Partie 3 : Java et la programmation événementielle

Dans cette partie, nous aborderons la programmation événementielle comme le stipule le titre.
Par là, entendez ***programmation d'interface graphique***, ou IHM, ou encore GUI.

Nous utiliserons essentiellement les bibliothèques **Swing** et **AWT** présentes d'office dans Java.

Nous verrons ce qui forme, je pense, les fondements de base ! Nous n'entrerons pas dans les détails, enfin pas trop...
Je ne vais pas faire de long discours maintenant, je sais que vous êtes impatients... alors go !

Notre première fenêtre

Dans cette partie, nous aborderons les interfaces graphiques (on parle aussi d'`IHM` pour `I`nterfaces `H`omme `M`achine ou de `GUI` pour `G`raphical `U`ser `I`nterfaces) et, par extension, la programmation événementielle. Par là, vous devez comprendre que votre programme ne réagira plus à des saisies au clavier mais à des événements provenant d'un composant graphique : un bouton, une liste, un menu…

Le langage Java propose différentes bibliothèques pour programmer des IHM, mais dans cet ouvrage, nous utiliserons essentiellement les packages `javax.swing` et `java.awt` présents d'office dans Java. Ce chapitre vous permettra d'apprendre à utiliser l'objet `JFrame`, présent dans le package `java.swing`. Vous serez alors à même de créer une fenêtre, de définir sa taille, etc.

Le fonctionnement de base des IHM vous sera également présenté et vous apprendrez qu'en réalité, une fenêtre n'est qu'une multitude de composants posés les uns sur les autres et que chacun possède un rôle qui lui est propre.
Mais trêve de bavardages inutiles, commençons tout de suite !

L'objet JFrame

Avant de nous lancer à corps perdu dans cette partie, vous devez savoir de quoi nous allons nous servir. Dans ce cours, nous traiterons de `javax.swing` et de `java.awt`. Nous n'utiliserons pas de composants `awt`, nous travaillerons uniquement avec des composants `swing` ; en revanche, des objets issus du package `awt` seront utilisés afin d'interagir et de communiquer avec les composants `swing`. Par exemple, un composant peut être représenté par un bouton, une zone de texte, une case à cocher, etc.

Afin de mieux comprendre comment tout cela fonctionne, vous devez savoir que lorsque le langage Java a vu le jour, dans sa version 1.0, seul `awt` était utilisable ; `swing` n'existait pas, il est apparu dans la version 1.2 de Java (appelée aussi Java 2). Les composants `awt` sont considérés comme lourds (on dit aussi `HeavyWeight`) car ils sont fortement liés au système d'exploitation, c'est ce dernier qui les gère. Les composants `swing`, eux, sont comme dessinés dans un conteneur, ils sont dit légers (on dit aussi `LightWeight`) ; ils n'ont pas le même rendu à l'affichage, car ce n'est plus le système d'exploitation qui les gère. Il existe également d'autres différences, comme le nombre de composants utilisables, la gestion des bordures...

Pour toutes ces raisons, il est très fortement recommandé de ne pas mélanger les composants `swing` et `awt` dans une même fenêtre ; cela pourrait occasionner des conflits ! Si vous associez les deux, vous aurez de très grandes difficultés à développer une IHM stable et valide. En effet, `swing` et `awt` ont les mêmes fondements mais diffèrent dans leur utilisation.

Cette parenthèse fermée, nous pouvons entrer dans le vif du sujet. Je ne vous demande pas de créer un projet contenant une classe `main`, celui-ci doit être prêt depuis des lustres ! Pour utiliser une fenêtre de type `JFrame`, vous devez l'instancier, comme ceci :

Code : Java

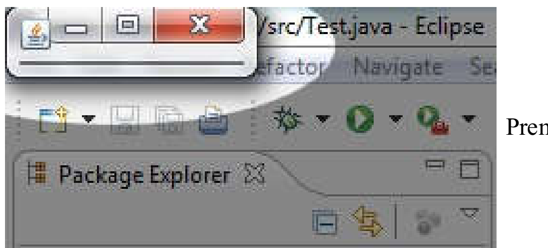
```java
import javax.swing.JFrame;

public class Test {
  public static void main(String[] args){
    JFrame fenetre = new JFrame();
  }
}
```

Lorsque vous exécutez ce code, vous n'obtenez rien, car par défaut, votre `JFrame` n'est pas visible. Vous devez donc lui dire « sois visible » de cette manière :

Code : Java

```
import javax.swing.JFrame;

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

Ainsi, lorsque vous exécutez ce code, vous obtenez la figure suivante.



Première fenêtre

À toutes celles et ceux qui se disent que cette fenêtre est toute petite, je réponds : « Bienvenue dans le monde de la programmation événementielle ! » Il faut que vous vous y fassiez, vos composants ne sont pas intelligents : il va falloir leur dire tout ce qu'ils doivent faire.

Pour obtenir une fenêtre plus conséquente, il faudrait donc :

- qu'elle soit plus grande ;
- qu'elle comporte un titre (ce ne serait pas du luxe !) ;
- qu'elle figure au centre de l'écran, ce serait parfait ;
- que notre programme s'arrête réellement lorsqu'on clique sur la croix rouge, car, pour ceux qui ne l'auraient pas remarqué, le processus Eclipse tourne encore même après la fermeture de la fenêtre.

Pour chacun des éléments que je viens d'énumérer, il y a aura une méthode à appeler afin que notre `JFrame` sache à quoi s'en tenir. Voici d'ailleurs un code répondant à toutes nos exigences :

Code : Java

```
import javax.swing.JFrame;

public class Test {
  public static void main(String[] args){

    JFrame fenetre = new JFrame();

    //Définit un titre pour notre fenêtre
    fenetre.setTitle("Ma première fenêtre Java");
    //Définit sa taille : 400 pixels de large et 100 pixels de haut
    fenetre.setSize(400, 100);
    //Nous demandons maintenant à notre objet de se positionner au
centre
    fenetre.setLocationRelativeTo(null);
    //Termine le processus lorsqu'on clique sur la croix rouge
    fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    //Et enfin, la rendre visible
    fenetre.setVisible(true);
  }
}
```

Voyez le rendu de ce code en figure suivante.

Une fenêtre plus adaptée

Afin de ne pas avoir à redéfinir les attributs à chaque fois, je pense qu'il serait utile que nous possédions notre propre objet. Comme ça, nous aurons notre propre classe !

Pour commencer, effaçons tout le code que nous avons écrit dans notre méthode `main`. Créons ensuite une classe que nous allons appeler `Fenetre` et faisons-la hériter de `JFrame`. Nous allons maintenant créer notre constructeur, dans lequel nous placerons nos instructions.
Cela nous donne :

Code : Java

```java
import javax.swing.JFrame;

public class Fenetre extends JFrame {
  public Fenetre(){
    this.setTitle("Ma première fenêtre Java");
    this.setSize(400, 500);
    this.setLocationRelativeTo(null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setVisible(true);
  }
}
```

Ensuite, vous avez le choix : soit vous conservez votre classe contenant la méthode `main` et vous créez une instance de `Fenetre`, soit vous effacez cette classe et vous placez votre méthode `main` dans votre classe `Fenetre`. Mais dans tous les cas, vous devez créer une instance de votre `Fenetre`. Personnellement, je préfère placer ma méthode `main` dans une classe à part… Mais je ne vous oblige pas à faire comme moi ! Quel que soit l'emplacement de votre `main`, la ligne de code suivante doit y figurer :

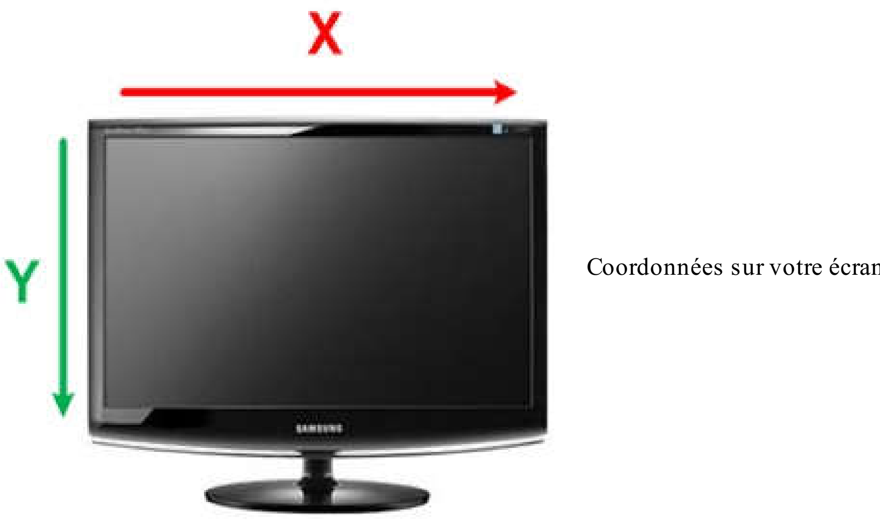
Code : Java

```java
Fenetre fen = new Fenetre();
```

Exécutez votre nouveau code, et… vous obtenez exactement la même chose que précédemment. Vous conviendrez que c'est tout de même plus pratique de ne plus écrire les mêmes instructions à chaque fois. Ainsi, vous possédez une classe qui va se charger de l'affichage de votre futur programme. Et voici une petite liste de méthodes que vous serez susceptibles d'utiliser.

Positionner la fenêtre à l'écran

Nous avons déjà centré notre fenêtre, mais vous voudriez peut-être la positionner ailleurs. Pour cela, vous pouvez utiliser la méthode `setLocation(int x, int y)`. Grâce à cette méthode, vous pouvez spécifier où doit se situer votre fenêtre sur l'écran. Les coordonnées, exprimées en pixels, sont basées sur un repère dont l'origine est représentée par le coin supérieur gauche (figure suivante).



Coordonnées sur votre écran

La première valeur de la méthode vous positionne sur l'axe *x*, 0 correspondant à l'origine ; les valeurs positives déplacent la fenêtre vers la droite tandis que les négatives la font sortir de l'écran par la gauche. La même règle s'applique aux valeurs de l'axe *y*, si ce n'est que les valeurs positives font descendre la fenêtre depuis l'origine tandis que les négatives la font sortir par le haut de l'écran.

Empêcher le redimensionnement de la fenêtre

Pour cela, il suffit d'invoquer la méthode `setResizable(boolean b)` : **`false`** empêche le redimensionnement tandis que **`true`** l'autorise.

Garder la fenêtre au premier plan

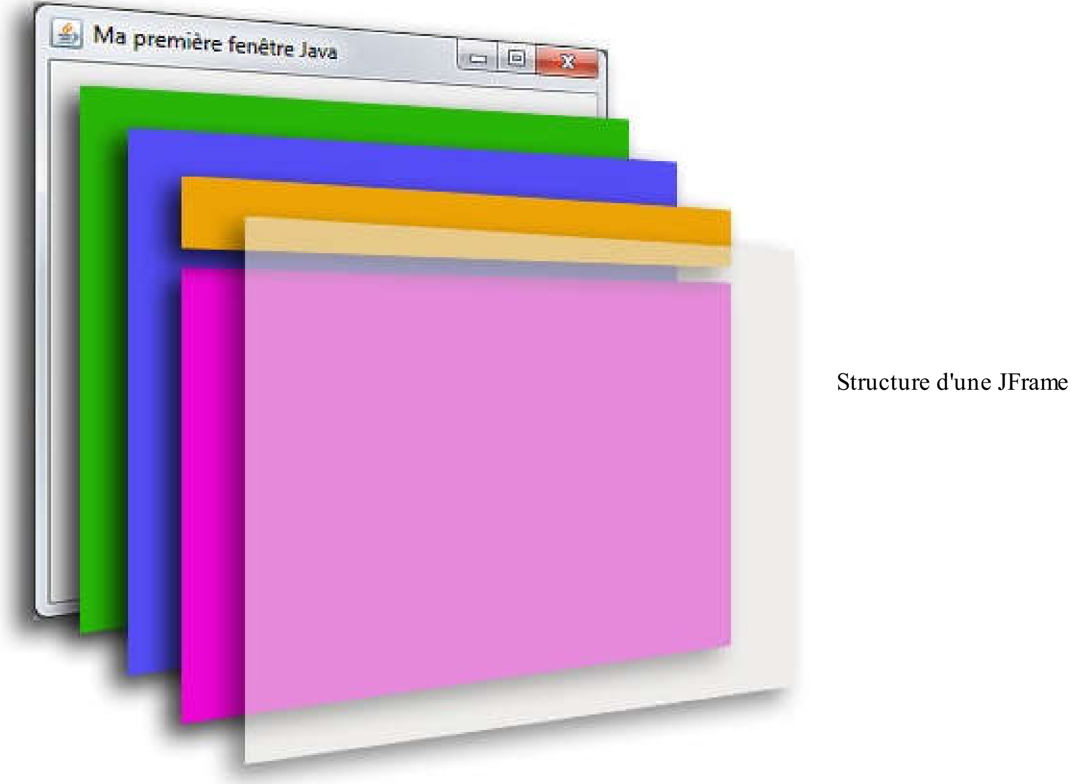
Il s'agit là encore d'une méthode qui prend un booléen en paramètre. Passer **`true`** laissera la fenêtre au premier plan quoi qu'il advienne, **`false`** annulera cela. Cette méthode est `setAlwaysOnTop(boolean b)`.

Retirer les contours et les boutons de contrôle

Pour ce faire, il faut utiliser la méthode `setUndecorated(boolean b)`.

Je ne vais pas faire le tour de toutes les méthodes maintenant, car de toute façon, nous allons nous servir de bon nombre d'entre elles très prochainement.Cependant, je suppose que vous aimeriez bien remplir un peu votre fenêtre. Je m'en doutais, mais avant il vous faut encore apprendre une bricole. En effet, votre fenêtre, telle qu'elle apparaît, vous cache quelques petites choses !

Vous pensez, et c'est légitime, que votre fenêtre est toute simple, dépourvue de tout composant (hormis les contours). Eh bien vous vous trompez ! Une `JFrame` est découpée en plusieurs parties superposées, comme le montre la figure suivante.



Structure d'une JFrame

Nous avons, dans l'ordre :

- la fenêtre ;
- le `RootPane` (en vert), le conteneur principal qui contient les autres composants ;
- le `LayeredPane` (en violet), qui forme juste un panneau composé du conteneur global et de la barre de menu (`MenuBar`) ;
- la `MenuBar` (en orange), la barre de menu, quand il y en a une ;
- le *content pane* (en rose) : c'est dans celui-ci que nous placerons nos composants ;
- le `GlassPane` (en transparence), couche utilisée pour intercepter les actions de l'utilisateur avant qu'elles ne parviennent aux composants.

Pas de panique, nous allons nous servir uniquement du content pane. Pour le récupérer, il nous suffit d'utiliser la méthode `getContentPane()` de la classe `JFrame`. Cependant, nous allons utiliser un composant autre que le content pane : un `JPanel` dans lequel nous insérerons nos composants.

Il existe d'autres types de fenêtre : la `JWindow`, une `JFrame` sans bordure et non *draggable* (déplaçable), et la `JDialog`, une fenêtre non redimensionnable. Nous n'en parlerons toutefois pas ici.

L'objet JPanel

Comme je vous l'ai dit, nous allons utiliser un `JPanel`, composant de type conteneur dont la vocation est d'accueillir d'autres objets de même type ou des objets de type composant (boutons, cases à cocher…).

Voici le marche à suivre :

1. Importer la classe `javax.swing.JPanel` dans notre classe héritée de `JFrame`.
2. Instancier un `JPanel` puis lui spécifier une couleur de fond pour mieux le distinguer.
3. Avertir notre `JFrame` que ce sera notre `JPanel` qui constituera son content pane.

Rien de bien sorcier, en somme. Qu'attendons-nous ?

Code : Java

```java
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Fenetre extends JFrame {
  public Fenetre(){
    this.setTitle("Ma première fenêtre Java");
    this.setSize(400, 100);
    this.setLocationRelativeTo(null);

    //Instanciation d'un objet JPanel
    JPanel pan = new JPanel();
    //Définition de sa couleur de fond
    pan.setBackground(Color.ORANGE);
    //On prévient notre JFrame que notre JPanel sera son content
pane
    this.setContentPane(pan);
    this.setVisible(true);
  }
}
```

Vous pouvez voir le résultat à la figure suivante.

Premier JPanel

C'est un bon début, mais je vois que vous êtes frustrés car il n'y a pas beaucoup de changement par rapport à la dernière fois. Eh bien, c'est maintenant que les choses deviennent intéressantes ! Avant de vous faire utiliser des composants (des boutons, par exemple), nous allons nous amuser avec notre `JPanel`. Plus particulièrement avec un objet dont le rôle est de dessiner et de peindre notre composant. Ça vous tente ? Alors, allons-y !

Les objets Graphics et Graphics2D

L'objet `Graphics`

Nous allons commencer par l'objet `Graphics`.Cet objet a une particularité de taille : vous ne pouvez l'utiliser que *si et seulement si* le système vous l'a donné via la méthode `getGraphics()` d'un composant `swing` ! Pour bien comprendre le fonctionnement de nos futurs conteneurs (ou composants), nous allons créer une classe héritée de `JPanel` : appelons-la `Panneau`. Nous allons faire un petit tour d'horizon du fonctionnement de cette classe, dont voici le code :

Code : Java

```java
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    //Vous verrez cette phrase chaque fois que la méthode sera
invoquée
    System.out.println("Je suis exécutée !");
    g.fillOval(20, 20, 75, 75);
  }
}
```

Qu'est-ce que c'est que cette méthode ?

Cette méthode est celle que l'objet appelle pour se dessiner sur votre fenêtre ; si vous réduisez cette dernière et que vous l'affichez de nouveau, c'est encore cette méthode qui est appelée pour afficher votre composant. Idem si vous redimensionnez votre fenêtre… De plus, nous n'avons même pas besoin de redéfinir un constructeur car cette méthode est appelée automatiquement !

C'est très pratique pour personnaliser des composants, car vous n'aurez *jamais* à l'appeler vous-mêmes : c'est automatique ! Tout ce que vous pouvez faire, c'est forcer l'objet à se repeindre ; ce n'est toutefois pas cette méthode que vous invoquerez, mais nous y reviendrons.

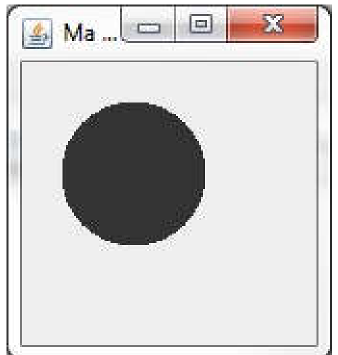
Vous aurez constaté que cette méthode possède un argument et qu'il s'agit du fameux objet `Graphics` tant convoité. Nous reviendrons sur l'instruction `g.fillOval(20, 20, 75, 75)`, mais vous verrez à quoi elle sert lorsque vous exécuterez votre programme.

Voici maintenant notre classe `Fenetre` :

Code : Java

```java
import javax.swing.JFrame;

public class Fenetre extends JFrame {
  public Fenetre(){
    this.setTitle("Ma première fenêtre Java");
    this.setSize(100, 150);
    this.setLocationRelativeTo(null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setContentPane(new Panneau());

    this.setVisible(true);
  }
}
```

Exécutez votre `main`, vous devriez obtenir la même chose qu'à la figure suivante.



Test de l'objet Graphics

Une fois votre fenêtre affichée, étirez-la, réduisez-la… À présent, vous pouvez voir ce qu'il se passe lorsque vous interagissez avec votre fenêtre : celle-ci met à jour ses composants à chaque changement d'état ou de statut. L'intérêt de disposer d'une classe héritée d'un conteneur ou d'un composant, c'est que nous pouvons redéfinir la façon dont est peint ce composant sur la fenêtre.

Après cette mise en bouche, explorons un peu plus les capacités de notre objet `Graphics`. Comme vous avez pu le voir, ce dernier permet, entre autres, de tracer des ronds ; mais il possède tout un tas de méthodes plus pratiques et amusantes les unes que les autres… Nous ne les étudierons pas toutes, mais vous aurez déjà de quoi faire.

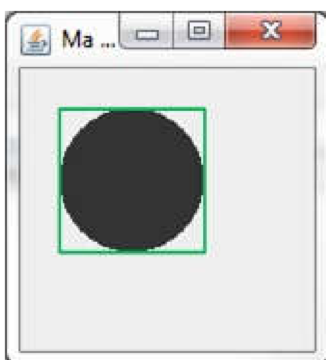
Pour commencer, reprenons la méthode utilisée précédemment : `g.fillOval(20, 20, 75, 75)`. Si nous devions traduire cette instruction en français, cela donnerait : « Trace un rond plein en commençant à dessiner sur l'axe *x* à 20 pixels et sur l'axe *y* à 20 pixels, et fais en sorte qu'il occupe 75 pixels de large et 75 pixels de haut. »

Oui, mais si je veux que mon rond soit centré et qu'il le reste ?

C'est dans ce genre de cas qu'il est intéressant d'utiliser une classe héritée. Puisque nous sommes dans notre objet `JPanel`, nous avons accès à ses données lorsque nous le dessinons.

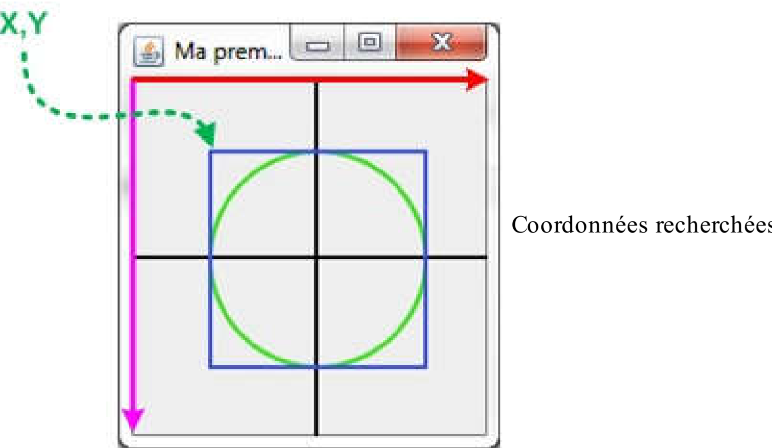
En effet, il existe des méthodes dans les objets composants qui retournent leur largeur (`getWidth()`) et leur hauteur (`getHeight()`). En revanche, réussir à centrer un rond dans un `JPanel` en toutes circonstances demande un peu de calcul mathématique de base, une pincée de connaissances et un soupçon de logique !

Reprenons notre fenêtre telle qu'elle se trouve en ce moment. Vous pouvez constater que les coordonnées de départ correspondent au coin supérieur gauche du carré qui entoure ce cercle, comme le montre la figure suivante.



Point de départ du cercle dessiné

Cela signifie que si nous voulons que notre cercle soit tout le temps centré, il faut que notre carré soit centré, donc que le centre de celui-ci corresponde au centre de notre fenêtre ! La figure suivante est un schéma représentant ce que nous devons obtenir.



Coordonnées recherchées

Ainsi, le principe est d'utiliser la largeur et la hauteur de notre composant ainsi que la largeur et la hauteur du carré qui englobe notre rond ; c'est facile, jusqu'à présent…

Maintenant, pour trouver où se situe le point depuis lequel doit commencer le dessin, il faut soustraire la moitié de la largeur du composant à la moitié de celle du rond afin d'obtenir la valeur sur l'axe *x*, et faire de même (en soustrayant les hauteurs, cette fois) pour l'axe *y*. Afin que notre rond soit le plus optimisé possible, nous allons donner comme taille à notre carré la moitié de la taille de notre fenêtre ; ce qui revient, au final, à diviser la largeur et la hauteur de cette dernière par quatre. Voici le code correspondant :

Code : Java

```java
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    int x1 = this.getWidth()/4;
    int y1 = this.getHeight()/4;
```

```
      g.fillOval(x1, y1, this.getWidth()/2, this.getHeight()/2);
    }
  }
```

Si vous testez à nouveau notre code, vous vous apercevez que notre rond est maintenant centré. Cependant, l'objet `Graphics` permet d'effectuer plein d'autres choses, comme peindre des ronds vides, par exemple. Sans rire ! Maintenant que vous avez vu comment fonctionne cet objet, nous allons pouvoir utiliser ses méthodes.

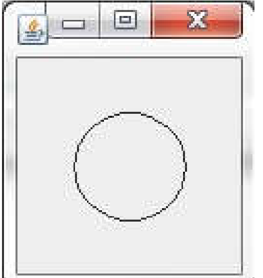
La méthode `drawOval()`

Il s'agit de la méthode qui permet de dessiner un rond vide. Elle fonctionne exactement de la même manière que la méthode `fillOval()`. Voici un code mettant en œuvre cette méthode :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    int x1 = this.getWidth()/4;
    int y1 = this.getHeight()/4;
    g.drawOval(x1, y1, this.getWidth()/2, this.getHeight()/2);
  }
}
```

Le résultat se trouve en figure suivante.



