

## Guide de l'utilisateur Swing<sup>1</sup>

*Ce document reprend en condensé les notions importantes à connaître pour développer des interfaces graphiques avec la librairie Swing. Pour des informations plus complètes, nous vous renvoyons sur le très bon tutoriel fourni par Sun (<http://java.sun.com/docs/books/tutorial/uiswing/>) duquel sont extraits la plupart des illustrations. Nous vous conseillons aussi de consulter la Javadoc.*

### Table des matières

Présentation des API graphiques en Java.....	2
La notion de Composant.....	2
La classe JComponent.....	3
Les composants principaux.....	3
La notion de conteneur.....	4
Exemple récapitulatif.....	5
La programmation événementielle.....	5
Les gestionnaires de mise en page.....	8
La boîte de dialogue.....	10
Boîtes de dialogue prêtes à l'emploi.....	10
Échange d'informations entre conteneurs.....	11
Les menus.....	12
Dessiner les composants.....	12
Gestion du focus.....	13

## PRÉSENTATION DES API GRAPHIQUES EN JAVA

**AWT (Abstract Window Toolkit)** est l'API des débuts créée par SUN. Elle offre tout ce qu'il faut pour développer des applications graphiques simples.

**Swing** est la deuxième réponse de SUN au problème. Swing possède des composants plus riches et offre des composants légers (au contraire de AWT où tous les composants sont lourds)

### La notion de composant

*De façon générale, un composant est un bout de code réutilisable. Pour cela, il doit respecter un certain nombre de conventions. Nous en reparlerons abondamment plus tard. Dans le cadre des interfaces graphiques, un composant est un élément graphique (qui va effectivement être réutilisé dans toutes les applications graphiques).*

### Composants légers vs composants lourds

*On parle de composant lourd lorsqu'on utilise l'API du système hôte pour les créer, ce que fait AWT. Ainsi, lorsqu'on crée un bouton AWT, cela entraîne un appel à la librairie graphique du système créant un bouton natif. Un composant léger, comme en Swing, est créé de toute pièce par l'API Java sans faire appel à son correspondant du système hôte. Ce fut un plus gros travail pour les concepteurs de l'API Swing mais cela permet d'avoir des applications qui ont exactement le même « look » quelle que soit la machine sur laquelle on la tourne.*

**SWT (Standard Widget Toolkit)** est une librairie *OpenSource* développée par IBM dans le cadre de son IDE (**Environnement de Développement Intégré**) Eclipse. Elle a la réputation d'être plus simple d'abord que Swing, complète mais moins souple. Elle plaît aussi plus au niveau de l'esthétique. Les composants sont lourds comme avec AWT (voir <http://www.eclipse.org/swt/> pour plus de détails).

Dans la suite, nous avons choisi de présenter Swing, car c'est une bibliothèque standard et très répandue.

## LA NOTION DE COMPOSANT

Tous les éléments graphiques (boutons, labels, champs de saisie, liste déroulante, ...) que nous allons manipuler sont proposés sous la forme de **JavaBean** (ou *composant*). Un **JavaBean**<sup>1</sup> est un code (une classe) respectant certaines contraintes qui vont lui permettre (entre autres) d'être reconnu et manipulé dans un IDE.

### Les propriétés

Un **JavaBean** est doté de *propriétés* (~ attributs). La manipulation de ces propriétés permet d'agir directement et facilement sur l'état ou le comportement d'un composant

- changer une couleur,
- spécifier le libellé à afficher,
- indiquer l'alignement du composant ou de son texte, etc...

Malgré l'illusion de manipuler une variable d'instance, les propriétés sont liées à des portions de code spécifiques : les *assesseurs*. (cf. cours Java). Dans le code Java, vous n'aurez donc pas directement accès aux attributs mais vous les manipulerez via des méthodes de lecture et d'écriture.

### Événements

Afin de permettre une réutilisabilité maximale des composants, leurs interactions principales sont définies à l'aide d'événements qui seront « écoutés » par des « observateurs ». Nous en reparlerons.

