Création interface graphique avec Swing : les bases

Par Baptiste Wicht

Date de publication : 12 juillet 2006

Dernière mise à jour : 3 décembre 2007

Au long de ce tutoriel, nous allons découvrir les bases de Swing. C'est-à-dire comment afficher une fenêtre, afficher du texte, demander du texte à l'utilisateur, ... Tout au long de l'article, nous allons appliquer les connaissances que nous verrons sur un petit projet.

1 - Introduction

Vous avez toujours voulu créer une application avec une interface graphique en java, mais vous n'avez pas réussi.

Ce tutoriel est fait pour vous. On va y découvrir les composants de base d'une interface graphique et comment les intégrer à votre interface et tout cela en utilisant Swing. A la fin de ce tutoriel, vous aurez créé une petite calculatrice, qui pourra vous servir de base à vos futurs programmes.

Ce tutoriel est divisé en différentes petites étapes et à la fin de chacune de celles-ci, vous pouvez télécharger le code source de l'étape pour toujours avoir une base de comparaison en cas d'erreur. Vous pouvez aussi télécharger l'intégralité des sources à la fin du tutoriel.

Cet article n'est pas destiné à couvrir la totalité de Swing. Il vous présentera les bases du framework, il sera complété à termes par d'autres articles plus poussés sur des sujets précis.

1.1 - Notre projet

Le projet que nous allons développer tout au long de ce tutoriel est très simple. C'est une petite calculette permettant de faire des additions, des soustractions, des multiplications et des divisions et tout cela avec 2 nombres. On utilisera une fenêtre pour afficher cette calculatrice.

Au niveau de l'interface que nous devrons développer, elle consistera donc en une fenêtre contenant 5 champs :

- Un champ pour saisir le premier nombre
- Un champ pour saisir le deuxième nombre
- Une liste pour choisir l'opérateur
- Un bouton pour effectuer le calcul
- Un champ pour afficher le résultat du calcul

Nous avancerons le projet au fur et à mesure que nous apprendrons à manipuler les composants d'une interface graphique.

2 - Swing

Tout au long de cet article, nous allons utiliser le framework graphique Swing. Mais qu'est-ce que ce Swing et pourquoi utiliser celui-là plutôt qu'un autre ?

Pour comprendre ce qu'est Swing, on va remonter un peu plus loin, au début de Java. Swing n'existait pas encore. On utilisait alors AWT pour créer des interfaces graphiques. AWT utilise directement les composants du système. Cela permet d'être rapide puisque c'est le système qui s'occupe de la création. Néanmoins, Java se voulant portable, on est obligé de limiter les composants AWT aux composants qu'on trouve sur tous les systèmes d'exploitation. On ne trouvait donc pas de tableau ni d'arbre.

C'est pour pallier à ce manque de composants que Swing a été introduit dans Java 1.2. Cette fois, les composants sont entièrement dessinés en Java, on peut donc créer autant de composants que l'on veut. C'est grâce à cela qu'on peut trouver maintenant un tableau ou un arbre dans les interfaces graphiques en Java. Néanmoins, Swing a été critiqué dans ses départs pour sa lenteur par rapport à AWT, en effet, cette fois, Java doit tout dessiner lui-même ce qui est logiquement plus lent que de demander au système d'afficher un composant. Mais depuis sa première version Swing a été très largement amélioré pour être maintenant une librairie très rapide et très complète. On peut noter que tous les composants Swing (reconnaissables à leur nom préfixés par J) héritent des composants AWT.

Dans les frameworks concurrents, on peut encore citer SWT. Il se situe entre AWT et Swing. C'est-à-dire qu'il utilise les composants systèmes, mais quand un composant n'existe pas sur le système, il l'emule en le dessinant directement. Le désavantage, c'est donc qu'on a une librairie SWT pour chaque système.

3 - Les fenêtres

Il existe plusieurs types de fenêtres dans Swing :

- **JWindow** : C'est la fenêtre la plus basique. C'est juste un conteneur que vous pouvez afficher sur votre écran. Il n'a pas de barre de titre, pas de boutons de fermeture/redimensionnement et n'est pas redimensionnable par défaut. Vous pouvez bien sûr lui ajouter toutes ces fonctionnalités. On utilise surtout les JWindow pour faire des SplashScreen, c'est-à-dire des interfaces d'attente qui se ferment automatiquement.
- **JDialog** : C'est une fenêtre destinée aux boîtes de dialogue. Ce type de fenêtre peut être modal, c'est-à-dire qu'elle bloque une autre fenêtre tant qu'elle est ouverte. Elles sont destinées à travailler de pair avec la fenêtre principale que nous allons voir maintenant.
- **JFrame** : C'est une fenêtre destinée à être la fenêtre principale de votre application. Elle n'est dépendante d'aucune autre fenêtre et ne peut pas être modale. Elle a une barre de titre et peut accueillir une barre de menu. Elle possède un bouton de fermeture, un bouton de redimensionnement et un bouton pour l'iconifier.

Nous allons maintenant voir comment créer ces fenêtres sans trop entrer dans les détails pour cet article.

3.1 - JWindow

Nous allons maintenant créer une simple JWindow qui ne fait rien pour le moment. Comme tel, ça n'est pas très utile, mais cela vous montrera à quoi ressemble une telle fenêtre.

Voici donc le code minimal nécessaire à la création d'une JWindow :

```
JWindow window = new JWindow();
```

On s'est donc juste contenté d'instancier une nouvelle JWindow. Maintenant pour l'afficher, on va utiliser la méthode setVisible qui vient de la classe Component et qui sera donc la même pour toutes les sortes de fenêtres.

```
SwingUtilities.invokeLater(new Runnable(){
 public void run(){
  //On crée une nouvelle instance de notre JWindow
  JWindow window = new JWindow();
  window.setSize(300, 200);//On lui donne une taille pour qu'on puisse la voir
  window.setVisible(true);//On la rend visible
 }
});
```

Comme vous pouvez le voir, c'est un peu plus compliqué que ce qu'on pense, mais rassurez-vous, vous allez vite comprendre à quoi sert tout ce code. En plus, certains me diront que ça marche sans tout ça. Oui, en effet, ça marche sans ça, mais dans ce tutoriel nous allons tout de suite apprendre la bonne façon de faire et c'est celle-ci

Alors, pourquoi tout ce code ? Pour une raison bien simple, Swing utilise son propre thread de traitement, qu'on appelle l'EDT (Event Dispatch Thread). C'est-à-dire que dans chaque application, on a 2 threads (au minimum), un thread pour le programme en lui-même et un thread pour Swing. La règle est simple, on ne fait rien de graphique dans le thread principal et on ne fait rien d'autres que du graphique dans le thread Swing. On crée donc une tâche (Runnable) contenant le code dédié à la création de la fenêtre et on l'ajoute dans le thread de Swing.

Un thread, c'est quoi ? *Vous pouvez vous représentez un thread comme un fil d'éxécution. C'est-à-dire une sorte de tube contenant des instructions à effectuer. Quand vous voulez faire quelque chose dans le thread de Swing, vous allez rajouter un nouvel élément dans le tube et il va être exécuté quand ce sera son tour.*

Pourquoi ne faut-il pas mélanger les deux ? Parce que si nous éxécutons une action longue dans l'EDT, nous bloquerons ce dernier et l'interface va être figée jusqu'à la fin de cette action. Tandis que si nous séparons les deux, on a notre action en tâche de fond et notre interface qui est toujours réactive.

Je ne vais pas m'étendre plus longtemps sur le sujet, si vous voulez plus d'informations, je vous invite à consulter cet excellent article : **Threads et performance avec swing**.

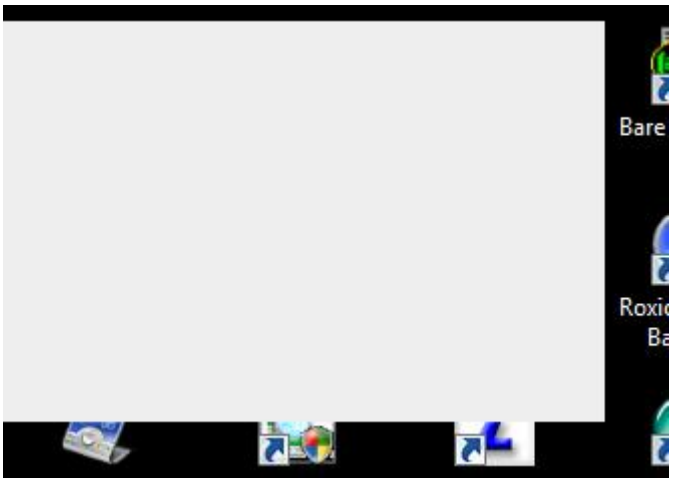
On va maintenant mettre notre code dans une méthode main :

```
public class TestJWindow {
 public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable(){
   public void run(){
    //On crée une nouvelle instance de notre JWindow
    JWindow window = new JWindow();
    window.setSize(300, 200);//On lui donne une taille pour qu'on puisse la voir
    window.setVisible(true);//On la rend visible
   }
  });
 }
}
```

Mais on ne va pas le laisser tel quel, car sinon, on ne pourrait pas fermer notre programme. On va donc rajouter une attente de 5 secondes ensuite fermer le programme. Pour ceux qui ne le savent pas, pour attendre un certain temps, on utiliser la méthode statique sleep de la classe Thread et pour quitter le programme, on utilise la méthode exit de la classe System :

```
public class TestJWindow {
 public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable(){
   public void run(){
    //On crée une nouvelle instance de notre JWindow
    JWindow window = new JWindow();
    window.setSize(300, 200);//On lui donne une taille pour qu'on puisse la voir
    window.setVisible(true);//On la rend visible
   }
  });

  try {
   Thread.sleep(5000);
  } catch (InterruptedException e) {}

  System.exit(0);
 }
}
```

Et voici le résultat :



JWindow

Comme vous le voyez, en l'état ce n'est pas très beau... C'est pour ça qu'il est assez rare de les utiliser, mis à part pour faire un écran de démarrage ou des fenêtres très personnalisés. Mais rassurez-vous, nous allons tout de suite passer à quelque chose de plus commun et de plus utile.

Le code complet : **TestJWindow**

3.2 - JDialog

Passons maintenant à une JDialog. Encore une fois, nous allons créer une fenêtre toute vide, nous apprendrons plus loin comment faire pour ajouter quelques choses dans cette fenêtre.

