```

1  void      add (String item)
2  String    getItem (int index)
3  int       getItemCount ()
4  int       getSelectedIndex ()
5  String    getSelectedItem ()

```

Exemple 2.8:

```

1  Panel p= new Panel ();
2  Choice ColorChooser = new Choice ();
3  ColorChooser.add ("Green");
4  ColorChooser.add ("Red");

```

```
5 ColorChooser.add("Blue");  
6 p.add(ColorChooser);
```

h. Exemple récapitulatif

```
1 /* http://www.eeng.dcu.ie/~ee553/ee402notes/html/java/ComponentApplet.  
   java */  
2 package ma.ensao;  
3  
4 import java.awt.*;  
5  
6 public class Fen extends Frame  
7 {  
8     private Button b ;  
9     private Checkbox cb;  
10    private Choice choice;  
11    private Label l ;  
12    private TextField t;  
13    private List lst;  
14    public Fen(String titre){  
15        super(titre);  
16        Panel p =new Panel();  
17        b = new Button("Test Button");  
18        p.add(b);  
19  
20        cb = new Checkbox("Test Checkbox");  
21        p.add(cb);  
22  
23        CheckboxGroup cbg = new CheckboxGroup();  
24        p.add(new Checkbox("CB Item 1", cbg, false));  
25        p.add(new Checkbox("CB Item 2", cbg, false));  
26        p.add(new Checkbox("CB Item 3", cbg, true));  
27  
28        choice = new Choice();  
29        choice.addItem("Choice Item 1");  
30        choice.addItem("Choice Item 2");  
31        choice.addItem("Choice Item 3");  
32        p.add(choice);  
33  
34        l = new Label("Test Label");  
35        p.add(l);  
36  
37        t = new TextField("Test TextField",30);  
38        p.add(t);  
39  
40        lst = new List(4, false);  
41        lst.setMultipleMode(true);  
42        lst.add("Mercury");  
43        lst.add("Venus");  
44        lst.add("Earth");  
45        lst.add("JavaSoft");
```

```
46     lst.add("Mars");
47     lst.add("Jupiter");
48     lst.add("Saturn");
49     lst.add("Uranus");
50     lst.add("Neptune");
51     lst.add("Pluto");
52     p.add(lst);
53
54     this.add(p);
55     pack();
56     setVisible(true);
57 }
58 public static void main(String[] args)
59 {
60     Fen f =new Fen("Exemple Awt");
61 }
62 }
```

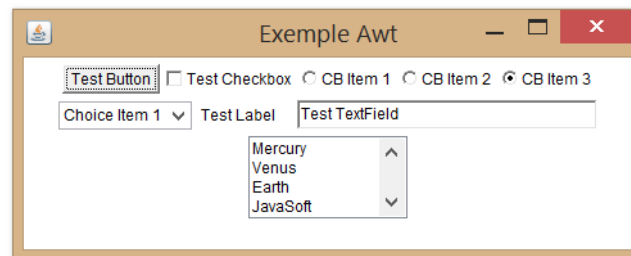


Figure 2.4: Exemple récapitulatif

3. Architecture du Framework Swing

1. Introduction

La création d'interfaces homme-machine avec l'api AWT repose sur l'utilisation de composants graphiques natifs du système d'exploitation (SE) hébergeant la JVM qui exécute le code de votre application awt. Il en découle que le rendu visuel change en fonction du SE utilisé. Pour un langage dont la devise est la portabilité il serait naturelle de vouloir étendre cette portabilité d'exécution au rendu visuel des application graphique.

L'api Swing a été initialement crée car les composants awt étaient très restrictifs pour créer des applications orientée formulaire. Par exemple, il n'était pas possible de créer un bouton sous forme d'image. Swing propose de substituer les composants graphiques awt par ses propres composants. Ces dernier incluent les fonctionnalités de awt et les étends. Sauf quelques exceptions les composants swing reprennent l'appellation de leurs équivalents awt préfixés de la lettre "J".

Swing propose un large ensemble de composants allant du simple label aux éléments graphiques complexes tels que arbres, tables et zones de saisies enrichies. Tous les composants swing dérivent de la classe *javax.swing.JComponent* qui est une classe fille de *java.awt.Container*. Ainsi, il s'avère que swing est une couche logicielle bâtie au dessus de awt.

2. Différences entre awt et Swing

Les composants swing sont qualifiés de légers par oppositions à ceux de awt qu'on qualifie de lourds. Une différence majeure entre les composants lourds et léger est la notion de profondeur ou superposition. Les composants légers sont regroupé au sein d'un composant lourds (conteneur) au sein duquel ils sont régis par le modèle d superposition de swing.

Une caractéristique importante des composants swing est qu'ils sont écrits purement en JAVA. Par conséquent, il n'y a pas recours aux composants graphiques du système d'exploitation. Ainsi, un bouton swing aura la même apparence (look) et fonctionnement sur des plateformes Linux, Windows et Mac. Cette portabilité rend inutile de faire des tests d'ergonomie spécifiques aux plateformes de déploiement.

3. Création d'une IHM Swing et arbre des composants

a. Création d'une application swing

Pour créer une application swing on procède comme suit:

1. Définir les composants graphiques.
2. Placer les composants dans un conteneur.
3. Spécifier le gestionnaire de positionnement.
4. Définir les actions associées aux événements (Listener).
5. Associer ces derniers aux composants.

b. Arbre des composants

Une application swing peut être modélisée par un arbre de composants modélisant les compositions entre les éléments graphiques la formant. Cette approche a pour intérêt de simplifier l'affichage de l'application qui se résume à une succession d'appels de fonctions graphiques dictée par la structure de l'arbre de composants. Les composants swing se conforment au design pattern "composite" illustré par la Figure 3.1.

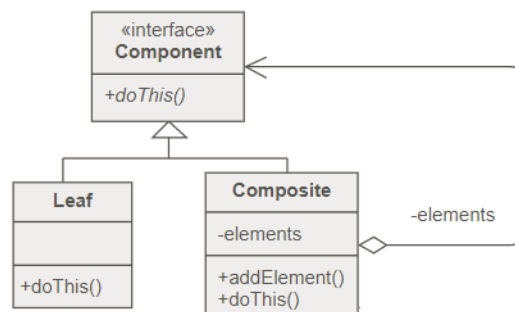


Figure 3.1: Design Pattern composite.

Notez que la classe *javax.swing.JComponent* hérite de *java.awt.Container* et que cette dernière possède une relation de composition avec zéro ou plusieurs instances de *java.awt.Component*, sa classe mère comme illustré sur la Figure 3.2.

javax.swing

Class JComponent

java.lang.Object

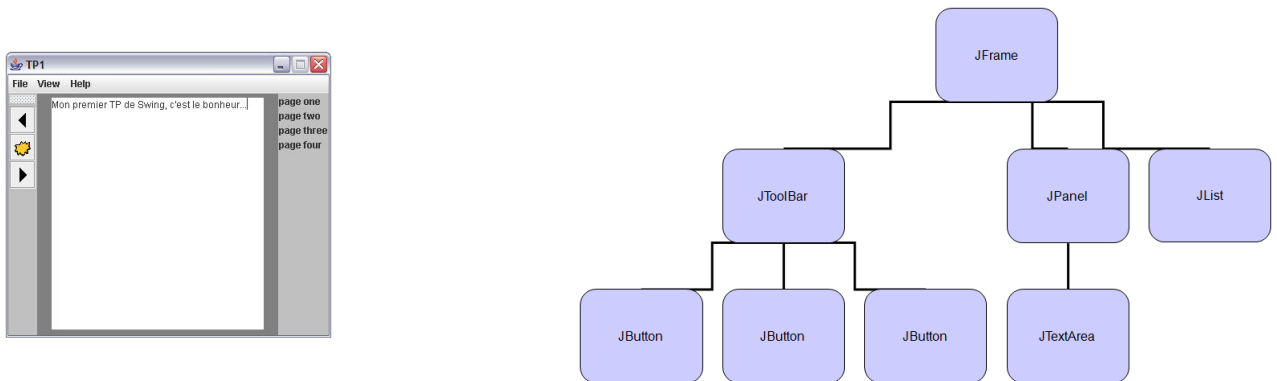
java.awt.Component

java.awt.Container

javax.swing.JComponent

Figure 3.2: Hiérarchie JComponent

Les figures ci-dessous illustrent la correspondance entre une interface swing et son arbre de composants.



4. Les différentes catégories de composants

Un composant swing est un élément graphique indépendant héritant de JComponent. Cette classe assure le support des fonctionnalités communes telles que: look-and-feel adhoc, accessibilité, drag and drop, ...

- Contrôles basiques : Composants atomiques permettant d'interagir avec l'utilisateur (Boutons, Cases à cocher, ...).
- Afficheurs non-éditables : Composants atomiques permettant d'afficher de l'information à l'intention de l'utilisateur.
- Afficheurs éditables avec formatage : Composants atomiques permettant d'afficher de l'information formaté potentiellement éditable par l'utilisateur.