### Méthodes

Outre les *assesseurs* (pour les propriétés), un composant peut définir également d'autres méthodes.

<sup>1</sup> Ce document a été rédigé initialement par nos prédécesseurs, qu'ils en soient ici remerciés.

<sup>1</sup> Voir le *Guide du développeur JavaBean* pour plus d'informations.

## LA CLASSE JComponent

C'est la classe (abstraite) dont hérite chaque composant. Voyons quelques éléments qui y sont définis (cf. l'API pour les détails)

- La propriété **Font**
- Les propriétés **background** et **foreground** (examinez la classe **Color**)
- Un ordre de focalisation (pour voyager à travers les composants d'un conteneur)
- La propriété **enabled** : le composant est activé (apparaît en grisé sinon)
- La propriété **visible** : il est visible
- Si vous ne définissez pas certaines propriétés, elles seront souvent déduites des propriétés identiques du parent (ex : couleur de fond)

## LES COMPOSANTS PRINCIPAUX

Nous allons examiner les composants les plus souvent utilisés. Cela permettra déjà d'écrire quelques applications simples. Les autres composants seront vus plus loin.

### JLabel

Ce composant représente tout simplement une zone de texte non éditable.

- Propriété **text** : définit le texte du label
- Propriété **horizontalAlignment** : alignement du texte dans sa zone. (LEFT, CENTER, RIGHT, LEADING, TRAILING, cf. la classe **SwingConstants**)
- On s'occupe rarement d'événements déclenchés sur ces composants

### JTextField

Ce composant représente un champ de saisie simple : une seule ligne (vs. **JTextArea**) sans formatage (vs. **JFormattedTextField**) et en clair (vs. **JPasswordField**)

- Propriété **text** : définit le texte contenu dans le champ de saisie (permet ainsi de récupérer ce qui a été tapé).
- Propriété **column** : définit la longueur du champ de saisie
- L'événement **ActionEvent/actionPerformed** est lancé lorsque l'utilisateur sort du champ de saisie (ou tape sur ENTER)

### JButton

Le bouton bien connu.

- Propriété **text** : définit le texte du bouton
- L'événement **ActionEvent/actionPerformed** est déclenché lorsque l'utilisateur appuie sur le bouton (pas forcément avec la souris !)

Les cases à cocher et les boutons radio sont des cas particuliers de boutons (voir ci-dessous).

### JCheckBox

Techniquement, une case à cocher est un cas particulier de bouton. Les éléments spécifiques sont

- La propriété **selected** : indique si la case est cochée ou pas
- L'événement **ItemEvent/itemStateChanged** est déclenché lorsque la case à cocher change d'état

### JRadioButton

Un bouton radio est proche d'une case à cocher mais un seul bouton d'un groupe peut être sélectionné à la fois. Cela s'obtient via un objet de la classe **ButtonGroup** auquel on ajoute les boutons radio qui doivent faire partie du même groupe.

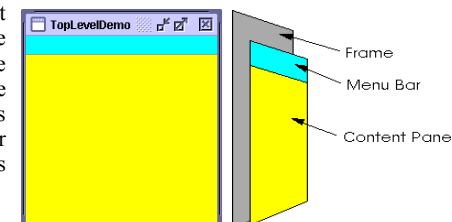
En général, on inscrit un même observateur de type **ActionListener** pour chaque bouton radio. On peut alors interroger l'objet de type **ButtonGroup** pour connaître le bouton qui est actuellement sélectionné.

## LA NOTION DE CONTENEUR

Les **conteneurs**, comme leur nom l'indique, sont des éléments contenant des composants. La grande puissance de cette architecture est de poser qu'un conteneur est aussi un composant. On obtient ainsi toute une arborescence de composants. Que trouve-t-on comme conteneur ?