Voilà donc le code minimal pour créer une JDialog avec un titre :

```
JDialog dialog = new JDialog();
dialog.setTitle("Première fenêtre"); //On lui donne un titre
```

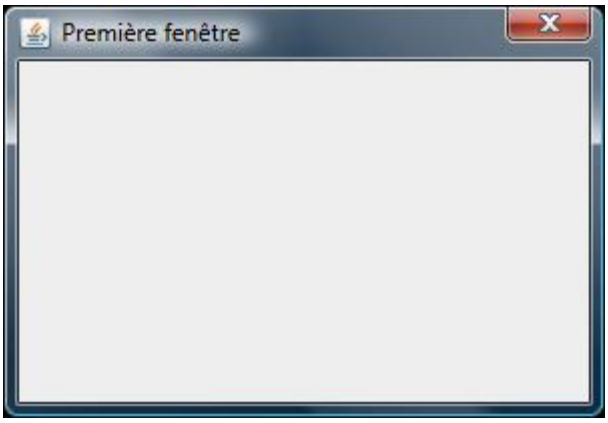
C'est une nouvelle fois tout simple, non ?

On remet notre code dans un main :

```
public class TestJDialog {
 public static void main(String[] args){
  SwingUtilities.invokeLater(new Runnable(){
   public void run(){
    //On crée une nouvelle instance de notre JDialog
    JDialog dialog = new JDialog();
    dialog.setSize(300, 200);//On lui donne une taille
    dialog.setTitle("Première fenêtre"); //On lui donne un titre
    dialog.setVisible(true);//On la rend visible
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //On dit à l'application de se fermer lors du
 clic sur la croix
   }
  });
 }
}
```

Vous pouvez voir que c'est très ressemblant au code de la JWindow. Le principe reste en effet le même pour toutes les fenêtres.

Voici donc ce que vous donne ce code :



JDialog

Le résultat est déjà plus proche d'une fenêtre qu'on rencontre habituellement.

En général, on n'utilise pas directement la classe JDialog comme ça. On passe souvent par la classe JOptionPane qui permet de créer des boites de dialogue très facilement. Nous verrons cela au chapitre 8.

Le code complet : **TestJDialog**

3.3 - JFrame

Passons maintenant à la fenêtre la plus complète et celle qui va nous intéresser le plus, la JFrame.

Cette fois, on ne va pas créer directement une instance de JFrame, mais on va créer une nouvelle classe héritant de JFrame. C'est-à-dire un nouveau type de JFrame si vous préférez. Il ne faut bien sûr pas oublier d'importer la classe JFrame.

Ainsi, vous pourriez faire quelque chose dans ce goût-là :

```
import javax.swing.JFrame;

public class SimpleFenetre extends JFrame{

 public SimpleFenetre(){
  super();
 }
}
```

Sauf qu'ainsi, on n'afficherait rien du tout, il va donc falloir ensuite configurer cette fenêtre pour qu'elle fasse ce que l'on veut. Pour cela, on va créer une méthode qui va initialiser notre fenêtre et appeler celle-ci depuis notre constructeur.

Voici donc ce à quoi va ressembler notre fenêtre :

```
import javax.swing.JFrame;

public class SimpleFenetre extends JFrame{

 public SimpleFenetre(){
  super();

  build();//On initialise notre fenêtre
 }

 private void build(){
  setTitle("Ma première fenêtre"); //On donne un titre à l'application
  setSize(320,240); //On donne une taille à notre fenêtre
  setLocationRelativeTo(null); //On centre la fenêtre sur l'écran
```

```
   setResizable(false); //On interdit la redimensionnement de la fenêtre
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //On dit à l'application de se fermer lors du
  clic sur la croix
  }
 }
```

Voici ce que font les différentes méthodes que l'on vient d'utiliser :

- **setTitle(String title)** : Cette méthode modifie le titre de la fenêtre. Ce titre va apparaître dans la barre de titre et également dans la barre des tâches.
- **setSize(int width, int height)** : Cette méthode permet de modifier la taille de la fenêtre. On verra plus tard qu'il y une autre façon de faire pour changer la taille de la fenêtre, mais cette deuxième façon n'est utile que pour les fenêtres non vides.
- **setLocationRelativeTo(Component c)** : Cette méthode permet de positionner la fenêtre par rapport à un composant. En indiquant un composant null, elle va se placer automatiquement au milieu de l'écran.
- **setResizable(boolean resizable)** : Cette méthode permet ou non le redimensionnement de la fenêtre.
- **setDefaultCloseOperation(int operation)** : Cette méthode permet de configurer l'action qui va être éxécutée lors de la fermeture de la fenêtre. On peut choisir de faire quitter le programme, de fermer la fenêtre ou de la rendre invisible (ce qui n'est pas pareil)

La méthode setLocationRelativeTo(null) pour centrer une fenêtre ne fonctionne pas avec plusieurs écrans. Elle va en effet centrer la fenêtre au milieu des deux écrans. Ce n'est pas un bug car elle positionne la fenêtre au milieu de l'espace affichable, mais ce n'est pas pratique. Si vous êtes dans un tel cas, je vous laisser consulter l'annexe à la fin de cet article.

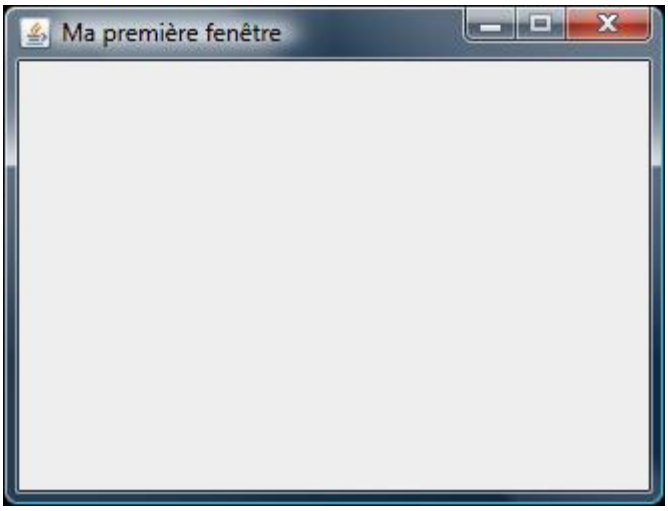
Avec ces méthodes, vous devriez pouvoir commencer à créer vos propres fenêtres. Il existe bien sûr encore bien d'autres choses que l'on peut faire avec les fenêtres, mais ce n'est pas l'objet de cet article.

On va créer une classe (TestJFrame) qui va nous permettre de lancer notre code maintenant :

```
public class TestJFrame {
 public static void main(String[] args){
  SwingUtilities.invokeLater(new Runnable(){
   public void run(){
    //On crée une nouvelle instance de notre JDialog
    SimpleFenetre fenetre = new SimpleFenetre();
    fenetre.setVisible(true);//On la rend visible
   }
  });
 }
}
```

A nouveau le code est semblable à celui des autres fenêtres, sauf que cette fois-ci on n'instancie pas directement un fenêtre Swing mais notre propre fenêtre.

Et voici ce que donne le code que l'on a développé :



JFrame

La seule différence visuelle avec la JDialog est le nombre de boutons. Sur la JDialog on n'a que le bouton de fermeture alors qu'avec la JFrame, on peut minimiser la fenêtre ou la réduire dans la barre des tâches.

Code complet :

- **TestJFrame**
- **SimpleFenetre**

3.4 - Application au projet

Maintenant que l'on sait créer une fenêtre, on va commencer notre projet de calculatrice. La fenêtre que l'on doit développer est la fenêtre principale de notre application. On va donc utiliser une JFrame. Pour le moment, on va fixer la taille de notre JFrame à 400 sur 200, on l'adaptera plus tard en fonction de nos composants. Cette fenêtre portera comme titre "Calculatrice", sera centrée à l'écran, ne sera pas redimensionnable et fermera l'application lors de la fermeture de la fenêtre.

Essayez de réfléchir à la solution avant de regarder. C'est vraiment très simple, mais c'est toujours mieux si c'est vous qui le faites.

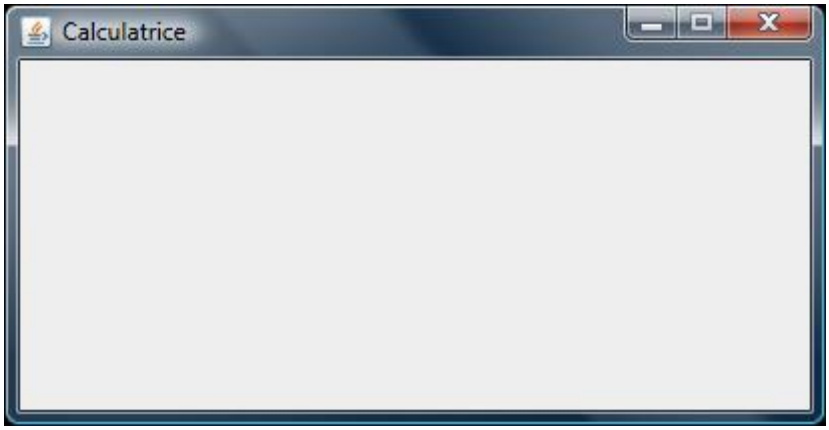
Voici ce que pourrait donner notre code :

```
import javax.swing.JFrame;

public class CalculatriceFenetre extends JFrame{

 public CalculatriceFenetre(){
  super();

  build();//On initialise notre fenêtre
 }

 private void build(){
  setTitle("Calculatrice"); //On donne un titre à l'application
  setSize(400,200); //On donne une taille à notre fenêtre
  setLocationRelativeTo(null); //On centre la fenêtre sur l'écran
  setResizable(false); //On interdit la redimensionnement de la fenêtre
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //On dit à l'application de se fermer lors du
 clic sur la croix
 }
}
```

Comme vous le voyez, je n'ai pas mis de méthode main dans cette classe. On va en effet mettre cette méthode dans une classe lanceur qui sera la classe principale. Pensez à toujours correctement séparer les responsabilités de vos objets dans des classes différentes. On va donc créer une classe Calculatrice qui va lancer notre fenêtre :

```
public class Calculatrice {
 public static void main(String[] args){
  SwingUtilities.invokeLater(new Runnable(){
   public void run(){
    CalculatriceFenetre fenetre = new CalculatriceFenetre();
    fenetre.setVisible(true);
   }
  });
 }
}
```

Et une fois cette classe lancée, vous devriez voir notre application se lancer :



Notre calculatrice

Elle est encore toute simple et ne contient rien, mais on a déjà la fenêtre qui s'affiche.

Code complet :

- **Calculatrice**
- **CalculatriceFenetre**

4 - Afficher du texte dans la fenêtre

Maintenant que l'on sait comment créer des fenêtres, il va bien falloir mettre quelque chose dedans, on va donc commencer par apprendre à afficher du texte dans une fenêtre.

Il y a deux sortes de composants textes dans une interface graphique :

- Les composants de saisie : Ce sont les composants qui permettent à l'utilisateur de saisir du texte. On va les décrire plus loin.
- Les composants d'affichage : Ce sont les composants qui permettent d'afficher du texte. Ce sont sur ceux-là qu'on va se concentrer. Je dis ceux-là, mais en fait, on ne va en utiliser qu'un.