Rendu de la méthode drawOval()

Si vous spécifiez une largeur différente de la hauteur, ces méthodes dessineront une forme ovale.

La méthode `drawRect()`

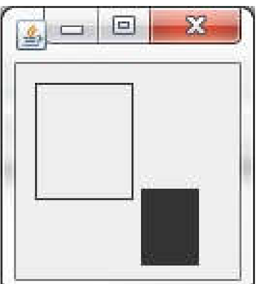
Cette méthode permet de dessiner des rectangles vides. Bien sûr, son homologue `fillRect()` existe. Ces deux méthodes fonctionnent de la même manière que les précédentes, voyez plutôt ce code :

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    //x1, y1, width, height
    g.drawRect(10, 10, 50, 60);
    g.fillRect(65, 65, 30, 40);
```

```
    }
  }
```

Le résultat se trouve à la figure suivante.


Rendu des méthodes drawRect() et fillRect()

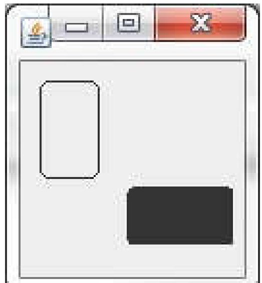
La méthode drawRoundRect()

Il s'agit du même élément que précédemment, hormis le fait que le rectangle sera arrondi. L'arrondi est défini par la valeur des deux derniers paramètres.

Code : Java

```java
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    //x1, y1, width, height, arcWidth, arcHeight
    g.drawRoundRect(10, 10, 30, 50, 10, 10);
    g.fillRoundRect(55, 65, 55, 30, 5, 5);
  }
}
```

Voyez le résultat en figure suivante.


Rendu de la méthode drawRoundRect()

La méthode drawLine()

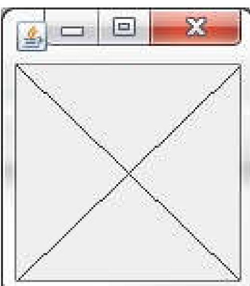
Cette méthode permet de tracer des lignes droites. Il suffit de lui spécifier les coordonnées de départ et d'arrivée de la ligne. Dans ce code, je trace les diagonales du conteneur :

Code : Java

```java
import java.awt.Graphics;
```

```java
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    //x1, y1, x2, y2
    g.drawLine(0, 0, this.getWidth(), this.getHeight());
    g.drawLine(0, this.getHeight(), this.getWidth(), 0);
  }
}
```

Le résultat se trouve à la figure suivante.


Rendu de la méthode drawLine()

La méthode drawPolygon()

Grâce à cette méthode, vous pouvez dessiner des polygones de votre composition. Eh oui, c'est à vous de définir les coordonnées de tous les points qui les forment ! Voici à quoi elle ressemble

Code : Java

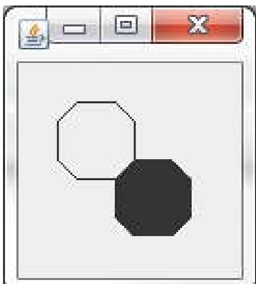
```java
drawPolygon(int[] x, int[] y, int nbrePoints);
```

Le dernier paramètre est le nombre de points formant le polygone. Ainsi, vous n'aurez pas besoin d'indiquer deux fois le point d'origine pour boucler votre figure : Java la fermera automatiquement en reliant le dernier point de votre tableau au premier. Cette méthode possède également son homologue pour dessiner des polygones remplis : `fillPolygon()`.

Code : Java

```java
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    int x[] = {20, 30, 50, 60, 60, 50, 30, 20};
    int y[] = {30, 20, 20, 30, 50, 60, 60, 50};
    g.drawPolygon(x, y, 8);

    int x2[] = {50, 60, 80, 90, 90, 80, 60, 50};
    int y2[] = {60, 50, 50, 60, 80, 90, 90, 80};
    g.fillPolygon(x2, y2, 8);
  }
}
```

Voyez le résultat à la figure suivante.


Rendu des méthodes drawPolygon() et fillPolygon()

Il existe également une méthode qui prend exactement les mêmes arguments mais qui, elle, trace plusieurs lignes : `drawPolyline()`.

Cette méthode va dessiner les lignes correspondant aux coordonnées définies dans les tableaux, sachant que lorsque son indice s'incrémente, la méthode prend automatiquement les valeurs de l'indice précédent comme point d'origine. Cette méthode ne fait pas le lien entre la première et la dernière valeur de vos tableaux. Vous pouvez essayer le code précédent en remplaçant `drawPolygon()` par cette méthode.

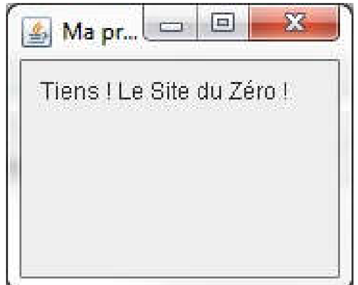
La méthode drawString()

Voici la méthode permettant d'écrire du texte. Elle est très simple à utiliser : il suffit de lui passer en paramètre la phrase à écrire et de lui spécifier à quelles coordonnées commencer.

Code : Java

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    g.drawString("Tiens ! Le Site du Zéro !", 10, 20);
  }
}
```

Le résultat se trouve à la figure suivante.



Rendu de la méthode drawString()

Vous pouvez aussi modifier la couleur (la modification s'appliquera également pour les autres méthodes) et la police d'écriture. Pour redéfinir la police d'écriture, vous devez créer un objet `Font`. Le code suivant illustre la façon de procéder.

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;

import javax.swing.JPanel;
```

```java
public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    Font font = new Font("Courier", Font.BOLD, 20);
    g.setFont(font);
    g.setColor(Color.red);
    g.drawString("Tiens ! Le Site du Zéro !", 10, 20);
  }
}
```

Le résultat correspond à la figure suivante.

Changement de couleur et de police d'écriture

La méthode `drawImage()`

Voici à quoi elle ressemble :

Code : Java

```java
drawImage(Image img, int x, int y, Observer obs);
```

Vous devez charger votre image grâce à trois objets :

- un objet `Image` ;
- un objet `ImageIO` ;
- un objet `File`.

Vous allez voir que l'utilisation de ces objets est très simple. Il suffit de déclarer un objet de type `Image` et de l'initialiser en utilisant une méthode statique de l'objet `ImageIO` qui, elle, prend un objet `File` en paramètre. Ça peut sembler compliqué, mais vous allez voir que ce n'est pas le cas… Notre image sera stockée à la racine de notre projet, mais ce n'est pas une obligation. Dans ce cas, faites attention au chemin d'accès de votre image.

En ce qui concerne le dernier paramètre de la méthode `drawImage`, il s'agit de l'objet qui est censé observer l'image. Ici, nous allons utiliser notre objet `Panneau`, donc **`this`**.

Cette méthode dessinera l'image avec ses propres dimensions. Si vous voulez qu'elle occupe l'intégralité de votre conteneur, utilisez le constructeur suivant : `drawImage(Image img, int x, int y, int width, int height, Observer obs)`.

Code : Java

```java
import java.awt.Graphics;
import java.awt.Image;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
```

```
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    try {
      Image img = ImageIO.read(new File("images.jpg"));
      g.drawImage(img, 0, 0, this);
      //Pour une image de fond
      //g.drawImage(img, 0, 0, this.getWidth(), this.getHeight(),
this);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Les résultats se trouvent aux deux figures suivantes (pour bien vous montrer la différence, j'ai créé une fenêtre plus grande que l'image).

Conservation de la taille d'origine de l'image

Adaptation de la taille de l'image

L'objet `Graphics2D`

Ceci est une amélioration de l'objet `Graphics`, et vous allez vite comprendre pourquoi.

Pour utiliser cet objet, il nous suffit en effet de caster l'objet `Graphics` en `Graphics2D` (`Graphics2D g2d = (Graphics2D) g`), et de ne surtout pas oublier d'importer notre classe qui se trouve dans le package `java.awt`. L'une des possibilités qu'offre cet objet n'est autre que celle de peindre des objets avec des dégradés de couleurs. Cette opération n'est pas du tout difficile à réaliser : il suffit d'utiliser un objet `GradientPaint` et une méthode de l'objet `Graphics2D`.

Nous n'allons pas reprendre tous les cas que nous avons vus jusqu'à présent, mais juste deux ou trois afin que vous voyiez bien la différence. Commençons par notre objet `GradientPaint` ; voici comment l'initialiser (vous devez mettre à jour vos imports en ajoutant `import java.awt.GradientPaint`) :

Code : Java

```
GradientPaint gp = new GradientPaint(0, 0, Color.RED, 30, 30,
Color.cyan, true);
```

Alors, que signifie tout cela ? Voici le détail du constructeur utilisé dans ce code :

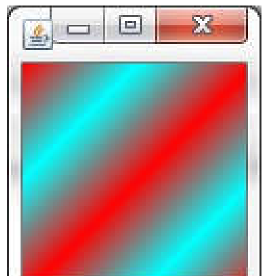
- premier paramètre : la coordonnée *x* où doit commencer la première couleur ;
- deuxième paramètre : la coordonnée *y* où doit commencer la première couleur ;
- troisième paramètre : la première couleur ;
- quatrième paramètre : la coordonnée *x* où doit commencer la seconde couleur ;
- cinquième paramètre : la coordonnée *y* où doit commencer la seconde couleur ;
- sixième paramètre : la seconde couleur ;
- septième paramètre : le booléen indiquant si le dégradé doit se répéter.

Ensuite, pour utiliser ce dégradé dans une forme, il faut mettre à jour notre objet `Graphics2D`, comme ceci :

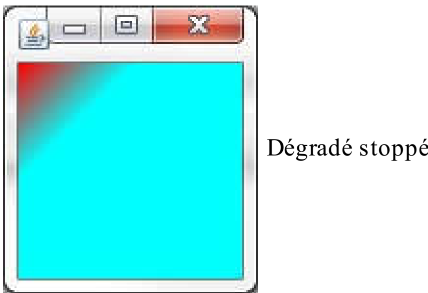
Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    Graphics2D g2d = (Graphics2D)g;
    GradientPaint gp = new GradientPaint(0, 0, Color.RED, 30, 30,
Color.cyan, true);
    g2d.setPaint(gp);
    g2d.fillRect(0, 0, this.getWidth(), this.getHeight());
  }
}
```

Les deux figures suivantes représentent les résultats obtenus, l'un avec le booléen à **`true`**, et l'autre à **`false`**.

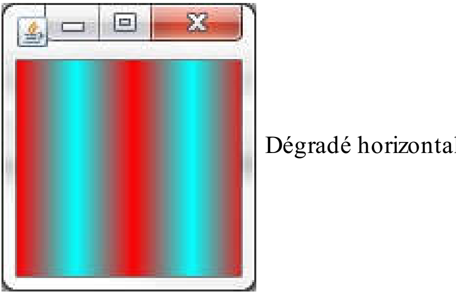


Dégradé répété



Dégradé stoppé

Votre dégradé est oblique (rien ne m'échappe, à moi :-p). Ce sont les coordonnées choisies qui influent sur la direction du dégradé. Dans notre exemple, nous partons du point de coordonnées (0, 0) vers le point de coordonnées (30, 30). Pour obtenir un dégradé vertical, il suffit d'indiquer la valeur de la seconde coordonnée *x* à 0, ce qui correspond à la figure suivante.



Dégradé horizontal

Voici un petit cadeau :

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.imageio.ImageIO;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  public void paintComponent(Graphics g){
    Graphics2D g2d = (Graphics2D)g;
    GradientPaint gp, gp2, gp3, gp4, gp5, gp6;
    gp = new GradientPaint(0, 0, Color.RED, 20, 0, Color.magenta,
true);
    gp2 = new GradientPaint(20, 0, Color.magenta, 40, 0, Color.blue,
true);
    gp3 = new GradientPaint(40, 0, Color.blue, 60, 0, Color.green,
true);
    gp4 = new GradientPaint(60, 0, Color.green, 80, 0, Color.yellow,
true);
    gp5 = new GradientPaint(80, 0, Color.yellow, 100, 0,
Color.orange, true);
    gp6 = new GradientPaint(100, 0, Color.orange, 120, 0, Color.red,
true);

    g2d.setPaint(gp);
    g2d.fillRect(0, 0, 20, this.getHeight());
    g2d.setPaint(gp2);
    g2d.fillRect(20, 0, 20, this.getHeight());
    g2d.setPaint(gp3);
    g2d.fillRect(40, 0, 20, this.getHeight());
    g2d.setPaint(gp4);
    g2d.fillRect(60, 0, 20, this.getHeight());
    g2d.setPaint(gp5);
    g2d.fillRect(80, 0, 20, this.getHeight());
    g2d.setPaint(gp6);
```

```
        g2d.fillRect(100, 0, 40, this.getHeight());
    }
}
```

Maintenant que vous savez utiliser les dégradés avec des rectangles, vous savez les utiliser avec toutes les formes. Je vous laisse essayer cela tranquillement chez vous.

- Pour créer des fenêtres, Java fournit les composants `swing` (dans `javax.swing`) et `awt` (dans `java.awt`).
- Il ne faut pas mélanger les composants `swing` et `awt`.
- Une `JFrame` est constituée de plusieurs composants.
- Par défaut, une fenêtre a une taille minimale et n'est pas visible.
- Un composant doit être bien paramétré pour qu'il fonctionne à votre convenance.
- L'objet `JPanel` se trouve dans le package `javax.swing`.
- Un `JPanel` peut contenir des composants ou d'autres conteneurs.
- Lorsque vous ajoutez un `JPanel` principal à votre fenêtre, n'oubliez pas d'indiquer à votre fenêtre qu'il constituera son content pane.
- Pour redéfinir la façon dont l'objet est dessiné sur votre fenêtre, vous devez utiliser la méthode `paintComponent()` en créant une classe héritée.
- Cette méthode prend en paramètre un objet `Graphics`.
- Cet objet doit vous être fourni par le système.
- C'est lui que vous allez utiliser pour dessiner dans votre conteneur.
- Pour des dessins plus évolués, vous devez utiliser l'objet `Graphics2D` qui s'obtient en effectuant un cast sur l'objet `Graphics`.

Le fil rouge : une animation

Dans ce chapitre, nous allons voir comment créer une animation simple. Il ne vous sera pas possible de réaliser un jeu au terme de ce chapitre, mais je pense que vous y trouverez de quoi vous amuser un peu.

Nous réutiliserons cette animation dans plusieurs chapitres de cette troisième partie afin d'illustrer le fonctionnement de divers composants graphiques. L'exemple est rudimentaire, mais il a l'avantage d'être efficace et de favoriser votre apprentissage de la programmation événementielle.

Je sens que vous êtes impatients de commencer alors… allons-y !

Création de l'animation

Voici un résumé de ce que nous avons déjà codé :

- une classe héritée de `JFrame` ;
- une classe héritée de `JPanel` avec laquelle nous faisons de jolis dessins. Un rond, en l'occurrence.

En utilisant ces deux classes, nous allons pouvoir créer un effet de déplacement.
Vous avez bien lu : j'ai parlé d'un effet de déplacement ! Le principe réside dans le fait que vous allez modifier les coordonnées de votre rond et forcer votre objet `Panneau` à se redessiner. Tout cela - vous l'avez déjà deviné - dans une boucle.

Jusqu'à présent, nous avons utilisé des valeurs fixes pour les coordonnées du rond, mais il va falloir dynamiser tout ça. Nous allons donc créer deux variables privées de type `int` dans la classe `Panneau` : appelons-les `posX` et `posY`. Dans l'animation sur laquelle nous allons travailler, notre rond viendra de l'extérieur de la fenêtre. Partons du principe que celui-ci a un diamètre de cinquante pixels : il faut donc que notre panneau peigne ce rond en dehors de sa zone d'affichage. Nous initialiserons donc nos deux variables d'instance à « -50 ». Voici le code de notre classe `Panneau` :

Code : Java

```java
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  private int posX = -50;
  private int posY = -50;

  public void paintComponent(Graphics g){
    g.setColor(Color.red);
    g.fillOval(posX, posY, 50, 50);
  }

  public int getPosX() {
    return posX;
  }

  public void setPosX(int posX) {
    this.posX = posX;
  }

  public int getPosY() {
    return posY;
  }

  public void setPosY(int posY) {
    this.posY = posY;
  }
}
```

Il ne nous reste plus qu'à faire en sorte que notre rond se déplace. Nous allons devoir trouver un moyen de changer ses coordonnées grâce à une boucle. Afin de gérer tout cela, ajoutons une méthode privée dans notre classe `Fenetre` que nous appellerons en dernier lieu dans notre constructeur. Voici donc ce à quoi ressemble notre classe `Fenetre` :

Code : Java

```java
import java.awt.Dimension;
import javax.swing.JFrame;

public class Fenetre extends JFrame{
  private Panneau pan = new Panneau();

  public Fenetre(){
    this.setTitle("Animation");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    this.setContentPane(pan);
    this.setVisible(true);
    go();
  }

  private void go(){
    for(int i = -50; i < pan.getWidth(); i++){
      int x = pan.getPosX(), y = pan.getPosY();
      x++;
      y++;
      pan.setPosX(x);
      pan.setPosY(y);
      pan.repaint();
      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Vous vous demandez sûrement l'utilité des deux instructions à la fin de la méthode `go()`. La première de ces deux nouvelles instructions est `pan.repaint()`. Elle demande à votre composant, ici un `JPanel`, de se redessiner.

La toute première fois, dans le constructeur de notre classe `Fenetre`, votre composant avait invoqué la méthode `paintComponent()` et avait dessiné un rond aux coordonnées que vous lui aviez spécifiées. La méthode `repaint()` ne fait rien d'autre qu'appeler à nouveau la méthode `paintComponent()` ; mais puisque nous avons changé les coordonnées du rond par le biais des accesseurs, la position de celui-ci sera modifiée à chaque tour de boucle.

La deuxième instruction, `Thread.sleep()`, est un moyen de suspendre votre code. Elle met en attente votre programme pendant un laps de temps défini dans la méthode `sleep()` exprimé en millièmes de seconde (plus le temps d'attente est court, plus l'animation est rapide). `Thread` est en fait un objet qui permet de créer un nouveau processus dans un programme ou de gérer le processus principal.

Dans tous les programmes, *il y a au moins un processus* : celui qui est en cours d'exécution. Vous verrez plus tard qu'il est possible de diviser certaines tâches en plusieurs processus afin de ne pas perdre du temps et des performances. Pour le moment, sachez que vous pouvez effectuer des pauses dans vos programmes grâce à cette instruction :

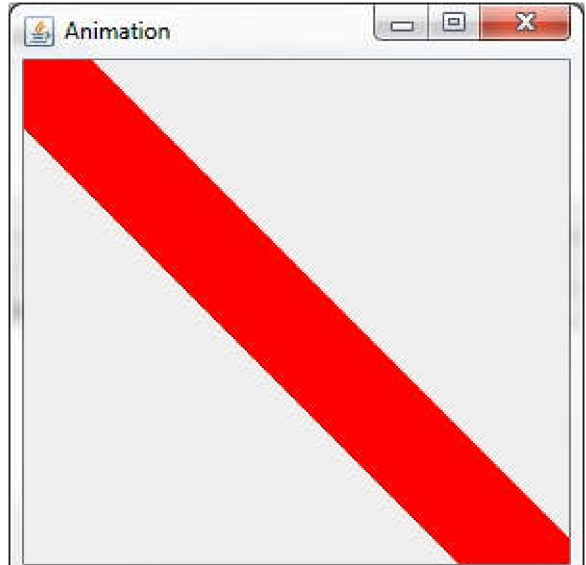
Code : Java

```java
try{
  Thread.sleep(1000); //Ici, une pause d'une seconde
}catch(InterruptedException e) {
  e.printStackTrace();
}
```

Cette instruction est dite « à risque », vous devez donc l'entourer d'un bloc **try**{…} **catch**{…} afin de capturer les exceptions potentielles. Sinon, erreur de compilation !

Ce PDF vous est offert par Accenture
http://careers.accenture.com/fr-fr

Maintenant que la lumière est faite sur cette affaire, exécutez ce code, vous obtenez la figure suivante.



Rendu final de l'animation

Bien sûr, cette image est le résultat final : vous devez avoir vu votre rond bouger. Sauf qu'il a laissé une traînée derrière lui… L'explication de ce phénomène est simple : vous avez demandé à votre objet `Panneau` de se redessiner, mais il a également affiché les précédents passages de votre rond ! Pour résoudre ce problème, il faut effacer ces derniers avant de redessiner le rond.

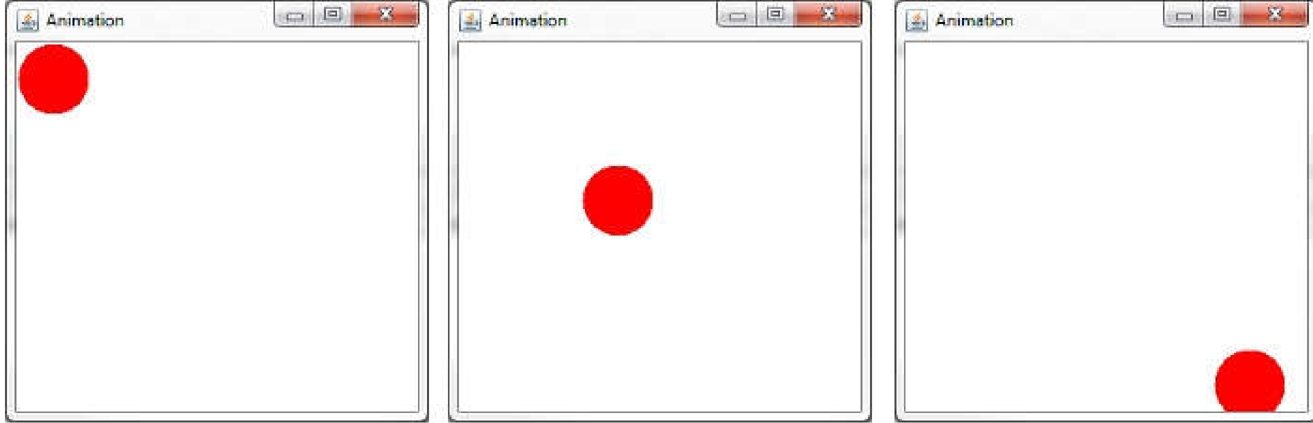
Comment ? Dessinez un rectangle de n'importe quelle couleur occupant toute la surface disponible avant de peindre votre rond. Voici le nouveau code de la classe `Panneau` :

Code : Java

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {
  private int posX = -50;
  private int posY = -50;

  public void paintComponent(Graphics g){
    //On choisit une couleur de fond pour le rectangle
    g.setColor(Color.white);
    //On le dessine de sorte qu'il occupe toute la surface
    g.fillRect(0, 0, this.getWidth(), this.getHeight());
    //On redéfinit une couleur pour le rond
    g.setColor(Color.red);
    //On le dessine aux coordonnées souhaitées
    g.fillOval(posX, posY, 50, 50);
  }

  public int getPosX() {
    return posX;
  }

  public void setPosX(int posX) {
    this.posX = posX;
  }

  public int getPosY() {
    return posY;
  }
```

```
        public void setPosY(int posY) {
          this.posY = posY;
        }
      }
```

la figure suivante représente l'animation à différents moments.



Capture de l'animation à trois moments différents

Cela vous plairait-il que votre animation se poursuive tant que vous ne fermez pas la fenêtre ? Oui ? Alors, continuons.

Améliorations

Voici l'un des moments délicats que j'attendais. Si vous vous rappelez bien ce que je vous ai expliqué sur le fonctionnement des boucles, vous vous souvenez de mon avertissement à propos des boucles infinies. Eh bien, ce que nous allons faire ici, c'est justement utiliser une boucle infinie.