### JFrame

Une **JFrame** est un conteneur évident et incontournable. C'est une fenêtre classique d'une application graphique (un rectangle avec une barre de titre et des boutons de manipulation de la fenêtre). Il est toutefois spécial en ce sens qu'il s'agit d'un conteneur de haut niveau qui ne peut pas être inclus dans un autre conteneur



### JPanel

Le **JPanel** se rencontre également souvent. Invisible, il contient d'autres composants. Il est utilisé pour la mise en page mais également lors de l'écriture de composants.

### Ajouter un composant à un conteneur

Pour ajouter un composant à un conteneur, on utilise la méthode `add()` de ce dernier.

Exemple :

```
conteneur.add(composant)
```

Avec une **JFrame** c'est un peu différent car elle est plus qu'un simple conteneur. Dans ce cas, il faut écrire

```
fenetre.getContentPane().add(composant)
```

Pour faciliter la vie du programmeur, la version 1.5 de Java permet d'ajouter un composant directement à la fenêtre qui va déléguer à son « content pane »

```
fenetre.add(composant)
```

### Afficher une fenêtre

Une fois tous les composants ajoutés, il faut pour l'afficher

1. Appeler sa méthode `pack()` qui lui demande de calculer sa taille et la taille de tous ses composants. Cela se fait via un algorithme complexe qui tient compte de la taille préférée des éléments mais aussi des contraintes de mise en page (cf. infra).  
Alternativement, on peut fixer sa taille mais c'est à éviter le plus possible car cela ne permet pas à la fenêtre de s'adapter à l'environnement précis (taille des polices par exemple)
2. La rendre visible via la commande `setVisible(true)`.

### Fermer une fenêtre

Que se passe-t-il lorsque l'utilisateur ferme une fenêtre ? c'est déterminé par la propriété `defaultCloseOperation` qui peut prendre les valeurs `DO_NOTHING_ON_CLOSE`, `HIDE_ON_CLOSE`, `DISPOSE_ON_CLOSE` et `EXIT_ON_CLOSE`. (cf. la classe **WindowConstants**).

## EXEMPLE RÉCAPITULATIF

```
import java.awt.*; // Pour la mise en page
import javax.swing.*; // Pour les composants

public class MaFenetre extends JFrame {

    private JButton    okB;
    private JTextField texteTF;

    public MaFenetre() {
        // Necessary pour la mise en page. On expliquera plus loin
        setLayout (new FlowLayout());
        texteTF = new JTextField(10);
        okB = new JButton("OK");
        add( new JLabel("Entrez un texte"));
        add( texteTF );
        add( okB );
        this.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        pack();
    }

    public static void main (String[] args) {
        MaFenetre fenetre = new MaFenetre();
        fenetre.setVisible(true);
    }
}
```

## LA PROGRAMMATION ÉVÉNEMENTIELLE

Dans les applications disposant d'une interface graphique (la quasi-totalité des applications donc), le programme ne suit pas un flot prédéfini mais est **dirigé par les événements** (l'appui sur un bouton, le choix d'un élément de menu, ...).

L'ordre des événements est impossible à déterminer puisqu'il dépend de l'utilisateur du programme. Cette forme de programmation (non déterminisme du flux, réaction aux événements) est à la base de ce qu'on appelle la programmation « événementielle ».

### Les événements

La notion d'événement est donc primordiale. Ils sont groupés en catégories.

- **Action** : une action a été déclenchée sur le composant.
- **Graphique** : le composant a été caché, montré, déplacé ou redimensionné.
- Liés aux **conteneurs** : un composant a été ajouté ou enlevé du conteneur.
- Liés à la **focalisation** : le composant a reçu ou perdu le focus.
- **Clavier** : frappe des touches au clavier.
- **Souris** : clic, entrée ou sortie du curseur.
- **Interaction à la souris** : opérations de déplacement de la souris et de glisser-déplacer.
- Élément ou **texte** : le contenu principal a changé, l'état d'un de ses éléments a été modifié.
- **Fenêtre** : Ensemble des événements liés aux fenêtres (fermeture, ouverture,...).