Les conteneurs sont destinés à regrouper un ensemble de composants et fournissent un espace d'affichage à ces derniers. On distingue trois catégories de conteneur:

- Les conteneurs de haut niveau héritant de *java.awt.Container*, il n'est pas possible d'en faire une composition. Les classes JFrame/Jwindow, JDialog et JApplet en sont des exemples. Notons que ces conteneurs remplissent des missions spécifiques (Fenêtre décorée/non décorée, boîte de dialogue et applet.)

- Les conteneurs léger sont des éléments génériques utilisés pour organiser des composants ayant une relation fonctionnel et/ou sémantique. La classe JPanel en est l'exemple type.
 - Les conteneurs à usage spécifique jouent un rôle spécifique dans l'IHM (JTabbedPane, JScrollPane, JSplitPane).
-

5. Architecture MVC des composants Swing

L'architecture Model-View-Controller (MVC) est un motif de conception notamment utilisé pour la création d'IHM. Les composants swing sont formée de trois entités: un modèle, une vue et un contrôleur.

Le modèle est responsable de maintenir toutes les informations liées à l'état d'un composant. Il s'agit par exemple de l'état courant d'un bouton (clické ou non), de la chaîne de caractère saisie dans un champs textuel, Le modèle pourra communiquer indirectement avec le contrôleur et la vue. Cette communication indirecte a lieu via la production d'évènements.

La vue détermine la représentation graphique du modèle d'un composant. Le look fournit par la vue couvre la couleur, la font et l'aspect du composant. Pour un bouton l'aspect est par exemple l'apparence enfoncée suite à click utilisateur. La vue est responsable de mettre à jour la représentation graphique du composant sur l'écran suite à une notification indirecte provenant du modèle (changement d'état) ou d'un message issu du contrôleur.

Le contrôleur détermine si le composant doit réagir ou non à des actions de l'utilisateur généré par la souris ou le clavier par exemple. Le contrôleur régit le comportement du composant et indique les actions réalisées quand un composant est utilisé.

a. Exemple:

Considérons le scénario suivant: une case à cocher de l'interface graphique a été activé par l'utilisateur. Les éléments de l'architecture MVC de la case vont interagir comme suit:

- Le contrôleur détermine que l'utilisateur a réalisé un click de souris et envoi un message à la vue.
- Si la vue juge que le click a eu lieu sur la case à cocher, elle envoi un message au modèle.
- Le modèle met à jour l'état du composant et diffuse un message qui sera intercepté par la vue l'informant qu'elle doit se mettre à jour conformément au nouvel état du modèle.

Cette flexibilité permet de découpler le modèle des vues et contrôleurs. Il est ainsi possible d'associer un même modèle à plusieurs contrôleurs et vues.

Les composants Swing respectent une version compacte du modèle MVC qui associe la vue et le contrôleur dans une entité appelé Délégué UI. Le délégué UI gère l'apparence visuelle du composant et ses interactions avec l'utilisateur. Un délégué UI différent produit

une apparence différente et plusieurs délégués UI peuvent être associés au même modèle.

Exemple de modèles

- ButtonModel : JButton, JMenu, JMenuItem, ...
- Document : JTextPane, JTextArea, JPasswordField, ...
- BoundedRangeModel: utilisé par JProgressBar, JScrollBar, JSlider.
- ListModel utilisé par JList.

6. Pluggable look and feel

Swing contient des ensembles de délégués UI, chacun contenant des implémentations pour la majorité des composants swing. On appelle chacun de ces ensembles look and feel. Il existe trois pluggable look and feel:

- Windows: `com.sun.java.swing.plaf.windows.WindowsLookAndFeel`
- CDE-Motif: `com.sun.java.swing.plaf.motif.MotifLookAndFeel`
- Metal (default): `javax.swing.plaf.metal.MetalLookAndFeel`

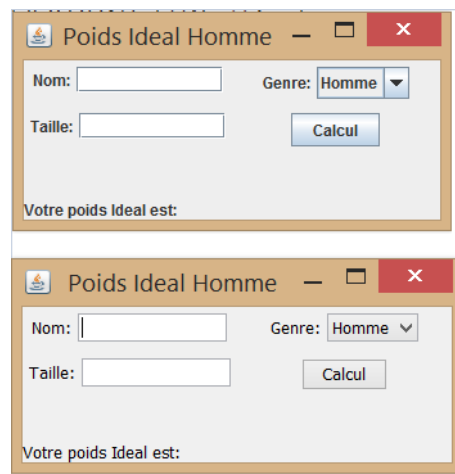
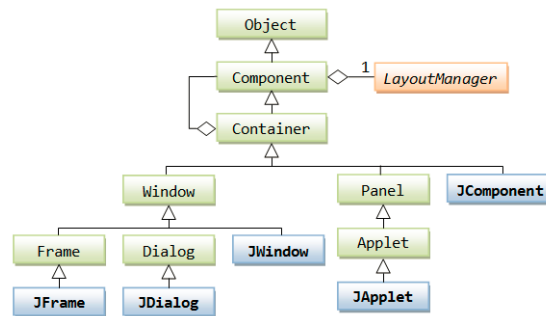


Figure 3.3: System vs Metal plaf.

7. Les conteneurs



a. JFrame

La classe `javax.swing.JFrame` représente une fenêtre décorée qui possède une barre de titre, une barre de menu (optionnelle) et une zone d'affichage du contenu. Le comportement par défaut est qu'elle devient invisible quand l'utilisateur tente de la fermer. Ce comportement peut être changé en utilisant la méthode `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`. Il est possible de spécifier la taille de la JFrame directement en pixels en utilisant la méthode `setSize(int height, int width)`. Toutefois, pour éviter les comportements aberrants on lui préfère la méthode `pack()` pour un calcul de la taille idéale en fonction des tailles préférées des composants élémentaire la formant et du gestionnaire de positionnement utilisé (Organisation visuelle des composants).

Exemple 3.1:

```

1 import java.awt.*;
2 import javax.swing.JButton;
3 import javax.swing.JFrame;
4 import javax.swing.JLabel;
5 import javax.swing.JPanel;
6 public class JFrameExample {
7     public static void main(String s[]) {
8         JFrame frame = new JFrame("JFrame Example");
9         JPanel panel = new JPanel();
10        panel.setLayout(new FlowLayout());
11        JLabel label = new JLabel("JFrame By Example");
12        JButton button = new JButton();
13        button.setText("Button");
14        panel.add(label);
15        panel.add(button);
16        Container c = frame.getContentPane();
17        c.add(panel);
18
19        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20        // frame.setSize(200, 300);
21        pack();
22        frame.setVisible(true);
  
```

```

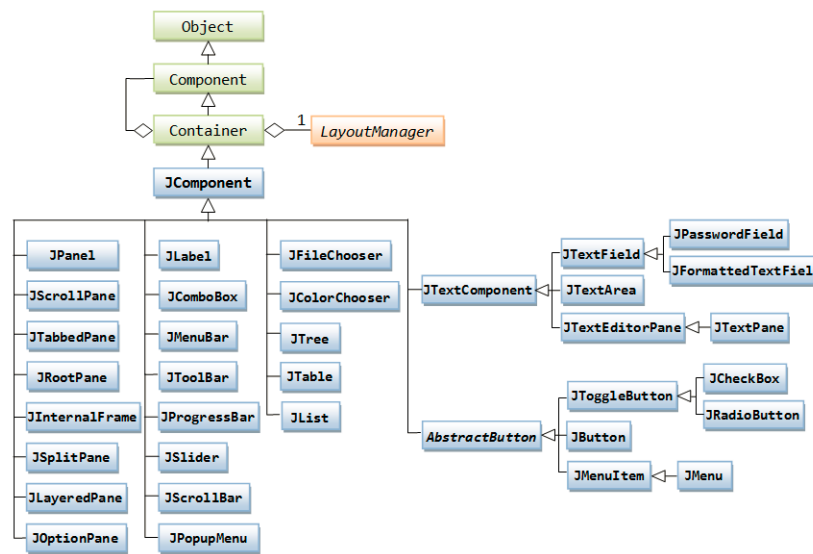
23     }
24 }

```

b. JPanel

La classe *javax.swing.JPanel* représente un conteneur générique. Elle permet d'organiser les éléments qu'elle contient (Composant ou conteneurs) en fonction de son gestionnaire de présentation. En effet pour avoir un rendu visuel complexe mélangeant différents gestionnaires de présentation, il est de coutume de combiner des JPanel avec des Layouts différents.

