

## TP IHM : rapide introduction à la programmation événementielle et à Swing

- Les fichiers dont il est question dans ce document se trouvent dans le semainier sur le portail.
- attention à la confusion possible du à l’usage du terme *interface* dans deux sens : *interface* graphique et *interface* au sens JAVA du terme. A priori le contexte d’emploi du terme supprime l’ambiguïté.

### 1 Généralités

#### 1.1 Introduction

**Swing** est une boîte à outils permettant la création d’interfaces utilisateur en Java. Swing est une amélioration de **AWT** (*Abstract Window Toolkit*) qui était la boîte à outils originale tout en s’appuyant dessus. Les interfaces graphiques ainsi créées sont indépendantes de l’OS et de l’environnement graphique sous-jacents. Swing fournit les classes pour la représentation des différents éléments d’interfaces graphiques : fenêtres, boutons, menus, etc., et la gestion de la souris (les événements). Ces classes se trouvent dans le paquetage `javax.swing` et ses sous-paquetages. Pour construire une interface graphique avec Swing, vous utilisez des éléments préfabriqués (boutons, zone de textes, listes déroulantes, etc.) que vous assemblez et arrangez à votre convenance. Vous pouvez ainsi également construire de nouveaux éléments plus complexes.

Swing utilise des **gestionnaires de placement** pour disposer des **composants** à l’intérieur de **conteneurs** et contrôler leur taille et leur position.

#### 1.2 Composants

Tous les éléments graphiques de Swing sont des **composants**. Ils dérivent de la classe `javax.swing.JComponent`. Pour être utilisé, un composant doit le plus souvent être placé dans un **conteneur** (`java.awt.Container`). Les objets conteneurs regroupent les composants, les organisent pour les afficher grâce à un gestionnaire de placement. Les fonctionnalités des composants se décomposent essentiellement en deux catégories : celles responsables de l’**apparence** et celles chargées du **comportement**.

Pour ce qui est de l’apparence, il s’agit par exemple de la gestion d’attributs de base comme la visibilité, la taille, la couleur, la police, etc.

Pour le comportement, il s’agit de la réaction du composant aux **événements** contrôlés par l’utilisateur. Un événement est une action sur l’interface qui est susceptible de déclencher une réaction. Ces actions peuvent être un click de souris, le déplacement de la souris, le redimensionnement d’un objet, son masquage, sa sélection, la frappe d’une touche du clavier, etc. Pour ces actions, Swing (pour simplifier) envoie une notification d’événement aux objets qui se sont abonnés en tant que “**listener**” auprès du composant qui est à l’origine de l’événement. Ce listener doit posséder les méthodes de réaction à l’événement et celles-ci sont alors invoquées. C’est le **gestionnaire d’événements** de Swing qui est en charge de ce mécanisme.

#### 1.3 Redessiner

À tout moment un composant peut être appelé à se redessiner. Pour demander à un composant de dessiner, Swing invoque la méthode `paint()` de ce composant. Celle-ci est au moins appelée lors du premier affichage du composant. Vous n’avez pas a priori à vous occuper de cette méthode ni de son invocation. Lorsque vous souhaitez redessiner un composant, il faut invoquer sa méthode `repaint`. Celle-ci demande à Swing de prévoir un appel à `paint()` “dès que possible”. Swing se charge ensuite d’organiser et d’optimiser au mieux les différentes requêtes pour redessiner les composants.

#### 1.4 Conteneurs

Un **conteneur** est un composant qui contient d’autres composants et qui les gouvernent. Les conteneurs les plus utilisés sont `JFrame` (à nuancer), `JPanel` et `JApplet` (il existe également `JWindow`). Un `JPanel` est un conteneur générique qui permet de regrouper différents composants à l’intérieur d’un `JFrame` en créant des “sous-groupes” de composants en quelque sorte. Nous ne nous occuperons pas de `JApplet` pour l’instant.

Un conteneur peut lui-même contenir d’autres conteneurs et on crée ainsi une hiérarchie d’imbrications de composants. La méthode `add` des conteneurs permet de leur ajouter de nouveaux composants. Il est rare qu’un composant (autre qu’un conteneur) ait une existence en dehors d’un conteneur.