Le composant pour afficher du texte en Java s'appelle un JLabel. C'est juste un composant sans réelle apparence qui affiche un texte.

Mais avant d'utiliser un JLabel, on va commencer par apprendre comment ajouter des composants dans une fenêtre. En fait, on ne va pas ajouter directement des composants dans la fenêtre. Je m'explique : Une Jframe est découpée en plusieurs parties : la fenêtre, le RootPane, le LayeredPane, la MenuBar, le ContentPane et le GlassPane. Pas besoin de s'affoller, on n'aura pas besoin de tout manipuler. Le RootPane est le container principal, c'est lui qui contient les autres, le LayeredPane forme juste un panneau composé du ContentPane et de la barre de menu (MenuBar) et le GlassPane est une couche par dessus le tout qui est utilisée pour intercepter les actions de l'utilisateur avant qu'elles ne parviennent aux composants. Et finalement, il nous reste le ContentPane qui est le composant qui contient la partie utile de la fenêtre, c'est-à-dire, celle dans laquelle on va afficher nos composants.

En Swing, on peut dire qu'il y deux types de composants, les containers et les composants. Les containers sont destinés à contenir d'autres containers et composants et les composants sont destinés à quelque chose de précis, par exemple afficher du texte ou permettre à l'utilisateur de saisir du texte. Le ContentPane est donc un container et le JLabel un composant.

Pour ajouter des composants dans la fenêtre, on va donc créer notre propre JPanel, ajouter notre JLabel à l'intérieur et ensuite indiquer à la fenêtre que notre JPanel sera le contentPane.

On va maintenant passer un chapitre suivant qui va traiter du positionnement des composants.

4.1 - Le positionnement des composants

Avant d'apprendre à ajouter des composants dans notre JPanel, on va voir comment Java permet de positionner les composants à l'intérieur d'un container.

Il y a deux manières de faire ceci :

- Le positionnement absolu : On va placer nos composant nous-même en utilisant un système de coordonnées en x et en y. Ce placement qui semble à première vue très simple, n'est en fait pas pratique du tout. Tout d'abord, si on veut ajouter des composants, il faut souvent changer une grande partie des coordonnées des autres composants et ensuite, il est quasi-impossible de faire du contenu redimensionnable avec cette méthode à moins de recalculer les coordonnées de tous les composants à chaque redimensionnement. Cette méthode n'est pas à utiliser.
- L'utilisation d'un gestionnaire de placement : On peut aussi utiliser les gestionnaires de placements (layout) qui vont s'occuper de placer correctement nos composants dans la fenêtre en fonction des paramètres qu'on leur a donné pour certains et bien évidemment en fonction des composants eux-mêmes. C'est cette solution que nous développer maintenant. Elle se révèle bien plus souple et bien plus pratique que le positionnement absolu.

Il existe de nombreux gestionnaires de layout dans Swing, en voici quelques uns :

- FlowLayout : C'est le layout par défaut des panels dans Swing. Il place les composants sur une ligne et recommence une nouvelle ligne à chaque fois qu'il atteint la fin de la ligne.
- BorderLayout : Ce layout place les composants dans 5 zones du container : La zone du haut, la zone du bas, la zone de gauche, celle de droite et la zone du centre. Il est très utile pour gérer le positionement du contentpane.
- CardLayout : Ce layout place les composants sur des couches disposées les unes sur les autres. On ne peut voir qu'une couche à la fois. On utilise surtout ce layout quand on a une série de composants qui s'affichent en fonction de quelques choses (liste déroulante, boutons, ...).
- GridLayout : Ce composant place les composants dans une grille. Il va redimensionner les composants pour les rendre tous de la même taille. C'est surtout utile quand on a plusieurs boutons ou champs texte en colonne et ligne qui doivent avoir la même taille, par exemple, les boutons d'une calculatrice.
- GridBagLayout : C'est le layout le plus avancé et le plus puissant. Néanmoins, c'est également le plus complexe à utiliser. Il permet de positionner les composants dans une grille et il permet d'indiquer beaucoup de données pour indiquer comment placer tel composant dans la grille. Ainsi on pourra dire à un composant de se redimensionner en hauteur mais pas en largeur et lui mettre des marges à gauche et à droite.

Il y a encore d'autres layouts dans Java et on peut également en trouver d'autres sur Internet.

Je ne vais pas détailler tous les gestionnaires de placements dans cet article. Ce n'est pas le but de cet article, je le ferai peut-être dans un prochain article. Pour le moment, on va rester avec le FlowLayout qui est très simple à utiliser.

4.2 - Afficher du texte avec un JLabel

Maintenant que l'on sait tout ce qu'il faut, on va afficher du texte dans notre fenêtre. Pour faire simple, on va afficher un message de bienvenue dans notre fenêtre.

Pour commencer, on va donc créer un JPanel et lui indiquer quel layout utiliser :

```
JPanel panel = new JPane();
panel.setLayout(new FlowLayout());
```

Rien de compliqué pour le moment, donc.

On va ensuite ajouter dans ce panel notre JLabel de bienvenue :

```
JLabel label = new JLabel("Bienvenue dans ma modeste application");

panel.add(label);
```

On a donc passé la chaîne à afficher au constructeur de JLabel. On aurait aussi pu utiliser la méthode setText pour indiquer au JLabel le texte à afficher.

*On peut afficher du texte multiligne dans un JLabel avec de l'HTML. Il suffit pour cela de mettre un texte comme celui-ci : "<html><body>Ligne 1
Ligne 2</body></html>"*

On a ensuite utilisé la méthode add pour ajouter le label dans le panel. Ensuite le gestionnaire de placement se chargera du reste. Comme vous pouvez le voir, c'est un jeu d'enfant.

On va maintenant reprendre le code la première fenêtre qu'on faîte, qu'on va renommer en FenetreTexte. On pourrait modifier le contentpane directement dans le constructeur, dans le build() ou dans une méthode à part. Personnellement, je fais une méthode à part qui créée et qui me renvoie le panel et je l'appelle depuis la méthode build. Voilà donc ce que ça nous donnerait :

```
import javax.swing.JFrame;

public class FenetreTexte extends JFrame{
```

```
 public FenetreTexte(){
  super();

  build();//On initialise notre fenêtre
 }

 private void build(){
  setTitle("Fenêtre qui affiche du texte"); //On donne un titre à l'application
  setSize(320,240); //On donne une taille à notre fenêtre
  setLocationRelativeTo(null); //On centre la fenêtre sur l'écran
  setResizable(true); //On permet le redimensionnement
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //On dit à l'application de se fermer lors du
clic sur la croix
  setContentPane(buildContentPane());
 }
}
```

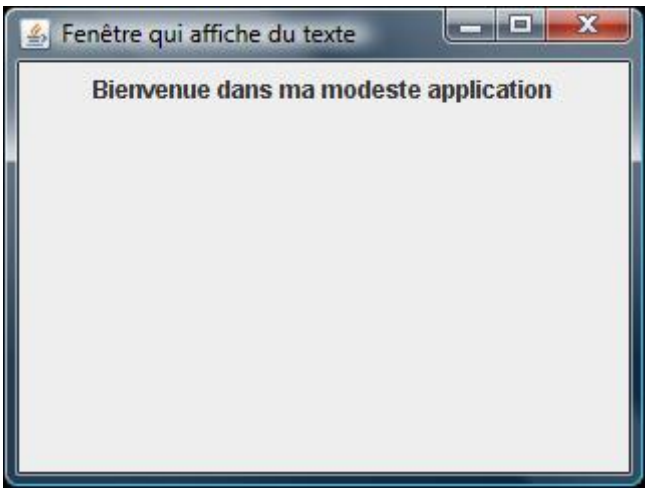
La seule chose qui ait vraiment changé est le fait qu'on appelle la méthode buildContentPane pour modifier le ContentPane. J'ai également modifié le titre et permis le redimensionnement pour que vous voyiez comment se comporte le FlowLayout.

On va maintenant coder notre méthode buildContentPane() avec ce qu'on a appris avant :

```
private JPanel buildContentPane(){
 JPanel panel = new JPanel();
 panel.setLayout(new FlowLayout());

 JLabel label = new JLabel("Bienvenue dans ma modeste application");

 panel.add(label);

 return panel;
}
```

Cette fois-ci, on va directement ajouter la méthode main dans la classe Fenetre pour avoir moins de classes :

Si on met cette méthode dans FenetreTexte et qu'on la lance, voici ce que ça devrait vous donner :



Affichage de texte

Le texte s'ajoute donc automatiquement sur la première ligne de notre fenêtre.

Code complet : **FenetreTexte**

4.3 - Application au projet

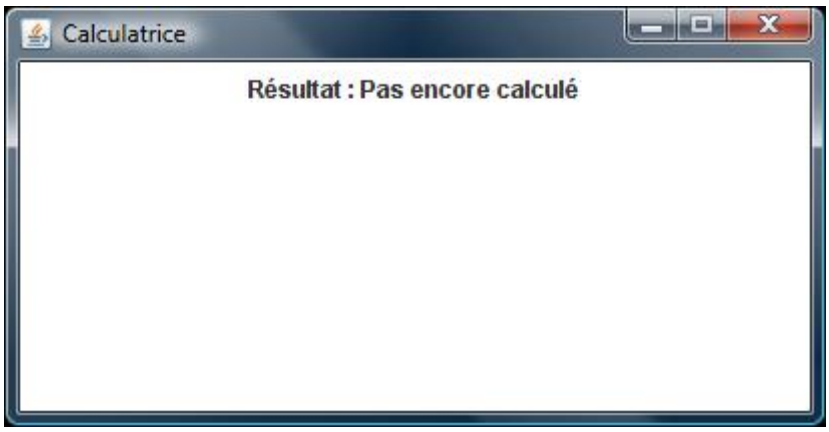
Comme vous l'avez vu, c'est vraiment très simple d'afficher du texte dans une JFrame. On va maintenant afficher du texte dans notre Calculatrice. Pour le moment, comme rien d'autre n'est fait, on va juste afficher "Résultat : Pas encore calculé". De plus, on va mettre un fond blanc à notre fenêtre plutôt que ce gris triste qui apparaît par défaut. Pour cela, on va utiliser la méthode setBackground(Color) sur notre JPanel.