8. Les composants simples



Tous les composants swing partagent des propriétés communes:

- Trois tailles pour un composant : minimum, maximum, preferred

```

1 public Dimension getMinimumSize() ;
2 public Dimension getMaximumSize() ;
3 public Dimension getPreferredSize() ;
4 public void setMinimumSize(Dimension minimumSize) ;
5 public void setMaximumSize(Dimension maximumSize) ;
6 public void setPreferredSize(Dimension preferredSize) ;

```

- Une info-bulle pour fournir une aide contextuelle sur chaque composant:

```

1 public void setToolTipText (String txt) ;

```

- Un raccourcis clavier

```
1 public void registerKeyboardAction( ActionListener anAction ,  
2   String aCommand, KeyStroke aKeyStroke , int aCondition) ;
```

- Une bordure:

```
1 public void setBorder( Border b);
```

a. Les images

La classe `ImageIcon` implémente l'interface `Icon` et dessine des icônes à partir d'images spécifiées par une URL, un nom de fichier ou un tableau de bytes.

```
1 // Constructeurs  
2 ImageIcon( String nomfichier )  
3 ImageIcon( Image image )  
4 ImageIcon( URL url )  
5  
6 // Methodes  
7 int getIconHeight ()  
8 int getIconWidth ()  
9 Image getImage ()
```

b. Les boutons

Les boutons sont des instances de `JButton` avec différentes possibilités de création:

```
1 JButton ()  
2 JButton( Icon icon )  
3 JButton( String text )  
4 JButton( String text , Icon icon )
```

Les méthodes suivantes permettent de récupérer et de modifier le texte d'un bouton après sa création:

```
1 void setText( String ) // Set the text displayed by the button  
2 String getText() // Get the text displayed by the button
```

Il est aussi possible de spécifier l'alignement horizontal et vertical du texte du bouton avec les méthodes Héritées de `AbstractButton`:

```

1 public void setHorizontalAlignment(int alignment)
2 public int getHorizontalAlignment()
3 public void setVerticalTextPosition(int textPosition)
4 public int getVerticalTextPosition()

```

Les constantes d'alignement prédéfinies sont:

```

1 AbstractButton.RIGHT
2 AbstractButton.LEFT
3 AbstractButton.CENTER
4 AbstractButton.LEADING // Texte a gauche
5 AbstractButton.TRAILING // Texte a droite
6
7 AbstractButton.CENTER // valeur par défaut
8 AbstractButton.TOP
9 AbstractButton.BOTTOM

```

Exemple 3.2:

```

1 ....
2 protected JButton b1, b2, b3;
3 protected ImageIcon createImageIcon(String path,
4                                     String description) {
5     java.net.URL imgURL = getClass().getResource(path);
6     if (imgURL != null) {
7         return new ImageIcon(imgURL, description);
8     } else {
9         System.err.println("Couldn't find file: " + path);
10        return null;
11    }
12 }
13
14 public Demo() {
15     ImageIcon leftButtonIcon = createImageIcon("images/right.gif");
16     ImageIcon middleButtonIcon = createImageIcon("images/middle.gif");
17     ImageIcon rightButtonIcon = createImageIcon("images/left.gif");
18
19     b1 = new JButton("Desactiver bouton du milieu", leftButtonIcon);
20     b1.setVerticalTextPosition(AbstractButton.CENTER);
21     b1.setHorizontalTextPosition(AbstractButton.LEADING);
22
23     b2 = new JButton("Bouton du milieu", middleButtonIcon);
24     b2.setVerticalTextPosition(AbstractButton.BOTTOM);
25     b2.setHorizontalTextPosition(AbstractButton.CENTER);
26
27     b3 = new JButton("Activer bouton du milieu", rightButtonIcon);
28

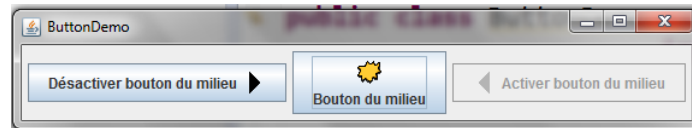
```



```

29     b1.setToolTipText("Click pour desactiver le bouton du milieu.");
30     b2.setToolTipText("Bouton du milieu ne reponds pas au clicks.");
31     b3.setToolTipText("Click pour activer le bouton du milieu.");
32 }

```



c. JCheckBox

Une case à cocher est une instance de JCheckBox et possède un état on/off. Pour créer une case à cocher on peut utiliser l'un des différents constructeurs proposés par la jdk:

```

1 JCheckBox()
2 JCheckBox(Icon icon)
3 JCheckBox(Icon icon, boolean selected)
4 JCheckBox(String text)
5 JCheckBox(String text, boolean selected)
6 JCheckBox(String text, Icon icon)
7 JCheckBox(String text, Icon icon, boolean selected)

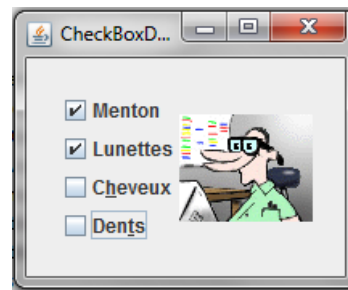
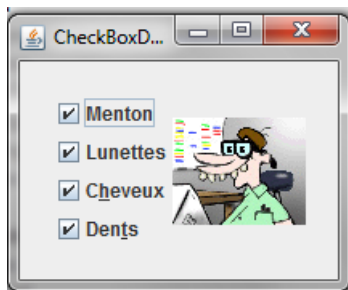
```

Exemple 3.3:

```

1 public CheckBoxDemo() {
2     chinButton = new JCheckBox("menton");
3     chinButton.setSelected(true);
4     glassesButton = new JCheckBox("Lunettes");
5     glassesButton.setSelected(true);
6     hairButton = new JCheckBox("Cheveux");
7     hairButton.setSelected(true);
8     teethButton = new JCheckBox("Dents");
9     teethButton.setSelected(true);
10 }

```



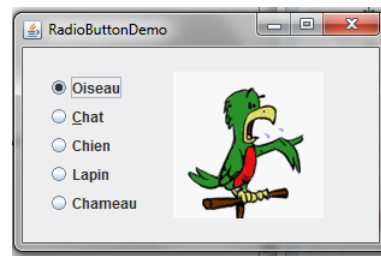
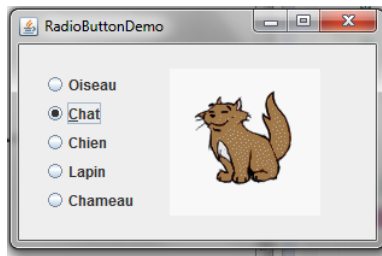
d. JRadioButton

Contrairement à awt, Swing représente un bouton radio par une instance de la classe JRadioButton. Pour conférer un comportement mutuellement exclusif à un ensemble de boutons radio on utilise un objet de type ButtonGroup.

```
1 JRadioButton()
2 //Creates an initially unselected radio button with no set text.
3 JRadioButton(Icon icon)
4 //Creates an initially unselected radio button with the specified image
  but no text.
5 JRadioButton(Icon icon, boolean selected)
6 //Creates a radio button with the specified image and selection state,
  but no text.
7 JRadioButton(String text)
8 //Creates an unselected radio button with the specified text.
9 JRadioButton(String text, boolean selected)
10 //Creates a radio button with the specified text and selection state.
11 JRadioButton(String text, Icon icon)
12 //Creates a radio button that has the specified text and image, and that
   is initially unselected.
13 JRadioButton(String text, Icon icon, boolean selected)
14 //Creates a radio button that has the specified text, image, and
   selection state.
```

Exemple 3.4:

```
1 public RadioButtonDemo() {
2
3     JRadioButton birdButton = new JRadioButton(birdString);
4     birdButton.setActionCommand(birdString);
5     birdButton.setSelected(true);
6     JRadioButton catButton = new JRadioButton(catString);
7     catButton.setActionCommand(catString);
8     JRadioButton dogButton = new JRadioButton(dogString);
9     dogButton.setActionCommand(dogString);
10    JRadioButton rabbitButton = new JRadioButton(rabbitString);
11    rabbitButton.setActionCommand(rabbitString);
12    JRadioButton pigButton = new JRadioButton(pigString);
13    pigButton.setActionCommand(pigString);
14
15    //Grouper les radio buttons.
16    ButtonGroup group = new ButtonGroup();
17    group.add(birdButton);
18    group.add(catButton);
19    group.add(dogButton);
20    group.add(rabbitButton);
21    group.add(pigButton);
22 }
```



e. Les listes

Une *javax.swing.JComboBox* permet à l'utilisateur de choisir une parmi plusieurs options et se présente sous l'une de deux formes: éditable ou non. La deuxième constitue l'option par défaut tandis qu'une zone de texte s'ajoute à la liste déroulante en mode éditable pour permettre en plus des choix, la saisie directe d'une option inexistante.

La classe *JComboBox* expose les méthodes suivantes:

```

1 JComboBox ()
2 JComboBox (ComboBoxModel)
3 JComboBox (Object [])
4 JComboBox (Vector)
5 void addItem (Object)
6 void insertItemAt (Object, int)
7 Object getItemAt (int)
8 Object getSelectedItem ()

```

Exemple 3.5:

```

1 String [] patternExamples = {
2     "dd MMMM yyyy",
3     "dd.MM.yy",
4     "MM/dd/yy",
5     "yyyy.MM.dd G 'at' hh:mm:ss z",
6     "EEE, MMM d, ''yy",
7     "h:mm a",
8     "H:mm:ss:SSS",
9     "K:mm a,z",
10    "yyyy.MMMM.dd GGG hh:mm aaa"
11 };
12 . . .
13 JComboBox patternList = new JComboBox(patternExamples);
14 patternList.setEditable(true);

```

En addition à la classe *JComboBox* on dispose de la classe *javax.swing.JList* qui fournit à l'utilisateur une liste d'objets affichés sur une ou plusieurs colonnes pour en choisir un ou plusieurs éléments.