Il existe plusieurs manières de réaliser une boucle infinie : vous avez le choix entre une boucle **for**, **while** ou **do**… **while**. Regardez ces déclarations :

Code : Java

```
//Exemple avec une boucle while
while(true){
  //Ce code se répétera à l'infini, car la condition est toujours
vraie !
}

//Exemple avec une boucle for
for(;;)
{
  //Idem que précédemment : il n'y a pas d'incrémentation donc la
boucle ne se terminera jamais.
}

//Exemple avec do… while
do{
  //Encore une boucle que ne se terminera pas.
}while(true);
```

Nous allons donc remplacer notre boucle finie par une boucle infinie dans la méthode `go()` de l'objet `Fenetre`. Cela donne :

Code : Java

```
private void go(){
  for(;;){
    int x = pan.getPosX(), y = pan.getPosY();
```

```
      x++;
      y++;
      pan.setPosX(x);
      pan.setPosY(y);
      pan.repaint();
      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

Si vous avez exécuté cette nouvelle version, vous vous êtes rendu compte qu'il reste un problème à régler… En effet, notre rond ne se replace pas à son point de départ une fois qu'il a atteint l'autre côté de la fenêtre !

Si vous ajoutez une instruction `System.out.println()` dans la méthode `paintComponent()` inscrivant les coordonnées du rond, vous verrez que celles-ci ne cessent de croître.

Le premier objectif est bien atteint, mais il nous reste à résoudre ce dernier problème. Pour cela, il faut réinitialiser les coordonnées du rond lorsqu'elles atteignent le bout de notre composant. Voici donc notre méthode `go()` revue et corrigée :

Code : Java

```
  private void go(){
    for(;;){
      int x = pan.getPosX(), y = pan.getPosY();
      x++;
      y++;
      pan.setPosX(x);
      pan.setPosY(y);
      pan.repaint();
      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      //Si nos coordonnées arrivent au bord de notre composant
      //On réinitialise
      if(x == pan.getWidth() || y == pan.getHeight()){
        pan.setPosX(-50);
        pan.setPosY(-50);
      }
    }
  }
```

Ce code fonctionne parfaitement (en tout cas, comme nous l'avons prévu), mais avant de passer au chapitre suivant, nous pouvons encore l'améliorer. Nous allons maintenant rendre notre rond capable de détecter les bords de notre `Panneau` et de ricocher sur ces derniers !

Jusqu'à présent, nous n'attachions aucune importance au bord que notre rond dépassait. Cela est terminé ! Dorénavant, nous séparerons le dépassement des coordonnées `posX` et `posY` de notre `Panneau`.

Pour les instructions qui vont suivre, gardez en mémoire que les coordonnées du rond correspondent en réalité aux coordonnées du coin supérieur gauche du carré entourant le rond.

Voici la marche à suivre :

- si la valeur de la coordonnée *x* du rond est inférieure à la largeur du composant et que le rond avance, on continue

d'avancer ;
- sinon, on recule.

Nous allons faire de même pour la coordonnée *y*.

Comment savoir si l'on doit avancer ou reculer ? Grâce à un booléen, par exemple. Au tout début de notre application, deux booléens seront initialisés à **`false`**, et si la coordonnée *x* est supérieure à la largeur du `Panneau`, on recule ; sinon, on avance. Idem pour la coordonnée *y*.

Dans ce code, j'utilise deux variables de type `int` pour éviter de rappeler les méthodes `getPosX()` et `getPosY()`.

Voici donc le nouveau code de la méthode `go()` :

Code : Java

```
private void go(){
  //Les coordonnées de départ de notre rond
  int x = pan.getPosX(), y = pan.getPosY();
  //Le booléen pour savoir si l'on recule ou non sur l'axe x
  boolean backX = false;
  //Le booléen pour savoir si l'on recule ou non sur l'axe y
  boolean backY = false;

  //Dans cet exemple, j'utilise une boucle while
  //Vous verrez qu'elle fonctionne très bien
  while(true){
    //Si la coordonnée x est inférieure à 1, on avance
    if(x < 1)
      backX = false;

    //Si la coordonnée x est supérieure à la taille du Panneau
moins la taille du rond, on recule
    if(x > pan.getWidth()-50)
      backX = true;

    //Idem pour l'axe y
    if(y < 1)
      backY = false;
    if(y > pan.getHeight()-50)
      backY = true;

    //Si on avance, on incrémente la coordonnée
    //backX est un booléen, donc !backX revient à écrire
    //if (backX == false)
    if(!backX)
      pan.setPosX(++x);

    //Sinon, on décrémente
    else
      pan.setPosX(--x);

    //Idem pour l'axe Y
    if(!backY)
      pan.setPosY(++y);
    else
      pan.setPosY(--y);

    //On redessine notre Panneau
    pan.repaint();

    //Comme on dit : la pause s'impose ! Ici, trois millièmes de
seconde
    try {
      Thread.sleep(3);
    } catch (InterruptedException e) {
```

```
            e.printStackTrace();
          }
        }
      }
```

Exécutez l'application : le rond ricoche contre les bords du `Panneau`. Vous pouvez même étirer la fenêtre ou la réduire, ça marchera toujours ! On commence à faire des choses sympa, non ?

Voici le code complet du projet si vous le souhaitez :

Classe Panneau

Code : Java

```java
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel {

  private int posX = -50;
  private int posY = -50;

  public void paintComponent(Graphics g) {
    // On décide d'une couleur de fond pour notre rectangle
    g.setColor(Color.white);
    // On dessine celui-ci afin qu'il prenne tout la surface
    g.fillRect(0, 0, this.getWidth(), this.getHeight());
    // On redéfinit une couleur pour notre rond
    g.setColor(Color.red);
    // On le dessine aux coordonnées souhaitées
    g.fillOval(posX, posY, 50, 50);
  }

  public int getPosX() {
    return posX;
  }

  public void setPosX(int posX) {
    this.posX = posX;
  }

  public int getPosY() {
    return posY;
  }

  public void setPosY(int posY) {
    this.posY = posY;
  }
}
```

Classe Fenetre

Code : Java

```java
import java.awt.Dimension;
import javax.swing.JFrame;

public class Fenetre extends JFrame {

  public static void main(String[] args) {
    new Fenetre();
```

```
    }

    private Panneau pan = new Panneau();

    public Fenetre() {
      this.setTitle("Animation");
      this.setSize(300, 300);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      this.setLocationRelativeTo(null);
      this.setContentPane(pan);
      this.setVisible(true);

      go();
    }

    private void go() {
      // Les coordonnées de départ de notre rond
      int x = pan.getPosX(), y = pan.getPosY();
      // Le booléen pour savoir si l'on recule ou non sur l'axe x
      boolean backX = false;
      // Le booléen pour savoir si l'on recule ou non sur l'axe y
      boolean backY = false;

      // Dans cet exemple, j'utilise une boucle while
      // Vous verrez qu'elle fonctionne très bien
      while (true) {
        // Si la coordonnée x est inférieure à 1, on avance
        if (x < 1)
          backX = false;
        // Si la coordonnée x est supérieure à la taille du Panneau
moins la taille du rond, on recule
        if (x > pan.getWidth() - 50)
          backX = true;
        // Idem pour l'axe y
        if (y < 1)
          backY = false;
        if (y > pan.getHeight() - 50)
          backY = true;

        // Si on avance, on incrémente la coordonnée
        // backX est un booléen, donc !backX revient à écrire
        // if (backX == false)
        if (!backX)
          pan.setPosX(++x);
        // Sinon, on décrémente
        else
          pan.setPosX(--x);
        // Idem pour l'axe Y
        if (!backY)
          pan.setPosY(++y);
        else
          pan.setPosY(--y);

        // On redessine notre Panneau
        pan.repaint();
        // Comme on dit : la pause s'impose ! Ici, trois millièmes de
seconde
        try {
          Thread.sleep(3);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
  }
```

- À l'instanciation d'un composant, la méthode `paintComponent()` est automatiquement appelée.
- Vous pouvez forcer un composant à se redessiner en invoquant la méthode `repaint()`.

- Pensez bien à ce que va produire votre composant une fois redessiné.
- Pour éviter que votre animation ne bave, réinitialisez le fond du composant.
- Tous les composants fonctionnent de la même manière.
- L'instruction `Thread.sleep()` permet d'effectuer une pause dans le programme.
- Cette méthode prend en paramètre un entier qui correspond à une valeur temporelle exprimée en millièmes de seconde.
- Vous pouvez utiliser des boucles infinies pour créer des animations.

Positionner des boutons

Voici l'un des moments que vous attendiez avec impatience ! Vous allez enfin pouvoir utiliser un bouton dans votre application. Cependant, ne vous réjouissez pas trop vite : vous allez effectivement insérer un bouton, mais vous vous rendrez rapidement compte que les choses se compliquent dès que vous employez ce genre de composant… Et c'est encore pire lorsqu'il y en a plusieurs !

Avant de commencer, nous devrons apprendre à positionner des composants dans une fenêtre. Il nous faut en effet gérer la façon dont le contenu est affiché dans une fenêtre.

Utiliser la classe JButton

Comme indiqué dans le titre, nous allons utiliser la classe `JButton` issue du package `javax.swing`. Au cours de ce chapitre, notre projet précédent sera mis à l'écart : oublions momentanément notre objet `Panneau`.

Créons un nouveau projet comprenant :

- une classe contenant une méthode `main` que nous appellerons `Test` ;
- une classe héritée de `JFrame` (contenant la totalité du code que l'on a déjà écrit, hormis la méthode `go()`), nous la nommerons `Fenetre`.

Dans la classe `Fenetre`, nous allons créer une variable d'instance de type `JPanel` et une autre de type `JButton`. Faisons de `JPanel` le content pane de notre `Fenetre`, puis définissons le libellé (on parle aussi d'étiquette) de notre bouton et mettons-le sur ce qui nous sert de content pane (en l'occurrence, `JPanel`).

Pour attribuer un libellé à un bouton, il y a deux possibilités :

Code : Java

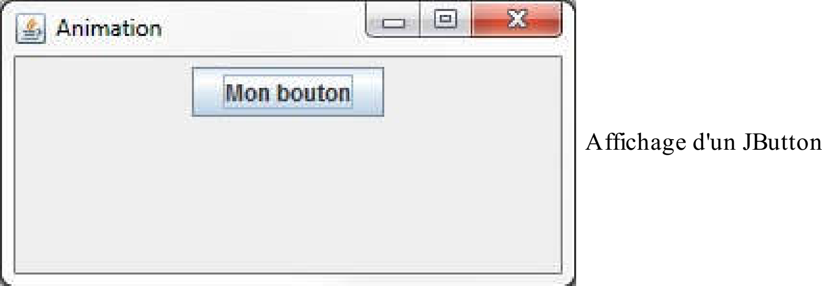
```java
//Possibilité 1 : instanciation avec le libellé
JButton bouton = new JButton("Mon premier bouton");

//Possibilité 2 : instanciation puis définition du libellé
JButton bouton2 = new JButton();
bouton2.setText("Mon deuxième bouton");
```

Il ne nous reste plus qu'à ajouter ce bouton sur notre content pane grâce à la méthode `add()` de l'objet `JPanel`. Voici donc notre code :

Code : Java

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

public class Fenetre extends JFrame{
  private JPanel pan = new JPanel();
  private JButton bouton = new JButton("Mon bouton");

  public Fenetre(){
    this.setTitle("Animation");
    this.setSize(300, 150);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    //Ajout du bouton à notre content pane
    pan.add(bouton);
    this.setContentPane(pan);
    this.setVisible(true);
  }
}
```

Voyez le résultat en figure suivante.



Affichage d'un JButton

Je ne sais pas si vous avez remarqué, mais le bouton est centré sur votre conteneur ! Cela vous semble normal ? Ça l'est, car par défaut, `JPanel` gère la mise en page. En fait, il existe en Java des objets qui servent à agencer vos composants, et `JPanel` en instancie un par défaut.

Pour vous le prouver, je vais vous faire travailler sur le content pane de votre `JFrame`. Vous constaterez que pour obtenir la même chose que précédemment, nous allons être obligés d'utiliser un de ces fameux objets d'agencement.

Tout d'abord, pour utiliser le content pane d'une `JFrame`, il faut appeler la méthode `getContentPane()` : nous ajouterons nos composants au content pane qu'elle retourne. Voici donc le nouveau code :

Code : Java

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

public class Fenetre extends JFrame{
  private JButton bouton = new JButton("Mon bouton");

  public Fenetre(){
    this.setTitle("Bouton");
    this.setSize(300, 150);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    //On ajoute le bouton au content pane de la JFrame
    this.getContentPane().add(bouton);
    this.setVisible(true);
  }
}
```

La figure suivante montre que le résultat n'est pas du tout concluant.

Bouton positionné sur le content pane

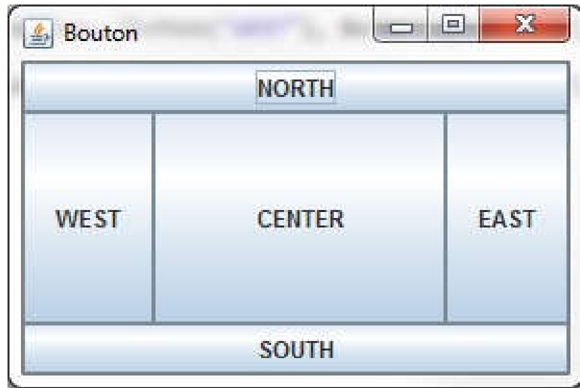
Votre bouton est énorme ! En fait, il occupe toute la place disponible, parce que le content pane de votre `JFrame` ne possède pas d'objet d'agencement. Faisons donc un petit tour d'horizon de ces objets et voyons comment ils fonctionnent.

Positionner son composant : les layout managers

Vous allez voir qu'il existe plusieurs sortes de *layout managers*, plus ou moins simples à utiliser, dont le rôle est de gérer la position des éléments sur la fenêtre. Tous ces layout managers se trouvent dans le package `java.awt`.

L'objet `BorderLayout`

Le premier objet que nous aborderons est le `BorderLayout`. Il est très pratique si vous voulez placer vos composants de façon simple par rapport à une position cardinale de votre conteneur. Si je parle de positionnement cardinal, c'est parce que vous devez utiliser les valeurs `NORTH`, `SOUTH`, `EAST`, `WEST` ou encore `CENTER`. Mais puisqu'un aperçu vaut mieux qu'un exposé sur le sujet, voici un exemple à la figure suivante mettant en œuvre un `BorderLayout`.



Exemple de BorderLayout

Cette fenêtre est composée de cinq `JButton` positionnés aux cinq endroits différents que propose un `BorderLayout`. Voici le code de cette fenêtre :

Code : Java

```java
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Fenetre extends JFrame{
  public Fenetre(){
    this.setTitle("Bouton");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    //On définit le layout à utiliser sur le content pane
    this.setLayout(new BorderLayout());
    //On ajoute le bouton au content pane de la JFrame
    //Au centre
    this.getContentPane().add(new JButton("CENTER"),
BorderLayout.CENTER);
    //Au nord
    this.getContentPane().add(new JButton("NORTH"),
BorderLayout.NORTH);
    //Au sud
    this.getContentPane().add(new JButton("SOUTH"),
BorderLayout.SOUTH);
    //À l'ouest
    this.getContentPane().add(new JButton("WEST"),
BorderLayout.WEST);
    //À l'est
    this.getContentPane().add(new JButton("EAST"),
BorderLayout.EAST);
    this.setVisible(true);
  }
}
```

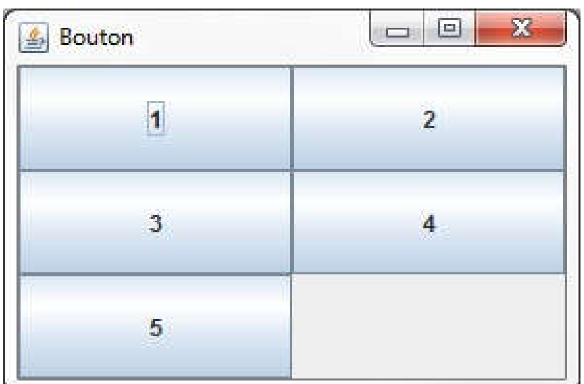
Ce n'est pas très difficile à comprendre. Vous définissez le layout à utiliser avec la méthode `setLayout()` de l'objet `JFrame` ; ensuite, pour chaque composant que vous souhaitez positionner avec `add()`, vous utilisez en deuxième paramètre un attribut **`static`** de la classe `BorderLayout` (dont la liste est citée plus haut).

Utiliser l'objet `BorderLayout` soumet vos composants à certaines contraintes. Pour une position `NORTH` ou `SOUTH`, la hauteur de votre composant sera proportionnelle à la fenêtre, mais il occupera toute la largeur ; tandis qu'avec `WEST` et `EAST`, ce sera la largeur qui sera proportionnelle alors que toute la hauteur sera occupée ! Et bien entendu, avec `CENTER`, tout l'espace est utilisé.

Vous devez savoir que `CENTER` est aussi le layout par défaut du content pane de la fenêtre, d'où la taille du bouton lorsque vous l'avez ajouté pour la première fois.

L'objet GridLayout

Celui-ci permet d'ajouter des composants suivant une grille définie par un nombre de lignes et de colonnes. Les éléments sont disposés à partir de la case située en haut à gauche. Dès qu'une ligne est remplie, on passe à la suivante. Si nous définissons une grille de trois lignes et de deux colonnes, nous obtenons le résultat visible sur la figure suivante.



Exemple de rendu avec un GridLayout

Voici le code de cet exemple :

Code : Java

```
import java.awt.GridLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Fenetre extends JFrame{
  public Fenetre(){
    this.setTitle("Bouton");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    //On définit le layout à utiliser sur le content pane
    //Trois lignes sur deux colonnes
    this.setLayout(new GridLayout(3, 2));
    //On ajoute le bouton au content pane de la JFrame
    this.getContentPane().add(new JButton("1"));
    this.getContentPane().add(new JButton("2"));
    this.getContentPane().add(new JButton("3"));
    this.getContentPane().add(new JButton("4"));
    this.getContentPane().add(new JButton("5"));
    this.setVisible(true);
  }
}
```

Ce code n'est pas bien différent du précédent : nous utilisons simplement un autre layout manager et n'avons pas besoin de définir le positionnement lors de l'ajout du composant avec la méthode `add()`.

Sachez également que vous pouvez définir le nombre de lignes et de colonnes en utilisant ces méthodes :

Code : Java

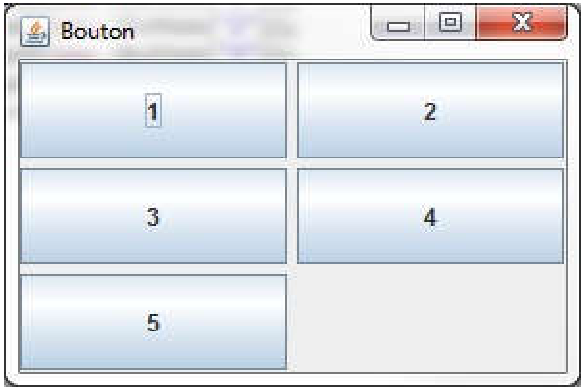
```
GridLayout gl = new GridLayout();
gl.setColumns(2);
gl.setRows(3);
this.setLayout(gl);
```

Vous pouvez aussi ajouter de l'espace entre les colonnes et les lignes.

Code : Java

```
GridLayout gl = new GridLayout(3, 2);
gl.setHgap(5); //Cinq pixels d'espace entre les colonnes (H comme
Horizontal)
gl.setVgap(5); //Cinq pixels d'espace entre les lignes (V comme
Vertical)
//Ou en abrégé : GridLayout gl = new GridLayout(3, 2, 5, 5);
```

On obtient ainsi la figure suivante.



Ajout d'espaces entre les lignes et les colonnes

L'objet `BoxLayout`

Grâce à lui, vous pourrez ranger vos composants à la suite soit sur une ligne, soit sur une colonne. Le mieux, c'est encore un exemple de rendu (voir figure suivante) avec un code.

Exemple de BoxLayout

Voici le code correspondant :

Code : Java

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

```java
public class Fenetre extends JFrame{

  public Fenetre(){

    this.setTitle("Box Layout");
    this.setSize(300, 120);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    JPanel b1 = new JPanel();
    //On définit le layout en lui indiquant qu'il travaillera en
ligne
    b1.setLayout(new BoxLayout(b1, BoxLayout.LINE_AXIS));
    b1.add(new JButton("Bouton 1"));

    JPanel b2 = new JPanel();
    //Idem pour cette ligne
    b2.setLayout(new BoxLayout(b2, BoxLayout.LINE_AXIS));
    b2.add(new JButton("Bouton 2"));
    b2.add(new JButton("Bouton 3"));

    JPanel b3 = new JPanel();
    //Idem pour cette ligne
    b3.setLayout(new BoxLayout(b3, BoxLayout.LINE_AXIS));
    b3.add(new JButton("Bouton 4"));
    b3.add(new JButton("Bouton 5"));
    b3.add(new JButton("Bouton 6"));

    JPanel b4 = new JPanel();
    //On positionne maintenant ces trois lignes en colonne
    b4.setLayout(new BoxLayout(b4, BoxLayout.PAGE_AXIS));
    b4.add(b1);
    b4.add(b2);
    b4.add(b3);

    this.getContentPane().add(b4);
    this.setVisible(true);
  }
}
```

Ce code est simple : on crée trois `JPanel` contenant chacun un certain nombre de `JButton` rangés en ligne grâce à l'attribut `LINE_AXIS`. Ces trois conteneurs créés, nous les rangeons dans un quatrième où, cette fois, nous les agençons dans une colonne grâce à l'attribut `PAGE_AXIS`. Rien de compliqué, vous en conviendrez, mais vous devez savoir qu'il existe un moyen encore plus simple d'utiliser ce layout : *via* l'objet `Box`. Ce dernier n'est rien d'autre qu'un conteneur paramétré avec un `BoxLayout`. Voici un code affichant la même chose que le précédent :

Code : Java

```java
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Fenetre extends JFrame{

  public Fenetre(){
    this.setTitle("Box Layout");
    this.setSize(300, 120);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    //On crée un conteneur avec gestion horizontale
    Box b1 = Box.createHorizontalBox();
    b1.add(new JButton("Bouton 1"));
    //Idem
    Box b2 = Box.createHorizontalBox();
```

```
        b2.add(new JButton("Bouton 2"));
        b2.add(new JButton("Bouton 3"));
        //Idem
        Box b3 = Box.createHorizontalBox();
        b3.add(new JButton("Bouton 4"));
        b3.add(new JButton("Bouton 5"));
        b3.add(new JButton("Bouton 6"));
        //On crée un conteneur avec gestion verticale
        Box b4 = Box.createVerticalBox();
        b4.add(b1);
        b4.add(b2);
        b4.add(b3);

        this.getContentPane().add(b4);
        this.setVisible(true);
    }
}
```

L'objet CardLayout

Vous allez à présent pouvoir gérer vos conteneurs comme un tas de cartes (les uns sur les autres), et basculer d'un contenu à l'autre en deux temps, trois clics. Le principe est d'assigner des conteneurs au layout en leur donnant un nom afin de les retrouver plus facilement, permettant de passer de l'un à l'autre sans effort. La figure suivante est un schéma représentant ce mode de fonctionnement.

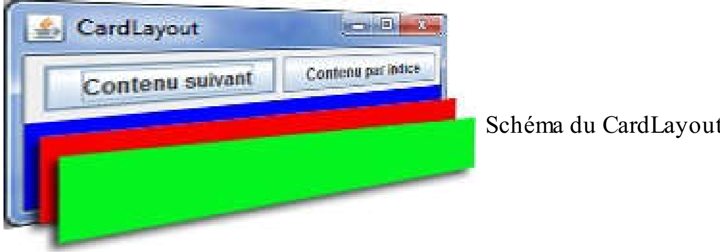


Schéma du CardLayout

Je vous propose un code utilisant ce layout. Vous remarquerez que j'ai utilisé des boutons afin de passer d'un conteneur à un autre et n'y comprendrez peut-être pas tout, mais ne vous inquiétez pas, nous allons apprendre à réaliser tout cela avant la fin de ce chapitre. Pour le moment, ne vous attardez donc pas trop sur les actions : concentrez-vous sur le layout en lui-même.

Code : Java

```
import java.awt.BorderLayout;
import java.awt.CardLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

  CardLayout cl = new CardLayout();
  JPanel content = new JPanel();
  //Liste des noms de nos conteneurs pour la pile de cartes
  String[] listContent = {"CARD_1", "CARD_2", "CARD_3"};
  int indice = 0;

  public Fenetre(){
    this.setTitle("CardLayout");
    this.setSize(300, 120);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
```

```
    //On crée trois conteneurs de couleur différente
    JPanel card1 = new JPanel();
    card1.setBackground(Color.blue);
    JPanel card2 = new JPanel();
    card2.setBackground(Color.red);
    JPanel card3 = new JPanel();
    card3.setBackground(Color.green);

    JPanel boutonPane = new JPanel();
    JButton bouton = new JButton("Contenu suivant");
    //Définition de l'action du bouton
    bouton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event){
        //Via cette instruction, on passe au prochain conteneur de
la pile
        cl.next(content);
      }
    });