Voici donc ce que donnerait notre classe modifiée :

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;

public class CalculatriceFenetre extends JFrame{

 public CalculatriceFenetre(){
  super();

  build();//On initialise notre fenêtre
 }

 private void build(){
  setTitle("Calculatrice"); //On donne un titre à l'application
  setSize(400,200); //On donne une taille à notre fenêtre
  setLocationRelativeTo(null); //On centre la fenêtre sur l'écran
  setResizable(false); //On interdit la redimensionnement de la fenêtre
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //On dit à l'application de se fermer lors du
 clic sur la croix
  setContentPane(buildContentPane());
 }

 private JPanel buildContentPane(){
  JPanel panel = new JPanel();
  panel.setLayout(new FlowLayout());
  panel.setBackground(Color.white);

  JLabel label = new JLabel("Résultat : Pas encore calculé");

  panel.add(label);

  return panel;
 }
}
```

On a également changé la couleur de fond de notre JPanel pour que le résultat soit plus joli.

La classe Calculatrice ne change pas. Voici donc notre calculatrice en l'état :



Notre calculatrice

Vous pouvez maintenant essayer de modifier les différents paramètres que nous avons définis pour ce label et essayer d'ajouter d'autres JLabel à votre fenêtre pour vous familiariser avec ce composant, qui reste très simple et basique.

- **TestJFrame**
- **SimpleFenetre**

5 - Ajouter des boutons dans la fenêtre

La principale fonctionnalité d'une application graphique est la programmation événementielle, c'est-à-dire le fait que l'utilisateur puisse déclencher des événements et réagir à ce qui se passe dans la fenêtre. Au contraire d'un programme en mode console, dans lequel le programme régit les actions de l'utilisateur à sa guise.

Pour que l'utilisateur puisse interagir avec le programme, on dispose par exemple, des boutons. Vous allez donc apprendre dans ce chapitre, à vous en servir.

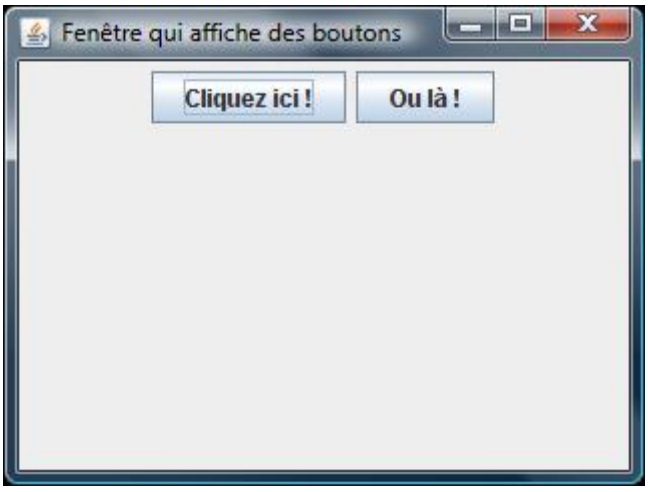
Mais qu'est-ce qu'un bouton ? C'est tout simplement un élément graphique sur lequel l'utilisateur peut cliquer pour déclencher une action. Le bouton ne fait rien tant que l'utilisateur n'a pas cliqué dessus.

Le composant pour créer un bouton est le JButton. On va créer une fenêtre qui comprendra deux boutons.

On va reprendre le code de la classe FenetreTexte (renommé en FenetreBoutons) et l'on va juste s'occuper de la méthode buildContentPane() pour ajouter 2 boutons à la fenêtre :

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;

public class FenetreAvecBouton extends JFrame{
 private JPanel buildContentPane(){
  JPanel panel = new JPanel();
  panel.setLayout(new FlowLayout());

  JButton bouton = new JButton("Cliquez ici !");
  panel.add(bouton);

  JButton bouton2 = new JButton("Ou là !");
  panel.add(bouton2);

  return panel;
 }
}
```

Vous devriez donc avoir un affichage comme celui-ci :



Utilisation de boutons

C'est bien beau, vous me direz, mais le bouton ne fait rien... C'est normal, on va devoir lui attribuer une action. Nous verrons cela au chapitre qui suit.

5.1 - Attribuer une action à un bouton

On va maintenant apprendre à attribuer une action à un bouton. C'est-à-dire indiquer ce qui va devoir se passer en cas de clic.

Le fonctionnement des actions en Swing suit toujours le même principe, celui des évênements et des écouteurs. Donc lors du clic sur un bouton, on aura donc un événement (ActionEvent dans ce cas) qui sera envoyé à tous les écouteurs du bouton. Donc si on veut savoir quand se passe l'action d'un bouton, il faut mettre un écouteur sur ce bouton et réagir en cas d'événement.

Il y a 2 façons de faire avec un bouton :

- Utiliser une classe qui implémente ActionListener et écouter le bouton : Cette méthode est toute simple. Il faut avoir une classe implémentant ActionListener et l'ajouter comme écouteur aux boutons. Ensuite, à chaque clic, la méthode actionPerformed va être appelée et il faudra tester le bouton qui a envoyé l'événement, ce qui n'est pas très pratique avec un grand nombre de boutons.
- Utiliser une AbstractAction : Cette classe représente une action de Swing. On peut lui donner un texte et une icône et on peut évidemment lui dire que faire en cas de clic dans la méthode actionPerformed. Personnellement, je préfère cette méthode qui permet de bien séparer l'action du composant graphique, mais c'est un choix personnel.

On va maintenant essayer les 2 techniques.

Code complet : **FenetreBoutons**

5.1.1 - ActionListener

On va reprendre la classe FenetreBoutons et la renommer FenetreBoutonsListener.

On va commencer par la méthode à base d'ActionListener. La première chose à faire est de dire que la FenetreBoutonsListener est un listener d'actions. Comment faire ça ? Tout simplement en implémentant l'interface ActionListener et en redéfinissant la méthode actionPerformed :

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.event.ActionListener;

public class FenetreBoutonsListener extends JFrame implements ActionListener{
 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPane();
  panel.setLayout(new FlowLayout());

  JButton bouton = new JButton("Cliquez ici !");
  panel.add(bouton);

  JButton bouton2 = new JButton("Ou là !");
  panel.add(bouton2);

  return panel;
 }

 public void actionPerformed(ActionEvent e) {

 }
}
```

Ensuite, on va dire aux boutons que la fenêtre va écouter leurs actions :

```
JButton bouton = new JButton("Cliquez ici !");
bouton.addActionListener(this);
panel.add(bouton);

JButton bouton2 = new JButton("Ou là !");
bouton2.addActionListener(this);
panel.add(bouton2);
```

On a donc ajouté un écouteur au bouton avec la méthode addActionListener. Maintenant à chaque fois qu'on va cliquer sur un bouton, là méthode actionPerformed va être appelée. Vous pouvez vérifier en ajoutant cela dans la méthode actionPerformed :

```
System.out.println("Clic");
```

Vous verrez donc apparaître "Clic" dans la console à chaque clic.

Mais, comme fait-on pour différencier les deux boutons ? En fait, ce n'est pas très compliqué, mais ça va nécessiter quelques changements. On va en effet devoir comparer la référence de la source de l'événement avec nos boutons pour savoir qui a lancé l'événement :

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.event.ActionListener;

public class FenetreAvecBouton extends JFrame implements ActionListener{
 private JButton bouton;
 private JButton bouton2;

 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPane();
  panel.setLayout(new FlowLayout());

  bouton = new JButton("Cliquez ici !");
  panel.add(bouton);

  bouton2 = new JButton("Ou là !");
  panel.add(bouton2);

  return panel;
 }

 public void actionPerformed(ActionEvent e) {
  Object source = e.getSource();

  if(source == bouton){
   System.out.println("Vous avez cliqué ici.");
  } else if(source == bouton2){
   System.out.println("Vous avez cliqué là.");
  }
 }
}
```

Vous aurez cette fois un message différent en fonction du bouton. Vous pouvez tester avec plus de boutons, ça va marcher aussi. Mais le code de la méthode actionPerformed deviendra vite lourd s'il y a trop de boutons différents. On va maintenant passer à la deuxième technique. Elle nécessite un peu plus de code, mais je la considère plus propre et plus claire.

Code complet : **FenetreBoutonsListener**

5.1.2 - AbstractAction

On va reprendre la classe FenetreBoutons et la renommer FenetreBoutonsActions.

Nous allons maintenant passer à la deuxième méthode : l'utilisation d'AbstractAction. Une AbstraAction est un ActionListener qui nous permet de gérer non seulement l'action générée par le composant mais aussi le texte, l'icône et l'état du composant. Par l'état, je veux dire activé/désactivé. En plus, on peut utiliser une action pour les éléments du menu comme pour les boutons ou alors les JToolbar. Donc si on a un bouton et un élément de menu qui font la même chose, il suffit d'utiliser la même action.

La classe AbstractAction est une implémentation abstraite d'Action qui nous évite de devoir implémenter toutes les méthodes d'Action. On va donc utiliser cette classe.

Maintenant, pour ajouter une Action à un programme, on peut soit créer une nouvelle classe dans un nouveau fichier soit créer une classe interne soit utiliser une classe anonyme. On va utiliser la première solution qui à mon avis, la plus appropriée.

Créons donc une classe IciAction pour notre bouton 1 :

```
public class IciAction extends AbstractAction {
 public IciAction(String texte){
  super(texte);
 }

 public void actionPerformed(ActionEvent e) {
  System.out.println("Vous avez cliqué ici");
 }
}
```

Tout ce qu'on a eu à faire, c'est redéfinir la méthode actionPerformed et créer un constructeur qui prend un texte en paramètre pour pouvoir lui passer le texte du bouton. Faisons de même pour le deuxième bouton :

```
public class LaAction extends AbstractAction {
 public LaAction(String texte){
  super(texte);
 }

 public void actionPerformed(ActionEvent e) {
  System.out.println("Vous avez cliqué là");
 }
}
```

On pourrait aussi directement indiquer le texte de l'action dans le constructeur de l'action sans le demander en paramètre, mais cela limiterait un peu la classe, le texte pouvant être différent en fonction d'ou on met l'action.