La classe *JList* expose les méthodes suivantes:

```

1 JList ()
2 JList (E[] listData)

```

```

3 JList(ListModel<E> dataModel)
4 int getSelectedIndex()
5 int[] getSelectedIndices()
6 E getSelectedValue()
7 void setSelectedIndex(int index)
8 void setSelectedIndices(int[] indices)
9 void setSelectionMode(int selectionMode)

```

Notons qu'il existe trois mode de sélection:

```

1 ListSelectionModel.SINGLE_SELECTION
2 ListSelectionModel.SINGLE_INTERVAL_SELECTION
3 ListSelectionModel.MULTIPLE_INTERVAL_SELECTION

```

Le mode par défaut étant: *ListSelectionModel.SINGLE_SELECTION*.

Exemple 3.6:

```

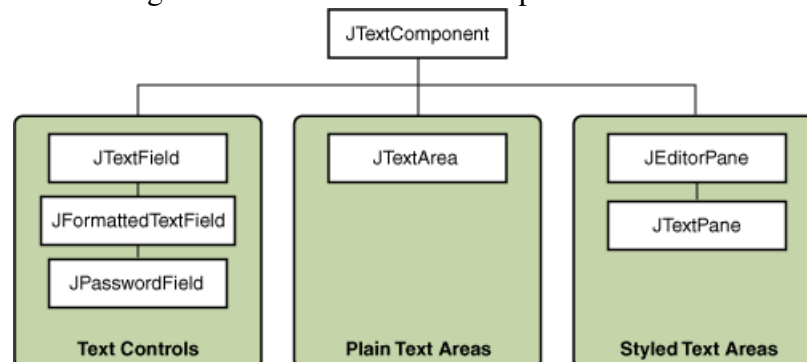
1 String[] data = {"one", "two", "three", "four"};
2 JList<String> myList = new JList<String>(data);

```

f. Les champs textuels

La Figure 3.4 illustre la hiérarchie des composants textuels de l'api Swing.

Figure 3.4: Hiérarchie des composants textuels



Les zones de texte mono-ligne sont des instances de *JTextField* à créer avec l'un des constructeurs:

```

1 JTextField()
2 JTextField(Document doc, String text, int columns)
3 JTextField(int columns)
4 JTextField(String text)
5 JTextField(String text, int columns)

```

En vue de manipuler les instances de *JTextField*, on dispose des méthodes suivantes:

```
1 int getColumns ()
2 protected int getColumnWidth ()
3 int getHorizontalAlignment ()
4 void setHorizontalAlignment (int alignment)
5 void setColumns (int columns)
```

Pour l'alignement les constantes suivantes peuvent être utilisées:

```
1 JTextField.LEFT
2 JTextField.CENTER
3 JTextField.RIGHT
4 JTextField.LEADING
5 JTextField.TRAILING
```

Notons que les saisies de l'utilisateur dans les zones *JTextField* ne sont pas formatées. En effet, il n'est pas possible de forcer ce dernier à saisir des données dans un format spécifique (data, pourcentage, ...). Pour obtenir un formatage des zones textuels, il faut plutôt utiliser des instances de *JFormattedTextField*:

```
1 JFormattedTextField ()
2 JFormattedTextField (Format format)
3 JFormattedTextField (JFormattedTextField.AbstractFormatter formatter)
4 JFormattedTextField (JFormattedTextField.AbstractFormatterFactory fact)
5 JFormattedTextField (JFormattedTextField.AbstractFormatterFactory fact)
```

Les classe *NumberFormat* est une classe fille de *Format* correspondant au format numérique. Elle expose des méthodes publiques pour fixer le nombre de digits formant la partie entière et la partie réelle.

Exemple 3.7:

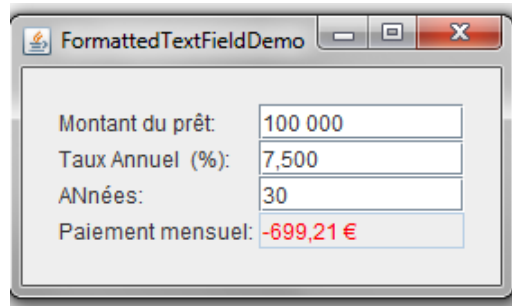
```
1 private void setUpFormats () {
2     amountFormat = NumberFormat.getNumberInstance ();
3     percentFormat = NumberFormat.getNumberInstance ();
4     percentFormat.setMinimumFractionDigits (3);
5     paymentFormat = NumberFormat.getCurrencyInstance ();
6 }
7 ...
8 private double amount = 100000;
9 private double rate = 7.5; // 7.5%
10 private int numPeriods = 30;
11 private JLabel amountLabel;
12 private JLabel rateLabel;
13 private JLabel numPeriodsLabel;
14 private JLabel paymentLabel;
15 private static String amountString = "Montant du pret: ";
```

```
16     private static String rateString = "Taux Annuel (%): ";
17     private static String numPeriodsString = "Anees: ";
18     private static String paymentString = "Paieement mensuel: ";
19     private JFormattedTextField amountField;
20     private JFormattedTextField rateField;
21     private JFormattedTextField numPeriodsField;
22     private JFormattedTextField paymentField;
23
24     private NumberFormat amountFormat;
25     private NumberFormat percentFormat;
26     private NumberFormat paymentFormat;
27
28     public FormattedTextFieldDemo() {
29         setUpFormats();
30         double payment = computePayment(amount, rate, numPeriods);
31         amountLabel = new JLabel(amountString);
32         rateLabel = new JLabel(rateString);
33         numPeriodsLabel = new JLabel(numPeriodsString);
34         paymentLabel = new JLabel(paymentString);
35         amountField = new JFormattedTextField(amountFormat);
36         amountField.setValue(new Double(amount));
37
38         amountField.setColumns(10);
39         rateField = new JFormattedTextField(percentFormat);
40         rateField.setValue(new Double(rate));
41         rateField.setColumns(10);
42         rateField.addPropertyChangeListener("value", this);
43         numPeriodsField = new JFormattedTextField();
44         numPeriodsField.setValue(new Integer(numPeriods));
45         numPeriodsField.setColumns(10);
46         numPeriodsField.addPropertyChangeListener("value", this);
47         paymentField = new JFormattedTextField(paymentFormat);
48         paymentField.setValue(new Double(payment));
49         paymentField.setColumns(10);
50         paymentField.setEditable(false);
51         paymentField.setForeground(Color.red);
52         amountLabel.setLabelFor(amountField);
53         rateLabel.setLabelFor(rateField);
54         numPeriodsLabel.setLabelFor(numPeriodsField);
55         paymentLabel.setLabelFor(paymentField);
56
57         JPanel labelPane = new JPanel(new GridLayout(0,1));
58         labelPane.add(amountLabel);
59         labelPane.add(rateLabel);
60         labelPane.add(numPeriodsLabel);
61         labelPane.add(paymentLabel);
62         JPanel fieldPane = new JPanel(new GridLayout(0,1));
63         fieldPane.add(amountField);
64         fieldPane.add(rateField);
65         fieldPane.add(numPeriodsField);
66         fieldPane.add(paymentField);
67         setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
```

```

68         add(labelPane , BorderLayout.CENTER);
69         add(fieldPane , BorderLayout.LINE_END);
70     }
71
72 }

```



9. Les composants swing élaborés

a. Les arbres

Les arbres sont des instance de la classe `JTree` qui fournit une représentation hiérarchique de données. On utilise les arbres pour représenter graphiquement des données hérarchisées avec des relation de type père→#fils. Il s'agit par exemple de la représentation graphique d'un workspace dans Eclipse regroupant les projets et les fichiers et autres ressources les composant.

La création d'une instance de type `JTree` requiert l'utilisation d'un certains nombre de classe annexes à choisir parmi:

- `TreeModel` : contient les données représenté dans l'arbre
- `TreeNode` : implémentation des noeuds et de la structure d'arbre
- `TreeSelectionModel` : Contient le ou les noeuds sélectionnés
- `TreePath` : Contient le chemin de la racine de l'arbre.
- `TreeCellRenderer` : Appelé pour dessiner un noeud
- `TreeCellEditor` : Editeur pour un noeud (éditable)
- `TreeUI` : look-and-feel

Toujours, dans le paradigme MVC, le modèle représente l'état du composant. Il s'agit de la structure qui permet de stocker les données de ce dernier. La classe `TreeModel` expose les méthodes suivantes:

```

1 //Retourne l'enfant d'ordre index
2 public Object getChild(Object parent , int index);
3 // Retourne le noeud parent
4 public Object getRoot();
5 //Pour verifier q'un objet est une feuille
6 public boolean isLeaf(Object node);

```

En effet, tout modèle arborescent pourra être parcouru entièrement en utilisant les trois méthodes citées plus-haut avec une approche récursive.