    JButton bouton2 = new JButton("Contenu par indice");
    //Définition de l'action du bouton2
    bouton2.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event){
        if(++indice > 2)
          indice = 0;
        //Via cette instruction, on passe au conteneur
correspondant au nom fourni en paramètre
        cl.show(content, listContent[indice]);
      }
    });

    boutonPane.add(bouton);
    boutonPane.add(bouton2);
    //On définit le layout
    content.setLayout(cl);
    //On ajoute les cartes à la pile avec un nom pour les retrouver
    content.add(card1, listContent[0]);
    content.add(card2, listContent[1]);
    content.add(card3, listContent[2]);

    this.getContentPane().add(boutonPane, BorderLayout.NORTH);
    this.getContentPane().add(content, BorderLayout.CENTER);
    this.setVisible(true);
  }
}
```

La figure suivante correspond aux résultats de ce code à chaque clic sur les boutons.

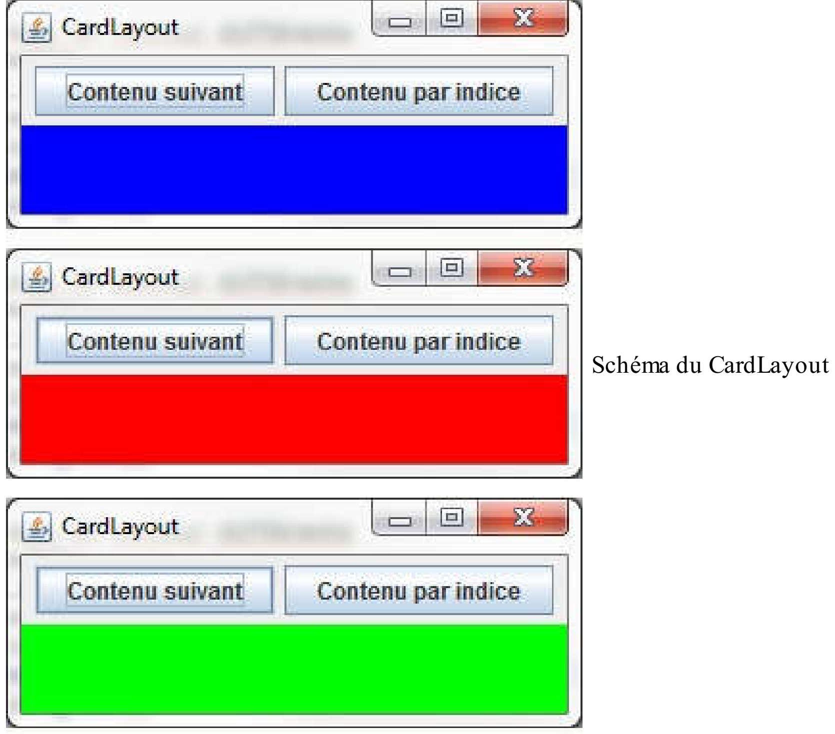
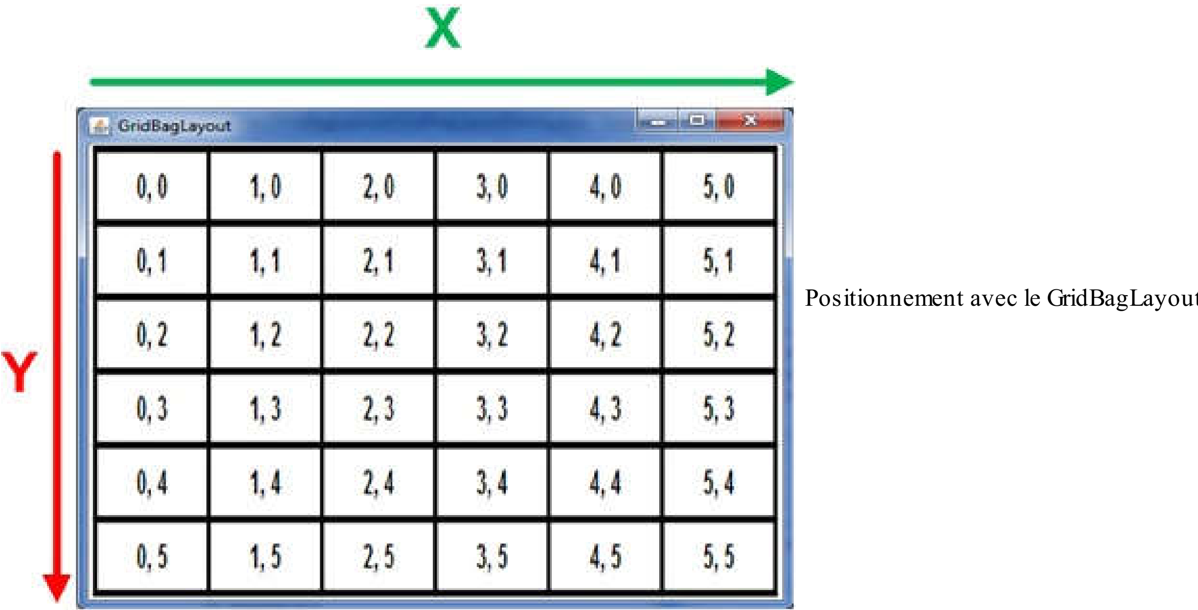


Schéma du CardLayout

L'objet `GridBagLayout`

Cet objet est certainement le plus difficile à utiliser et à comprendre (ce qui l'a beaucoup desservi auprès des développeurs Java). Pour faire simple, ce layout se présente sous la forme d'une grille à la façon d'un tableau Excel : vous devez positionner vos composants en vous servant des coordonnées des cellules (qui sont définies lorsque vous spécifiez leur nombre). Vous devez aussi définir les marges et la façon dont vos composants se répliquent dans les cellules... Vous voyez que c'est plutôt dense comme gestion du positionnement. Je tiens aussi à vous prévenir que je n'entrerai pas trop dans les détails de ce layout afin de ne pas trop compliquer les choses.

La figure suivante représente la façon dont nous allons positionner nos composants.

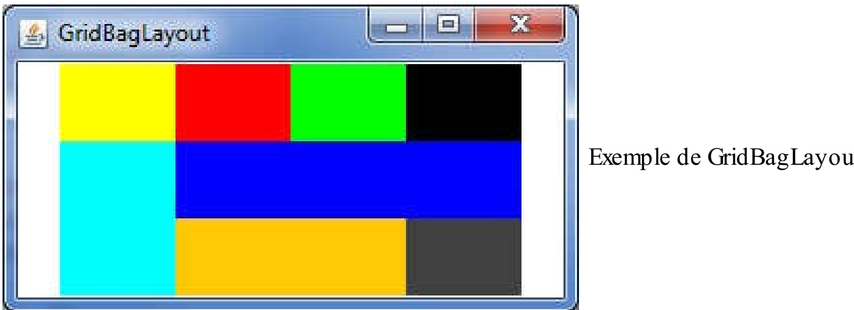


Positionnement avec le GridBagLayout

Imaginez que le nombre de colonnes et de lignes ne soit pas limité comme il l'est sur le schéma (c'est un exemple et j'ai dû limiter

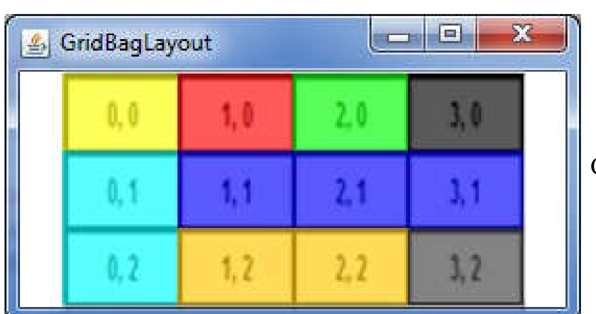
sa taille, mais le principe est là). Vous paramétreriez le composant avec des coordonnées de cellules, en précisant si celui-ci doit occuper une ou plusieurs d'entre elles. Afin d'obtenir un rendu correct, vous devriez indiquer au layout manager lorsqu'une ligne se termine, ce qui se fait en spécifiant qu'un composant est le dernier élément d'une ligne, et vous devriez en plus spécifier au composant débutant la ligne qu'il doit suivre le dernier composant de la précédente.

Je me doute que c'est assez flou et confus, je vous propose donc de regarder la figure suivante, qui est un exemple de ce que nous allons obtenir.



Exemple de GridBagLayout

Tous les éléments que vous voyez sont des conteneurs positionnés suivant une matrice, comme expliqué ci-dessus. Afin que vous vous en rendiez compte, regardez comment le tout est rangé sur la figure suivante.



Composition du GridBagLayout

Vous pouvez voir que nous avons fait en sorte d'obtenir un tableau de quatre colonnes sur trois lignes. Nous avons positionné quatre éléments sur la première ligne, spécifié que le quatrième élément terminait celle-ci, puis nous avons placé un autre composant au début de la deuxième ligne d'une hauteur de deux cases, en informant le gestionnaire que celui-ci suivait directement la fin de la première ligne. Nous ajoutons un composant de trois cases de long terminant la deuxième ligne, pour passer ensuite à un composant de deux cases de long puis à un dernier achevant la dernière ligne.

Lorsque des composants se trouvent sur plusieurs cases, vous devez spécifier la façon dont ils s'étalent : horizontalement ou verticalement.

Le moment est venu de vous fournir le code de cet exemple, mais je vous préviens, ça pique un peu les yeux :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

  public Fenetre(){
    this.setTitle("GridBagLayout");
    this.setSize(300, 160);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    //On crée nos différents conteneurs de couleur différente
```

```
        //On crée nos différents conteneurs de couleur différente
        JPanel cell1 = new JPanel();
        cell1.setBackground(Color.YELLOW);
        cell1.setPreferredSize(new Dimension(60, 40));
        JPanel cell2 = new JPanel();
        cell2.setBackground(Color.red);
        cell2.setPreferredSize(new Dimension(60, 40));
        JPanel cell3 = new JPanel();
        cell3.setBackground(Color.green);
        cell3.setPreferredSize(new Dimension(60, 40));
        JPanel cell4 = new JPanel();
        cell4.setBackground(Color.black);
        cell4.setPreferredSize(new Dimension(60, 40));
        JPanel cell5 = new JPanel();
        cell5.setBackground(Color.cyan);
        cell5.setPreferredSize(new Dimension(60, 40));
        JPanel cell6 = new JPanel();
        cell6.setBackground(Color.BLUE);
        cell6.setPreferredSize(new Dimension(60, 40));
        JPanel cell7 = new JPanel();
        cell7.setBackground(Color.orange);
        cell7.setPreferredSize(new Dimension(60, 40));
        JPanel cell8 = new JPanel();
        cell8.setBackground(Color.DARK_GRAY);
        cell8.setPreferredSize(new Dimension(60, 40));

        //Le conteneur principal
        JPanel content = new JPanel();
        content.setPreferredSize(new Dimension(300, 120));
        content.setBackground(Color.WHITE);
        //On définit le layout manager
        content.setLayout(new GridBagLayout());

        //L'objet servant à positionner les composants
        GridBagConstraints gbc = new GridBagConstraints();

        //On positionne la case de départ du composant
        gbc.gridx = 0;
        gbc.gridy = 0;
        //La taille en hauteur et en largeur
        gbc.gridheight = 1;
        gbc.gridwidth = 1;
        content.add(cell1, gbc);
        //---------------------------------------------
        gbc.gridx = 1;
        content.add(cell2, gbc);
        //---------------------------------------------
        gbc.gridx = 2;
        content.add(cell3, gbc);
        //---------------------------------------------
        //Cette instruction informe le layout que c'est une fin de
    ligne
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.gridx = 3;
        content.add(cell4, gbc);
        //---------------------------------------------
        gbc.gridx = 0;
        gbc.gridy = 1;
        gbc.gridwidth = 1;
        gbc.gridheight = 2;
        //Celle-ci indique que la cellule se réplique de façon
    verticale
        gbc.fill = GridBagConstraints.VERTICAL;
        content.add(cell5, gbc);
        //---------------------------------------------
        gbc.gridx = 1;
        gbc.gridheight = 1;
        //Celle-ci indique que la cellule se réplique de façon
    horizontale
        gbc.fill = GridBagConstraints.HORIZONTAL;
```

Ce PDF vous est offert par Accenture
http://careers.accenture.com/fr-fr

```
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        content.add(cell6, gbc);
        //-----------------------------------------------
        gbc.gridx = 1;
        gbc.gridy = 2;
        gbc.gridwidth = 2;
        content.add(cell7, gbc);
        //-----------------------------------------------
        gbc.gridx = 3;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        content.add(cell8, gbc);
        //-----------------------------------------------
        //On ajoute le conteneur
        this.setContentPane(content);
        this.setVisible(true);
    }
}
```

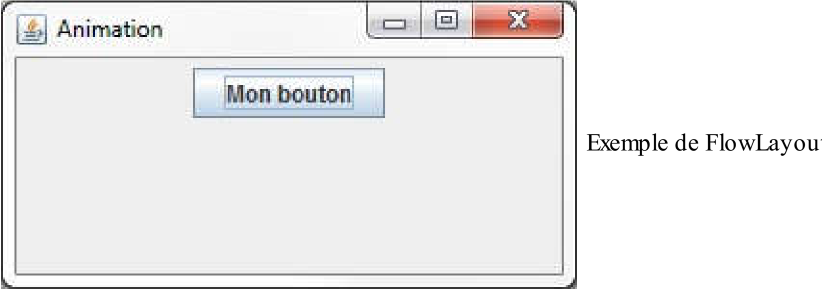
Vous pouvez vous rendre compte que c'est *via* l'objet `GridBagConstraints` que tout se joue. Vous pouvez utiliser ses différents arguments afin de positionner vos composants, en voici une liste :

- `gridx` : position en *x* dans la grille.
- `gridy` : position en *y* dans la grille.
- `gridwidth` : nombre de colonnes occupées.
- `gridheight` : nombre de lignes occupées.
- `weightx` : si la grille est plus large que l'espace demandé, l'espace est redistribué proportionnellement aux valeurs de `weightx` des différentes colonnes.
- `weighty` : si la grille est plus haute que l'espace demandé, l'espace est redistribué proportionnellement aux valeurs de `weighty` des différentes lignes.
- `anchor` : ancrage du composant dans la cellule, c'est-à-dire son alignement dans la cellule (en bas à droite, en haut à gauche…). Voici les différentes valeurs utilisables :
    - `FIRST_LINE_START` : en haut à gauche ;
    - `PAGE_START` : en haut au centre ;
    - `FIRST_LINE_END` : en haut à droite ;
    - `LINE_START` : au milieu à gauche ;
    - `CENTER` : au milieu et centré ;
    - `LINE_END` : au milieu à droite ;
    - `LAST_LINE_START` : en bas à gauche ;
    - `PAGE_END` : en bas au centre ;
    - `LAST_LINE_END` : en bas à droite.
- `fill` : remplissage si la cellule est plus grande que le composant. Valeurs possibles : `NONE`, `HORIZONTAL`, `VERTICAL` et `BOTH`.
- `insets` : espace autour du composant. S'ajoute aux espacements définis par les propriétés `ipadx` et `ipady` ci-dessous.
- `ipadx` : espacement à gauche et à droite du composant.
- `ipady` : espacement au-dessus et au-dessous du composant.

Dans mon exemple, je ne vous ai pas parlé de tous les attributs existants, mais si vous avez besoin d'un complément d'information, n'hésitez pas à consulter le site d'Oracle.

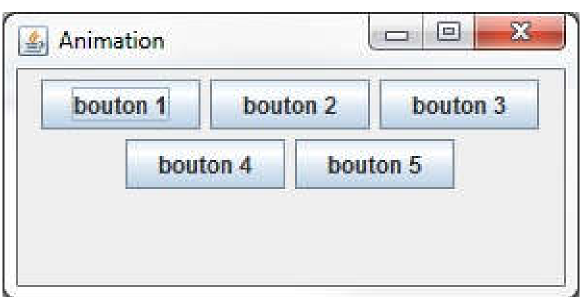
L'objet `FlowLayout`

Celui-ci est certainement le plus facile à utiliser ! Il se contente de centrer les composants dans le conteneur. Regardez plutôt la figure suivante.



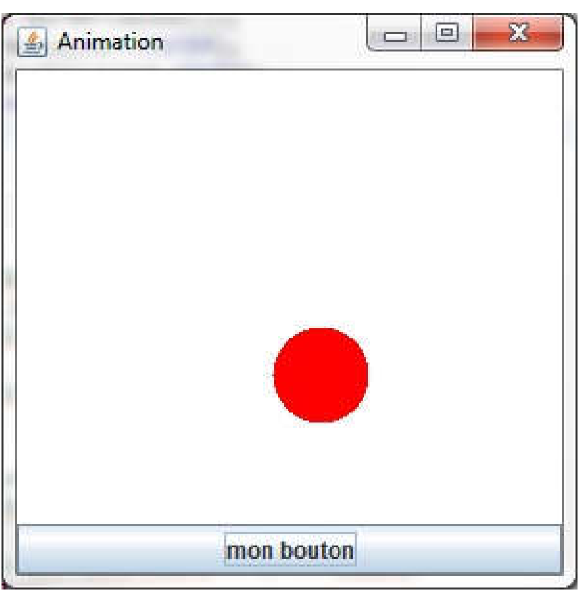
Exemple de FlowLayout

On dirait bien que nous venons de trouver le layout manager défini par défaut dans les objets `JPanel`. Lorsque vous insérez plusieurs composants dans ce gestionnaire, il passe à la ligne suivante dès que la place est trop étroite. Voyez l'exemple de la figure suivante.



FlowLayout contenant plusieurs composants

Il faut que vous sachiez que les IHM ne sont en fait qu'une imbrication de composants positionnés grâce à des layout managers. Vous allez tout de suite voir de quoi je veux parler : nous allons maintenant utiliser notre conteneur personnalisé avec un bouton. Vous pouvez donc revenir dans le projet contenant notre animation créée au cours des chapitres précédents. Le but est d'insérer notre animation au centre de notre fenêtre et un bouton en bas de celle-ci, comme le montre la figure suivante.



Bouton et animation dans la même fenêtre

Voici le nouveau code de notre classe `Fenetre` :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class Fenetre extends JFrame{
```

```
    private Panneau pan = new Panneau();
    private JButton bouton = new JButton("mon bouton");
    private JPanel container = new JPanel();

    public Fenetre(){
      this.setTitle("Animation");
      this.setSize(300, 300);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      this.setLocationRelativeTo(null);
      container.setBackground(Color.white);
      container.setLayout(new BorderLayout());
      container.add(pan, BorderLayout.CENTER);
      container.add(bouton, BorderLayout.SOUTH);
      this.setContentPane(container);
      this.setVisible(true);
      go();
    }

    private void go(){
      //Les coordonnées de départ de notre rond
      int x = pan.getPosX(), y = pan.getPosY();
      //Le booléen pour savoir si l'on recule ou non sur l'axe x
      boolean backX = false;
      //Le booléen pour savoir si l'on recule ou non sur l'axe y
      boolean backY = false;

      //Dans cet exemple, j'utilise une boucle while
      //Vous verrez qu'elle fonctionne très bien
      while(true){
        //Si la coordonnée x est inférieure à 1, on avance
        if(x < 1)backX = false;
        //Si la coordonnée x est supérieure à la taille du Panneau
  moins la taille du rond, on recule
        if(x > pan.getWidth()-50)backX = true;
        //Idem pour l'axe y
        if(y < 1)backY = false;
        if(y > pan.getHeight()-50)backY = true;

        //Si on avance, on incrémente la coordonnée
        if(!backX)
          pan.setPosX(++x);
        //Sinon, on décrémente
        else
          pan.setPosX(--x);
        //Idem pour l'axe Y
        if(!backY)
          pan.setPosY(++y);
        else
          pan.setPosY(--y);

        //On redessine notre Panneau
        pan.repaint();
        //Comme on dit : la pause s'impose ! Ici, trois millièmes de
  seconde
        try {
          Thread.sleep(3);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
  }
```

- Un bouton s'utilise avec la classe `JButton` présente dans le package `javax.swing`.
- Pour ajouter un bouton dans une fenêtre, vous devez utiliser la méthode `add()` de son content pane.
- Il existe des objets permettant de positionner les composants sur un content pane ou un conteneur : les layout managers.
- Les layout managers se trouvent dans le package `java.awt`.
- Le layout manager par défaut du content pane d'un objet `JFrame` est le `BorderLayout`.

- Le layout manager par défaut d'un objet `JPanel` est le `FlowLayout`.
- Outre le `FlowLayout` et le `BorderLayout`, il existe le `GridLayout`, le `CardLayout`, le `BoxLayout` et le `GridBagLayout`. La liste n'est pas exhaustive.
- On définit un layout sur un conteneur grâce à la méthode `setLayout()`.

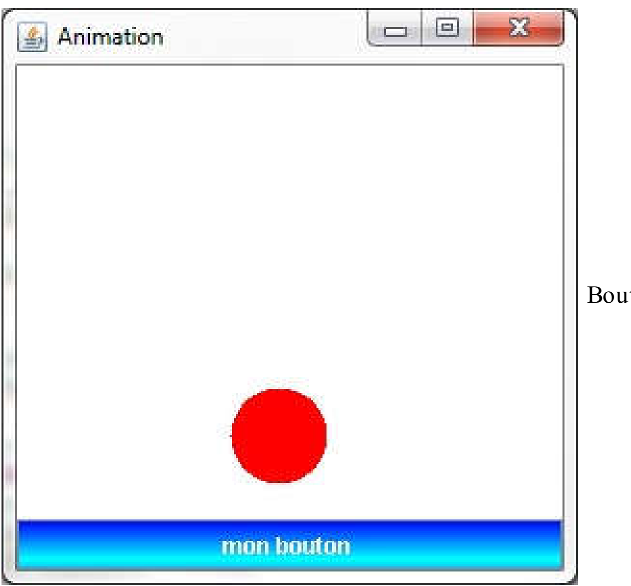
Interagir avec des boutons

Nous avons vu dans le chapitre précédent les différentes façons de positionner des boutons et, par extension, des composants (car oui, ce que nous venons d'apprendre pourra être réutilisé avec tous les autres composants que nous verrons par la suite).

Maintenant que vous savez positionner des composants, il est grand temps de leur indiquer ce qu'ils doivent faire. C'est ce que je vous propose d'aborder dans ce chapitre. Mais avant cela, nous allons voir comment personnaliser un bouton. Toujours prêts ?

Une classe Bouton personnalisée

Créons une classe héritant de `javax.swing.JButton` que nous appellerons `Bouton` et redéfinissons sa méthode `paintComponent()`. Vous devriez y arriver tout seuls. Cet exemple est représenté à la figure suivante :



Bouton personnalisé

Voici la classe `Bouton` de cette application :

Code : Java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JButton;

public class Bouton extends JButton {
  private String name;
  public Bouton(String str){
    super(str);
    this.name = str;
  }

  public void paintComponent(Graphics g){
    Graphics2D g2d = (Graphics2D)g;
    GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20,
Color.cyan, true);
    g2d.setPaint(gp);
    g2d.fillRect(0, 0, this.getWidth(), this.getHeight());
    g2d.setColor(Color.white);
    g2d.drawString(this.name, this.getWidth() / 2 -
(this.getWidth()/ 2 /4), (this.getHeight() / 2) + 5);
  }
}
```

J'ai aussi créé un bouton personnalisé avec une image de fond, comme le montre la figure suivante.

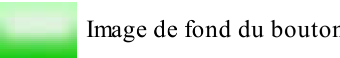
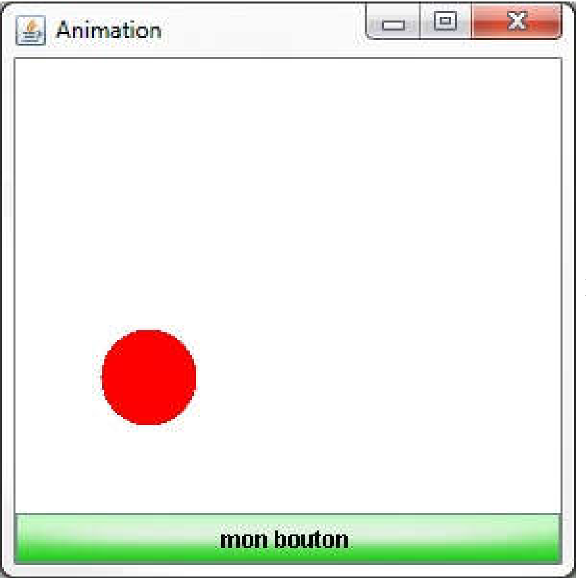


Image de fond du bouton

Voyez le résultat en figure suivante.



