# 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'ambigüité.

## 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 des événements. Les classes de Swing 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.

### 1.2 Composants

Les principaux 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 clic 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 possèder 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. Le mécanisme est similaire à celui d'autres langage, comme javascript par exemple.

### 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`[1] et `JPanel` (il existe également `JWindow` ...).
Un `JPanel` est un conteneur générique. Il permet de regrouper différents composants afin de créer un nouveau composant[2] plus complexe qui est propre à l'application. On peut ensuite combiner à leur tour les composants ainsi obtenus pour en former de nouveau encore "plus complexes".
En effet 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] Qui hérite en fait de `java.awt.Component` et pas de `javax.swing.JComponent`
[2] N'oublions pas que les `Panel` sont aussi des composants.

### 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'interfaces grpahiques avec Swing. Cependant, une consultation de la `javadoc` des différentes classe sera certainement 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 `FirestFrameAndWindow.java`. Compilez-le et expérimentez son exécution.

**Q 2 .** **Etudiez le code source** pour faire le lien entre ce qui a été affiché.

**Q 3 .** Décommentez dans le `main`, l'invocation `t.jWindowExperiment()()`, compilez, expérimentez et, à nouveau, comparez avec le code source correspondant.

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

**Q 4 .** En décommentant puis expérimentant la ligne `t.dialogExperiment()` 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`.

**NB :** 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 quand la fenêtre est fermée, sinon celle-ci est simplement « masquée » sans que le programme (« `main` »)ne se termine.

### 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 au besoin le contenu d'une fenêtre en lui attribuant au choix l'un de ces objets `JPanel`.

**Q 5 .** Etudiez le code et expérimentez avec la classe `PanelExperiments`.

Vous pouvez constater avec le test `setContentPaneSecondExperiment` 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).

### 2.3 Layout

On va ici découvrir rapidement les différents gestionnaire de placements.

**Q 6 .** Etudiez, testez et surtout comparez les méthodes `????LayoutExperiment` dans le fichier `LayoutExperiments.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 7 .** Exépreimentez et étudiez la méthode `combinedLayoutExperiment` de `LayoutExperiments.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`

## 2.4 Evénements

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 osbervateurs doivent donc être des instances de classes qui implémente l'interface prévue pour l'événement considé et donc qui implémentent ces méthodes.
Par convention, le plus souvent, pour tout événement de nom `XXXEvent`, le listener associé s'appelle `XXXListener`[3] 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 clic 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 8 .** Examinez, puis expérimentez, le contenu des fichiers `EventExperiments.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

Java offre la possibilité de créer des classes à l'intérieur d'une autre classe. On parle alors de *classe interne*.
L'utilisation des classes internes est une application le principe d'encapsulation. Elle permet 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, soulignons 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 `ClasseExterne` 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 `CloseWindowEvent` 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 évènements) et donc l'application n'a pas de raison de se terminer.

**Q 9 .** Etudiez et expérimentez `InnerEventExperiments.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 `$`, dans notre cas : `InnerEventExperiments$InnerActionListener.class`.
Remarquez que dans la classe interne on a pu accéder à l'attribut `aButton` 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 10 .** Testez et étudiez `InteractionExperiments.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 `aLabel`

[3] ce qui facilite le travail de mémorisation du programmeur (exception par exemple avec le `MouseMotionListener`)

### 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 11 .** Expérimentez et étudiez `TextFieldExperiments.java`, essayez par exemple les caractères `X` et `x` dans le champ de texte.

## Menus (très vite)

Les classes permettant de gérer des menus sont dans le paquetage `javax.swing` :

**JMenuBar** la barre de menu

**JMenu** un menu, que l'on ajoute à un `JMenuBar` par `add`

**JMenuItem** un élément du menu, que l'on ajoute à un `JMenu` par `add`. On peut abonner des `ActionListener` à un `JMenuItem`, leur méthode `actionPerformed` est déclenchée lors que l'on clique sur l'item. (`JMenuItem` est en fait une sous-classe de `AbstractButton`).

Jetez un œil au lien "*How to Use Menus*" aux documentations des classes `JMenu`, `JMenuItem` et `JMenuBar` du paquetage `javax.swing`.

# 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 graqhiques prédéfinis classiques tels que : `JButton` (les cases à coser 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