Divers modèles de sélection sont disponibles:

- sélection d'un seul élément
- sélection de plusieurs éléments contigus
- sélection de plusieurs éléments non contigus

La classe `JTree` fournit une vue du modèle. Pour personnaliser l'affichage des noeuds de l'arbre on passe par un objet `TreeCellRenderer`. Aussi, en vue de personnaliser l'édition des noeuds de l'arbre on passe par un objet `TreeCellEditor`

Pour construire un arbre on utilise l'un des constructeurs suivants:

```
1 JTree ()
2 JTree (TreeNode racine)
3 JTree (TreeNode racine , boolean allowsChildren)
4 JTree (TreeModel modele)
5 JTree (TreeModel modele , boolean allowsChildren)
```

La classe `DefaultMutableTreeNode` fournit le modèle d'un noeud.

```
1 DefaultMutableTreeNode ()
2 DefaultMutableTreeNode (Object userObject)
3 DefaultMutableTreeNode (Object userObject , boolean allowsChildren)
```

Le contenu d'un `userObject` est affiché via appel de la méthode `toString()`.

Exemple 3.8:

```
1 class Arbre extends JPanel {
2     JTree tree;
3     public Arbre () {
4         DefaultMutableTreeNode top, noeud, fils, n;
5         top = new DefaultMutableTreeNode("Top");
6         tree = new JTree(top);
7         noeud = new DefaultMutableTreeNode("Repertoire 1");
8         top.add(noeud);
9         n = new DefaultMutableTreeNode("1a"); noeud.add(n);
10        n = new DefaultMutableTreeNode("1b"); noeud.add(n);
11
12        noeud = new DefaultMutableTreeNode("Repertoire 2");
13        top.add(noeud);
14        n = new DefaultMutableTreeNode("2a"); noeud.add(n);
15
16        fils = new DefaultMutableTreeNode("2d"); noeud.add(fils);
17        n = new DefaultMutableTreeNode("3a"); fils.add(n);
18    }
19 }
```

Le rendu visuel est réalisé par défaut par une instance de `DefaultTreeCellRenderer`


```

1  ...
2  JTree tree = new JTree(top);
3  DefaultTreeCellRenderer rd = (DefaultTreeCellRenderer) tree .
   getCellRenderer();

```

Pour modifier le rendu visuel on peut utiliser le snippet de code ci-dessous:

```

1  void setBackground(Color color)
2  void setBackgroundNonSelectionColor(Color newColor)
3  void setBackgroundSelectionColor(Color newColor)
4  void setBorderSelectionColor(Color newColor)
5  void setClosedIcon(Icon newIcon)
6  void setFont(Font font)
7  void setLeafIcon(Icon newIcon)
8  void setOpenIcon(Icon newIcon)
9  void setTextNonSelectionColor(Color newColor)
10 void setTextSelectionColor(Color newColor)

```

Un objet de type `TreeSelectionListener` informe sur les changements dans les sélections de l'arbre:

```

1  class Selecteur implements TreeSelectionListener {
2      public void valueChanged( TreeSelectionEvent e ) {
3          message.setText( "Nouveau : " + e.getNewLeadSelectionPath() );
4      }
5  }
6  tree.addTreeSelectionListener(new Selecteur());

```

Le parcours d'un arbre se fait de manière récursive selon l'une des méthodes suivantes:

```

1  breadthFirstEnumeration
2  depthFirstEnumeration
3  postorderEnumeration
4  preorderEnumeration

```

Exemple 3.9:

```

1  DefaultMutableTreeNode n, top;
2  Enumeration e;
3  top = (DefaultMutableTreeNode) tree.getModel().getRoot();
4  e = top.breadthFirstEnumeration();
5  while (e.hasMoreElements()) {
6      n = (DefaultMutableTreeNode) e.nextElement();
7      System.out.println(n.getUserObject()+" ");
8  }

```

Exemple 3.10:

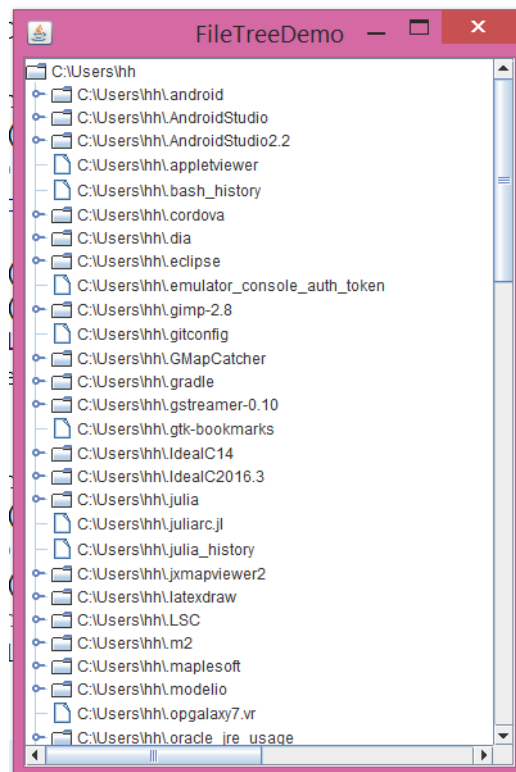
```
1 import javax.swing.*;
2 import javax.swing.event.*;
3 import javax.swing.tree.*;
4
5 import java.awt.BorderLayout;
6 import java.io.File;
7
8 public class FileTreeDemo {
9     public static void main(String[] args) {
10         // Figure out where in the filesystem to start displaying
11         File root;
12         if (args.length > 0) root = new File(args[0]);
13         else root = new File(System.getProperty("user.home"));
14
15         // Create a TreeModel object to represent our tree of files
16         FileTreeModel model = new FileTreeModel(root);
17
18         // Create a JTree and tell it to display our model
19         JTree tree = new JTree();
20         tree.setModel(model);
21
22         // The JTree can get big, so allow it to scroll
23         JScrollPane scrollpane = new JScrollPane(tree);
24
25         // Display it all in a window and make the window appear
26         JFrame frame = new JFrame("FileTreeDemo");
27         frame.getContentPane().add(scrollpane, BorderLayout.CENTER);
28         frame.setSize(400,600);
29         frame.setVisible(true);
30     }
31 }
32
33 class FileTreeModel implements TreeModel {
34     protected File root;
35     public FileTreeModel(File root) { this.root = root; }
36
37     public Object getRoot() { return root; }
38
39     public boolean isLeaf(Object node) { return ((File)node).isFile(); }
40
41     public int getChildCount(Object parent) {
42         String[] children = ((File)parent).list();
43         if (children == null) return 0;
44         return children.length;
45     }
46
47     public Object getChild(Object parent, int index) {
48         String[] children = ((File)parent).list();
49         if ((children == null) || (index >= children.length)) return null;
50         return new File((File) parent, children[index]);
```

```

51 }
52
53 public int getIndexOfChild(Object parent, Object child) {
54     String[] children = ((File)parent).list();
55     if (children == null) return -1;
56     String childname = ((File)child).getName();
57     for(int i = 0; i < children.length; i++) {
58         if (childname.equals(children[i])) return i;
59     }
60     return -1;
61 }
62 public void valueForPathChanged(TreePath path, Object newvalue) {}
63 public void addTreeModelListener(TreeModelListener l) {}
64 public void removeTreeModelListener(TreeModelListener l) {}
65 }

```

Le code ci-dessous produit le résultat suivant:



b. Les tables

La classe `JTable` affiche des données issues d'un modèle tabulaire `TableModel` dans un tableau. Un exemple est le classeur Excel de Microsoft (ou Calc de OpenOffice :)) Il peut s'agir par exemple d'un modèle par défaut (`DefaultTableModel`) basé sur un tableau d'objet bi-dimensionnel: `Object[][]`. Pour des modèles tabulaires basés sur d'autres structures

qu'une matrice bi-dimensionnel d'objet il faudra étendre *AbstractTableModel*. La sélection est régie par un modèle de sélection est un modèle de colonnes est utilisé.

Pour construire une instance *JTable* on utilise l'un des constructeurs suivants:

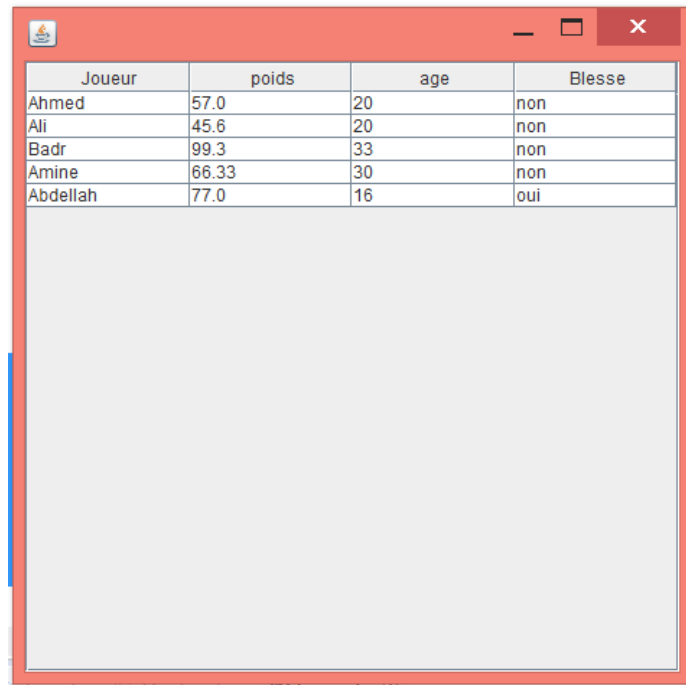
- *JTable()* modèles par défaut pour les trois modèles
- *JTable(int numRows, int numColumns)* avec autant de cellules vides
- *JTable(Object[][] rowData, Object[] columnNames)* avec les valeurs des cellules de *rowData* et noms de colonnes *columnNames*.
- *JTable(TableModel dm)* avec le modèle de données *dm*, les autres par défaut.
- *JTable(TableModel dm, TableColumnModel cm)* avec modèle de données et modèle de colonnes fournis.
- *JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)* Les trois modèles sont fournis.
- *JTable(Vector rowData, Vector columnNames)* ici, les données sont fournies par colonne.