Bouton avec une image de fond

J'ai appliqué l'image (bien sûr, ladite image se trouve à la racine de mon projet !) sur l'intégralité du fond, comme je l'ai montré lorsque nous nous amusions avec notre `Panneau`. Voici le code de cette classe `Bouton` :

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JButton;

public class Bouton extends JButton{
  private String name;
  private Image img;

  public Bouton(String str){
    super(str);
    this.name = str;
    try {
      img = ImageIO.read(new File("fondBouton.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void paintComponent(Graphics g){
    Graphics2D g2d = (Graphics2D)g;
    GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20,
```

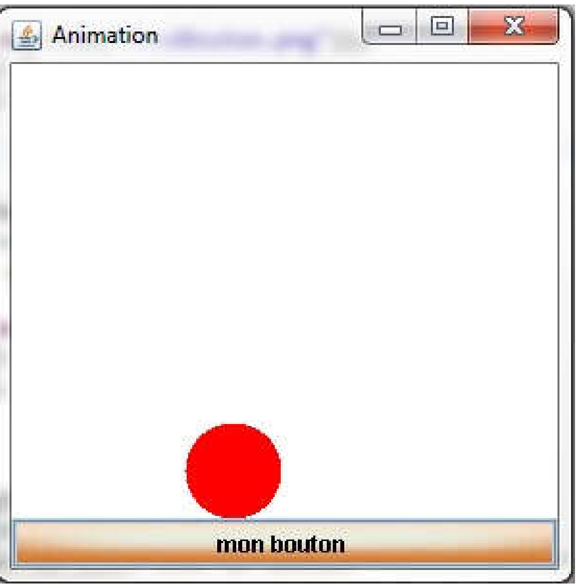
```
        Color.cyan, true);
            g2d.setPaint(gp);
            g2d.drawImage(img, 0, 0, this.getWidth(), this.getHeight(),
        this);
            g2d.setColor(Color.black);
            g2d.drawString(this.name, this.getWidth() / 2 - (this.getWidth()
        / 2 /4), (this.getHeight() / 2) + 5);
          }
        }
```

Rien de compliqué jusque-là… C'est à partir de maintenant que les choses deviennent intéressantes !

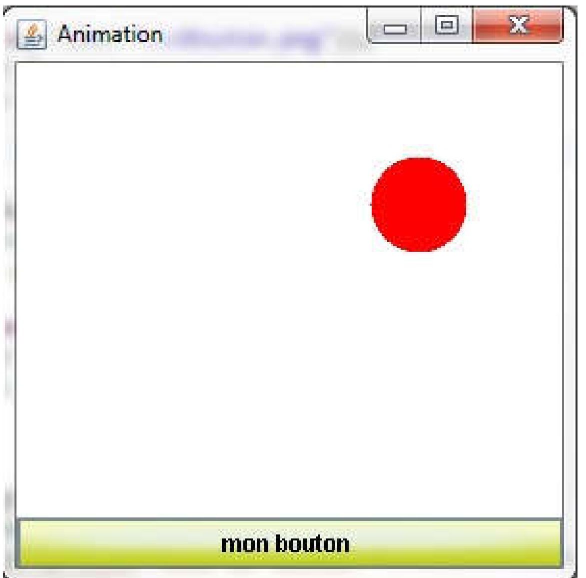
Et si je vous proposais de changer l'aspect de votre objet lorsque vous cliquez dessus avec votre souris et lorsque vous relâchez le clic ? Il existe des interfaces à implémenter qui permettent de gérer toutes sortes d'événements dans votre IHM. Le principe est un peu déroutant au premier abord, mais il est assez simple lorsqu'on a un peu pratiqué. N'attendons plus et voyons cela de plus près !

Interactions avec la souris : l'interface `MouseListener`

Avant de nous lancer dans l'implémentation, vous pouvez voir le résultat que nous allons obtenir sur les deux figures suivantes.



Apparence du bouton au survol de la souris



Apparence du bouton lors d'un clic de souris

Il va tout de même falloir passer par un peu de théorie avant d'arriver à ce résultat. Pour détecter les événements qui surviennent sur votre composant, Java utilise ce qu'on appelle le *design pattern observer*. Je ne vous l'expliquerai pas dans le détail tout de suite, nous le verrons à la fin de ce chapitre.

Vous vous en doutez, nous devrons implémenter l'interface `MouseListener` dans notre classe `Bouton`. Nous devrons aussi préciser à notre classe qu'elle devra tenir quelqu'un au courant de ses changements d'état par rapport à la souris. Ce quelqu'un n'est autre… qu'elle-même ! Eh oui : notre classe va s'écouter, ce qui signifie que dès que notre objet observable (notre bouton) obtiendra des informations concernant les actions effectuées par la souris, il indiquera à l'objet qui l'observe (c'est-à-dire à lui-même) ce qu'il doit effectuer.

Cela est réalisable grâce à la méthode `addMouseListener(MouseListener obj)` qui prend un objet `MouseListener` en paramètre (ici, elle prendra **`this`**). Rappelez-vous que **vous pouvez utiliser le type d'une interface comme supertype** : ici, notre classe implémente l'interface `MouseListener`, nous pouvons donc utiliser cet objet comme référence de cette interface.

Voici à présent notre classe `Bouton` :

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JButton;

public class Bouton extends JButton implements MouseListener{
  private String name;
  private Image img;
  public Bouton(String str){
    super(str);
    this.name = str;
    try {
      img = ImageIO.read(new File("fondBouton.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
    //Grâce à cette instruction, notre objet va s'écouter
    //Dès qu'un événement de la souris sera intercepté, il en sera
```

```java
averti
      this.addMouseListener(this);
    }

    public void paintComponent(Graphics g){
      Graphics2D g2d = (Graphics2D)g;
      GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20,
Color.cyan, true);
      g2d.setPaint(gp);
      g2d.drawImage(img, 0, 0, this.getWidth(), this.getHeight(),
this);
      g2d.setColor(Color.black);
      g2d.drawString(this.name, this.getWidth() / 2 - (this.getWidth()
/  2 /4), (this.getHeight() / 2) + 5);
    }

    //Méthode appelée lors du clic de souris
    public void mouseClicked(MouseEvent event) { }

    //Méthode appelée lors du survol de la souris
    public void mouseEntered(MouseEvent event) { }

    //Méthode appelée lorsque la souris sort de la zone du bouton
    public void mouseExited(MouseEvent event) { }

    //Méthode appelée lorsque l'on presse le bouton gauche de la
souris
    public void mousePressed(MouseEvent event) { }

    //Méthode appelée lorsque l'on relâche le clic de souris
    public void mouseReleased(MouseEvent event) { }
  }
```

C'est en redéfinissant ces différentes méthodes présentes dans l'interface `MouseListener` que nous allons gérer les différentes images à dessiner dans notre objet.
Rappelez-vous en outre que même si vous n'utilisez pas toutes les méthodes d'une interface, vous devez malgré tout insérer le squelette des méthodes non utilisées (avec les accolades), cela étant également valable pour les classes abstraites.

Dans notre cas, la méthode `repaint()` est appelée de façon implicite : lorsqu'un événement est déclenché, notre objet se redessine automatiquement ! Comme lorsque vous redimensionniez votre fenêtre dans les premiers chapitres.

Nous n'avons alors plus qu'à modifier notre image en fonction de la méthode invoquée. Notre objet comportera les caractéristiques suivantes :

- il aura une teinte jaune au survol de la souris ;
- il aura une teinte orangée lorsque l'on pressera le bouton gauche ;
- il reviendra à la normale si on relâche le clic.

Pour ce faire, je vous propose de télécharger les fichiers PNG dont je me suis servi (rien ne vous empêche de les créer vous-mêmes).

Télécharger les images

Je vous rappelle que dans le code qui suit, les images sont placées à la racine du projet.

Voici maintenant le code de notre classe `Bouton` personnalisée :

Code : Java

```
import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JButton;

public class Bouton extends JButton implements MouseListener{
  private String name;
  private Image img;

  public Bouton(String str){
    super(str);
    this.name = str;
    try {
      img = ImageIO.read(new File("fondBouton.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
  this.addMouseListener(this);
  }

  public void paintComponent(Graphics g){
    Graphics2D g2d = (Graphics2D)g;
    GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20,
Color.cyan, true);
    g2d.setPaint(gp);
    g2d.drawImage(img, 0, 0, this.getWidth(), this.getHeight(),
this);
    g2d.setColor(Color.black);
    g2d.drawString(this.name, this.getWidth() / 2 - (this.getWidth()
/ 2 /4), (this.getHeight() / 2) + 5);
  }

  public void mouseClicked(MouseEvent event) {
    //Inutile d'utiliser cette méthode ici
  }

  public void mouseEntered(MouseEvent event) {
    //Nous changeons le fond de notre image pour le jaune lors du
survol, avec le fichier fondBoutonHover.png
    try {
      img = ImageIO.read(new File("fondBoutonHover.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void mouseExited(MouseEvent event) {
  //Nous changeons le fond de notre image pour le vert lorsque nous
quittons le bouton, avec le fichier fondBouton.png
    try {
      img = ImageIO.read(new File("fondBouton.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void mousePressed(MouseEvent event) {
    //Nous changeons le fond de notre image pour le jaune lors du
clic gauche, avec le fichier fondBoutonClic.png
    try {
      img = ImageIO.read(new File("fondBoutonClic.png"));
    } catch (IOException e) {
```

Ce PDF vous est offert par Accenture
http://careers.accenture.com/fr-fr

```
            e.printStackTrace();
          }
        }

        public void mouseReleased(MouseEvent event) {
          //Nous changeons le fond de notre image pour le orange lorsque
      nous relâchons le clic, avec le fichier fondBoutonHover.png
          try {
            img = ImageIO.read(new File("fondBoutonHover.png"));
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
```

Et voilà le travail ! Si vous avez enregistré mes images, elles ne possèdent probablement pas le même nom que dans mon code : vous devez alors modifier le code en fonction de celui que vous leur avez attribué ! D'accord, ça va de soi… mais on ne sait jamais.

Vous possédez dorénavant un bouton personnalisé qui réagit au passage de votre souris. Je sais qu'il y aura des « p'tits malins » qui cliqueront sur le bouton et relâcheront le clic en dehors du bouton : dans ce cas, le fond du bouton deviendra orange, puisque c'est ce qui doit être effectué vu la méthode `mouseReleased()`. Afin de pallier ce problème, nous allons vérifier que lorsque le clic est relâché, la souris se trouve toujours sur le bouton.

Nous avons implémenté l'interface `MouseListener` ; il reste cependant un objet que nous n'avons pas encore utilisé. Vous ne le voyez pas ? C'est le paramètre présent dans toutes les méthodes de cette interface : oui, c'est `MouseEvent` !

Cet objet nous permet d'obtenir beaucoup d'informations sur les événements. Nous ne détaillerons pas tout ici, mais nous verrons certains côtés pratiques de ce type d'objet tout au long de cette partie. Dans notre cas, nous pouvons récupérer les coordonnées *x* et *y* du curseur de la souris par rapport au `Bouton` grâce aux méthodes `getX()` et `getY()`. Cela signifie que si nous relâchons le clic en dehors de la zone où se trouve notre objet, la valeur retournée par la méthode `getY()` sera négative.

Voici le correctif de la méthode `mouseReleased()` de notre classe `Bouton` :

Code : Java

```java
      public void mouseReleased(MouseEvent event) {
        //Nous changeons le fond de notre image pour le orange lorsque
      nous relâchons le clic avec le fichier fondBoutonHover.png si la
      souris est toujours sur le bouton
        if((event.getY() > 0 && event.getY() < bouton.getHeight()) &&
      (event.getX() > 0 && event.getX() < bouton.getWidth())){
          try {
            img = ImageIO.read(new File("fondBoutonHover.png"));
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
        //Si on se trouve à l'extérieur, on dessine le fond par défaut
        else{
          try {
            img = ImageIO.read(new File("fondBouton.png"));
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
```

Vous verrez dans les chapitres qui suivent qu'il existe plusieurs interfaces pour les différentes actions possibles sur une IHM. Sachez qu'il existe aussi une convention pour ces interfaces : leur nom commence par le type de l'action, suivi du mot `Listener`. Nous avons étudié ici les actions de la souris, voyez le nom de l'interface : `MouseListener`.

Nous possédons à présent un bouton réactif, mais qui n'effectue rien pour le moment. Je vous propose de combler cette lacune.

Interagir avec son bouton

Déclencher une action : l'interface ActionListener

Afin de gérer les différentes actions à effectuer selon le bouton sur lequel on clique, nous allons utiliser l'interface `ActionListener`.

Nous n'allons pas implémenter cette interface dans notre classe `Bouton` mais dans notre classe `Fenetre`, le but étant de faire en sorte que lorsque l'on clique sur le bouton, il se passe quelque chose dans notre application : changer un état, une variable, effectuer une incrémentation… Enfin, n'importe quelle action !

Comme je vous l'ai expliqué, lorsque nous appliquons un `addMouseListener()`, nous informons l'objet observé qu'un autre objet doit être tenu au courant de l'événement. Ici, nous voulons que ce soit notre application (notre `Fenetre`) qui écoute notre `Bouton`, le but étant de pouvoir lancer ou arrêter l'animation dans le `Panneau`.

Avant d'en arriver là, nous allons faire plus simple : nous nous pencherons dans un premier temps sur l'implémentation de l'interface `ActionListener`. Afin de vous montrer toute la puissance de cette interface, nous utiliserons un nouvel objet issu du package `javax.swing` : le `JLabel`. Cet objet se comporte comme un libellé : il est spécialisé dans l'affichage de texte ou d'image. Il est donc idéal pour notre premier exemple !

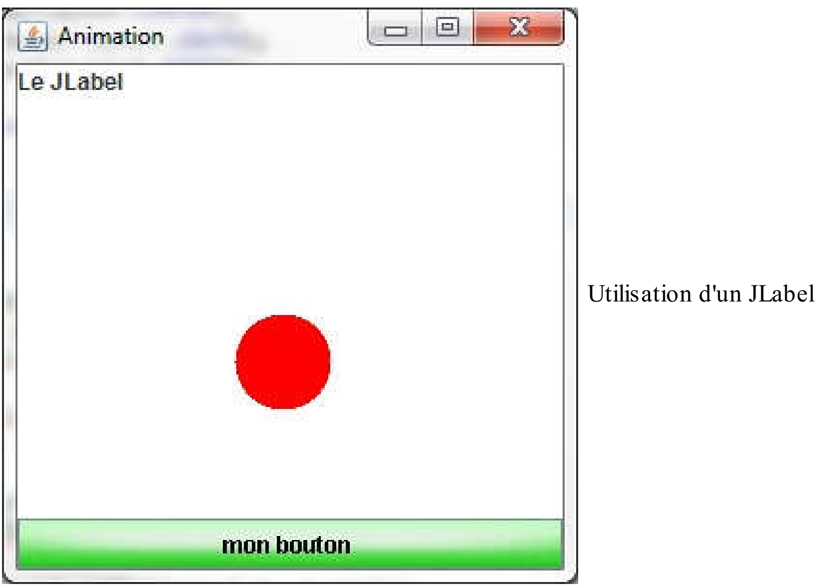
On l'instancie et l'initialise plus ou moins de la même manière que le `JButton` :

Code : Java

```java
JLabel label1 = new JLabel();
label1.setText("Mon premier JLabel");
//Ou encore
JLabel label2 = new JLabel("Mon deuxième JLabel");
```

Créez une variable d'instance de type `JLabel` (appelez-la `label`) et initialisez-la avec le texte qui vous plaît ; ajoutez-la ensuite à votre content pane en position `BorderLayout.NORTH`.

Le résultat se trouve en figure suivante.



Utilisation d'un JLabel

Voici le code correspondant :

Code : Java

```
public class Fenetre extends JFrame {
  private Panneau pan = new Panneau();
  private Bouton bouton = new Bouton("mon bouton");
  private JPanel container = new JPanel();
  private JLabel label = new JLabel("Le JLabel");

  public Fenetre(){
    this.setTitle("Animation");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    container.setBackground(Color.white);
    container.setLayout(new BorderLayout());
    container.add(pan, BorderLayout.CENTER);
    container.add(bouton, BorderLayout.SOUTH);
    container.add(label, BorderLayout.NORTH);

    this.setContentPane(container);
    this.setVisible(true);
    go();
  }
  //Le reste ne change pas
}
```

Vous pouvez voir que le texte de cet objet est aligné par défaut en haut à gauche. Il est possible de modifier quelques paramètres tels que :

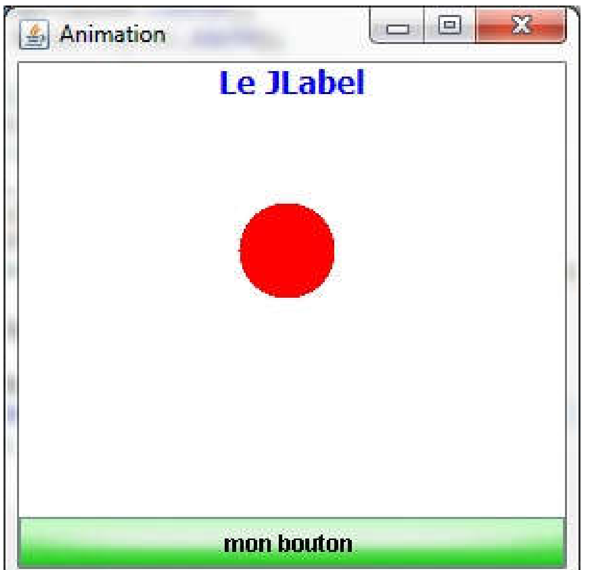
- l'alignement du texte ;
- la police à utiliser ;
- la couleur du texte ;
- d'autres paramètres.

Voici un code mettant tout cela en pratique :

Code : Java

```
public Fenetre(){
  this.setTitle("Animation");
  this.setSize(300, 300);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);

  container.setBackground(Color.white);
  container.setLayout(new BorderLayout());
  container.add(pan, BorderLayout.CENTER);
  container.add(bouton, BorderLayout.SOUTH);

  //Définition d'une police d'écriture
  Font police = new Font("Tahoma", Font.BOLD, 16);
  //On l'applique au JLabel
  label.setFont(police);
  //Changement de la couleur du texte
  label.setForeground(Color.blue);
  //On modifie l'alignement du texte grâce aux attributs statiques
  //de la classe JLabel
  label.setHorizontalAlignment(JLabel.CENTER);

  container.add(label, BorderLayout.NORTH);
  this.setContentPane(container);
  this.setVisible(true);
  go();
}
```

La figure suivante donne un aperçu de ce code.



Utilisation plus fine d'un JLabel

Maintenant que notre libellé se présente exactement sous la forme que nous voulons, nous pouvons implémenter l'interface `ActionListener`. Vous remarquerez que cette interface ne contient qu'une seule méthode !

Code : Java

```
//CTRL + SHIFT + O pour générer les imports
public class Fenetre extends JFrame implements ActionListener{
  private Panneau pan = new Panneau();
  private Bouton bouton = new Bouton("mon bouton");
  private JPanel container = new JPanel();
  private JLabel label = new JLabel("Le JLabel");

  public Fenetre(){
    //Ce morceau de code ne change pas
  }

  //Méthode qui sera appelée lors d'un clic sur le bouton
  public void actionPerformed(ActionEvent arg0) {

  }
}
```

Nous allons maintenant informer notre objet `Bouton` que notre objet `Fenetre` l'écoute. Vous l'avez deviné : ajoutons notre `Fenetre` à la liste des objets qui écoutent notre `Bouton` grâce à la méthode `addActionListener(ActionListener obj)` présente dans la classe `JButton`, donc utilisable avec la variable `bouton`. Ajoutons cette instruction dans le constructeur en passant **`this`** en paramètre (puisque c'est notre `Fenetre` qui écoute le `Bouton`).

Une fois l'opération effectuée, nous pouvons modifier le texte du `JLabel` avec la méthode `actionPerformed()`. Nous allons compter le nombre de fois que l'on a cliqué sur le bouton : ajoutons une variable d'instance de type `int` dans notre class et appelons-la `compteur`, puis dans la méthode `actionPerformed()`, incrémentons ce compteur et affichons son contenu dans notre libellé.

Voici le code de notre objet mis à jour :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
```

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame implements ActionListener{
  private Panneau pan = new Panneau();
  private Bouton bouton = new Bouton("mon bouton");
  private JPanel container = new JPanel();
  private JLabel label = new JLabel("Le JLabel");
  //Compteur de clics
  private int compteur = 0;

  public Fenetre(){
    this.setTitle("Animation");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    container.setBackground(Color.white);
    container.setLayout(new BorderLayout());
    container.add(pan, BorderLayout.CENTER);

    //Nous ajoutons notre fenêtre à la liste des auditeurs de notre
bouton
    bouton.addActionListener(this);

    container.add(bouton, BorderLayout.SOUTH);

    Font police = new Font("Tahoma", Font.BOLD, 16);
    label.setFont(police);
    label.setForeground(Color.blue);
    label.setHorizontalAlignment(JLabel.CENTER);
    container.add(label, BorderLayout.NORTH);
    this.setContentPane(container);
    this.setVisible(true);
    go();
  }

  private void go(){
    //Cette méthode ne change pas
  }

   public void actionPerformed(ActionEvent arg0) {
    //Lorsque l'on clique sur le bouton, on met à jour le JLabel
    this.compteur++;
    label.setText("Vous avez cliqué " + this.compteur + " fois");
  }
}
```

Voyez le résultat à la figure suivante.

Interaction avec le bouton

Et nous ne faisons que commencer… Eh oui, nous allons maintenant ajouter un deuxième bouton à notre `Fenetre`, à côté du premier (vous êtes libres d'utiliser la classe personnalisée ou un simple `JButton`). Pour ma part, j'utiliserai des boutons normaux ; en effet, dans notre classe personnalisée, la façon dont le libellé est écrit dans notre bouton n'est pas assez souple et l'affichage peut donc être décevant (dans certains cas, le libellé peut ne pas être centré)…

Bref, nous possédons à présent deux boutons écoutés par notre objet `Fenetre`.

Vous devez créer un deuxième `JPanel` qui contiendra nos deux boutons, puis l'insérer dans le content pane en position `BorderLayout.SOUTH`. Si vous tentez de positionner deux composants au même endroit grâce à un `BorderLayout`,seul le dernier composant ajouté apparaîtra : en effet, le composant occupe toute la place disponible dans un `BorderLayout` !