Et maintenant, on va attribuer ces actions à nos boutons. Pour cela rien de plus, il suffit de passer l'action au constructeur du JButton :

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.event.ActionListener;

public class FenetreBoutonsActions extends JFrame implements ActionListener{
 private JPanel buildContentPane(){
  JPanel panel = new JPane();
  panel.setLayout(new FlowLayout());

  JButton bouton = new JButton(new IciAction("Cliquez ici !"));
```

```
   panel.add(bouton);

   JButton bouton2 = new JButton(new LaAction("Ou là"));
   panel.add(bouton2);

   return panel;
  }
 }
```

Comme vous le voyez c'est très simple et personnellement, je trouve cela bien plus claire que la méthode avec l'ActionListener unique, mais c'est à vous de juger. Le désavantage étant qu'il faut créer autant de classe que de boutons, mais le nombre de classe n'étant pas limitées, est-ce bien important ?

Dans un tel cas, on aurait pu créer une seule action qui prend en paramètre ce qu'il faut afficher dans la console pour éviter d'être trop redondant au niveau du code.

Code complet :

- **IciAction**
- **LaAction**
- **FenetreBoutonsActions**

5.2 - Application au projet

Maintenant que nous avons appris à utiliser des boutons et à leur attribuer une action, nous allons appliquer cela à notre application en y ajoutant un bouton "Calculer".

On va donc créer une classe "CalculAction" qui sera vide pour le moment :

```
public class CalculAction extends AbstractAction {
 private CalculatriceFenetre fenetre;

 public CalculAction(CalculatriceFenetre fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  //Action lors du clic sur le bouton calculer
 }
}
```

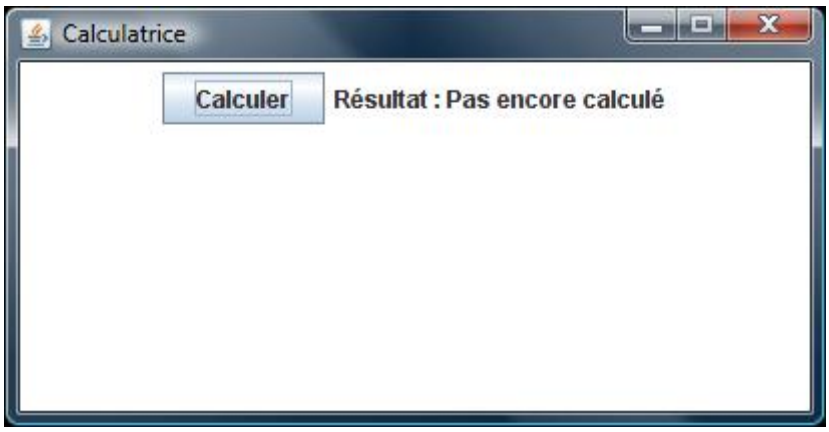
On a juste mis en paramètre de l'action la référence vers la fenêtre pour que l'action puisse aller chercher des infos dans la fenêtre.

Et on va maintenant ajouter un bouton avec cette action dans notre fenêtre :

```
//...
import javax.swing.JButton;

public class CalculatriceFenetre extends JFrame {

 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPane();
  panel.setLayout(new FlowLayout());
  panel.setBackground(Color.White);

  JButton bouton = new JButton(new CalculAction(this, "Calculer"));
```

```
    panel.add(bouton);

    JLabel label = new JLabel("Résultat : Pas encore calculé");

    panel.add(label);

    return panel;
  }
}
```

Ce qui devrait nous donner pour le moment une calculatrice de ce genre qui ne fait encore rien :



Notre calculatrice

Code complet :

- **CalculAction**
- **Calculatrice**
- **CalculatriceFenetre**

6 - Demander du texte à l'utilisateur

Maintenant que vous savez afficher du texte à l'écran et employer des boutons pour créer un événement, on va apprendre à demander à l'utilisateur du texte. Pour cela, il existe plusieurs composants :

- Le JTextField : C'est le plus simple des composants textes. Il permet d'entrer un texte sur une seule ligne.
- Le JTextArea : Il permet d'entrer un texte complet sur plusieurs lignes.
- Le JEditorPane : Très complet, vous pouvez modifier la police, la taille, la couleur de votre texte. Il permet même d'afficher des pages html. Vous pouvez y insérer du texte et des images.
- Le JTextPane : C'est le composant texte le plus évolué de Swing. Il permet la même chose que le JEditorPane mais il permet également d'insérer des composant personnalisés, c'est-àire que vous pouvez y insérer des composants Swing.

On ne va pas s'étendre sur les 3 derniers dans cet article, car cela dépasserait le cadre de celui-ci. J'essayerai de faire un article dédié pour les composants textes pour compléter celui-ci.

Nous allons maintenant apprendre à utiliser un JTextField.

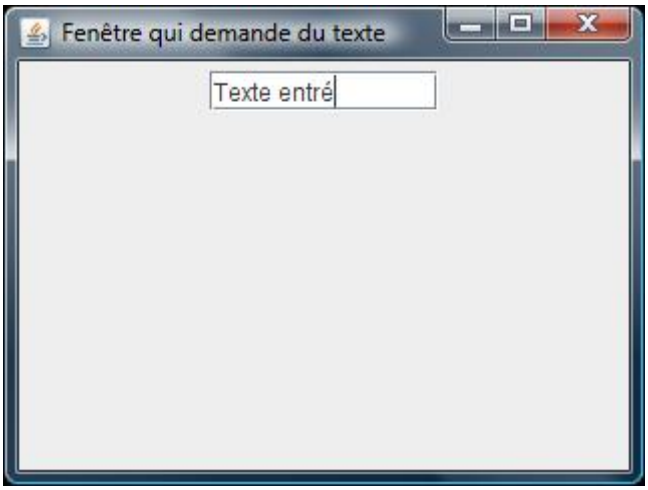
6.1 - Utilisation d'un JTextField

On va reprendre la base des autres fenêtres. Un JTextField s'utilise comme un autre composant, on utilise donc la même manière pour l'insérer dans une page. On va donc commencer par ajouter un JTextField dans une fenêtre :

```
//...
import javax.swing.JButton;

public class FenetreSaisie extends JFrame {

 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPanel();
  panel.setLayout(new FlowLayout());

  JTextField textField = new JTextField();
  textField.setColumns(10); //On lui donne un nombre de colonnes à afficher

  panel.add(textField);

  return panel;
 }
}
```

Si vous voulez mettre un texte par défaut dans le JTextField, il suffit de passer une chaîne de caractères au constructeur ou alors d'utiliser la méthode setText pour lui donner un texte.

Ce qui devrait vous donner :



Fenêtre de saisie

Vous pouvez donc écrire du texte dans le JTextField. On va maintenant ajouter un bouton à notre fenêtre. Ce bouton va récupérer le texte dans le JTextField et l'afficher dans un JLabel qu'on va aussi ajouter à l'interface.

Un problème va maintenant se poser. On va devoir faire une nouvelle classe Action pour notre bouton alors comment récupérer la valeur du JTextField et modifier la valeur du JLabel ? Dans une application de cette taille, la meilleure solution est d'ajouter des accesseurs sur notre classe d'interface et passer une référence de cette classe à notre action.

On va commencer par modifier notre interface pour ajouter un label et pour pouvoir accéder aux deux composants depuis une autre classe :

```
//...
import javax.swing.JButton;

public class FenetreSaisie extends JFrame {
 private JTextField textField;
 private JLabel label;

 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPanel();
  panel.setLayout(new FlowLayout());

  textField = new JTextField();

  panel.add(textField);

  label = new JLabel("Rien pour le moment");

  panel.add(label);

  return panel;
 }

 public JTextField getTextField(){
  return textField;
 }

 public JLabel getLabel(){
  return label;
 }
}
```

Il serait tentant de juste mettre le modificateur d'accès à public pour nos deux champs pour générer moins de code, mais ceci est à proscrire. Il faut limiter le plus possible les attributs publics d'une classe.

Ensuite, on va créer notre GetAction :

```
public class GetAction extends AbstractAction {
 private FenetreSaisie fenetre;

 public GetAction(FenetreSaisie fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  String texteUtilisateur = fenetre.getTextField().getText();
  fenetre.getLabel().setText(texteUtilisateur);
 }
}
```

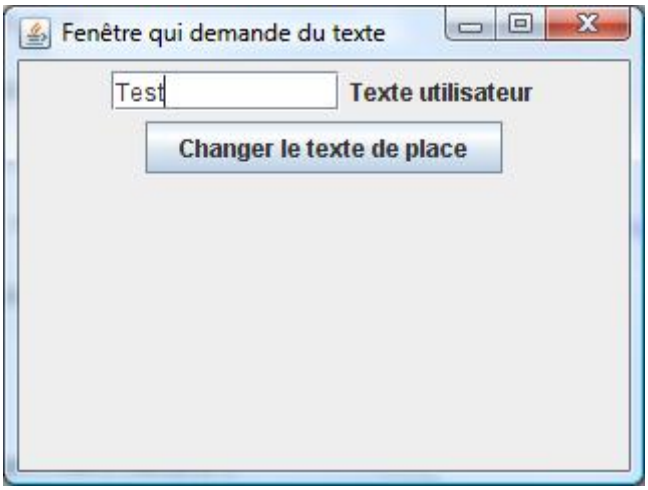
Rien de bien compliqué donc. On passe une référence de la fenêtre au constructeur de l'action et on utilise ensuite cette référence dans la méthode actionPerformed pour récupérer le texte dans le JTextField avec getText() et modifier le texte du JLabel avec setText().

On va maintenant ajouter notre bouton à notre interface :

```
//...

public class FenetreSaisie extends JFrame {
 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPane();
  panel.setLayout(new FlowLayout());

  textField = new JTextField();

  panel.add(textField);

  label = new JLabel("Rien pour le moment");

  panel.add(label);

  JButton bouton = new JButton(new GetAction(this, "Changer le texte de place"));

  panel.add(bouton);

  return panel;
 }

 //...
}
```

On a donc passé this (référence à la classe courante) à notre GetAction et on a appliqué cette action sur notre bouton.

Pour finir, voilà ce que donne notre petite application :



Récupération du texte saisi

La manipulation des textFields est donc très simple.

Code complet :

- **GetAction**
- **FenetreSaisie**

6.2 - Application au projet

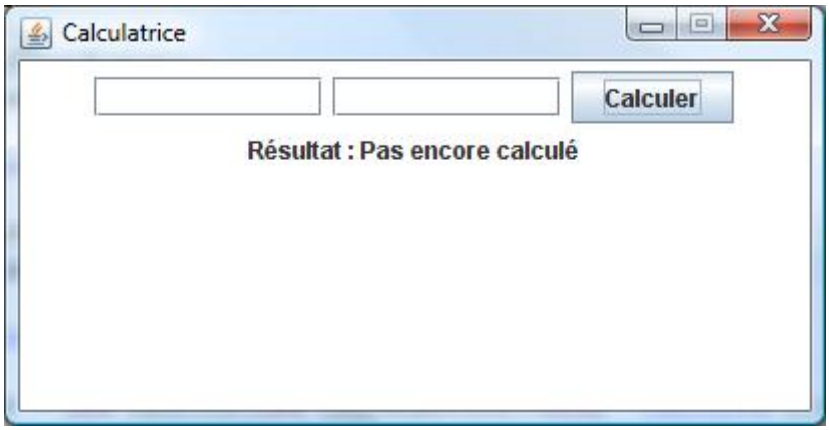
Maintenant que nous savons comment utiliser des JTextField, on va ajouter deux champs textes à notre calculatrice.

Voilà ce que ça va donner :

```
//...
import javax.swing.JTextField;

public class CalculatriceFenetre extends JFrame {
 private JTextField field1;
 private JTextField field2;