Exemple 3.11:

```

1 import java.awt.BorderLayout;
2 import javax.swing.*;
3 public class Table extends JPanel {
4     private Object[][] cellules = {
5         { "Ahmed", new Double(57), new Integer(20), "non" },
6         { "Ali", new Double(45.6), new Integer(20), "non" },
7         { "Badr", new Double(99.3), new Integer(33), "non" },
8         { "Amine", new Double(66.33), new Integer(30), "non" },
9         { "Abdellah", new Double(77), new Integer(16), "oui" }
10    };
11    private String[] columnNames = { "Joueur", "poids", "age", "Blesse" };
12
13    Table() {
14        setLayout(new BorderLayout());
15        JTable table = new JTable(cellules, columnNames);
16        add(new JScrollPane(table), BorderLayout.CENTER);
17    }
18    public static void main(String[] args){
19        JFrame f = new JFrame();
20        f.setContentPane(new Table());
21        f.pack();
22        f.setVisible(true);
23    }
24 }
```

Le code ci-dessous produit le résultat suivant:



Joueur	poids	age	Blessé
Ahmed	57.0	20	non
Ali	45.6	20	non
Badr	99.3	33	non
Amine	66.33	30	non
Abdellah	77.0	16	oui

Comme pour les arbres, Le JTable utilise un modèle pour stocker les données:

- Les données sont accessibles par un modèle. Ils peuvent être stockés ou calculés, de façon transparente.
- La classe AbstractTableModel implémente les méthodes d'un modèle de table:

```

1 public int getRowCount()
2 // Retourne le nombre de lignes
3
4 public int getColumnCount()
5 // Retourne le nombre de colonnes
6
7 public Object getValueAt(int ligne, int colonne)
8 // Retourne l'objet à afficher dans les ligne et colonne indiquées
9 // via appel de sa méthode toString.
```

Exemple 3.12:

```

1 import java.awt.BorderLayout;
2 import javax.swing.*;
3 import javax.swing.table.*;
4 class MonModele extends AbstractTableModel{
5     int nbC, nbL;
6     public MonModele(int nbC, int nbL){
7         this.nbC=nbC; this.nbL=nbL;
8     }
9     public int getRowCount() {
10         return nbL;
```

```

11     }
12     public int getColumnCount() {
13         return nbC;
14     }
15     public Object getValueAt(int rowIndex, int columnIndex) {
16         return new Integer((1+rowIndex)*(1+columnIndex));
17     }
18 }
19 public class SimpleTable extends JPanel {
20     MonModele dataModel =new MonModele(10, 10);
21     SimpleTable() {
22         setLayout(new BorderLayout());
23         JTable table = new JTable(dataModel);
24         add(new JScrollPane(table));
25     }
26     public static void main(String[] args){
27         JFrame f = new JFrame();
28         f.setContentPane(new SimpleTable());
29         f.pack();
30         f.setVisible(true);
31     }
32 }

```

	A	B	C	D	E	F	G	H	I	J
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Exemple 3.13:

```

1 import java.text.NumberFormat;
2 import java.util.Locale;
3

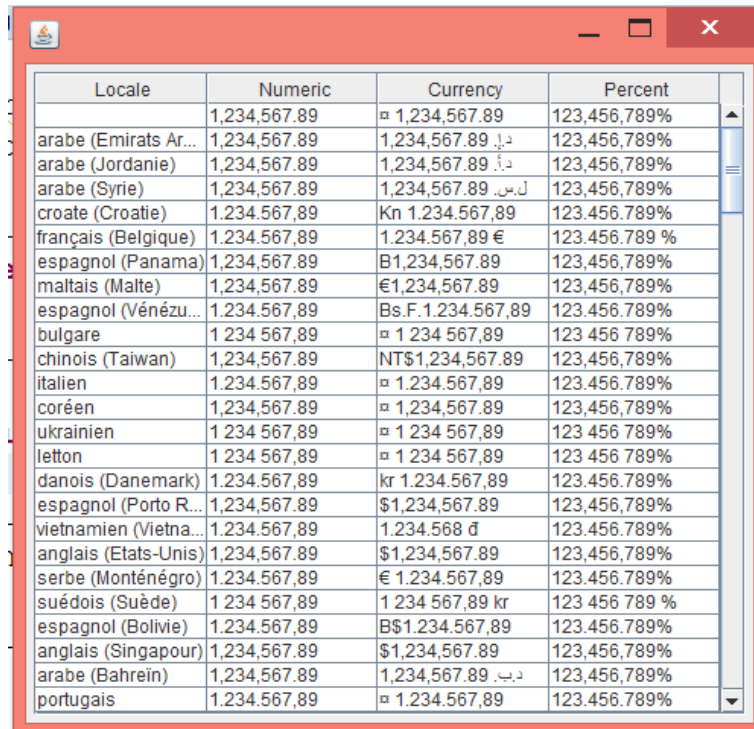
```

```
4 import javax.swing.*;
5 import javax.swing.JTable;
6 import javax.swing.table.*;
7
8 class LocaleTableModel extends AbstractTableModel {
9     protected double currentValue = 1234567.89;
10
11     protected static Locale[] availableLocales=Locale.
        getAvailableLocales();
12
13     public final static int LOCALE_COLUMN = 0;
14     public final static int NUMBER_COLUMN = 1;
15     public final static int CURRENCY_COLUMN = 2;
16     public final static int PERCENT_COLUMN = 3;
17
18     public final static String[] columnHeaders = { "Locale", "Numeric"
        , "Currency", "Percent" };
19
20
21     public int getRowCount() {
22         return availableLocales.length;
23     }
24
25     public int getColumnCount() {
26         return columnHeaders.length;
27     }
28
29     public Object getValueAt(int row, int column) {
30         Locale locale = availableLocales[row];
31         NumberFormat formatter = NumberFormat.getNumberInstance();
32         switch (column) {
33             case LOCALE_COLUMN:
34                 return locale.getDisplayName();
35             case NUMBER_COLUMN:
36                 formatter = NumberFormat.getNumberInstance(locale);
37                 break;
38             case CURRENCY_COLUMN:
39                 formatter = NumberFormat.getCurrencyInstance(locale);
40                 break;
41             case PERCENT_COLUMN:
42                 formatter = NumberFormat.getPercentInstance(locale);
43                 break;
44         }
45         return formatter.format(currentValue);
46     }
47
48     public String getColumnName(int column) {
49         return columnHeaders[column];
50     }
51 }
52
53 public class NumberViewer extends JPanel {
```

```

54 protected AbstractTableModel tableModel;
55
56 public NumberViewer() {
57     tableModel = new LocaleTableModel();
58     JTable table = new JTable(tableModel);
59     add(new JScrollPane(table));
60 }
61 public static void main(String[] args) {
62     JFrame f = new JFrame("");
63     f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
64     f.getContentPane().add(new NumberViewer());
65     f.pack();
66     f.setVisible(true);
67 }
68
69 }

```



Locale	Numeric	Currency	Percent
	1,234,567.89	¤ 1,234,567.89	123,456,789%
arabe (Emirats Ar...)	1,234,567.89	1,234,567.89 د.إ.	123,456,789%
arabe (Jordanie)	1,234,567.89	1,234,567.89 د.إ.	123,456,789%
arabe (Syrie)	1,234,567.89	1,234,567.89 ل.س.	123,456,789%
croate (Croatie)	1.234.567,89	Kn 1.234.567,89	123.456.789%
français (Belgique)	1.234.567,89	1.234.567,89 €	123.456.789 %
espagnol (Panama)	1,234,567.89	B1,234,567.89	123,456,789%
maltais (Malte)	1,234,567.89	€1,234,567.89	123,456,789%
espagnol (Vénézu...)	1.234.567,89	Bs.F. 1.234.567,89	123.456.789%
bulgare	1 234 567,89	¤ 1 234 567,89	123 456 789%
chinois (Taiwan)	1,234,567.89	NT\$1,234,567.89	123,456,789%
italien	1,234,567.89	¤ 1,234,567,89	123,456,789%
coréen	1,234,567.89	¤ 1,234,567,89	123,456,789%
ukrainien	1 234 567,89	¤ 1 234 567,89	123 456 789%
letton	1 234 567,89	¤ 1 234 567,89	123 456 789%
danois (Danemark)	1,234,567.89	kr 1,234,567,89	123,456,789%
espagnol (Porto R...)	1,234,567.89	\$1,234,567.89	123,456,789%
vietnamien (Vietna...)	1,234,567.89	1,234,568 đ	123,456,789%
anglais (Etats-Unis)	1,234,567.89	\$1,234,567.89	123,456,789%
serbe (Monténégro)	1,234,567.89	€ 1,234,567,89	123,456,789%
suédois (Suède)	1 234 567,89	1 234 567,89 kr	123 456 789 %
espagnol (Bolivie)	1,234,567.89	B\$1,234,567,89	123,456,789%
anglais (Singapour)	1,234,567.89	\$1,234,567.89	123,456,789%
arabe (Bahrein)	1,234,567.89	1,234,567.89 د.ب.	123,456,789%
portugais	1.234.567,89	¤ 1.234.567,89	123.456.789%

4. Les gestionnaires de présentation

Pour des raisons d'ergonomie, il est fortement déconseillé de décrire la position des composants graphiques explicitement dans le référentiel d'affichage (Ecran 2D). En effet, toute modification de la taille de la fenêtre entraînera des incohérences (zones vides/cachées, ...) et affectera l'aspect visuelle de cette dernière. La solution à cette problématique passe par l'utilisation d'objet JAVA appelé gestionnaires de positionnement (LayoutManager). Ces objets sont associés à des conteneurs et décident de la manière avec laquelle les composants qu'ils contiennent vont être présentés à l'utilisateur. Un avantage majeur de cette approche est que le gestionnaire de présentation adaptera l'affichage des composants à toute modification de la zone d'affichage en fonction de l'espace disponible en essayant de grader une cohérence visuelle.