## 1.5 Gestionnaires de placement

Un **gestionnaire de placement** (“*layout manager*”) est un objet contrôlant le placement et le dimensionnement des composants situés à l’intérieur de la zone d’affichage d’un conteneur. Il existe un gestionnaire de placement par défaut mais chaque conteneur peut spécifier son propre gestionnaire de placement grâce à sa méthode `setLayout`.

Le gestionnaire par défaut d’un `JPanel` est un `java.awt.FlowLayout`. Il essaie de placer les objets selon leur taille préférentielle de gauche à droite, tant que c’est possible (c-à-d qu’il a assez de place pour le composant à afficher), puis de bas en haut.

Le gestionnaire par défaut d’un `JFrame` est un `java.awt.BorderLayout`. Ce dernier place les composants à des emplacements désignés dans la fenêtre par les positions (`java.awt.BorderLayout`.)`NORTH`, `SOUTH`, `EAST`, `WEST` et `CENTER`.

Deux autres gestionnaires de placement existant sont le `java.awt.GridLayout` et (plus complexe) le `java.awt.GridBagLayout`

## 2 Manipulations

Nous allons effectuer diverses manipulations, à travers celles-ci nous essaierons de découvrir les grands principes de construction d’interface. Cependant, une consultation de la `javadoc` des différentes classe sera nécessaire et indispensable, ainsi que de la pratique...

### 2.1 JFrame et JWindow

Seules ces deux classes peuvent être affichées en dehors de tout conteneur.

**Q 1 .** Récupérez le fichier `Test.java`, décommentez dans le `main`, l’invocation `t.testFenetre()`, compilez, testez et regardez le code source correspondant<sup>1</sup>.

Vous pouvez découvrir quelques méthodes de base comme `setVisible`, `setLocation` et `setSize`.

**Q 2 .** En décommentant puis testant la ligne `t.testDialog` vous pouvez avoir un aperçu de la création de fenêtres type popup. Vous trouverez tous les détails sur ces fenêtres dans la classe `javax.swing.JOptionPane`.

### 2.2 Panneaux de contenu

Les `JFrame` (et `JWindow`) sont des conteneurs particuliers car ils passent par un “panneau” pour regrouper des composants. Ce panneau est de type `java.awt.Container`. Il est possible d’y accéder ou de le modifier par les méthodes `getContentPane` ou `setContentPane`. C’est à ce panneau que l’on s’adresse pour ajouter ou retirer des composants à un objet `JFrame`.

Il est par exemple possible (et facile) de disposer de plusieurs objets “`JPanel`” différents, puis de changer le contenu d’une fenêtre en lui attribuant au choix l’un de ces objets `JPanel`.

**Q 3 .** Etudiez et testez avec la classe `TestPanel`.

Vous pouvez constater avec le test `testSetContentPaneBis` qui fait alterner l’utilisation de deux `JPanel` différents pour la même `JFrame` (il s’agit ici d’en illustrer l’usage, le code donné n’est certainement pas de toute beauté...).

**NB :** La méthode `pack()` demande à la `JFrame` de se redimensionner afin de s’ajuster exactement avec les composants qu’elle contient (en fait que contient son panneau).

La ligne `f.addWindowListener(...)` et la classe interne en fin de fichier sont laissées de côté pour l’instant, vous la comprendrez mieux lorsque nous aborderons la gestion des événements dans quelques minutes. Elle permet de quitter le programme sans le `CTRL+C`, en passant par les boutons de fermeture dans les bandeaux des fenêtres.

### 2.3 Layout

On va ici découvrir rapidement les différents gestionnaire de placements.

**Q 4 .** Etudiez, testez et surtout comparez les méthodes `test????Layout` dans le fichier `TestLayout.java`.

Il est notamment intéressant de modifier les dimensions de la fenêtre et de voir comment se comporte le placement des différents composants (boutons ici). Les tailles des composants graphiques sont calculées et ajustées en fonction des propriétés propres de chaque objet et des contraintes du gestionnaire de placement.

