TP IHM : introduction 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 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 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.

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.
Récupérez le fichier `Test.java`, décommentez dans le `main`, l'invocation `t.testFenetre()`, compilez, testez **et regardez le code source correspondant**.
Vous pouvez découvrir quelques méthodes de base comme `setVisible`, `setLocation` et `setSize`.

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.

2.2 Panneaux de contenu

Les `JFrame` (et `JWindow`) sont des conteneurs particuliers car ils passent par un "**panneau**" pour regrouper des composants. Ce tableau est de type `java.awt.Container`. Il est possible d'accéder à ce panneau 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 `JFrame`
Décommentez du `main` de `Test.java` et testez les invocations `t.testGetContentPane()` et `t.testSetContentPane()`. À nouveau consultez le code.
En apparence tout est pareil, ce qui a changé c'est la gestion du "Container" de la JFrame, dans le premier cas on utilise celui par défaut, dans le second on en définit un que l'on "impose" ensuite à la fenêtre. Avec cette seconde méthode il est facile de disposer de plusieurs "JPanel" puis de changer le contenu de la fenêtre.
Vous pouvez constater cela 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...).
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

▷ Etudiez, testez et surtout comparez les méthodes `test????Layout` dans `Test.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).

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

Testez et étudiez le fichier `TestLayout.java`

2.4 Evénements

Les différents composants Swing sont susceptibles d'émettre des événements. À chaque événement correspond une ou plusieurs méthodes susceptibles d'être déclenchées par le gestionnaire d'événements auprès d'objets appelés *listeners* (ou *observateurs*). Les listeners doivent donc être des instances de classes qui implémentent ces méthodes. Pour être `listener` il faut être instance d'une classe qui implémentera l'interface listener prévue pour l'événement considéré. 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.

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`[1]. 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.
Récupérez les fichiers `TestEvenement.java` et `ActionListenerTest.java`, testez et regardez le code.
On abonne un objet aux événements émis par un composant grâce à une méthode dont le nom et de la forme `addXXXListener`.

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*.
Sans entrer dans tous les détails liées à cette notion,[2] disons que D'autre part les classes internes sont un autre moyen d'appliquer le principe d'encapsulation et de ne pas forcément publier ce qui n'a pas à l'être. Cela permet au concepteur de contrôler ce qui est fait et d'empêcher tout mauvais usage des notions qu'il met en œuvre.
L'un des intérêts de cette possibilité 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. En plus de l'utilisation que nous verrons ci-dessoua avec les interfaces graphiques. On peut citer le cas des `Iterator` dont l'implémentation pour chaque type de collection est gérée par une classe interne à celle de la collection concernée. Cela permet ainsi à l'objet `Iterator` d'avoir accès à la structure de reprsentation de la collection sous-jacente sans pour autant que cette notion soit publique.
Pour finir cette présentaion très succincte, indiquons 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 `ClasssExterne` est le nom de la classe externe.

Cette notion est utile pour la réalisation des interfaces graphiques, notamment pour les *listeners*.
Testez `TestInterneEvenement.java`.
En regardant le code, vous pouvez constater que deux classes sont définies dans le fichier, mais une seule classe est publique (le contraire n'est pas possible). L'autre classe est dite interne à la première : elle est défini à l'intérieur des accolades de définition de la classe "principale". Si vous regardez 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 `$`.
Remarquez que dans la classe interne on a pu utiliser l'attribut (bien que privé) de la classe englobante, 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 allons illustrer cela dans la section suivante.

2.6 Interactions

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.

2.7 `JTextField`

Les `JTextField` sont des composants texte éditables ou non (par défaut ils le sont, voir `setEditable(...)`). Ils émettent un `ActionEvent` quand la touche ENTRÉE est pressée. Comme tous les autres composants ils sont également sensibles aux `KeyEvent`.
Testez et étudiez `TestTextField.java`, essayez par exemple les caractères `X` et `x` dans le champ de texte.

Vous pouvez au passage noter aussi la gestion de la fermeture de l'application lorsque l'on ferme la fenêtre avec l'abonnement à la `JFrame` d'un `WindowListener` (créé à partir d'un surcharge de `WindowAdapter`).
Sans ce traitement, lorsque vous fermez la fenêtre, celle-ci est 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.

2.8 Exercice

Exercice 1 : On considère une classe abstraite `Counter`

[1] Par convention, pour tout événement de nom `XXXEvent`, le listener associé s'appelle `XXXListener`, ce qui facilite le travail du développeur – à l'exception du `MouseMotionListener` – et la méthode pour ajouter un listener à un composant se nomme `addXXXListener`.

[2] vous êtes **très** fortement invité à vous informer sur cette notion par vous mêmes, consultez par exemple la partie qui y est consacrée dans "*Thinking in Java*" dont l'URL est proposée sur le portail.

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

De la même manière que cela avait été fait en POO au S4 (pour ceux ayant suivi cette UE) avec les interfaces, 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()` (le code est 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 devrait ressembler à :

GraphicalCounter
counter : Counter -init:JButton -increment:JButton -saisie:JTextField -affichage:JLabel -frame:JFrame
+GraphicalCounter(counter:Counter) ...

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.