 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPanel();
  panel.setLayout(new FlowLayout());
  panel.setBackground(Color.white);

  field1 = new JTextField();
  field1.setColumns(10);

  panel.add(field1);

  field2 = new JTextField();
  field2.setColumns(10);

  panel.add(field2);

  JButton bouton = new JButton(new CalculAction(this, "Calculer"));

  panel.add(bouton);

  JLabel label = new JLabel("Résultat : Pas encore calculé");

  panel.add(label);
```

```
    return panel;
  }

  public JTextField getField1(){
    return field1;
  }

  public JTextField getField2(){
    return field2;
  }
}
```

Ce qui devrait nous donner à l'affichage ceci :



Notre calculatrice

Code complet :

- **CalculAction**
- **Calculatrice**
- **CalculatriceFenetre**

7 - Proposer une liste de choix à l'utilisateur

Après avoir vu les champs de texte, on va maintenant voir un nouveau champ de saisie, les listes déroulantes. En Swing, il existe 2 sortes de listes déroulantes :

- JList, liste déroulante assez avancée qui permet d'afficher plusieurs éléments à la fois.
- JComboBox, c'est une liste déroulante normale, de celle que vous voyez partout dans les applications.

On va se contenter d'utiliser la JComboBox dans cet article, la JList sera peut-être pour plus tard dans un autre article.

7.1 - Deux manières d'utiliser JComboBox

Il existe 2 manières de manipuler des JComboBox, soit on utilise directement les méthodes de manipulations des éléments de la JComboBox soit on développe notre propre modèle de liste.

On va maintenant détailler ces deux méthodes.

7.1.1 - Méthodes de JComboBox

La première manière de faire, c'est simplement d'utiliser les méthodes que nous propose JComboBox :

- addItem(Objet item), cette méthode va ajouter un nouvel objet à la liste
- removeItem(Objet item), cette méthode va enlever l'objet de la liste
- removeAllItems(), cette méthode va vider la liste déroulante
- getSelectedItem(), cette méthode retourne l'objet qui est actuellement sélectionné

Comme vous le voyez, c'est très simple.

On va maintenant utiliser ces méthodes. Pour cela, on va imaginer une fenêtre avec plusieurs boutons et 2 listes déroulantes. La première liste déroulante contiendra une série d'éléments qu'on pourra copier dans la deuxième liste avec un bouton et un deuxième bouton permettra d'enlever des éléments de la deuxième liste. On va également ajouter un troisième bouton qui permettra de vider la deuxième liste.

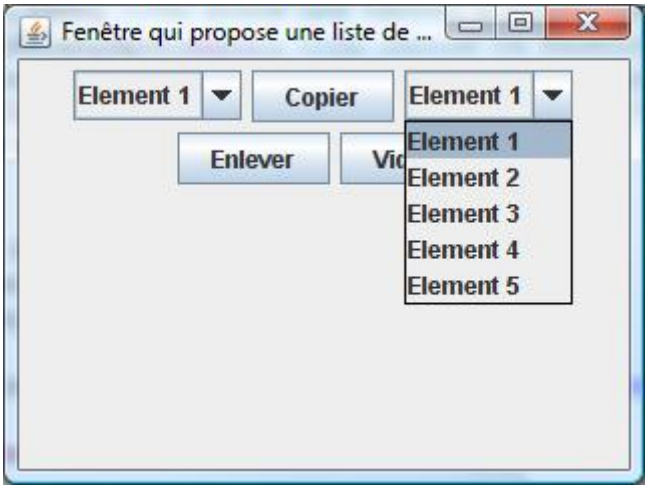
On va commencer par créer notre fenêtre :

```
import javax.swing.JComboBox;
//...

public class FenetreListes extends JFrame {
 private JComboBox liste1;
 private JComboBox liste2;

 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPanel();
  panel.setLayout(new FlowLayout());

  Object[] elements = new Object[]{"Element 1", "Element 2", "Element 3", "Element 4", "Element
 5"};

  liste1 = new JComboBox(elements);

  panel.add(liste1);

  JButton bouton = new JButton(new CopierAction(this, "Copier"));

  panel.add(bouton);
```

```
  liste2 = new JComboBox(elements);

  panel.add(liste2);

  JButton bouton2 = new JButton(new SupprimmerAction(this, "Enlever"));

  panel.add(bouton2);

  JButton bouton3 = new JButton(new ViderAction(this, "Vider"));

  panel.add(bouton3);

  return panel;
 }

 public JComboBox getListe1(){
  return liste1;
 }

 public JComboBox getListe2(){
  return liste2;
 }
}
```

Pour remplir la première liste, on a donné un tableau d'objets au constructeur du JComboBox qui va se charger de remplir la liste automatiquement avec les objets du tableau. Pour le reste rien de bien compliqué.

Ce qui devrait nous donner une interface de ce style :



Listes de choix

On va maintenant créer nos 3 actions :

```
public class CopierAction extends AbstractAction {
 private FenetreAvecJComboBox fenetre;

 public CopierAction(FenetreAvecJComboBox fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  Object selected = fenetre.getListe1().getSelectedItem();
  fenetre.getListe2().addItem(selected);
 }
}
```

```
public class SupprimmerAction extends AbstractAction {
 private FenetreAvecJComboBox fenetre;

 public SupprimmerAction(FenetreAvecJComboBox fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  Object selected = fenetre.getListe2().getSelectedItem();
  fenetre.getListe2().removeItem(selected);
 }
}
```

```
public class ViderAction extends AbstractAction {
 private FenetreAvecJComboBox fenetre;

 public ViderAction(FenetreAvecJComboBox fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  fenetre.getListe2().removeAllItems();
 }
}
```

La non plus, rien de bien compliqué. On s'est contenté d'utiliser les méthodes de JComboBox pour faire nos modifications sur les listes.

On va maintenant passer à la seconde façon de faire.

Code complet :

- **FenetreListes**
- **CopierAction**
- **ViderAction**
- **SuprimmerAction**

7.1.2 - Création d'un modèle de liste

Cette fois-ci, on ne va pas manipuler directement le JComboBox, mais son modèle. En fait, plusieurs composants Swing utilisent la notion de modèle pour stocker leurs données. La vue est donc complètement séparée des données qui sont stockés dans un modèle. En fait quand on utilise les méthodes de JComboBox, on va aussi modifier le modèle, mais par l'intermédiaire de JComboBox. Cette fois-ci, on va créer nos propres modèles.

On pourrait tout à fait se contenter d'utiliser les modèles par défaut, étant donné que ce qu'on va faire n'est pas très compliqué. Mais le but de ce tutoriel est de s'entraîner, il n'est donc pas superflu de découvrir comment coder ses propres modèles de listes.

Pour créer un modèle de liste, il suffit d'hériter de DefaultComboBoxModel. On peut aussi seulement implémenter ComboBoxModel, mais après, il faudra recoder une bonne partie qui est là-même à chaque fois.

Que doit faire le premier modèle ? Il ne doit rien faire de spécial. Il n'est pas modifiable et doit juste se contenter de contenir nos 5 éléments. Comme cette classe est destinée à contenir des chaînes de caractères, on va l'appeler StringModel.

Voilà à quoi va ressembler notre classe :

```
public class StringModel extends DefaultComboBoxModel {
 private ArrayList<String> strings;

 public StringModel(){
  super();

  strings = new ArrayList<String>();
 }

 public StringModel(String[] strings){
  super();

  this.strings = new ArrayList<String>();

  for(String string : strings){
   this.strings.add(string);
  }
 }

 protected ArrayList<String> getStrings(){
  return strings;
 }

 public String getSelectedString(){
  return (String)getSelectedItem();
 }

 @Override
 public Object getElementAt(int index) {
  return strings.get(index);
 }

 @Override
 public int getSize() {
  return strings.size();
 }

 @Override
 public int getIndexOf(Object element) {
  return strings.indexOf(element);
 }
}
```

Et voilà, rien de plus simple que ça ;) On s'est contenté de redéfinir les méthodes nécessaires pour leur dire d'aller chercher les éléments dans notre ArrayList.

On va maintenant passer au deuxième model. Celui-ci est modifiable, on va donc lui ajouter les méthodes addString, removeString et clearStrings. Comme il a les mêmes fonctionnalités de base, on va étendre StringModel et on va l'appeler ModifiableStringModel. Voilà ce que ça va donner :

```
public class ModifiableStringModel extends StringModel {
 public ModifiableStringModel(){
  super();
 }

 public ModifiableStringModel(String[] strings){
  super(strings);
 }

 public void clearStrings(){
  int oldSize = getStrings().size();
  getStrings().clear();
  fireIntervalRemoved(this, 0, oldSize);
 }

 public void addString(String string){
  getStrings().add(string);
  int size = getStrings().size();
  fireIntervalAdded(this, size, size);
```

```
  }

  public void removeString(String string){
   int position = getStrings().indexOf(string);
   getStrings().remove(position);
   fireIntervalRemoved(this, position, position);
  }
}
```

Cette fois-ci, les méthodes sont un peu compliquées. En fait, les méthodes fireXXX permettent d'avertir la liste déroulante qu'il y a eu du changement. Sinon, elle ne pourra pas savoir que les données ont changés. C'est le même principe que les écouteurs sur le bouton. Ici la liste écoute son modèle pour savoir quand elle doit se mettre à jour.

On va maintenant appliquer nos modèles. On va reprendre exactement le même code que pour la méthode précédente, mais cette fois, on va initialiser nos deux listes ainsi :

```
String[] strings = new String[]{"Element 1", "Element 2", "Element 3", "Element 4", "Element 5"};

liste1 = new JComboBox(new StringModel(strings));

//...

liste2 = new JComboBox(new ModifiableStringModel(strings));
```

On va également supprimmer les méthodes pour récupérer les JComboBox et les retourner par des méthodes qui nous retourne les modèles :

```
public StringModel getModeleListe1(){
 return (StringModel)liste1.getModel();
}

public ModifiableStringModel getModeleListe2(){
 return (ModifiableStringModel)liste2.getModel();
}
```

Et on peut maintenant passer à la modification de nos actions :

```
public class CopierAction extends AbstractAction {
 private FenetreAvecJComboBox fenetre;

 public CopierAction(FenetreAvecJComboBox fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  String selected = fenetre.getModeleListe1().getSelectedString();
  fenetre.getModeleListe2().addString(selected);
 }
}
```

```
public class SupprimmerAction extends AbstractAction {
 private FenetreAvecJComboBox fenetre;

 public SupprimmerAction(FenetreAvecJComboBox fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  String selected = fenetre.getModeleListe2().getSelectedString();
  fenetre.getModeleListe2().removeString(selected);
 }
}
```