On distingue deux familles de gestionnaires de positionnement à savoir les LayoutManager avec et sans contraintes.

1. LayoutManager sans contrainte :

a. FlowLayout

C'est le layout manager par défaut des JPanel. Il affiche les composants à leur taille préférée, "de la gauche vers la droite", et revient à la ligne si nécessaire. L'ordre des composants est celui de leur ajout dans le container.

```
1 FlowLayout(int align, int hgap, int vgap)
2 FlowLayout(int align)
3 FlowLayout()
```

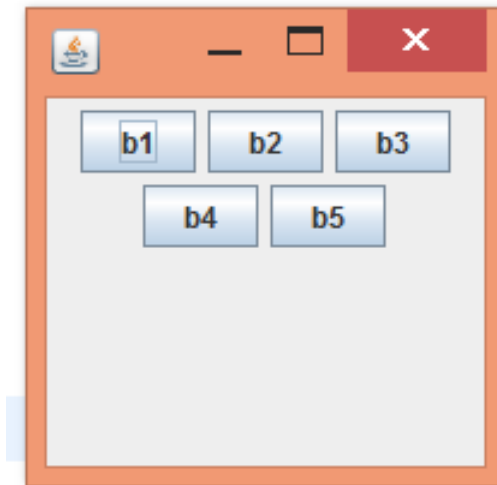
L'alignement sur chaque ligne peut prendre l'une des valeurs prédéfinies suivantes:

1 `FlowLayout.LEADING, FlowLayout.CENTER` ou `FlowLayout.TRAILING`

Aussi, il est possible de contrôler l'espacement vertical et horizontal entre les composants que gère le `FlowLayout` en spécifiant les valeurs *hgap/vgap*.

Exemple 4.1:

```
1 import java.awt.*;
2
3 import javax.swing.*;
4 public class FL extends JFrame{
5     protected JButton b1,b2,b3,b4,b5;
6     public FL(){
7         b1=new JButton("b1");
8         b2=new JButton("b2");
9         b3=new JButton("b3");
10        b4=new JButton("b4");
11        b5=new JButton("b5");
12
13        JPanel p =new JPanel();
14        //JPanel p =new JPanel(new FlowLayout());
15
16        p.add(b1);
17        p.add(b2);
18        p.add(b3);
19        p.add(b4);
20        p.add(b5);
21
22        setContentPane(p);
23        setSize(200,200);
24        setVisible(true);
25    }
26    public static void main(String[] args){
27        FL f=new FL();
28    }
29 }
```



b. GridLayout

Avec Un *GridLayout*, les composants sont distribués dans une grille conformément à leur ordre d'ajout au conteneur. Les composants sont affichés avec la même taille et occupe la totalité de la case réservée pour leur affichage. Le layout s'adapte si le nombre de composants est plus petit que celui des cases.

```
1 GridLayout(int rows, int cols, int hgap, int vgap)
2 GridLayout(int rows, int cols)
3 - rows/columns=lignes/colonnes
4 - hgap/vgap: espace entre les composants
```

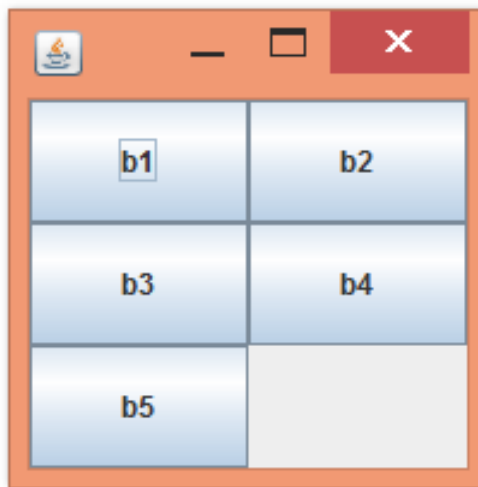
Exemple 4.2:

```
1 import java.awt.*;
2
3 import javax.swing.*;
4 public class GL extends JFrame{
5     protected JButton b1, b2, b3, b4, b5;
6     public GL(){
7         b1=new JButton("b1");
8         b2=new JButton("b2");
9         b3=new JButton("b3");
10        b4=new JButton("b4");
11        b5=new JButton("b5");
12
13        JPanel p =new JPanel(new GridLayout(3,2));
14
15        p.add(b1);
16        p.add(b2);
17        p.add(b3);
18        p.add(b4);
19        p.add(b5);
```

```

20         setContentPane(p);
21         setSize(200,200);
22         setVisible(true);
23     }
24     public static void main(String[] args){
25         GL f=new GL();
26     }
27 }
28

```



c. BorderLayout

BoxLayout affiche les composants en ligne ou en colonne, selon leur taille préférée. L'ordre d'affichage est l'ordre d'ajout dans le container

```

1  // On commence par specifier un layout null
2  JPanel p=new JPanel(null);
3  p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
4  c.setAlignmentX(0f);
5  /* 0f alignement a gauche
6     1f alignement a droite
7     0.5f alignement au centre
8  */

```

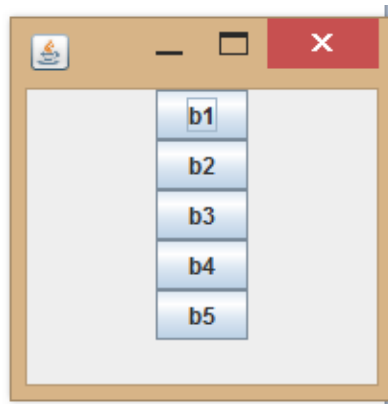
Exemple 4.3:

```

1  import java.awt.*;
2  import javax.swing.*;
3  public class BXL extends JFrame{
4      protected JButton b1,b2,b3,b4,b5;
5      public BXL(){

```

```
6      b1=new JButton("b1");
7      b2=new JButton("b2");
8      b3=new JButton("b3");
9      b4=new JButton("b4");
10     b5=new JButton("b5");
11     JPanel p =new JPanel();
12     p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
13     b1.setAlignmentX(0.5f);
14     p.add(b1);
15     b2.setAlignmentX(0.5f);
16     p.add(b2);
17     b3.setAlignmentX(0.5f);
18     p.add(b3);
19     b4.setAlignmentX(0.5f);
20     p.add(b4);
21     b5.setAlignmentX(0.5f);
22     p.add(b5);
23     setContentPane(p);
24     setSize(200,200);
25     setVisible(true);
26 }
27 public static void main(String[] args){
28     BXL f=new BXL();
29 }
30 }
```



2. LayoutManager avec contrainte :

La particularité de ces gestionnaires de positionnement et l' possibilité de spécifier une contrainte de placement lors de l'ajout d'un composant au container qui les utilisent. La méthode d'ajout *add* se voit attribuer un second argument:

```
1 add(java.awt.Component, Object constraint)
```

a. BorderLayout

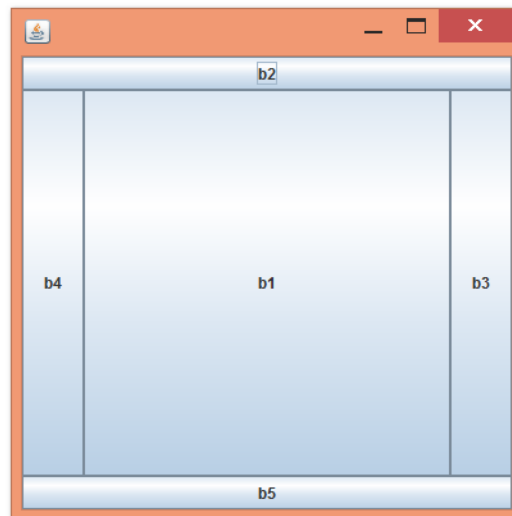
C'est le Layout par défaut du contentPane d'une JFrame. On rappelle que le contentPane est de type *java.awt.Container*. cinq zone de placement composent le *BorderLayout*:

- BorderLayout.NORTH
- BorderLayout.SOUTH
- BorderLayout.EAST
- BorderLayout.WEST
- BorderLayout.CENTER (valeur par défaut)

Notons que la capacité de chaque zone est exactement d'un seul composant. L'utilisation d'un JPanel permet de contourner cette limitation.