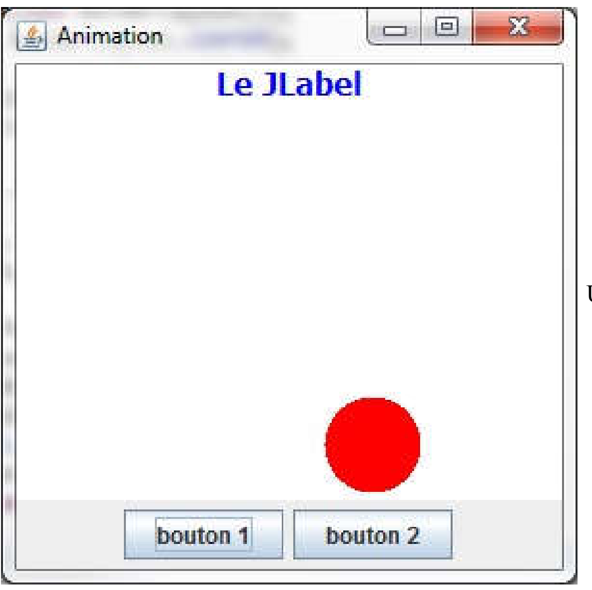
Voici notre nouveau code :

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame implements ActionListener{
  private Panneau pan = new Panneau();
  private JButton bouton = new JButton("bouton 1");
  private JButton bouton2 = new JButton("bouton 2");
  private JPanel container = new JPanel();
  private JLabel label = new JLabel("Le JLabel");
  private int compteur = 0;

  public Fenetre(){
    this.setTitle("Animation");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    container.setBackground(Color.white);
    container.setLayout(new BorderLayout());
    container.add(pan, BorderLayout.CENTER);
```

```
        bouton.addActionListener(this);
        bouton2.addActionListener(this);

        JPanel south = new JPanel();
        south.add(bouton);
        south.add(bouton2);
        container.add(south, BorderLayout.SOUTH);

        Font police = new Font("Tahoma", Font.BOLD, 16);
        label.setFont(police);
        label.setForeground(Color.blue);
        label.setHorizontalAlignment(JLabel.CENTER);
        container.add(label, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);
        go();
      }

      //…
    }
```

La figure suivante illustre le résultat que nous obtenons.



Un deuxième bouton dans la fenêtre

À présent, le problème est le suivant : comment effectuer deux actions différentes dans la méthode `actionPerformed()` ?

En effet, si nous laissons la méthode `actionPerformed()` telle quelle, les deux boutons exécutent la même action lorsqu'on les clique. Essayez, vous verrez le résultat.

Il existe un moyen de connaître l'élément ayant déclenché l'événement : il faut se servir de l'objet passé en paramètre dans la méthode `actionPerformed()`. Nous pouvons exploiter la méthode `getSource()` de cet objet pour connaître le nom de l'instance qui a généré l'événement. Testez la méthode `actionPerformed()` suivante et voyez si le résultat correspond à la figure suivante.

Code : Java

```
    public void actionPerformed(ActionEvent arg0) {
      if(arg0.getSource() == bouton)
        label.setText("Vous avez cliqué sur le bouton 1");

      if(arg0.getSource() == bouton2)
        label.setText("Vous avez cliqué sur le bouton 2");
```

```
        }
```

Détection de la source de l'événement

Notre code fonctionne à merveille ! Cependant, cette approche n'est pas très orientée objet : si notre IHM contient une multitude de boutons, la méthode `actionPerformed()` sera très chargée. Nous pourrions créer deux objets à part, chacun écoutant un bouton, dont le rôle serait de réagir de façon appropriée pour chaque bouton ; mais si nous avions besoin de modifier des données spécifiques à la classe contenant nos boutons, il faudrait ruser afin de parvenir à faire communiquer nos objets… Pas terrible non plus.

Parler avec sa classe intérieure

En Java, on peut créer ce que l'on appelle des **classes internes**. Cela consiste à déclarer une classe à l'intérieur d'une autre classe. Je sais, ça peut paraître tordu, mais vous allez bientôt constater que c'est très pratique.

En effet, les classes internes possèdent tous les avantages des classes normales, de l'héritage d'une superclasse à l'implémentation d'une interface. Elles bénéficient donc du polymorphisme et de la covariance des variables. En outre, elles ont l'avantage d'avoir accès aux attributs de la classe dans laquelle elles sont déclarées !

Dans le cas qui nous intéresse, cela permet de créer une implémentation de l'interface `ActionListener` détachée de notre classe `Fenetre`, mais pouvant utiliser ses attributs. La déclaration d'une telle classe se fait exactement de la même manière que pour une classe normale, si ce n'est qu'elle se trouve déjà dans une autre classe. Nous procédons donc comme ceci :

Code : Java

```java
public class MaClasseExterne{

  public MaClasseExterne(){
    //...
  }

  class MaClassInterne{
    public MaClassInterne(){
      //...
    }
  }
}
```

Grâce à cela, nous pourrons concevoir une classe spécialisée dans l'écoute des composants et qui effectuera un travail bien

déterminé. Dans notre exemple, nous créerons deux classes internes implémentant chacune l'interface `ActionListener` et redéfinissant la méthode `actionPerformed()` :

- `BoutonListener` écoutera le premier bouton ;
- `Bouton2Listener` écoutera le second.

Une fois ces opérations effectuées, il ne nous reste plus qu'à indiquer à chaque bouton « qui l'écoute » grâce à la méthode `addActionListener()`.

Voyez ci-dessous la classe `Fenetre` mise à jour.

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

  private Panneau pan = new Panneau();
  private JButton bouton = new JButton("bouton 1");
  private JButton bouton2 = new JButton("bouton 2");
  private JPanel container = new JPanel();
  private JLabel label = new JLabel("Le JLabel");
  private int compteur = 0;

  public Fenetre(){
    this.setTitle("Animation");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    container.setBackground(Color.white);
    container.setLayout(new BorderLayout());
    container.add(pan, BorderLayout.CENTER);

    //Ce sont maintenant nos classes internes qui écoutent nos
boutons
    bouton.addActionListener(new BoutonListener());
    bouton2.addActionListener(new Bouton2Listener());

    JPanel south = new JPanel();
    south.add(bouton);
    south.add(bouton2);
    container.add(south, BorderLayout.SOUTH);
    Font police = new Font("Tahoma", Font.BOLD, 16);
    label.setFont(police);
    label.setForeground(Color.blue);
    label.setHorizontalAlignment(JLabel.CENTER);
    container.add(label, BorderLayout.NORTH);
    this.setContentPane(container);
    this.setVisible(true);
    go();
  }

  private void go(){
    //Cette méthode ne change pas
  }

  //Classe écoutant notre premier bouton
  class BoutonListener implements ActionListener{
    //Redéfinition de la méthode actionPerformed()
```

```
    public void actionPerformed(ActionEvent arg0) {
      label.setText("Vous avez cliqué sur le bouton 1");
    }
  }

  //Classe écoutant notre second bouton
  class Bouton2Listener implements ActionListener{
    //Redéfinition de la méthode actionPerformed()
    public void actionPerformed(ActionEvent e) {
      label.setText("Vous avez cliqué sur le bouton 2");
    }
  }
}
```

Le résultat, visible à la figure suivante, est parfait.

Utilisation de deux actions sur deux boutons

Vous pouvez constater que nos classes internes ont même accès aux attributs déclarés **`private`** dans notre classe `Fenetre`.

Dorénavant, nous n'avons plus à nous soucier du bouton qui a déclenché l'événement, car nous disposons de deux classes écoutant chacune un bouton. Nous pouvons souffler un peu : une grosse épine vient de nous être retirée du pied.

Vous pouvez aussi faire écouter votre bouton par plusieurs classes. Il vous suffit d'ajouter ces classes supplémentaires à l'aide d'`addActionListener()`.

Eh oui, faites le test : créez une troisième classe interne et attribuez-lui le nom que vous voulez (personnellement, je l'ai appelée `Bouton3Listener`). Implémentez-y l'interface `ActionListener` et contentez-vous d'effectuer un simple `System.out.println()` dans la méthode `actionPerformed()`. N'oubliez pas de l'ajouter à la liste des classes qui écoutent votre bouton (n'importe lequel des deux ; j'ai pour ma part choisi le premier).

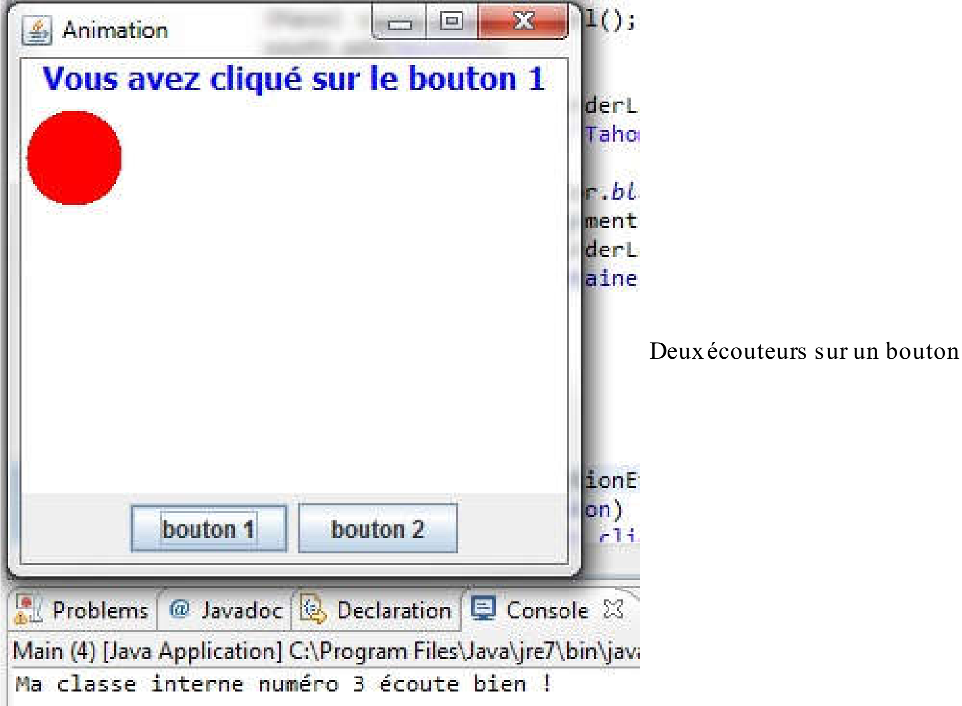
Je vous écris uniquement le code ajouté :

Code : Java

```
//Les imports…

public class Fenetre extends JFrame{
  //Les variables d'instance…
```

```
    public Fenetre(){
      this.setTitle("Animation");
      this.setSize(300, 300);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      this.setLocationRelativeTo(null);

      container.setBackground(Color.white);
      container.setLayout(new BorderLayout());
      container.add(pan, BorderLayout.CENTER);

      //Première classe écoutant mon premier bouton
      bouton.addActionListener(new BoutonListener());
      //Deuxième classe écoutant mon premier bouton
      bouton.addActionListener(new Bouton3Listener());

      bouton2.addActionListener(new Bouton2Listener());

      JPanel south = new JPanel();
      south.add(bouton);
      south.add(bouton2);
      container.add(south, BorderLayout.SOUTH);

      Font police = new Font("Tahoma", Font.BOLD, 16);
      label.setFont(police);
      label.setForeground(Color.blue);
      label.setHorizontalAlignment(JLabel.CENTER);
      container.add(label, BorderLayout.NORTH);
      this.setContentPane(container);
      this.setVisible(true);
      go();
    }

    //…

    class Bouton3Listener implements ActionListener{
      //Redéfinition de la méthode actionPerformed()
      public void actionPerformed(ActionEvent e) {
        System.out.println("Ma classe interne numéro 3 écoute bien
  !");
      }
    }
  }
```

Le résultat se trouve sur la figure suivante.



Deux écouteurs sur un bouton

Les classes internes sont vraiment des classes à part entière. Elles peuvent également hériter d'une superclasse. De ce fait, c'est presque comme si nous nous trouvions dans le cas d'un héritage multiple (ce n'en est pas un, même si cela y ressemble). Ce code est donc valide :

Code : Java

```
public class MaClasseExterne extends JFrame{

  public MaClasseExterne(){
    //...
  }

  class MaClassInterne extends JPanel{
    public MaClassInterne(){
      //…
    }
  }

  class MaClassInterne2 extends JButton{
    public MaClassInterne(){
      //…
    }
  }
}
```

Vous voyez bien que ce genre de classes peut s'avérer très utile.

Bon, nous avons réglé le problème d'implémentation : nous possédons deux boutons qui sont écoutés. Il ne nous reste plus qu'à lancer et arrêter notre animation à l'aide de ces boutons. Mais auparavant, nous allons étudier une autre manière d'implémenter des écouteurs et, par extension, des classes devant redéfinir les méthodes d'une classe abstraite ou d'une interface.

Les classes anonymes

Il n'y a rien de compliqué dans cette façon de procéder, mais je me rappelle avoir été déconcerté lorsque je l'ai rencontrée pour la première fois.

Les classes anonymes sont le plus souvent utilisées pour la gestion d'événements ponctuels, lorsque créer une classe pour un seul traitement est trop lourd. Rappelez-vous ce que j'ai utilisé pour définir le comportement de mes boutons lorsque je vous ai présenté l'objet `CardLayout` : c'étaient des classes anonymes. Pour rappel, voici ce que je vous avais amenés à coder :

Code : Java

```
JButton bouton = new JButton("Contenu suivant");
//Définition de l'action sur le bouton
bouton.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent event){
    //Action !
  }
});
```

L'une des particularités de cette méthode, c'est que l'écouteur n'écoutera que ce composant. Vous pouvez vérifier qu'il n'y se trouve aucune déclaration de classe et que nous instancions une interface par l'instruction `new ActionListener()`. Nous devons seulement redéfinir la méthode, que vous connaissez bien maintenant, dans un bloc d'instructions ; d'où les accolades après l'instanciation, comme le montre la figure suivante.

```
JButton bouton = new JButton("Contenu suivant");
bouton.addActionListener(new ActionListener()
  {
    public void actionPerformed(ActionEvent event){
        cl.next(content);
    }
  }
);
```

Découpage d'une classe anonyme

Pourquoi appelle-t-on cela une classe « anonyme » ?

C'est simple : procéder de cette manière revient à créer une classe fille sans être obligé de créer cette classe de façon explicite. L'héritage se produit automatiquement. En fait, le code ci-dessus revient à effectuer ceci :

Code : Java

```
class Fenetre extends JFrame{
  //…
  bouton.addActionListener(new ActionListenerBis());
  //…

  public class ActionListenerBis implements ActionListener{
    public void actionPerformed(ActionEvent event){
      //Action !
    }
  }
}
```

Seulement, la classe créée n'a pas de nom, l'héritage s'effectue de façon implicite ! Nous bénéficions donc de tous les avantages de la classe mère en ne redéfinissant que la méthode qui nous intéresse.

Sachez aussi que les classes anonymes peuvent être utilisées pour implémenter des classes abstraites. Je vous conseille d'effectuer de nouveaux tests en utilisant notre exemple du pattern strategy ; mais cette fois, plutôt que de créer des classes, créez des classes anonymes.

Les classes anonymes sont soumises aux mêmes règles que les classes « normales » :

- utilisation des méthodes non redéfinies de la classe mère ;
- obligation de redéfinir **toutes les méthodes** d'une interface ;
- obligation de redéfinir les méthodes abstraites d'une classe abstraite.

Cependant, ces classes possèdent des restrictions à cause de leur rôle et de leur raison d'être :

- elles ne peuvent pas être déclarées **abstract** ;
- elles ne peuvent pas non plus être déclarées **static** ;
- elles ne peuvent pas définir de constructeur ;
- elles sont automatiquement déclarées **final** : on ne peut dériver de cette classe, l'héritage est donc impossible !

Contrôler son animation : lancement et arrêt

Pour parvenir à gérer le lancement et l'arrêt de notre animation, nous allons devoir modifier un peu le code de notre classe `Fenetre`. Il va falloir changer le libellé des boutons de notre IHM : le premier affichera `Go` et le deuxième `Stop`. Pour éviter d'interrompre l'animation alors qu'elle n'est pas lancée et de l'animer quand elle l'est déjà, nous allons tantôt activer et désactiver les boutons. Je m'explique :

- au lancement, le bouton `Go` ne sera pas cliquable alors que le bouton `Stop` oui ;
- si l'animation est interrompue, le bouton `Stop` ne sera plus cliquable, mais le bouton `Go` le sera.

Ne vous inquiétez pas, c'est très simple à réaliser. Il existe une méthode gérant ces changements d'état :

Code : Java

```
JButton bouton = new JButton("bouton");
bouton.setEnabled(false); //Le bouton n'est plus cliquable
bouton.setEnabled(true);  //Le bouton est de nouveau cliquable
```

Ces objets permettent de réaliser pas mal de choses ; soyez curieux et testez-en les méthodes. Allez donc faire un tour sur le site d'Oracle : fouillez, fouinez…

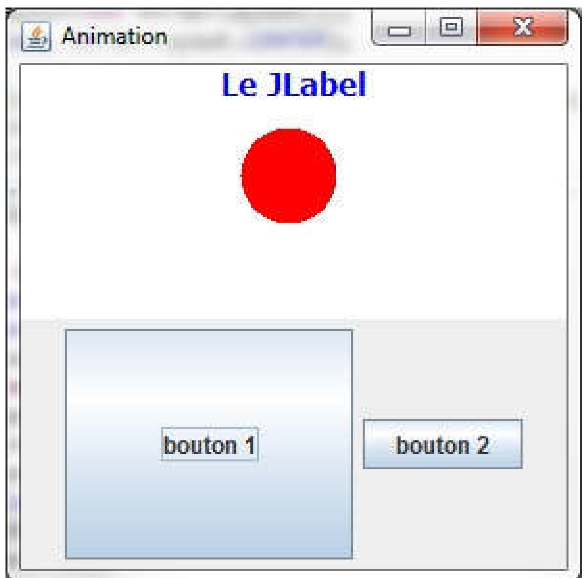
L'une de ces méthodes, qui s'avère souvent utile et est utilisable avec tous ces objets (ainsi qu'avec les objets que nous verrons par la suite), est la méthode de gestion de dimension. Il ne s'agit pas de la méthode `setSize()`, mais de la méthode `setPreferredSize()`. Elle prend en paramètre un objet `Dimension`, qui, lui, prend deux entiers comme arguments.

Voici un exemple :

Code : Java

```
bouton.setPreferredSize(new Dimension(150, 120));
```

En l'utilisant dans notre application, nous obtenons la figure suivante.



Gestion de la taille de nos boutons

Afin de bien gérer notre animation, nous devons améliorer notre méthode `go()`. Sortons donc de cette méthode les deux entiers dont nous nous servions afin de recalculer les coordonnées de notre rond. La boucle infinie doit dorénavant pouvoir être interrompue ! Pour réussir cela, nous allons déclarer un booléen qui changera d'état selon le bouton sur lequel on cliquera ; nous l'utiliserons comme paramètre de notre boucle.

Voyez ci-dessous le code de notre classe `Fenetre`.

Code : Java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame{

  private Panneau pan = new Panneau();
  private JButton bouton = new JButton("Go");
  private JButton bouton2 = new JButton("Stop");
  private JPanel container = new JPanel();
  private JLabel label = new JLabel("Le JLabel");
  private int compteur = 0;
  private boolean animated = true;
  private boolean backX, backY;
  private int x, y;

  public Fenetre(){
    this.setTitle("Animation");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    container.setBackground(Color.white);
    container.setLayout(new BorderLayout());
    container.add(pan, BorderLayout.CENTER);
    bouton.addActionListener(new BoutonListener());
    bouton.setEnabled(false);
    bouton2.addActionListener(new Bouton2Listener());

    JPanel south = new JPanel();
```

```
        south.add(bouton);
        south.add(bouton2);
        container.add(south, BorderLayout.SOUTH);
        Font police = new Font("Tahoma", Font.BOLD, 16);
        label.setFont(police);
        label.setForeground(Color.blue);
        label.setHorizontalAlignment(JLabel.CENTER);
        container.add(label, BorderLayout.NORTH);
        this.setContentPane(container);
        this.setVisible(true);
        go();
      }

      private void go(){
        //Les coordonnées de départ de notre rond
        x = pan.getPosX();
        y = pan.getPosY();
        //Dans cet exemple, j'utilise une boucle while
        //Vous verrez qu'elle fonctionne très bien
        while(this.animated){
          if(x < 1)backX = false;
          if(x > pan.getWidth()-50)backX = true;
          if(y < 1)backY = false;
          if(y > pan.getHeight()-50)backY = true;
          if(!backX)pan.setPosX(++x);
          else pan.setPosX(--x);
          if(!backY) pan.setPosY(++y);
          else pan.setPosY(--y);
          pan.repaint();

          try {
            Thread.sleep(3);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }

      class BoutonListener implements ActionListener{
        public void actionPerformed(ActionEvent arg0) {
          animated = true;
          bouton.setEnabled(false);
          bouton2.setEnabled(true);
          go();
        }
      }

      class Bouton2Listener implements ActionListener{
         public void actionPerformed(ActionEvent e) {
          animated = false;
          bouton.setEnabled(true);
          bouton2.setEnabled(false);
        }
      }
    }
```

À l'exécution, vous remarquez que :

- le bouton `Go` n'est pas cliquable et l'autre l'est ;
- l'animation se lance ;
- l'animation s'arrête lorsque l'on clique sur le bouton `Stop` ;
- le bouton `Go` devient alors cliquable ;
- lorsque vous cliquez dessus, l'animation ne se relance pas !

Comment est-ce possible ?

Comme je l'ai expliqué dans le chapitre traitant des conteneurs, un thread est lancé au démarrage de notre application : c'est le processus principal du programme. Au démarrage, l'animation est donc lancée dans le même thread que notre objet `Fenetre`. Lorsque nous lui demandons de s'arrêter, aucun problème : les ressources mémoire sont libérées, on sort de la boucle infinie et l'application continue à fonctionner.

Mais lorsque nous redemandons à l'animation de se lancer, l'instruction se trouvant dans la méthode `actionPerformed()` appelle la méthode `go()` et, étant donné que nous nous trouvons à l'intérieur d'une boucle infinie, nous restons dans la méthode `go()` et ne sortons plus de la méthode `actionPerformed()`.

Explication de ce phénomène

Java gère les appels aux méthodes grâce à ce que l'on appelle vulgairement **la pile**.

Pour expliquer cela, prenons un exemple tout bête ; regardez cet objet :

Code : Java

```java
public class TestPile {
  public TestPile(){
    System.out.println("Début constructeur");
    methode1();
    System.out.println("Fin constructeur");
  }

  public void methode1(){
      System.out.println("Début méthode 1");
      methode2();
      System.out.println("Fin méthode 1");
   }

   public void methode2(){
    System.out.println("Début méthode 2");
    methode3();
    System.out.println("Fin méthode 2");
  }

  public void methode3(){
    System.out.println("Début méthode 3");
    System.out.println("Fin méthode 3");
  }
}
```

Si vous instanciez cet objet, vous obtenez dans la console la figure suivante.

```
Problems  @ Javadoc  Declaration  Co
<terminated> Test (3) [Java Application] C:\Prograr
Début constructeur
Début méthode 1
Début méthode 2
Début méthode 3
Fin méthode 3
Fin méthode 2
Fin méthode 1
Fin constructeur
```

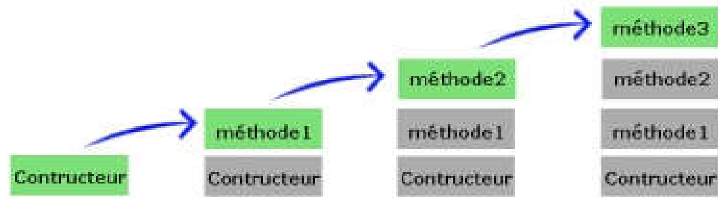
Exemple de pile d'invocations

Je suppose que vous avez remarqué avec stupéfaction que l'ordre des instructions est un peu bizarre. Voici ce qu'il se passe :

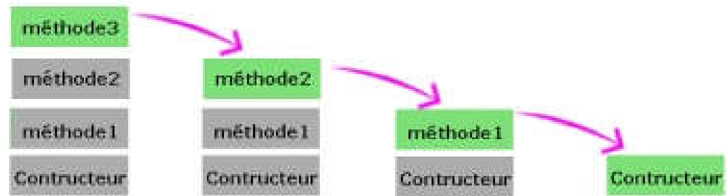
- à l'instanciation, notre objet appelle la méthode 1 ;
- cette dernière invoque la méthode 2 ;
- celle-ci utilise la méthode 3 : une fois qu'elle a terminé, la JVM retourne dans la méthode 2 ;
- lorsqu'elle a fini de s'exécuter, on remonte à la fin de la méthode 1, jusqu'à la dernière instruction appelante : le constructeur.

Lors de tous les appels, on dit que la JVM **empile** les invocations sur la pile. Une fois que la dernière méthode empilée a terminé de s'exécuter, la JVM la *dépile*.

La figure suivante présente un schéma résumant la situation.



Empilage et dépilage de méthodes



Dans notre programme, imaginez que la méthode `actionPerformed()` soit représentée par la méthode 2, et que notre méthode `go()` soit représentée par la méthode 3. Lorsque nous entrons dans la méthode 3, nous entrons dans une boucle infinie… Conséquence directe : nous ne ressortons jamais de cette méthode et la JVM ne dépile plus !

Afin de pallier ce problème, nous allons utiliser un nouveau thread. Grâce à cela, la méthode `go()` se trouvera dans une pile à part.

Attends : on arrive pourtant à arrêter l'animation alors qu'elle se trouve dans une boucle infinie. Pourquoi ?

Tout simplement parce que nous ne demandons d'effectuer qu'une simple initialisation de variable dans la gestion de notre événement ! Si vous créez une deuxième méthode comprenant une boucle infinie et que vous l'invoquez lors du clic sur le bouton `Stop`, vous aurez exactement le même problème.

Je ne vais pas m'éterniser là-dessus, nous verrons cela dans un prochain chapitre. À présent, je pense qu'il est de bon ton de vous parler du mécanisme d'écoute d'événements, le fameux pattern observer.

Être à l'écoute de ses objets : le design pattern Observer

Le design pattern `Observer` est utilisé pour gérer les événements de vos IHM. C'est une technique de programmation. La connaître n'est pas absolument indispensable, mais cela vous aide à mieux comprendre le fonctionnement de `Swing` et `AWT`. C'est par ce biais que vos composants effectueront quelque chose lorsque vous les cliquerez ou les survolerez.

Je vous propose de découvrir son fonctionnement à l'aide d'une situation problématique.

Posons le problème

Sachant que vous êtes des développeurs Java chevronnés, un de vos amis proches vous demande si vous êtes en mesure de

l'aider à réaliser une horloge digitale en Java. Il a en outre la gentillesse de vous fournir les classes à utiliser pour la création de son horloge. Votre ami a l'air de s'y connaître, car ce qu'il vous a fourni est bien structuré.

Package com.sdz.vue, classe Fenetre.java

Code : Java