```
public class ViderAction extends AbstractAction {
 private FenetreAvecJComboBox fenetre;

 public ViderAction(FenetreAvecJComboBox fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  fenetre.getModeleListe2().clearStrings();
 }
}
```

Rien de bien compliqué non plus de ce côté-là. On s'est juste contenté d'utiliser les méthodes qu'on a définies dans le modèle.

Comme vous le voyez, on a généré par mal de code pour pas grand chose. Les modèles sont plutôt à utiliser pour une liste qui est liée à une base de données ou à une autre classe, mais dans le cas qui nous préoccupe, ce n'est pas extrêmement utile. Néanmoins, vous pouvez aussi travailler toujours avec vos propres modèles si vous le désirez, ça offre quand même plus de souplesse.

Code complet :

- **FenetreListes**
- **CopierAction**
- **ViderAction**
- **SuprimmerAction**

7.2 - Application au projet

Maintenant que l'on sait utiliser les listes déroulantes, on va les utiliser dans notre calculatrice pour offrir à l'utilisateur le choix de l'opérateur. Ce sera donc un JComboBox contenant +-*/ entre les 2 JTextField.

Bien que ça ne soit pas indispensables, on va créer un modèle de listes. C'est donc un modèle qui contiendra nos opérateurs, on va donc l'appeler OperateursModel :

```
public class OperateursModel extends DefaultComboBoxModel {
 private ArrayList<String> operateurs;

 public OperateursModel(){
  super();

  operateurs = new ArrayList<String>();

  operateurs.add("+");
  operateurs.add("-");
  operateurs.add("*");
  operateurs.add("/");
 }

 public String getSelectedOperateur(){
  return (String)getSelectedItem();
 }

 @Override
 public Object getElementAt(int index) {
  return operateurs.get(index);
 }

 @Override
 public int getSize() {
  return operateurs.size();
```

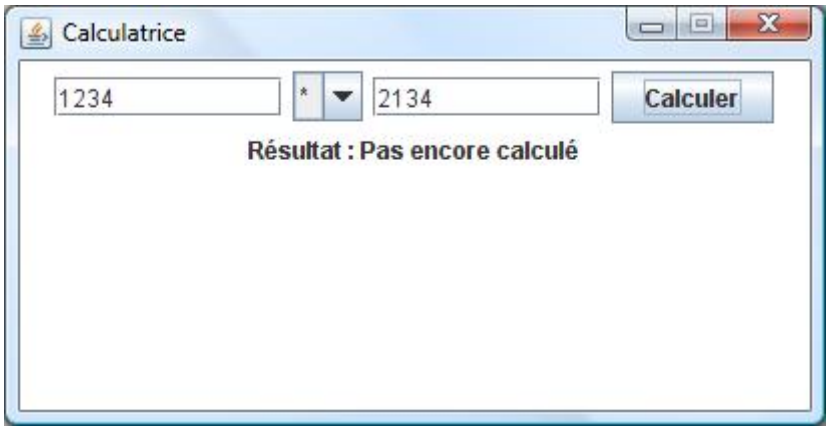
```
  }

  @Override
  public int getIndexOf(Object element) {
   return operateurs.indexOf(element);
  }
}
```

Rien de plus que ce qu'on a vu précédemment.

On va maintenant ajouter la liste déroulante à notre calculatrice :

```
//...
import javax.swing.JComboBox;

public class CalculatriceFenetre extends JFrame {
 private JComboBox liste;

 //...

 private JPanel buildContentPane(){
  JPanel panel = new JPane();
  panel.setLayout(new FlowLayout());
  panel.setBackground(Color.White);

  field1 = new JTextField();

  panel.add(field1);

  liste = new JComboBox(new OperateursModel());
  panel.add(liste);

  field2 = new JTextField();

  panel.add(field2);

  JButton bouton = new JButton(new CalculAction("Calculer"));

  panel.add(bouton);

  JLabel label = new JLabel("Résultat : Pas encore calculé");

  panel.add(label);

  return panel;
 }

 //...

 public OperateursModel getModelOperateurs(){
  return (OperateursModel)liste.getModel();
 }
}
```

Ce qui devrait vous donner finalement à l'affichage :



Notre calculatrice

Cette fois, tous les éléments graphiques de notre calculatrice ont été codés.

Et maintenant on va coder notre action pour que notre calculatrice puisse enfin fonctionner :

```
public class CalculAction extends AbstractAction {
 private CalculatriceFenetre fenetre;

 public CalculAction(CalculatriceFenetre fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  String nombre1String = fenetre.getField1().getText();//On récupère la valeur dans le premier
champ
  double nombre1 = Double.parseDouble(nombre1String);//On convertit cette valeur en un nombre

  String nombre2String = fenetre.getField2().getText();//On récupère la valeur dans le deuxième
champ
  double nombre2 = Double.parseDouble(nombre2String);//On convertit cette valeur en un nombre

  String operateur = fenetre.getModelOperateurs().getSelectedOperateur();

  double resultat = 0;

  if("+".equals(operateur)){
   resultat = nombre1 + nombre2;
  } else if ("-".equals(operateur)){
   resultat = nombre1 - nombre2;
  } else if ("*".equals(operateur)){
   resultat = nombre1 * nombre2;
  } else if ("/".equals(operateur)){
   resultat = nombre1 / nombre2;
  }

  fenetre.getLabel().setText("Résultat = " +  resultat);
 }
}
```

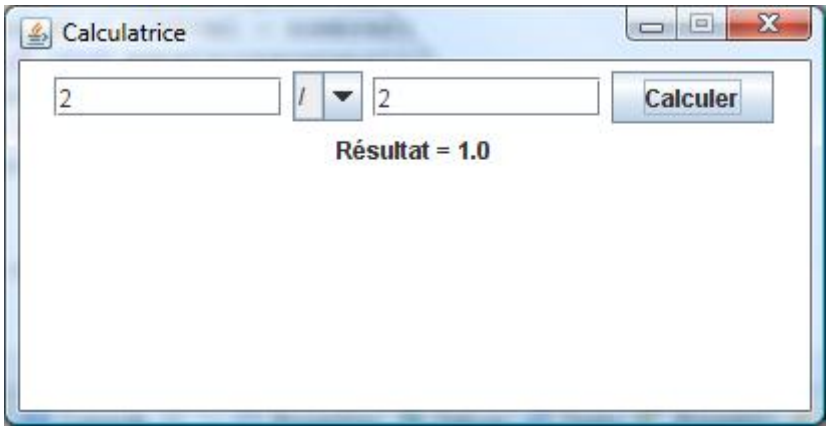
Rien de très compliqué. On commence par récupérer les 2 nombres saisis dans les champs textes, on les convertit en double, puis on récupére l'opérateur et en fonction de l'opérateur, on éxécute le calcul et on écrit le résultat dans le label de l'interface.

En théorie, toute la partie de calcul devrait être dans une classe à part et l'action ne devrait faire que récupérer les éléments de l'interface graphique, d'envoyer les deux nombres et l'opérateur à la classe de calcul et afficher le résultat que cette dernière lui aura retournée. Mais dans un

petit projet comme le nôtre, il n'est pas indispensable de le faire. Par contre, cela devient utile dès que l'on fait un projet de taille moyenne voire grande.

Et voilà ! Notre calculatrice est maintenant fonctionnelle.

Résultat :



Notre calculatrice

On pourrait bien sûr l'améliorer. On pourrait voir s'il y a une division par zéro, on pourrait vider les champs textes après la récupération des valeurs pour une autre utilisation. Mais comme maintenant vous vous y connaissez, essayez. La meilleure façon de progresser est de pratiquer.

Code complet :

- **CalculAction**
- **Calculatrice**
- **CalculatriceFenetre**
- **OperateursModel**

8 - Utiliser des boîtes de dialogue avec JOptionPane

Dans les composants Swing, il existe un composant très intéressant, le JOptionPane qui possède plusieurs méthodes statiques permettant d'ouvrir d'afficher diverses boîtes de dialogue :

- Les boîtes de messages, ces boîtes de dialogue permettent d'afficher un message pour l'utilisateur.
- Les boîtes de saisie, avec cette boite de dialogue, on va pouvoir demander une chaîne de caractères à l'utilisateur
- Les boites de confirmation, avec ces boîtes de dialogues, on peut demander une confirmation à l'utilisateur.
- Les boîtes de choix, qui permettent de donner le choix entre plusieurs choses, une liste déroulante en quelque sorte

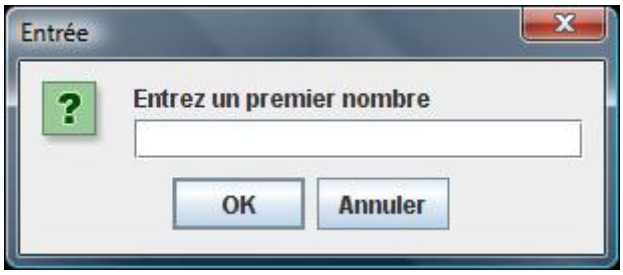
Pour utiliser ces différentes boîtes de dialogue, on va maintenant recréer une calculatrice mais sans interface graphique, uniquement avec des boîtes de dialogue.

Voilà donc ce qui va se passer dans notre programme :

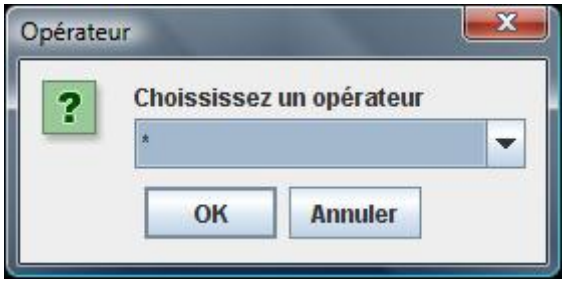
1 on va demander le premier nombre à l'utilisateur
2 On va demander un deuxième nombre à l'utilisateur
3 On va demander l'opérateur à l'utilisateur
4 On va afficher le résultat de l'addition de ces 2 nombres
5 On va demander à l'utilisateur s'il a apprécié ce programme
6 En fonction de sa réponse, on va afficher un message.