#### Événements de composants et événements sémantiques

Swing distingue 2 familles d'événements. Un événement de composant est de bas niveau et sa sémantique est indépendante du type de composant (« keyPressed », « mouseClicked », ...). Un événement sémantique est de plus haut niveau. Son sens précis dépend du composant. Ainsi, l'événement « Action » signifie « presser » pour un bouton et « choisir » pour une liste. On privilégiera au tant que possibles le deuxième type d'événements.

Une gestion d'événement implique 2 objets. Le premier, **l'observé ou source de l'événement**, sur lequel des événements peuvent se produire. Le deuxième, **l'observateur ou listener**, qui s'est déclaré intéressé par la gestion de cet événement sur ce premier objet. Lorsqu'un événement se produit sur un objet, tous les objets qui se sont inscrits comme observateur de cet objet pour cet événement sont prévenus et peuvent réagir.

Les événements similaires sont regroupés dans une même classe (cf. liste supra). Exemple : la classe « MouseEvent » regroupe les événements « mouseClicked », « mousePressed », « mouseEntered », « mouseExited », ...

Tous ces éléments interagissent selon un procédé défini par le Design-Pattern « Observateur-Observé ». Récapitulons. Un ou plusieurs objets (les **observateurs**) vont s'enregistrer auprès d'un autre objet (**l'observé**) comme observateur d'une classe particulière **d'événement**. Lorsqu'un événement de cette classe survient, l'observé prévient ses observateurs au travers de l'appel d'une **méthode précise**.

### Les observés (sources)

Pour être un observé valable, une classe doit fournir les méthodes pour s'inscrire/désinscrire. Pour la classe XXXEvent, elles sont de la forme :

```
addXXXListener (XXXListener listener)
removeXXXListener (XXXListener listener)
```

Exemple pour la classe d'événement ActionEvent :

```
addActionListener (ActionListener listener)
removeActionListener (ActionListener listener)
```

Bien souvent, on étendra un composant existant (comme un JPanel) et ces méthodes existeront déjà tout naturellement.

### Les observateurs (listeners)

Pour définir un observateur, il y a essentiellement 2 étapes

1. Être reconnu comme un observateur valable
2. Définir des méthodes qui seront appelées lorsque les événements surgiront

#### Être reconnu comme observateur

Une classe (ou plutôt un objet de cette classe) qui veut devenir un observateur doit être reconnu comme étant capable de le faire. Pour cela, elle doit implémenter l'interface correspondante ou hériter de la classe adhoc.

```
class observateur implements mouseListener
class observateur extends mouseAdapter
```

Pourquoi deux techniques ? L'héritage offre l'avantage de fournir des implémentations par défaut (et sans effet) des événements. Cela facilite l'écriture lorsqu'on ne veut réagir qu'à un petit nombre d'événements parmi ceux proposés par la classe. Par contre, l'héritage étant simple en Java, cela empêche d'hériter d'une autre classe. Ce qui est parfois inacceptable.

Notons que l'adapteur (la version sous forme de classe abstraite) n'existe pas pour les interfaces ne disposant que d'une seule méthode comme l'ActionListener par exemple.

#### Fournir les méthodes adéquates

L'observateur doit alors fournir une méthode pour chaque événement géré. Cette méthode a un nom imposé. Exemple : pour l'événement « Action », la méthode est « actionPerformed »

```
public void actionPerformed (ActionEvent e) {...}
```

Cette méthode recevra un objet représentant l'événement. On peut l'interroger si on veut en savoir

un peu plus. Une même méthode, par exemple, pourra observer un événement sur *plusieurs* objets observés. Interroger l'événement (l'objet « e ») permet de savoir sur quel objet précis a eu lieu l'événement.

### Lier un observateur et un observé

Une fois qu'on a un « observé » et un « observateur » il faut encore les mettre en contact, créer la dynamique.