En combinant différents `JPanel` ayant leur propre `Layout`, on peut obtenir des dispositions plus complexes :

**Q 5 .** Testez et étudiez la méthode `testCombinaisonLayout` de `TestLayout.java` qui :

- crée 2 conteneurs `panel1` et `panel2`, chacun avec leur propre «layout manager» et leurs propres composants graphiques,
- les place dans le conteneur par défaut de la `JFrame f`, dont le gestionnaire de placement par défaut est un `BorderLayout`

---

<sup>1</sup>Faites un superbe `CTRL+C` pour stopper le programme.

## 2.4 Événements

Les différents composants Swing sont susceptibles d'émettre des événements. Ces événements peuvent être perçus pas des objets appelés *listeners* (ou *observateurs*) si ils sont **abonnés** au composant graphique qui émet l'événement.

À chaque événement correspond une ou plusieurs méthodes de l'observateur qui sont déclenchées lorsque l'événement se produit. Ces méthodes sont spécifiées par une interface. Les observateurs doivent donc être des instances de classes qui implémentent l'interface prévue pour l'événement considéré et donc qui implémentent ces méthodes.

Par convention, le plus souvent, pour tout événement de nom `XXXEvent`, le listener associé s'appelle `XXXListener`<sup>2</sup> et la méthode pour abonner un observateur (listener) à un composant se nomme `addXXXListener`.

La plupart des événements sont définis dans le paquetage `java.awt.event`.

Par exemple, lors d'un click sur un `JButton`, celui-ci émet un `(java.awt.event.)ActionEvent`. Pour être listener de cet événement, il faut que la classe du listener implémente l'interface `ActionListener`. Cette interface exige l'implémentation de la méthode `public void actionPerformed(ActionEvent e)`. C'est cette méthode qui sera «automatiquement» invoquée par le gestionnaire d'événements, chez tous les listeners abonnés auprès du bouton émetteur de l'événement. L'objet événement généré est passé en paramètre.

**Q 6 .** Examinez puis testez le contenu des fichiers `TestEvenement.java` et `ActionListenerTest.java`.

Il peut y avoir autant de listeners qu'on peut le souhaiter abonnés à une source d'événements (un composant), tous les listeners sont prévenus lors de l'émission d'un événement.

## 2.5 Classes internes

Comme vu avec le TD sur les listes, Java offre la possibilité de créer des classes à l'intérieur d'une autre classe. On parle alors de *classe interne*.

Nous avons vu que l'utilisation des classes internes est une application le principe d'encapsulation et de masquer ce qui n'a pas à être connu. L'un des intérêts est que la classe interne étant définie dans la portée de la classe "externe", elle peut avoir un accès direct aux éléments privés de cette dernière. Enfin, rappelons que pour toute instance d'une classe interne il existe une instance de la classe externe qui lui est attaché. Au sein du code de la classe interne, cette "instance externe" se référence par `ClasseExterne.this` où `ClassseExterne` est le nom de la classe externe.

Cette notion est utile pour la réalisation des interfaces graphiques, notamment pour la mise en place des *listeners* d'événements. C'est ce qui était utilisé dans les exemples précédents pour gérer la fermeture des fenêtres avec la classe interne `FermeWindowEvent` avec l'abonnement de la `JFrame` à un `WindowListener` (créé à partir d'un surcharge de `WindowAdapter`). Sans ce traitement, lorsque vous fermez la fenêtre, celle-ci est en fait juste masquée (visuellement), l'objet existe encore ainsi que le flux d'exécution qui lui est attaché (notamment pour la gestion des événements) et donc l'application n'a pas de raison de se terminer.

**Q 7 .** Etudiez et testez `TestInterneEvenement.java`.

Si vous consultez les fichiers produits par la compilation, vous voyez apparaître le fichier `.class` correspondant à la classe interne avec un nom formé à partir du nom de la classe principale et de celui de la classe interne séparés par un `$`, dans notre cas : `TestInterneEvenement$InterneActionListenerTest.class`.

Remarquez que dans la classe interne on a pu accéder à l'attribut `unBouton` de la classe englobante bien qu'il soit privé. C'est normal puisque la définition de la classe interne est dans la *portée* de ces attributs privés (càd "entre les accolades de la classe"). Cela facilite les interactions entre les différents composants d'une interface graphique et évite de rendre publiques certains éléments. Nous continuerons à illustrer cela dans la section suivante.