```
public class CalculatriceDialogue {
 public static void main(String[] args){
  double resultat = 0;

  double nombre1 = Double.parseDouble(JOptionPane.showInputDialog(null, "Entrez un premier
 nombre")) ;

  double nombre2 = Double.parseDouble(JOptionPane.showInputDialog(null, "Entrez un second
 nombre")) ;

  String[] operateurs = new String[]{"+","-","*","/"};

  String operateur = (String)JOptionPane.showInputDialog(null,"Choississez un opérateur",
   "Opérateur",JOptionPane.QUESTION_MESSAGE, null, operateurs, operateurs[0]);

  if("+".equals(operateur)){
   resultat = nombre1 + nombre2;
  }
  else if ("-".equals(operateur)){
   resultat = nombre1 - nombre2;
  }
  else if ("*".equals(operateur)){
   resultat = nombre1 * nombre2;
  }
  else if ("/".equals(operateur)){
   resultat = nombre1 / nombre2;
  }

  JOptionPane.showMessageDialog(null, nombre1 + " " + operateur + " " + nombre2 + " = " +
 resultat);

  int reponse = JOptionPane.showConfirmDialog(null, "Êtes-vous content de ce programme ?",
   "Satisfaction", JOptionPane.YES_NO_OPTION);

  if(reponse == JOptionPane.YES_OPTION){
   JOptionPane.showMessageDialog(null, "Cool :) A la prochaine") ;
  }else if(reponse == JOptionPane.NO_OPTION){
   JOptionPane.showMessageDialog(null, "Dommage :( Peut-être trouverez-vous votre bonheur" +
    " dans les futurs programmes que je vais développer") ;
  }
```

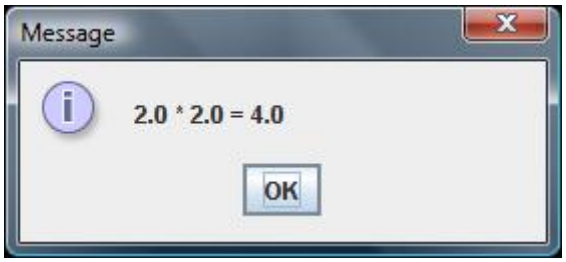
```
    }
}
```



Demande de texte à l'utilisateur



Proposition de choix à l'utilisateur



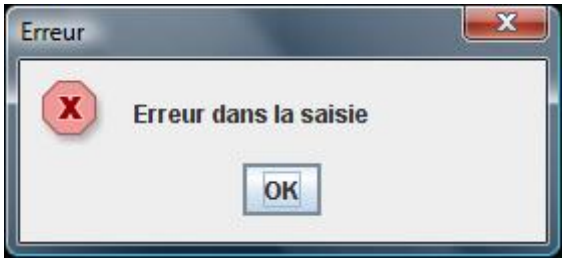
Envoyer un message à l'utilisateur

Dans cet exemple-là, on employait seulement des boites de dialogue, mais c'est très rare, les boîtes de dialogue sont plutôt faites pour seconder l'interface graphique. Vous pouvez employer ces boîtes de dialogue exactement de la même manière dans une interface graphique que dans cet exemple.

On peut aussi afficher des messages avec des icônes spéciales (warning, error) avec la syntaxe suivante :

```
showMessageDialog(parentComponent, message, title, messageType)
```

Par exemple, avec la propriété JOptionPane.ERROR_MESSAGE comme type de message, on obtient une boîte de dialogue de ce type :



Message d'erreur

Code complet : **CalculatriceDialogue**

9 - Ajouter une barre de menu à votre fenêtre

Maintenant que l'on a fait une calculatrice complète, on va y afficher un menu pour qu'elle fasse plus sérieuse. En Swing, pour mettre un menu sur votre JFrame, il faut employer une JMenuBar, qui sera composé de JMenu et de JMenuItem.

Un JMenu étant un bouton que l'on trouve directement dans la barre de tâches et il possède des JMenuITem qui sont eux des éléments du menu qui vont déclencher une action lors du clic.

Comme avec les boutons, il y a deux manières de gérer les actions des menus, les listeners ou les Action. Pour les listeners il suffit d'implémenter ActionListener sur la classe, d'utiliser la méthode addActionListener sur le JMenuItem pour ajouter le listener à l'élément de menu et de tester la source de l'événement pour savoir qui a cliqué. Je ne vais pas redétailler cette façon de faire ici. On va directement utiliser les actions.

Alors, on va faire deux menus :

- Un menu "Calculatrice" avec un élément "Calculer" et un élément "Quitter"
- Un menu "?" qui contiendra un bouton "A propos"

L'action de "Calculer" sera la même que celle du bouton Calculer, donc on va reprendre "CalculerAction". On va donc devoir créer "QuitterAction" et "AProposAction" :

```
public class AProposAction extends AbstractAction {
 private CalculatriceFenetre fenetre;

 public AProposAction(CalculatriceFenetre fenetre, String texte){
  super(texte);

  this.fenetre = fenetre;
 }

 public void actionPerformed(ActionEvent e) {
  JOptionPane.showMessageDialog(fenetre, "Ce programme a été développé par <Nom du
 développeur>");
 }
}
```

```
public class QuitterAction extends AbstractAction {
 public QuitterAction(String texte){
  super(texte);
 }

 public void actionPerformed(ActionEvent e) {
  System.exit(0);
 }
}
```

Rien qu'on ait pas vu dans ces deux actions.

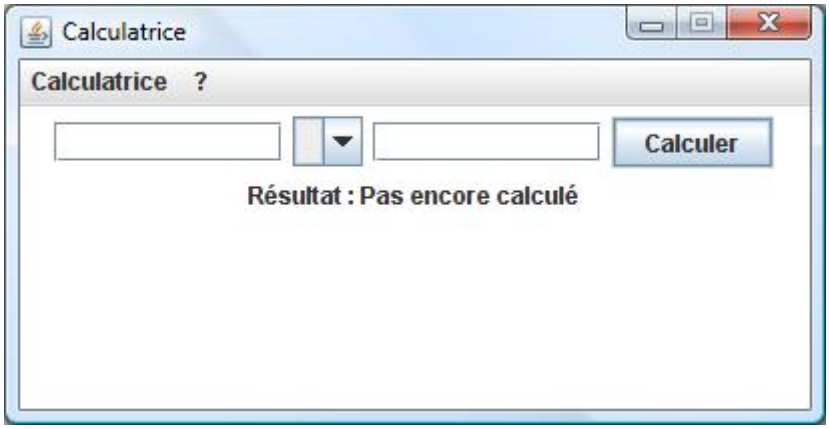
On va donc reprendre le code de la calculatrice et y ajouter les éléments du menu.

```
//...
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;

public class CalculatriceFenetre extends JFrame {
 //...

 private void build(){
  JMenuBar menuBar = new JMenuBar();
```

```
    JMenu menu1 = new JMenu("Calculatrice");

    JMenuItem calculer = new JMenuItem(new CalculAction(this, "Calculer"));
    menu1.add(calculer);

    JMenuItem quitter = new JMenuItem(new QuitterAction("Quitter"));
    menu1.add(quitter);

    menuBar.add(menu1);

    JMenu menu2 = new JMenu("?");

    JMenuItem aPropos = new JMenuItem(new AProposAction(this, "A propos"));
    menu2.add(aPropos);

    menuBar.add(menu2);

    setJMenuBar(menuBar);

    //...
  }

  //...
}
```

Comme vous le voyez, c'est tout simple. En plus, on n'a pas eu besoin de changer quelque chose pour le calcul, il suffit de prendre l'action déjà développée et tout fonctionne nickel.

Vous avez maintenant une jolie calculatrice avec un menu. Vous pourrez vous servir de ce menu en guise d'exemple pour toutes les autres applications que vous ferez ensuite.



Votre première application graphique complète, à quand les prochaines

Sachez que vous pouvez aussi créer des sous menus, pour cela, il vous suffit d'ajouter un JMenu à un autre JMenu et ensuite des JMenuItem dans le JMenu imbriqué. Vous pouvez continuer d'améliorer ce programme pour vous familiariser avec tous les composants de base de l'interfaçage graphique en java.

Code complet :

- **CalculAction**
- **Calculatrice**
- **CalculatriceFenetre**
- **OperateursModel**
- **QuitterAction**
- **AProposAction**

10 - Conclusion

Voilà, vous avez maintenant réalisé votre première application avec interface graphique en java. Nous arrivons donc à la fin de ce tutoriel, mais avant d'aller plus loin, je vous conseille d'essayer de faire des modifications et améliorations sur ce programme. Ainsi, vous apprendrez de vous-mêmes de nouvelles choses. Pour des infos sur des méthodes des classes que je vous ai présentées, le mieux est toujours d'aller sur **la javadoc** et si vous avez des problèmes sur certains points, il ne faut pas hésiter, si vos recherches ont été infructueuses, à poser la question sur les forums de developpez.com.

10.1 - Remerciements

J'aimerais remercier toute l'équipe java pour ses commentaires sur ce tutoriel, qui n'ont fait qu'en améliorer la qualité.

Je tiens aussi à remercier **Gorgonite** et **RideKick** qui ont respectivement corrigé la première et la seconde version de cet article.

10.2 - Les sources

Toutes les sources présentées dans ce tutoriel sont disponibles dans le fichier suivant : **Sources.zip**.

11 - Annexes

Cette section contient des codes annexes auxquels se réfère l'article.

11.1 - Centrer une JFrame avec plusieurs écrans

Cette classe permet de centrer une JFrame dans un environnement multi-écrans :

```
package org.jtheque.utils;

import java.awt.DisplayMode;
import java.awt.GraphicsDevice;
import java.awt.GraphicsEnvironment;
import java.awt.Insets;
import java.awt.Toolkit;

import javax.swing.JFrame;

/**
 * Provide utility methods for Swing Components.
 *
 * @author Baptiste Wicht
 */
public final class SwingUtils {
 private static DisplayMode mode;
 private static GraphicsDevice device;

 /**
  * Private constructor, this class isn't instanciable.
  *
  */
 private SwingUtils(){
  super();
 }

 /**
  * Load the display informations for this computer.
  *
  */
 private static void loadDisplayInfos(){
  GraphicsEnvironment gEnv = GraphicsEnvironment.getLocalGraphicsEnvironment();
  device = gEnv.getDefaultScreenDevice();
  mode = device.getDisplayMode();
 }

 /**
  * Return the insets of the screen.
  *
  * @return The insets
  */
 private static Insets getInsets(){
  return Toolkit.getDefaultToolkit().getScreenInsets(device.getDefaultConfiguration());
 }

 /**
  * Center a frame on the screen.
  *
  * @param frame The frame to be centered
  */
 public static void centerFrame(JFrame frame){
  if(mode == null){
   loadDisplayInfos();
  }

  frame.setLocation( (getWidth() - frame.getWidth()) / 2,
        (getHeight() - frame.getHeight()) / 2);
 }
```

```
  /**
   * Return the height of the screen.
   *
   * @return The height
   */
  public static int getHeight(){
   if(mode == null){
    loadDisplayInfos();
   }

   return mode.getHeight() - getInsets().bottom - getInsets().top;
  }

  /**
   * Return the widht of the screen.
   *
   * @return The width
   */
  public static int getWidth(){
   if(mode == null){
    loadDisplayInfos();
   }

   return mode.getWidth() - getInsets().left - getInsets().right;
  }
 }
```

Pour l'utiliser il suffit d'utiliser la méthode SwingUtils.centerFrame avec votre JFrame en paramètre après que vous lui ayez donnée une taille.