- 1) Il faut que l' « observateur » **s'enregistre** au près de l'objet qu'il « observe » via la méthode d'inscription de l'observé. Exemple : pour la classe « ActionEvent », on écrit

```
observé.addActionListener(observateur)
```

- 2) Lorsqu'un événement surgit sur un composant, **celui-ci prévient** tous les observateurs enregistrés (par un appel à la méthode ad-hoc, cf. 3.2)

### Le mécanisme observateur-observé en pratique

En pratique où se trouvera l'observateur ? Le plus souvent, c'est le conteneur direct d'un composant qui sait comment réagir. L'observateur pourra dès lors être:

#### Le conteneur lui même

Le code ressemble à ceci

```
class Conteneur ... implements ActionListener {
    ...
    JComponent composant = new JButton();
    add(composant);
    composant.addActionListener(this);

    public void actionPerformed(ActionEvent evt) {
        ...
    }
}
```

Cette solution a quelques inconvénients

1. Il est souvent impossible d'étendre le Listener car on étend déjà une autre classe. Il faut se rabattre alors sur l'implémentation qui est moins pratique.
2. Si on écoute un même type d'événement sur plusieurs composants différents, il faut tout gérer dans une même méthode.

#### Une classe interne

L'idée est de passer par une classe interne dédiée. Cela donne

```
class Conteneur ... {
    class Observateur implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            ...
        }
    }
    ...
    JComponent composant = new JButton();
    add(composant);
    composant.addActionListener( new Observateur() );
}
```

### Une classe anonyme

La classe interne ne servant qu'une fois, on peut se contenter d'une classe anonyme. Ce qui s'écrit :

```
class Conteneur ... {
    ...
    JComponent composant = new JButton();
    add(composant);
    composant.addActionListener( new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            ...
        }
    });
}
```

## LES GESTIONNAIRES DE MISE EN PAGE

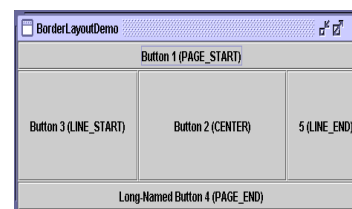
Les utilisateurs de JBuilder, lorsqu'ils utilisent l'outil de développement d'interface graphique, placent un composant sur le fenêtre et voient le composant rester à cet emplacement. C'est un peu différent avec Swing. Pourquoi ?

Un conteneur place ces composants en fonction de son « gestionnaire de mise en page ». Sous JBuilder, lorsqu'on crée une JFrame, un code est automatiquement inséré pour associer un gestionnaire « XYLayout » à ce Frame. Avec ce gestionnaire, les composants indiquent où ils veulent se placer lorsqu'il s'ajoutent à un conteneur. C'est probablement simple et efficace avec un outil de développement graphique mais ce n'est pas propre<sup>1</sup>.

Pour modifier la mise en page d'un conteneur, on appelle sa méthode `setLayout()` (cf. exemple supra)

#### Le BorderLayout

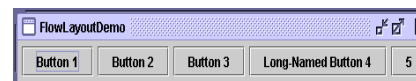
permet de placer jusqu'à 5 composants. Il définit 5 zones comme indiqué dans la figure. Si un composant ne spécifie rien, il est placé au centre. C'est le gestionnaire par défaut d'une JFrame.



```
setLayout(new BorderLayout());
add(button1, BorderLayout.PAGE_START);
add(button2, BorderLayout.CENTER);
add(button3, BorderLayout.LINE_START);
add(button4, BorderLayout.PAGE_END);
add(button5, BorderLayout.LINE_END);
```

#### Le FlowLayout

place ces composants les uns à la suite des autres, horizontalement, en passant à la ligne si nécessaire. C'est le gestionnaire par défaut d'un JPanel.