```
package com.sdz.vue;

import java.awt.BorderLayout;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.JLabel;

import com.sdz.model.Horloge;

public class Fenetre extends JFrame{
  private JLabel label = new JLabel();
  private Horloge horloge;

  public Fenetre(){
    //On initialise la JFrame
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    this.setResizable(false);
    this.setSize(200, 80);
    //On initialise l'horloge
    this.horloge = new Horloge();
    //On initialise le JLabel
    Font police = new Font("DS-digital", Font.TYPE1_FONT, 30);
    this.label.setFont(police);
    this.label.setHorizontalAlignment(JLabel.CENTER);
    //On ajoute le JLabel à la JFrame
    this.getContentPane().add(this.label, BorderLayout.CENTER);
  }

  //Méthode main() lançant le programme
  public static void main(String[] args){
    Fenetre fen = new Fenetre();
    fen.setVisible(true);
  }
}
```

Package com.sdz.model, classe Horloge.java

Code : Java

```
package com.sdz.model;

import java.util.Calendar;

public class Horloge{
  //Objet calendrier pour récupérer l'heure courante
  private Calendar cal;
  private String hour = "";

  public void run() {
    while(true){
    //On récupère l'instance d'un calendrier à chaque tour
    //Elle va nous permettre de récupérer l'heure actuelle
    this.cal = Calendar.getInstance();
    this.hour =  //Les heures
      this.cal.get(Calendar.HOUR_OF_DAY) + " : "
      +
      (    //Les minutes
```

```
                this.cal.get(Calendar.MINUTE) < 10
                ? "0" + this.cal.get(Calendar.MINUTE)
                : this.cal.get(Calendar.MINUTE)
              )
              + " : "
              +
              (    //Les secondes
                (this.cal.get(Calendar.SECOND)< 10)
                ? "0"+this.cal.get(Calendar.SECOND)
                : this.cal.get(Calendar.SECOND)
              );
              try {
                Thread.sleep(1000);
              } catch (InterruptedException e) {
                e.printStackTrace();
              }
            }
          }
        }
```

Si vous ne disposez pas de la police d'écriture que j'ai utilisée, utilisez-en une autre : `Arial` ou `Courrier`, par exemple.

Le problème auquel votre ami est confronté est simple : il est impossible de faire communiquer l'horloge avec la fenêtre.

Je ne vois pas où est le problème : il n'a qu'à passer son instance de `JLabel` dans son objet `Horloge`, et le tour est joué !

En réalité, votre ami, dans son infinie sagesse, souhaite - je le cite - que l'horloge ne dépende pas de son interface graphique, juste au cas où il devrait passer d'une IHM `swing` à une IHM `awt`.

Il est vrai que si l'on passe l'objet d'affichage dans l'horloge, dans le cas où l'on change le type de l'IHM, toutes les classes doivent être modifiées ; ce n'est pas génial. En fait, lorsque vous procédez de la sorte, on dit que vous couplez des objets : vous rendez un ou plusieurs objets dépendants d'un ou de plusieurs autres objets (entendez par là que vous ne pourrez plus utiliser les objets couplés indépendamment des objets auxquels ils sont attachés).

Le couplage entre objets est l'un des problèmes principaux relatifs à la réutilisation des objets. Dans notre cas, si vous utilisez l'objet `Horloge` dans une autre application, vous serez confrontés à plusieurs problèmes étant donné que cet objet ne s'affiche que dans un `JLabel`.

C'est là que le pattern observer entre en jeu : il fait communiquer des objets entre eux sans qu'ils se connaissent réellement ! Vous devez être curieux de voir comment il fonctionne, je vous propose donc de l'étudier sans plus tarder.

Des objets qui parlent et qui écoutent : le pattern observer

Faisons le point sur ce que vous savez de ce pattern pour le moment :

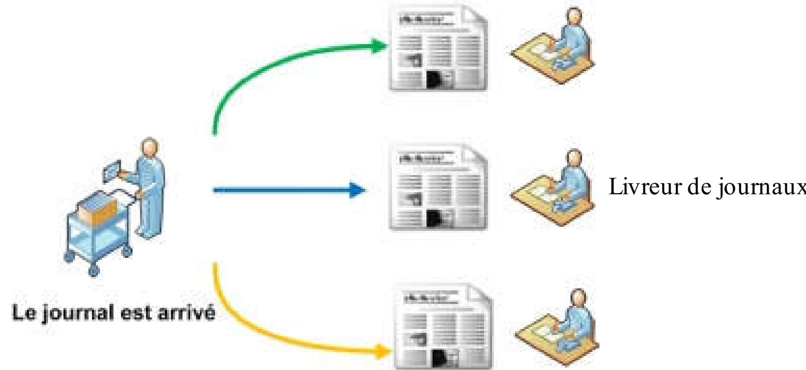
- il fait communiquer des objets entre eux ;
- c'est un bon moyen d'éviter le couplage d'objets.

Ce sont deux points cruciaux, mais un autre élément, que vous ne connaissez pas encore, va vous plaire : tout se fait automatiquement !

Comment les choses vont-elles alors se passer ? Réfléchissons à ce que nous voulons que notre horloge digitale effectue : elle doit pouvoir avertir l'objet servant à afficher l'heure lorsqu'il doit rafraîchir son affichage. Puisque les horloges du monde entier se mettent à jour toutes les secondes, il n'y a aucune raison pour que la nôtre ne fasse pas de même.

Ce qui est merveilleux avec ce pattern, c'est que notre horloge ne se contentera pas d'avertir un seul objet que sa valeur a changé : elle pourra en effet mettre plusieurs observateurs au courant !

En fait, pour faire une analogie, interprétez la relation entre les objets implémentant le pattern observer comme un éditeur de journal et ses clients (voir figure suivante).



Livreur de journaux

Grâce à ce schéma, vous pouvez sentir que notre objet défini comme observable pourra être surveillé par plusieurs objets : il s'agit d'une relation dite de un à plusieurs vers l'objet `Observateur`. Avant de vous expliquer plus en détail le fonctionnement de ce pattern, jetez un œil au diagramme de classes de notre application en figure suivante.

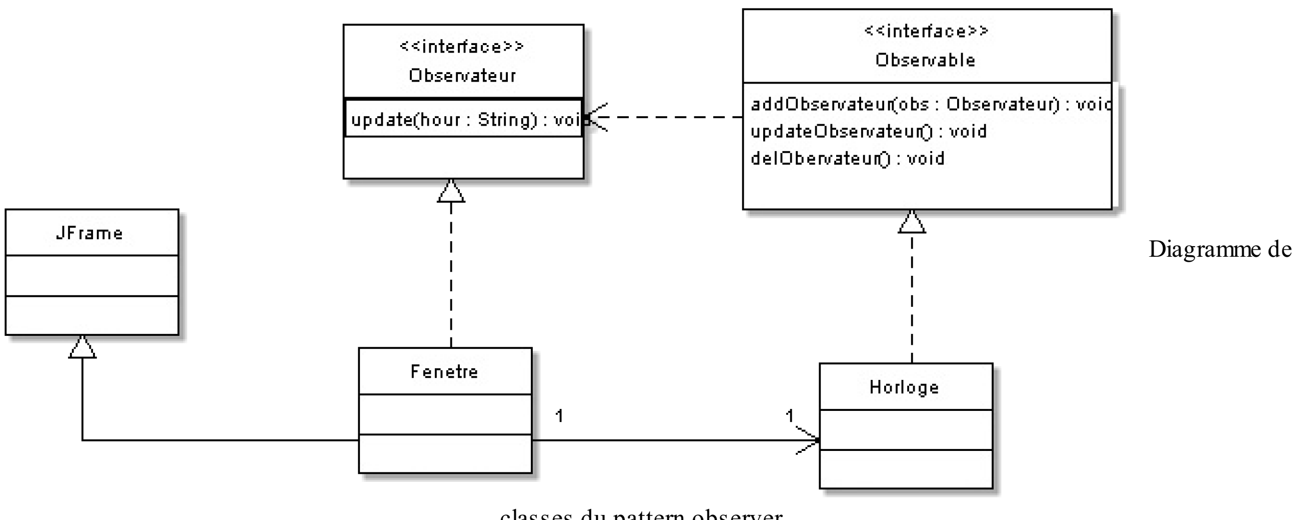


Diagramme de classes du pattern observer

Ce diagramme indique que ce ne sont pas les instances d'`Horloge` ou de `JLabel` que nous allons utiliser, mais des implémentations d'interfaces.

En effet, vous savez que les classes implémentant une interface peuvent être définies par le type de l'interface. Dans notre cas, la classe `Fenetre` implémentera l'interface `Observateur` : nous pourrons la voir comme une classe du type `Observateur`. Vous avez sans doute remarqué que la deuxième interface - celle dédiée à l'objet `Horloge` - possède trois méthodes :

- une permettant d'ajouter des observateurs (nous allons donc gérer une collection d'observateurs) ;
- une permettant de retirer les observateurs ;
- enfin, une mettant à jour les observateurs.

Grâce à cela, nos objets ne sont plus liés par leurs types, mais par leurs interfaces ! L'interface qui apportera les méthodes de mise à jour, d'ajout d'observateurs, etc. travaillera donc avec des objets de type `Observateur`.

Ainsi, le couplage ne s'effectue plus directement, il s'opère par le biais de ces interfaces. Ici, il faut que nos deux interfaces soient couplées pour que le système fonctionne. De même que, lorsque je vous ai présenté le pattern decorator, nos classes étaient très fortement couplées puisqu'elles devaient travailler ensemble : nous devions alors faire en sorte de ne pas les séparer.

Voici comment fonctionnera l'application :

- nous instancierons la classe `Horloge` dans notre classe `Fenetre` ;

- cette dernière implémentera l'interface `Observateur` ;
- notre objet `Horloge`, implémentant l'interface `Observable`, préviendra les objets spécifiés de ses changements ;
- nous informerons l'horloge que notre fenêtre l'observe ;
- à partir de là, notre objet `Horloge` fera le reste : à chaque changement, nous appellerons la méthode mettant tous les observateurs à jour.

Le code source de ces interfaces se trouve ci-dessous (notez que j'ai créé un package `com.sdz.observer`).

Observateur.java

Code : Java

```
package com.sdz.observer;

public interface Observateur {
  public void update(String hour);
}
```

Observer.java

Code : Java

```
package com.sdz.observer;

public interface Observable {
  public void addObservateur(Observateur obs);
  public void updateObservateur();
  public void delObservateur();
}
```

Voici maintenant le code de nos deux classes, travaillant ensemble mais n'étant que faiblement couplées.

Horloge.java

Code : Java

```
package com.sdz.model;

import java.util.ArrayList;
import java.util.Calendar;

import com.sdz.observer.Observable;
import com.sdz.observer.Observateur;

public class Horloge implements Observable{
  //On récupère l'instance d'un calendrier
  //Elle va nous permettre de récupérer l'heure actuelle
  private Calendar cal;
  private String hour = "";
  //Notre collection d'observateurs
  private ArrayList<Observateur> listObservateur = new
ArrayList<Observateur>();

  public void run() {
    while(true){
      this.cal = Calendar.getInstance();
      this.hour =  //Les heures
        this.cal.get(Calendar.HOUR_OF_DAY) + " : "
        +
        (      //Les minutes
```

```
          this.cal.get(Calendar.MINUTE) < 10
          ? "0" + this.cal.get(Calendar.MINUTE)
          : this.cal.get(Calendar.MINUTE)
        )
        + " : "
        +
        (       //Les secondes
          (this.cal.get(Calendar.SECOND)< 10)
          ? "0"+this.cal.get(Calendar.SECOND)
          : this.cal.get(Calendar.SECOND)
        );
      //On avertit les observateurs que l'heure a été mise à jour
      this.updateObservateur();

      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

  //Ajoute un observateur à la liste
  public void addObservateur(Observateur obs) {
    this.listObservateur.add(obs);
  }
  //Retire tous les observateurs de la liste
  public void delObservateur() {
    this.listObservateur = new ArrayList<Observateur>();
  }
  //Avertit les observateurs que l'objet observable a changé
  //et invoque la méthode update() de chaque observateur
  public void updateObservateur() {
    for(Observateur obs : this.listObservateur )
      obs.update(this.hour);
  }
}
```

Fenetre.java

Code : Java

```
package com.sdz.vue;

import java.awt.BorderLayout;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.JLabel;

import com.sdz.model.Horloge;
import com.sdz.observer.Observateur;

public class Fenetre extends JFrame {
  private JLabel label = new JLabel();
  private Horloge horloge;

  public Fenetre(){
    //On initialise la JFrame
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    this.setResizable(false);
    this.setSize(200, 80);

    //On initialise l'horloge
    this.horloge = new Horloge();
    //On place un écouteur sur l'horloge
```

```
        this.horloge.addObservateur(new Observateur(){
          public void update(String hour) {
            label.setText(hour);
          }
        });

        //On initialise le JLabel
        Font police = new Font("DS-digital", Font.TYPE1_FONT, 30);
        this.label.setFont(police);
        this.label.setHorizontalAlignment(JLabel.CENTER);
        //On ajoute le JLabel à la JFrame
        this.getContentPane().add(this.label, BorderLayout.CENTER);
        this.setVisible(true);
        this.horloge.run();
      }

      //Méthode main() lançant le programme
      public static void main(String[] args){
        Fenetre fen = new Fenetre();
      }
    }
```

Exécutez ce code, vous verrez que tout fonctionne à merveille. Vous venez de permettre à deux objets de communiquer en n'utilisant presque aucun couplage : félicitations !

Vous pouvez voir que lorsque l'heure change, la méthode `updateObservateur()` est invoquée. Celle-ci parcourt la collection d'objets `Observateur` et appelle sa méthode `update(String hour)`. La méthode étant redéfinie dans notre classe `Fenetre` afin de mettre `JLabel` à jour, l'heure s'affiche !

J'ai mentionné que ce pattern est utilisé dans la gestion événementielle d'interfaces graphiques. Vous avez en outre remarqué que leurs syntaxes sont identiques. En revanche, je vous ai caché quelque chose : il existe des classes Java permettant d'utiliser le pattern observer sans avoir à coder les interfaces.

Le pattern observer : le retour

En réalité, il existe une classe abstraite `Observable` et une interface `Observer` fournies dans les classes Java.

Celles-ci fonctionnent de manière quasiment identique à notre façon de procéder, seuls quelques détails diffèrent. Personnellement, je préfère de loin utiliser un pattern observer « fait maison ».

Pourquoi cela ? Tout simplement parce que l'objet que l'on souhaite observer **doit** hériter de la classe `Observable`. Par conséquent, il ne pourra plus hériter d'une autre classe étant donné que Java ne gère pas l'héritage multiple. La figure suivante présente la hiérarchie de classes du pattern observer présent dans Java.

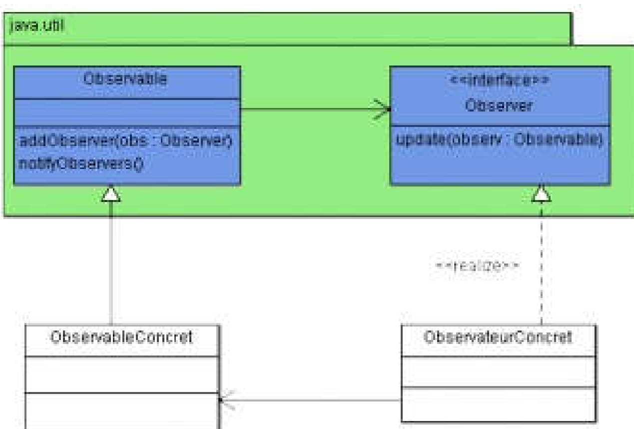


Diagramme de classes du pattern observer de Java

Vous remarquez qu'il fonctionne presque de la même manière que celui que nous avons développé. Il y a toutefois une différence dans la méthode `update(Observable obs, Object obj)` : sa signature a changé.
Cette méthode prend ainsi deux paramètres :

- un objet `Observable` ;
- un `Object` représentant une donnée supplémentaire que vous souhaitez lui fournir.

Vous connaissez le fonctionnement de ce pattern, il vous est donc facile de comprendre le schéma. Je vous invite cependant à effectuer vos propres recherches sur son implémentation dans Java : vous verrez qu'il existe des subtilités (rien de méchant, cela dit).

Cadeau : un bouton personnalisé optimisé

Terminons par une version améliorée de notre bouton qui reprend ce que nous avons appris :

Code : Java

```java
import java.awt.Color;
import java.awt.FontMetrics;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JButton;

public class Bouton extends JButton implements MouseListener{
  private String name;
  private Image img;

  public Bouton(String str){
    super(str);
    this.name = str;
    try {
      img = ImageIO.read(new File("fondBouton.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
    this.addMouseListener(this);
  }

  public void paintComponent(Graphics g){
```

```
    Graphics2D g2d = (Graphics2D)g;
    GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20,
Color.cyan, true);
    g2d.setPaint(gp);
    g2d.drawImage(img, 0, 0, this.getWidth(), this.getHeight(),
this);
    g2d.setColor(Color.black);

    //Objet permettant de connaître les propriétés d'une police,
dont la taille
    FontMetrics fm = g2d.getFontMetrics();
    //Hauteur de la police d'écriture
    int height = fm.getHeight();
    //Largeur totale de la chaîne passée en paramètre
    int width = fm.stringWidth(this.name);

    //On calcule alors la position du texte, et le tour est joué
    g2d.drawString(this.name, this.getWidth() / 2 - (width / 2),
(this.getHeight() / 2) + (height / 4));
  }

  public void mouseClicked(MouseEvent event) {
    //Inutile d'utiliser cette méthode ici
  }

  public void mouseEntered(MouseEvent event) {
    //Nous changeons le fond de notre image pour le jaune lors du
survol, avec le fichier fondBoutonHover.png
    try {
      img = ImageIO.read(new File("fondBoutonHover.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void mouseExited(MouseEvent event) {
    //Nous changeons le fond de notre image pour le vert lorsque
nous quittons le bouton, avec le fichier fondBouton.png
    try {
      img = ImageIO.read(new File("fondBouton.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void mousePressed(MouseEvent event) {
    //Nous changeons le fond de notre image pour le jaune lors du
clic gauche, avec le fichier fondBoutonClic.png
    try {
      img = ImageIO.read(new File("fondBoutonClic.png"));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void mouseReleased(MouseEvent event) {
    //Nous changeons le fond de notre image pour l'orange lorsque
nous relâchons le clic avec le fichier fondBoutonHover.png si la
souris est toujours sur le bouton
    if((event.getY() > 0 && event.getY() < this.getHeight()) &&
(event.getX() > 0 && event.getX() < this.getWidth())){
      try {
        img = ImageIO.read(new File("fondBoutonHover.png"));
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    //Si on se trouve à l'extérieur, on dessine le fond par défaut
    else{
      try {
```

```
                img = ImageIO.read(new File("fondBouton.png"));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Essayez, vous verrez que cette application fonctionne correctement.

- Vous pouvez interagir avec un composant grâce à votre souris en implémentant l'interface `MouseListener`.
- Lorsque vous implémentez une interface `< … >Listener`, vous indiquez à votre classe qu'elle doit se préparer à observer des événements du type de l'interface. Vous devez donc spécifier qui doit observer et ce que la classe doit observer grâce aux méthodes de type `add< … >Listener(< … >Listener)`.
- L'interface utilisée dans ce chapitre est `ActionListener` issue du package `java.awt`.
- La méthode à implémenter de cette interface est `actionPerformed()`.
- Une classe interne est une classe se trouvant à l'intérieur d'une classe.
- Une telle classe a accès à toutes les données et méthodes de sa classe externe.
- La JVM traite les méthodes appelées en utilisant une pile qui définit leur ordre d'exécution.
- Une méthode est empilée à son invocation, mais n'est dépilée que lorsque toutes ses instructions ont fini de s'exécuter.
- Le pattern observer permet d'utiliser des objets faiblement couplés. Grâce à ce pattern, les objets restent indépendants.

Ce PDF vous est offert par Accenture
http://careers.accenture.com/fr-fr

TP : une calculatrice

Ah ! Ça faisait longtemps… Un petit TP ! Dans celui-ci, nous allons - enfin, vous allez - pouvoir réviser tout ce qui a été vu au cours de cette partie :

- les fenêtres ;
- les conteneurs ;
- les boutons ;
- les interactions ;
- les classes internes ;

L'objectif est ici de réaliser une petite calculatrice basique.

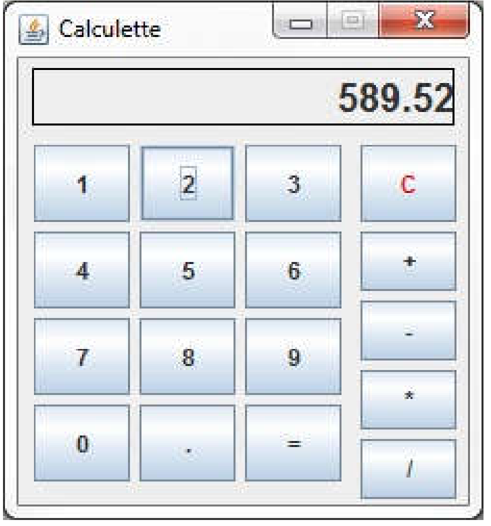
Élaboration

Nous allons tout de suite voir ce dont notre calculatrice devra être capable :

- Effectuer un calcul simple : 12 + 3 par exemple.
- Faire des calculs à la chaîne, par exemple : 1 + 2 + ... ; lorsqu'on clique à nouveau sur un opérateur, il faut afficher le résultat du calcul précédent.
- Donner la possibilité de tout recommencer à zéro.
- Gérer l'exception d'une division par 0 !

Conception

Vous devriez obtenir à peu de choses près la figure suivante.



Notre calculatrice

Voyons maintenant ce dont nous avons besoin pour parvenir à nos fins :

- autant de boutons qu'il en faut ;
- autant de conteneurs que nécessaire ;
- un `JLabel` pour l'affichage ;
- un booléen pour savoir si un opérateur a été sélectionné ;
- un booléen pour savoir si nous devons effacer ce qui figure à l'écran et écrire un nouveau nombre ;
- nous allons utiliser une variable de type `double` pour nos calculs ;
- il va nous falloir des classes internes qui implémenteront l'interface `ActionListener` ;
- et c'est à peu près tout.

Pour alléger le nombre de classes internes, vous pouvez en créer une qui se chargera d'écrire ce qui doit être affiché à l'écran. Utilisez la méthode `getSource()` pour savoir sur quel bouton on a cliqué.

Je ne vais pas tout vous dire, il faut que vous cherchiez par vous-mêmes : la réflexion est très importante ! En revanche, vous devez savoir que la correction que je vous fournis n'est pas **la** correction. Il y a plusieurs solutions possibles. Je vous propose seulement l'une d'elles.

Allez, au boulot !

Correction

Vous avez bien réfléchi ? Vous vous êtes brûlé quelques neurones ? Vous avez mérité votre correction ! Regardez bien comment tout interagit, et vous comprendrez comment fonctionne ce code.