## 2.6 Interactions

**Q 8 .** Testez et étudiez `TestInteraction.java` : cet exemple illustre l'influence d'un composant graphique sur un autre. Ici une action sur le bouton modifie le texte du label.

A nouveau on a pu accéder dans la classe interne à l'attribut privé `unLabel`

## 2.7 JTextField, autre événement : KeyEvent

Les `JTextField` sont des composants texte éditables ou non (par défaut ils le sont, voir méthode `setEditable(...)`). Ils émettent un événement de type `ActionEvent` quand la touche ENTRÉE est pressée. Comme tous les autres composants ils sont également sensibles aux `KeyEvent`.

**Q 9 .** Testez et étudiez `TestTextField.java`, essayez par exemple les caractères X et x dans le champ de texte.

---

<sup>2</sup>ce qui facilite le travail de mémorisation du programmeur (exception par exemple avec le `MouseMotionListener`)

### 3 Rapide synthèse

Les principes pour la mise en construction d'une interface graphique que nous avons évoqués sont ;

- les composants sont placés dans des conteneurs et leur agencement est organisé via un «layout»
- il existe des objets graphiques prédéfinis classiques tels que : `JButton` (les cases à cocher et radio boutons en sont des cas particulier), `JLabel`, pour les textes `JTextField/JTextArea/JTextPane`, `JMenu`, etc.  
Pour chacun il faut lire la documentation et en particulier on peut étudier le lien “how to use ...” dans l'entête de la javadoc de chaque classe
- il faut déterminer à quels événements ces composants peuvent et doivent répondre (clic, clavier, déplacement souris, etc.) et définir quel est leur réponse.
- pour gérer la réponse à un événement on crée une classe du type du «listener associé» à l'événement et surtout ne pas oublier de l'abonner au composant graphique

#### 3.1 Exercice

**Exercice 1 :** On considère une classe abstraite `Counter` (cf. ci-dessous)

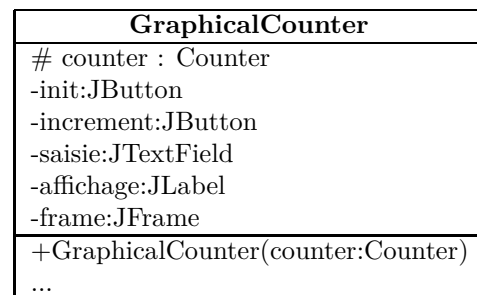
On souhaite disposer de plusieurs implémentations de compteurs. Sont définies par héritage des classes `SimpleCounter` (qui incrémente de 1 en 1), `ModularCounter` (qui est un compteur “qui boucle” sur les valeurs de 0 à  $n$ ), `CompteurGeometrique` (dont l'incrément consiste à multiplier la valeur par  $n$ ), etc. qui héritent de cette classe en surchargeant la méthode `increment()` (code disponible sur le portail).

Il vous est demandé de définir une interface graphique “générique” permettant de manipuler les compteurs. La même classe “graphique” doit permettre la manipulation de toutes les classes de compteur.

Elle présentera une zone de saisie (`JTextField`) pour donner la valeur initiale, une zone d'affichage `JLabel` qui restitue la valeur courante du compteur et deux boutons : `init` et `increment`, qui ont pour effet l'appel de la méthode de même nom, `init` prenant la valeur dans la zone de saisie et `increment` modifiant le label.

Cette classe pourrait ressembler à ce qui est présenté sur le diagramme UML ci-dessous.

```
public abstract class Counter {  
    protected int value;  
    public Counter(...) { ... }  
    public int getCurrentValue() { ... }  
    public abstract void increment();  
    public void init(int init) { ... }  
}
```



Vous utiliserez votre interface pour tester des instances des différentes classes de compteur créées. On doit donc pouvoir indifféremment écrire :

```
new GraphicalCounter(new SimpleCounter());  
new GraphicalCounter(new ModularCounter(7));  
new GraphicalCounter(new CompteurGeometrique(3));
```

et faire de même pour toute nouvelle classe de compteur que l'on pourrait créer par la suite.