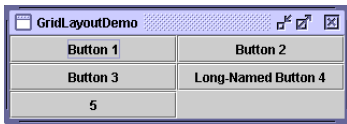


```
setLayout(new FlowLayout());
add(new JButton("Button 1"));
add(new JButton("Button 2"));
add(new JButton("Button 3"));
add(new JButton("Long-Named Button 4"));
add(new JButton("5"));
```

#### Le GridLayout

place ces composants dans une grille dont la taille peut être spécifiée. Toutes les cases auront la même taille.

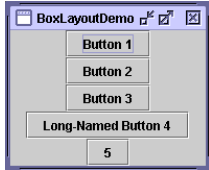
<sup>1</sup> Cela ne permet pas à l'application de s'adapter à des résolutions différentes d'écrans ni à des modifications de la taille des fenêtres.



```
setLayout(new GridLayout(0,2));
add(new JButton("Button 1"));
add(new JButton("Button 2"));
add(new JButton("Button 3"));
add(new JButton("Long-Named Button 4"));
add(new JButton("5"));
```

### Le BorderLayout

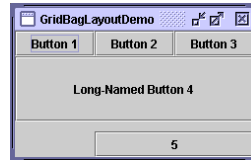
place ses composants les uns au dessus des autres ou les un à coté des autres. Il ressemble aux FlowLayout mais sans passage à la ligne et avec des tailles différentes pour les composants. L'orientation peut être définie à la construction via les constantes LINE\_AXIS et PAGE\_AXIS.



```
setLayout(new BorderLayout(pane, BorderLayout.Y_AXIS));
add(new JButton("Button 1"));
add(new JButton("Button 2"));
add(new JButton("Button 3"));
add(new JButton("Long-Named Button 4"));
add(new JButton("5"));
```

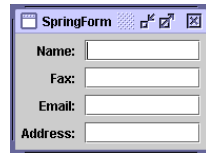
### Le GridBagLayout

est très riche mais il est difficile à appréhender. Il agit comme une grille mais avec des lignes et des colonnes de taille différentes. De plus, un composant peut occuper plusieurs cases ou une partie de case.



### Le SpringLayout

est apparu avec la version 1.4 de Java. Il est également très riche et complexe. En gros, les composants se placent les uns par rapport aux autres, certains ayant la liberté d'occuper tout l'espace libre restant. Il permet notamment d'obtenir ce genre de mise en page.



### Gérer la taille d'un composant

La gestion de la taille des composants est un processus complexe qui va dépendre de leur taille préférée (elle-même dépendante de leur contenu) mais aussi de la mise en page choisie. De manière générale, il vaut mieux ne pas fixer de dimensions et laisser le gestionnaire faire son travail au mieux. Si cela ne convient pas, on peut donner des recommandations. Il y a en fait 3 tailles différentes : la taille préférée, la taille minimale et la taille maximale. Les tailles minimales et maximales sont des recommandations que certaines mises en page peuvent être amenées à ignorer. La taille peut se définir depuis le conteneur du composant via une méthode set() (exemple : setPreferredSize(Dimension)) ou encore à l'intérieur même du composant en redéfinissant sa méthode get() (exemple : Dimension getPreferredSize())

## LA BOÎTE DE DIALOGUE

Vous connaissez déjà les conteneurs suivants :

- JFrame : une fenêtre « principale » classique
- JPanel : un conteneur sans apparence qui sera placé dans d'autres conteneurs

Un JDialog est très proche de la JFrame mais c'est une fenêtre **liée** qui peut fonctionner en mode **modal** ou non.

#### Fenêtre liée

Une fenêtre « liée » à une autre subit le même sort qu'elle. Elle est iconifiée quand l'autre l'est ; elle est tuée quand l'autre l'est, ..