Exemple 4.4:

```
1 import java . awt . BorderLayout ;
2
3 import javax . swing . * ;
4 public class BL extends JFrame {
5     protected JButton b1 , b2 , b3 , b4 , b5 ;
6     public BL () {
7         b1 = new JButton ( " b1 " ) ;
8         b2 = new JButton ( " b2 " ) ;
9         b3 = new JButton ( " b3 " ) ;
10        b4 = new JButton ( " b4 " ) ;
11        b5 = new JButton ( " b5 " ) ;
12
13        JPanel p = new JPanel ( new BorderLayout () ) ;
14
15        p . add ( b1 , BorderLayout . CENTER ) ;
16        p . add ( b2 , BorderLayout . NORTH ) ;
17        p . add ( b3 , BorderLayout . EAST ) ;
18        p . add ( b4 , BorderLayout . WEST ) ;
19        p . add ( b5 , BorderLayout . SOUTH ) ;
20
21        setContentPane ( p ) ;
22        setSize ( 400 , 400 ) ;
23        setVisible ( true ) ;
24    }
25    public static void main ( String [] args ) {
26        BL f = new BL () ;
27    }
28 }
```



b. GridBagLayout

Positionnement des composants selon une grille, dans l'ordre d'ajout, et en tenant compte de contraintes sur:

- la zone occupée par un composant
- le placement d'un composant dans sa zone
- le comportement de la zone en cas de redimensionnement

Les contraintes sont exprimées par une instance de `GridBagConstraints`. Comme la méthode `add` copie la contrainte, on peut réutiliser le même objet pour plusieurs contraintes, en ne modifiant que le nécessaire:

```
1 GridBagConstraints gbc=new GridBagConstraints();
2 gbc.gridwidth=3;
3 gbc.fill=GridBagConstraints.BOTH;
4 gbc.weightx=1f;
5 gbc.weighty=1f;
6 p.add(new JButton("J'occupe trois cases"),gbc);
7 gbc.gridwidth=GridBagConstraints.REMAINDER;
8 p.add(new JButton("J'occupe une seule case"),gbc);
```

Le `GridBoxLayout` offre beaucoup de propriétés et donc plus d'options de configurations que le `GridLayout`. Ainsi, la zone occupée par un composant est une portion rectangulaire de la grille et les cases n'ont pas toutes la même taille. De plus, les attributs `gridx` et `gridy` définissent les coordonnées de la cellule dans la zone d'affichage. La constante `GridBagConstraints.RELATIVE` permet le composant sera placé dans la cellule de droite (`gridx`) ou dans la cellule en-dessous (`gridy`) du composant précédent.

Les attributs `gridwidth` et `gridheight` permettent de spécifier respectivement le nombre de cases en colonnes et en lignes du composant courant.

Les constantes suivantes peuvent être utilisés avec `gridwidth` et `gridheight`:

- GridBagConstraints.REMAINDER Le composant est le dernier de sa ligne ou de sa colonne.
- GridBagConstraints.RELATIVE Le composant est l'avant-dernier de sa ligne ou de sa colonne.

L'attribut anchor permet de définir le point d'ancrage d'un composant dans la ou les cellules qu'il occupe. L'attribut fill détermine comment utiliser l'espace disponible lorsque la taille du composant est inférieure à celle qui lui est offerte :

- GridBagConstraint.NONE: ne pas redimensionner le composant.
- GridBagConstraint.HORIZONTAL: remplir l'espace horizontal offert.
- GridBagConstraint.VERTICAL: remplir l'espace vertical offert.
- GridBagConstraint.BOTH: remplir l'espace offert, horizontalement et verticalement.

Enfin, weightx et weighty déterminent comment se répartit l'espace supplémentaire entre les composants. Tandis que, ipadx et ipady définissent les marges internes minimales du composant.

Exemple 4.5:

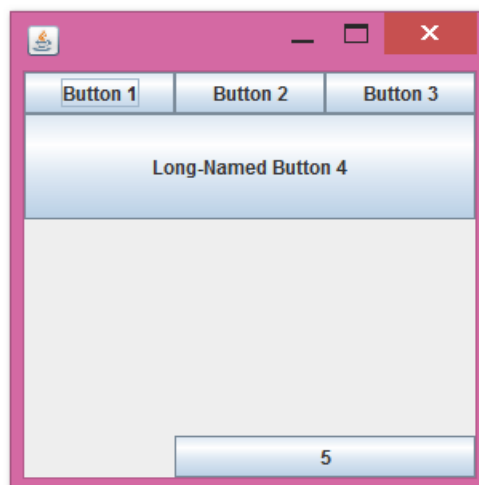
```

1 import java.awt.GridBagConstraints;
2 import java.awt.GridBagLayout;
3 import java.awt.Insets;
4
5 import javax.swing.JButton;
6 import javax.swing.JFrame;
7 import javax.swing.JPanel;
8
9 public class GridBagLayoutDemo extends JFrame {
10     final static boolean shouldFill = true;
11     final static boolean shouldWeightX = true;
12
13     public GridBagLayoutDemo() {
14         JButton button;
15         JPanel pane= new JPanel();
16         pane.setLayout(new GridBagLayout());
17         GridBagConstraints c = new GridBagConstraints();
18
19         c.fill = GridBagConstraints.HORIZONTAL;
20
21         button = new JButton("Button 1");
22         if (shouldWeightX) {
23             c.weightx = 0.5;
24         }
25         c.gridx = 0;
26         c.gridy = 0;
27         pane.add(button, c);
28
29         button = new JButton("Button 2");
30         c.gridx = 1;
31         c.gridy = 0;
32         pane.add(button, c);
33

```



```
34     button = new JButton("Button 3");
35     c.gridx = 2;
36     c.gridy = 0;
37     pane.add(button, c);
38
39     button = new JButton("Long-Named Button 4");
40     c.ipady = 40;           //make this component tall
41     c.weightx = 0.0;
42     c.gridwidth = 3;
43     c.gridx = 0;
44     c.gridy = 1;
45     pane.add(button, c);
46
47     button = new JButton("5");
48     c.ipady = 0;           //reset to default
49     c.weighty = 1.0;       //request any extra vertical space
50     c.anchor = GridBagConstraints.PAGE_END; //bottom of space
51     c.insets = new Insets(10,0,0,0); //top padding
52     c.gridx = 1;           //aligned with button 2
53     c.gridwidth = 2;       //2 columns wide
54     c.gridy = 2;           //third row
55     pane.add(button, c);
56     setSize(500,500);
57     setContentPane(pane);
58     setVisible(true);
59 }
60
61
62 public static void main(String[] args) {
63     GridBagLayoutDemo d = new GridBagLayoutDemo();
64 }
65 }
```



c. GroupLayout

Pour gérer des grilles de composants, sans qu'ils aient tous la même taille. Chaque composant doit être mis dans 2 groupes:

- Un pour gérer l'alignement horizontal.
- Un pour gérer l'alignement vertical.

Exemple 4.6:

```

1 import java.awt.*;
2 import javax.swing.*;
3 import static javax.swing.GroupLayout.Alignment.*;
4 public class GPL extends JFrame {
5     JLabel label = new JLabel("Find What:");
6     JTextField textField = new JTextField();
7     JCheckBox caseCheckBox = new JCheckBox("Match Case");
8     JCheckBox wrapCheckBox = new JCheckBox("Wrap Around");
9     JCheckBox wholeCheckBox = new JCheckBox("Whole Words");
10    JCheckBox backCheckBox = new JCheckBox("Search Backwards");
11    JButton findButton = new JButton("Find");
12    JButton cancelButton = new JButton("Cancel");
13    public GPL() {
14        GroupLayout layout = new GroupLayout(getContentPane());
15        getContentPane().setLayout(layout);
16        layout.setHorizontalGroup(
17            layout.createParallelGroup(GroupLayout.Alignment.LEADING)
18                .addComponent(label)
19                .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
20                    .addComponent(textField)
21                    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
22                        .addComponent(caseCheckBox)
23                        .addComponent(wholeCheckBox))
24                    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
25                        .addComponent(wrapCheckBox)
26                        .addComponent(backCheckBox)))
27                .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
28                    .addComponent(findButton)
29                    .addComponent(cancelButton))
30        );
31        layout.setVerticalGroup(
32            layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
33                .addComponent(label)
34                .addComponent(textField)
35                .addComponent(findButton)
36            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
37                .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
38                    .addComponent(caseCheckBox)
39                    .addComponent(wrapCheckBox))
40                .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
41                    .addComponent(wholeCheckBox)
42                    .addComponent(backCheckBox)))
43                .addComponent(cancelButton)
44    }

```

```
45     );  
46  
47     setTitle ("Trouver");  
48     pack();  
49     setDefaultCloseOperation (WindowConstants.EXIT_ON_CLOSE);  
50     setVisible (true);  
51 }  
52  
53 public static void main(String args[]) {  
54     GPL g =new GPL();  
55 }  
56 }
```