Code : Java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Calculatrice extends JFrame {
  private JPanel container = new JPanel();
  //Tableau stockant les éléments à afficher dans la calculatrice
  String[] tab_string = {"1", "2", "3", "4", "5", "6", "7", "8",
"9", "0", ".", "=", "C", "+", "-", "*", "/"};
  //Un bouton par élément à afficher
  JButton[] tab_button = new JButton[tab_string.length];
  private JLabel ecran = new JLabel();
  private Dimension dim = new Dimension(50, 40);
  private Dimension dim2 = new Dimension(50, 31);
  private double chiffre1;
  private boolean clicOperateur = false, update = false;
  private String operateur = "";

  public Calculatrice(){
    this.setSize(240, 260);
    this.setTitle("Calculette");
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    this.setResizable(false);
    //On initialise le conteneur avec tous les composants
   initComposant();
   //On ajoute le conteneur
    this.setContentPane(container);
    this.setVisible(true);
  }

  private void initComposant(){
    //On définit la police d'écriture à utiliser
    Font police = new Font("Arial", Font.BOLD, 20);
    ecran = new JLabel("0");
    ecran.setFont(police);
    //On aligne les informations à droite dans le JLabel
    ecran.setHorizontalAlignment(JLabel.RIGHT);
    ecran.setPreferredSize(new Dimension(220, 20));
    JPanel operateur = new JPanel();
    operateur.setPreferredSize(new Dimension(55, 225));
    JPanel chiffre = new JPanel();
    chiffre.setPreferredSize(new Dimension(165, 225));
    JPanel panEcran = new JPanel();
    panEcran.setPreferredSize(new Dimension(220, 30));

    //On parcourt le tableau initialisé
    //afin de créer nos boutons
    for(int i = 0; i < tab_string.length; i++){
      tab_button[i] = new JButton(tab_string[i]);
      tab_button[i].setPreferredSize(dim);
      switch(i){
        //Pour chaque élément situé à la fin du tableau
        //et qui n'est pas un chiffre
```

```
          //on définit le comportement à avoir grâce à un listener
          case 11 :
            tab_button[i].addActionListener(new EgalListener());
            chiffre.add(tab_button[i]);
            break;
          case 12 :
            tab_button[i].setForeground(Color.red);
            tab_button[i].addActionListener(new ResetListener());
            operateur.add(tab_button[i]);
            break;
          case 13 :
            tab_button[i].addActionListener(new PlusListener());
            tab_button[i].setPreferredSize(dim2);
            operateur.add(tab_button[i]);
            break;
          case 14 :
            tab_button[i].addActionListener(new MoinsListener());
            tab_button[i].setPreferredSize(dim2);
            operateur.add(tab_button[i]);
            break;
          case 15 :
            tab_button[i].addActionListener(new MultiListener());
            tab_button[i].setPreferredSize(dim2);
            operateur.add(tab_button[i]);
            break;
          case 16 :
            tab_button[i].addActionListener(new DivListener());
            tab_button[i].setPreferredSize(dim2);
            operateur.add(tab_button[i]);
            break;
          default :
            //Par défaut, ce sont les premiers éléments du tableau
            //donc des chiffres, on affecte alors le bon listener
            chiffre.add(tab_button[i]);
            tab_button[i].addActionListener(new ChiffreListener());
            break;
        }
      }
      panEcran.add(ecran);
      panEcran.setBorder(BorderFactory.createLineBorder(Color.black));
      container.add(panEcran, BorderLayout.NORTH);
      container.add(chiffre, BorderLayout.CENTER);
      container.add(operateur, BorderLayout.EAST);
    }

    //Méthode permettant d'effectuer un calcul selon l'opérateur
  sélectionné
    private void calcul(){
      if(operateur.equals("+")){
        chiffre1 = chiffre1 +
              Double.valueOf(ecran.getText()).doubleValue();
        ecran.setText(String.valueOf(chiffre1));
      }
      if(operateur.equals("-")){
        chiffre1 = chiffre1 -
              Double.valueOf(ecran.getText()).doubleValue();
        ecran.setText(String.valueOf(chiffre1));
      }
      if(operateur.equals("*")){
        chiffre1 = chiffre1 *
              Double.valueOf(ecran.getText()).doubleValue();
        ecran.setText(String.valueOf(chiffre1));
      }
      if(operateur.equals("/")){
        try{
          chiffre1 = chiffre1 /
                Double.valueOf(ecran.getText()).doubleValue();
          ecran.setText(String.valueOf(chiffre1));
        } catch(ArithmeticException e) {
          ecran.setText("0");
```

```
            }
        }
    }

    //Listener utilisé pour les chiffres
    //Permet de stocker les chiffres et de les afficher
    class ChiffreListener implements ActionListener {
      public void actionPerformed(ActionEvent e){
        //On affiche le chiffre additionnel dans le label
        String str = ((JButton)e.getSource()).getText();
        if(update){
          update = false;
        }
        else{
          if(!ecran.getText().equals("0"))
            str = ecran.getText() + str;
        }
        ecran.setText(str);
      }
    }

    //Listener affecté au bouton =
    class EgalListener implements ActionListener {
      public void actionPerformed(ActionEvent arg0){
        calcul();
        update = true;
        clicOperateur = false;
      }
    }

    //Listener affecté au bouton +
    class PlusListener implements ActionListener {
      public void actionPerformed(ActionEvent arg0){
        if(clicOperateur){
          calcul();
          ecran.setText(String.valueOf(chiffre1));
        }
        else{
          chiffre1 = Double.valueOf(ecran.getText()).doubleValue();
          clicOperateur = true;
        }
        operateur = "+";
        update = true;
      }
    }

    //Listener affecté au bouton -
    class MoinsListener implements ActionListener {
      public void actionPerformed(ActionEvent arg0){
        if(clicOperateur){
          calcul();
          ecran.setText(String.valueOf(chiffre1));
        }
        else{
          chiffre1 = Double.valueOf(ecran.getText()).doubleValue();
          clicOperateur = true;
        }
        operateur = "-";
        update = true;
      }
    }

    //Listener affecté au bouton *
    class MultiListener implements ActionListener {
      public void actionPerformed(ActionEvent arg0){
        if(clicOperateur){
          calcul();
          ecran.setText(String.valueOf(chiffre1));
        }
        else{
```

```java
        chiffre1 = Double.valueOf(ecran.getText()).doubleValue();
        clicOperateur = true;
      }
      operateur = "*";
      update = true;
    }
  }

  //Listener affecté au bouton /
  class DivListener implements ActionListener {
    public void actionPerformed(ActionEvent arg0){
      if(clicOperateur){
        calcul();
        ecran.setText(String.valueOf(chiffre1));
      }
      else{
        chiffre1 = Double.valueOf(ecran.getText()).doubleValue();
        clicOperateur = true;
      }
      operateur = "/";
      update = true;
    }
  }

  //Listener affecté au bouton de remise à zéro
  class ResetListener implements ActionListener {
    public void actionPerformed(ActionEvent arg0){
      clicOperateur = false;
      update = true;
      chiffre1 = 0;
      operateur = "";
      ecran.setText("");
    }
  }
}
```

Code : Java

```java
public class Main {
  public static void main(String[] args) {
    Calculatrice calculette = new Calculatrice();
  }
}
```

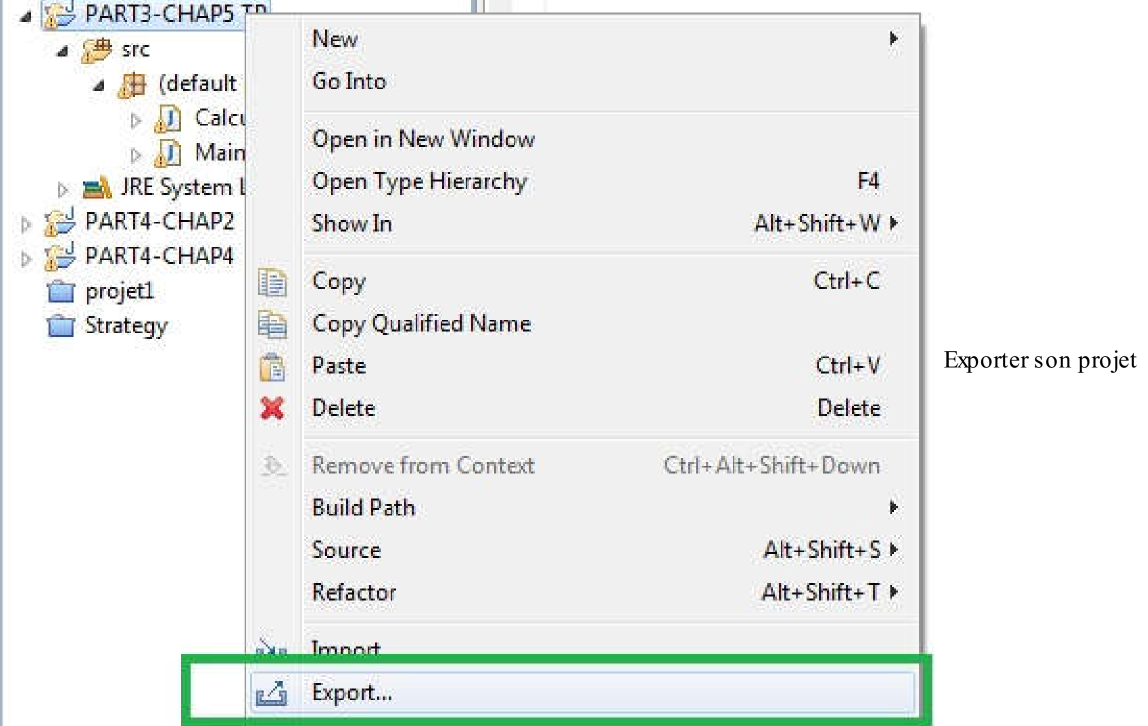
Je vais vous donner une petite astuce afin de créer un `.jar` exécutable en Java.

Générer un .jar exécutable

Tout d'abord, qu'est-ce qu'un `.jar` ? C'est une extension propre aux archives Java (Java ARchive). Ce type de fichier contient tout ce dont a besoin la JVM pour lancer un programme. Une fois votre archive créée, il vous suffit de double-cliquer sur celle-ci pour lancer l'application. C'est le meilleur moyen de distribuer votre programme.

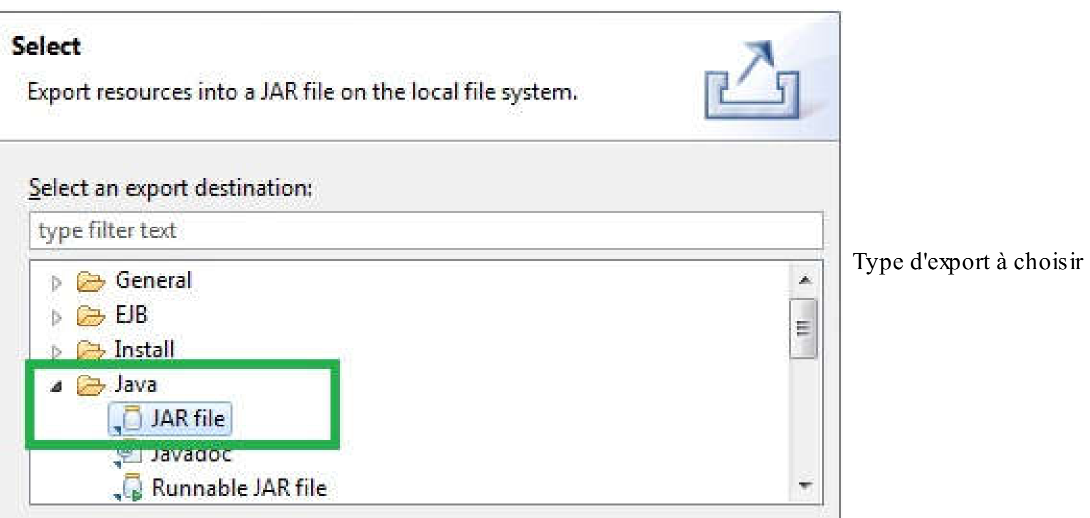
C'est exact pour peu que vous ayez ajouté les exécutables de votre JRE (présents dans le répertoire `bin`) dans votre variable d'environnement `PATH` ! Si ce n'est pas le cas, refaites un tour dans le premier chapitre du livre, section « Compilation en ligne de commande », et remplacez le répertoire du JDK par celui du JRE (si vous n'avez pas téléchargé le JDK ; sinon, allez récupérer ce dernier).

La création d'un `.jar` est un jeu d'enfant. Commencez par effectuer un clic droit sur votre projet et choisissez l'option `Export`, comme le montre la figure suivante.



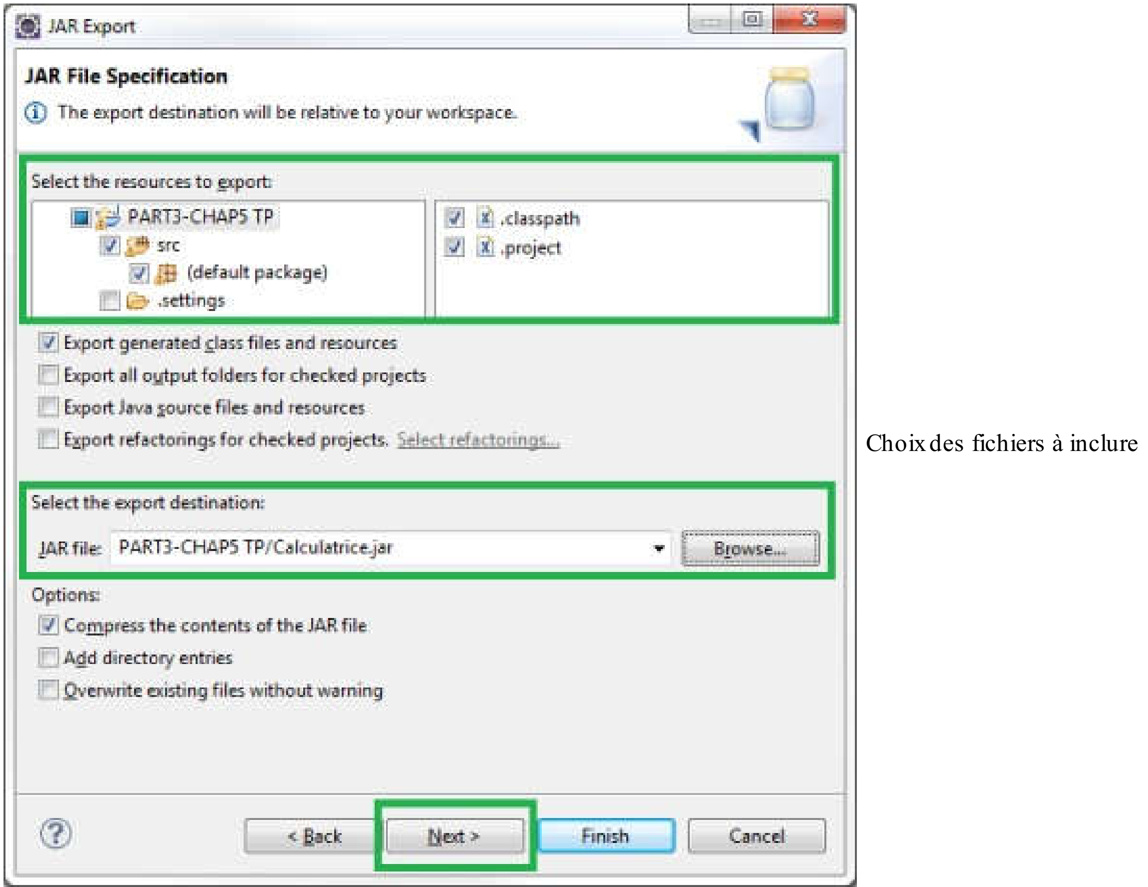
Exporter son projet

Vous voici dans la gestion des exports. Eclipse vous demande quel type d'export vous souhaitez réaliser, comme à la figure suivante.



Type d'export à choisir

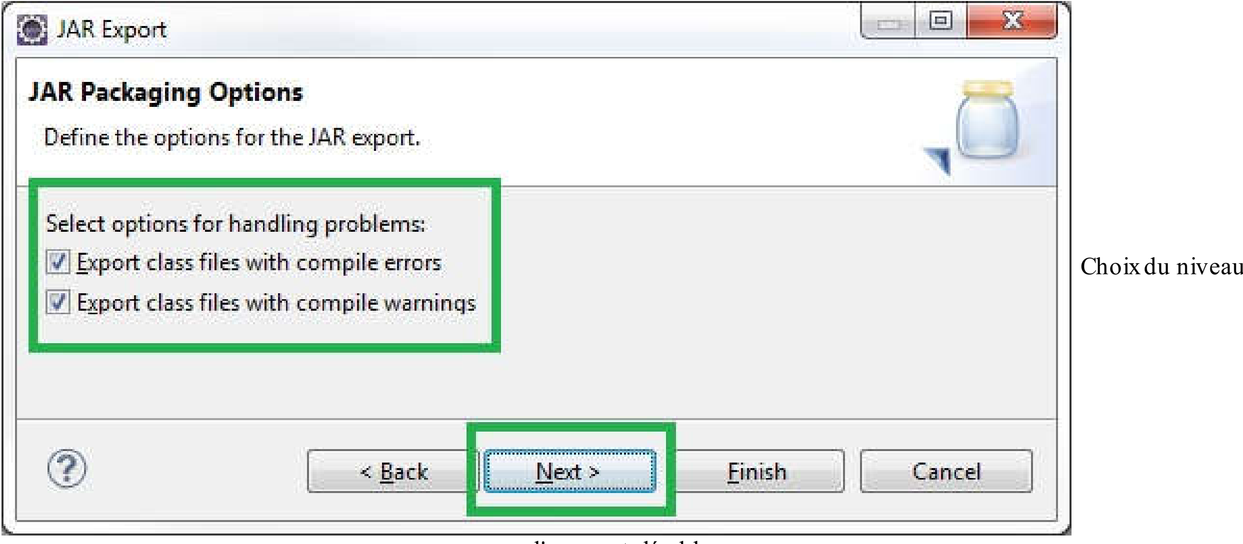
Comme l'illustre la figure précédent, sélectionnez `JAR File` puis cliquez sur `Next`. Vous voici maintenant dans la section qui vous demande les fichiers que vous souhaitez inclure dans votre archive, comme à la figure suivante.



Choix des fichiers à inclure

- Dans le premier cadre, sélectionnez tous les fichiers qui composeront votre exécutable `.jar`.
- Dans le second cadre, indiquez à Eclipse l'endroit où créer l'archive et le nom vous souhaitez lui donner.
- Ensuite, cliquez sur `Next`.

La page suivante n'est pas très pertinente ; je la mets cependant en figure suivante afin de ne perdre personne.

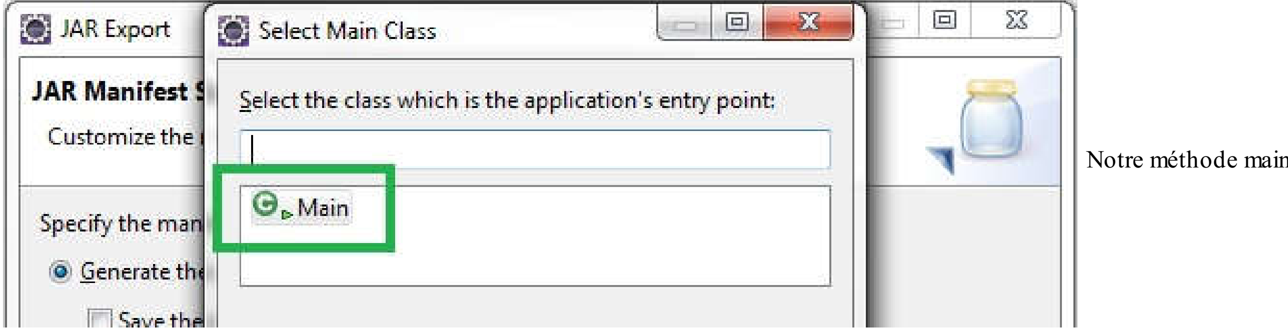


Choix du niveau d'erreurs tolérable

Cliquez sur `Next` : vous arrivez sur la page qui vous demande de spécifier l'emplacement de la méthode `main` dans votre programme (figure suivante).

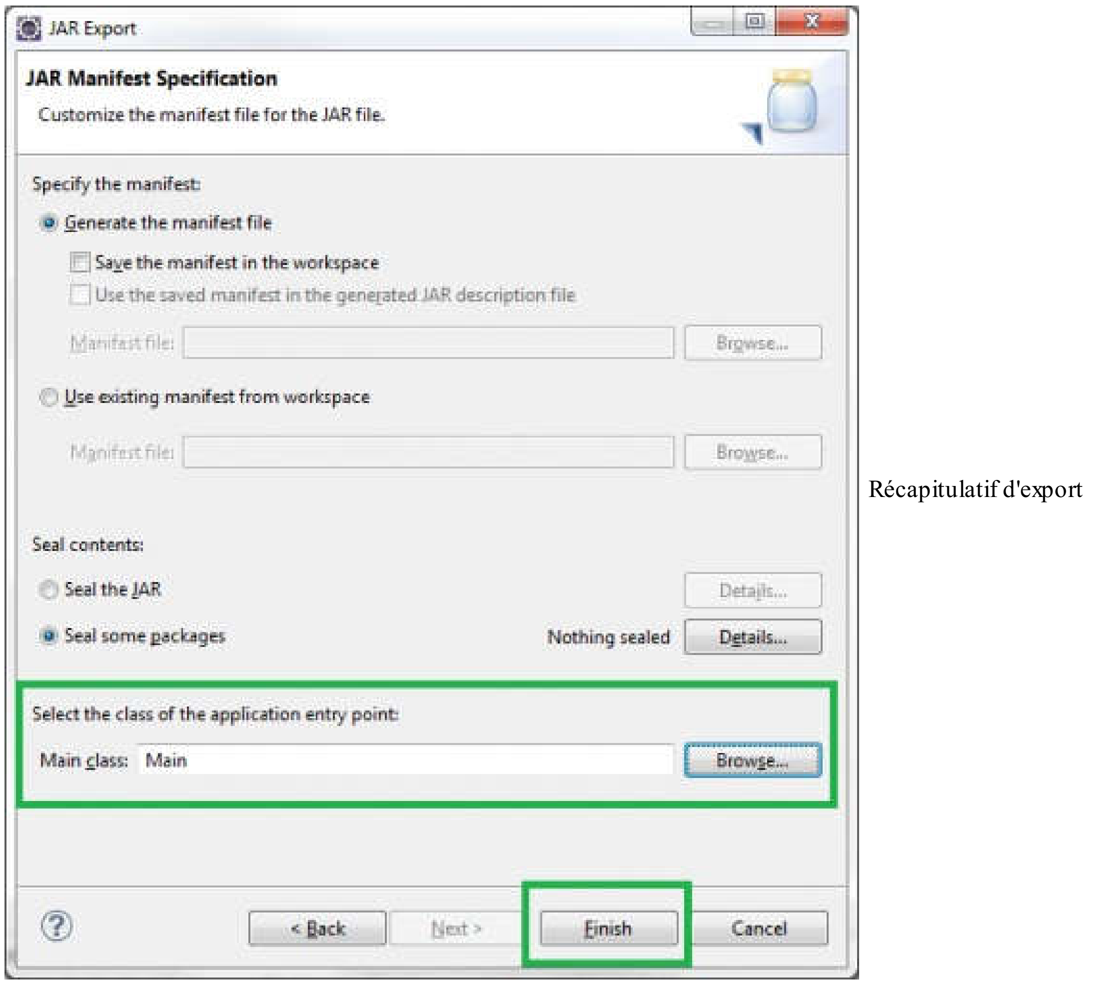
Choix du point de départ du programme

Cliquez sur `Browse...` pour afficher un *pop-up* listant les fichiers des programmes contenant une méthode `main`. Ici, nous n'en avons qu'une (voir figure suivante). Souvenez-vous qu'il est possible que plusieurs méthodes `main` soient déclarées, mais une seule sera exécutée !



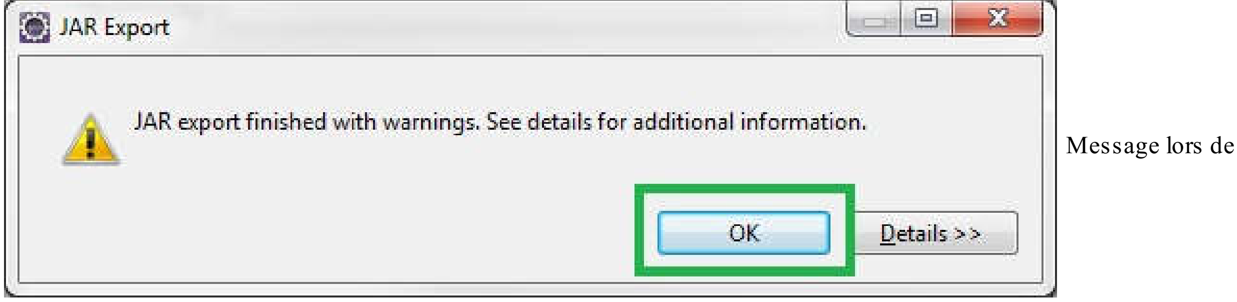
Notre méthode main

Sélectionnez le point de départ de votre application et validez. La figure suivante correspond à ce que vous devriez obtenir.



Récapitulatif d'export

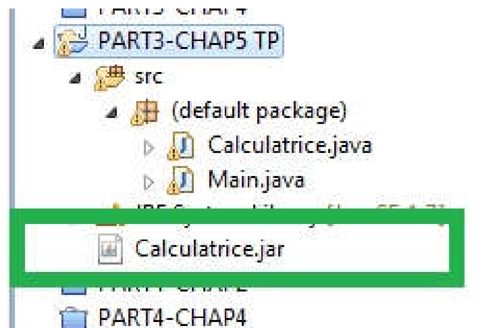
Vous pouvez maintenant cliquer sur `Finish` et voir s'afficher un message ressemblant à celui de la figure suivante.



Message lors de l'export

Ce type de message n'est pas alarmant : il vous signale qu'il existe des éléments qu'Eclipse ne juge pas très clairs. Ils n'empêcheront toutefois pas votre application de fonctionner, contrairement à un message d'erreur que vous repérerez facilement : il est en rouge.

Une fois cette étape validée, vous pouvez voir avec satisfaction qu'un fichier `.jar` a bien été généré dans le dossier spécifié, comme à la figure suivante.



Fichier exécutable .jar

Double-cliquez sur ce fichier : votre calculatrice se lance !