#### Fenêtre modale et non modale

Lorsqu'une fenêtre A ouvre une fenêtre B en mode « modal », A n'est plus utilisable tant que B n'est pas fermé. C'est le phénomène qu'on observe avec les boîtes de dialogue. A l'inverse, si A ouvre B en mode « non modal », les deux fenêtres resteront actives et auront un parcours indépendant (une boîte d'outils par exemple)

### Constructeur

Vous pouvez spécifier :

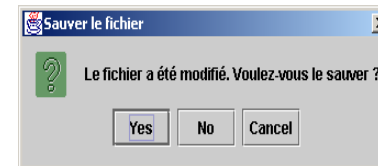
- Le nom de la JFrame à laquelle le dialogue est attaché
- Le titre de la boîte
- Un booléen indiquant si on est en « modal ».

## BOITES DE DIALOGUE PRÊTES A L'EMPLOI

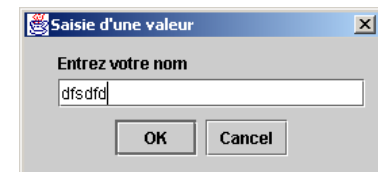
Pour notre facilité, Swing propose la classe JOptionPane qui offre des méthodes statiques qui permettent de construire facilement les boîtes de dialogue les plus courantes.



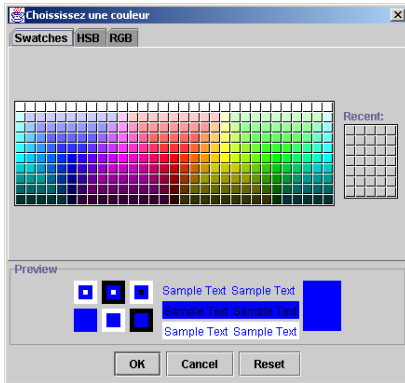
```
JOptionPane.showMessageDialog(frame,
    "Eggs aren't supposed to be green.");
// On peut aussi changer l'icône via un 3ème paramètre
```



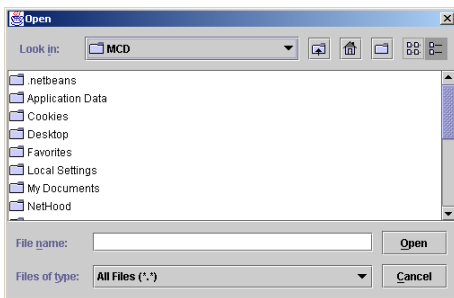
```
int n = JOptionPane.showConfirmDialog(
    this,
    "Le fichier a été modifié. "+
    "Voulez-vous le sauver ?",
    "Sauver le fichier",
    JOptionPane.YES_NO_CANCEL_OPTION);
```



```
String s = JOptionPane.showInputDialog(
    this,
    "Entrez votre nom",
    "Saisie d'une valeur",
    JOptionPane.PLAIN_MESSAGE);
```



```
Color couleurChoisie =
JColorChooser.showDialog(this,
"Choisissez une couleur",
Color.blue);
```



```
JFileChooser fc = new
JFileChooser();
int returnVal =
fc.showOpenDialog(this);
if (returnVal ==
JFileChooser.APPROVE_OPTION ) {
java.io.File file =
fc.getSelectedFile();
}
```

## ÉCHANGE D'INFORMATIONS ENTRE CONTENEURS

Cette partie est une mise en pratique du mécanisme « observateur-observé » au travers d'un problème particulier, l'échange d'information entre fenêtres. On identifie les différents cas de figures et les solutions possibles. Ces solutions seront en accord avec les bonnes pratiques de l'orienté objet. En particulier, on évitera de rendre les attributs publics.

L'information peut devoir transiter :

### De la fenêtre source (appelante) à la fenêtre créée (appelée).

**Exemple** : Une fenêtre contient un champ de saisie et un bouton. Appuyer sur le bouton ouvre une fenêtre modale reprenant le contenu du champ de saisie.

**Solution** : Une approche propre est de passer cette information au constructeur de la fenêtre créée.

### De la fenêtre créée (appelée) à la fenêtre source (appelante).

**Exemple** : Une boîte de dialogue a été ouverte. Elle contient un champ de saisie. Lorsqu'un ferme cette boîte, le texte saisi doit être affiché dans la fenêtre qui a initié le dialogue.

**Solution 1 (en mode modal)** : Il suffit d'attendre la fermeture de la boîte de dialogue et de l'interroger via des méthodes adhoc qui retournent l'information voulue.

**Solution 2 (non modal)** : De temps en temps, l'appelant utilise des méthodes de l'appelé pour obtenir des informations de l'appelé. La difficulté est de savoir « quand » utiliser ces méthodes. Il s'agit d'une mauvaise solution.

**Solution 3 (non modal)** : Via un événement, l'appelé signale à l'appelant qu'une information est disponible. L'appelant utilise alors la méthode ad-hoc de l'appelé. C'est la solution la plus souple et la plus propre.

**Solution 4 (non modal)** : L'appelé utilise une méthode de l'appelant pour lui communiquer de l'information. Il ne faut jamais oublier que l'objet appelé ne sait pas par qui et comment il sera utilisé ; c'est le principe même de la réutilisabilité dans l'orienté objet. Tout doit se faire via une interface précise que l'appelant doit implémenter. De plus, l'appelant se passera lui-même au constructeur de l'appelé. C'est plus direct à implémenter que la solution précédente mais moins souple. Pourquoi ?

## LES MENUS

Dans une JFrame, une zone spéciale est réservée pour un éventuel menu. Les classes impliquées sont :

- JMenuBar : Une barre de menu. La méthode `setJMenuBar(menuBar)` de JFrame permet d'indiquer la barre de menu que doit afficher une JFrame.
- JMenu : Un menu. Il s'ajoute à la barre de menu.
- JMenuItem : une entrée de menu. On peut lui adjoindre un ActionListener pour réagir à la sélection de cette entrée de menu.

```
...
JMenuBar menuBar = new JMenuBar();
JMenu menuFile = new JMenu("File");
JMenuItem menuFileExit = new JMenuItem("Exit");
menuFileExit.addActionListener( new ActionListener() {
public void actionPerformed(ActionEvent e) {
System.exit(0);
}});
menuFile.add(menuFileExit);
menuBar.add(menuFile);
setJMenuBar(menuBar);
...
```

## DESSINER LES COMPOSANTS

Pour personnaliser l'apparence graphique d'un de vos composants, vous pouvez redéfinir sa méthode `paintComponent()`.

**Exemple** : avec un composant héritant de JPanel et affichant un carré 10x10 de couleur bleue.

```
protected void paintComponent(Graphics g) {
g.setColor(Color.blue);
g.fillRect(0,0,10,10);
}
```

Le paramètre de type Graphics offre toute une série de méthodes pour le dessin (dessin de lignes, rectangles, carrés, ...). Notez que le point de coordonnées (0,0) est le point supérieur droit et que le point (0,10) est à droite du point (0,0).

Lorsqu'un composant doit être dessiné (quand son conteneur est rendu visible, quand son contenu change), sa méthode `repaint()` est appelée. Généralement, le système se rend correctement compte que le composant doit être redessiné mais ce n'est pas toujours le cas, surtout si vous avez redéfini la méthode `paintComponent()`. Dans ce cas, appelez la méthode `repaint()` quand vous le jugez utile (mais n'appellez jamais directement la méthode `paintComponent()` !).

## GESTION DU FOCUS

---

Le composant qui a le focus est celui qui recevra les événements liés au clavier. Par défaut, les composants reçoivent le focus dans l'ordre dans lequel il sont ajoutés à leur conteneur. Ce comportement par défaut peut être changé en associant une (extension de) `LayoutFocusTraversalPolicy` au conteneur.

Parmi les méthodes définies dans tous les composants on trouve :

- `requestFocusInWindow()` pour demander à avoir le focus
- `setFocusable(boolean)` pour indiquer si on peut avoir le focus ou pas

La classe d'événements associée est `FocusEvent` et les méthodes sont

- `focusGained(FocusEvent)`
- `focusLost(FocusEvent)`