Interfaces graphiques

Pratique de la programmation orientée-objet
Michel Schinz – 2015-05-18

AWT, Swing, Java FX

La bibliothèque Java contient trois ensembles de classes permettant la création d'interfaces graphiques. Dans l'ordre chronologique de leur apparition, ce sont :

1. **AWT** (*Abstract Window Toolkit*), tombé en désuétude mais servant de base à Swing,
2. **Swing**, utilisé pour la majorité des applications existantes mais en cours de remplacement par Java FX,
3. **Java FX**.

Cette leçon décrit Swing, mais la plupart des concepts se retrouvent dans Java FX et dans d'autres bibliothèques similaires pour d'autres langages.

Swing

Swing est composé d'un très grand nombre de classes appartenant toutes (ou presque) au paquetage `javax.swing` et à ses sous-paquetages.
Etant donné que Swing se base sur AWT, les classes du paquetage `java.awt` et ses sous-paquetages sont parfois nécessaires.

Composants

Composants

Une interface graphique Swing se construit en combinant un certain nombre de **composants** (*components*), imbriqués les uns dans les autres.
Par exemple, les fenêtres, les menus, les boutons, les zones de texte, etc. sont des composants.
Les classes représentant ces composants ont toutes un nom commençant par `J`, et – à quelques exceptions près – héritent de la classe abstraite `javax.swing.JComponent`.

Types de composants

Il existe deux types de composants Swing :

1. les **composants de base**, qui ne contiennent pas d'autres composants (boutons, etc.),
2. les **conteneurs**, dont le but principal est de contenir et d'organiser d'autres composants (fenêtres, etc.).

Les conteneurs sont séparés en deux catégories :

1. les **conteneurs de niveau intermédiaire**, qui peuvent eux-même être contenus par d'autres conteneurs,
2. les **conteneurs de niveau supérieur** (*top-level containers*), qui ne peuvent pas l'être.

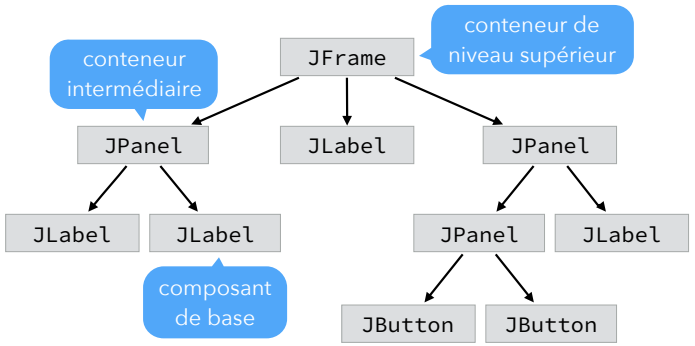
Hiérarchie

Les composants d'une application Swing sont organisés en (au moins) une hiérarchie arborescente dont :

- la racine est un conteneur de niveau supérieur,
- les nœuds – autres que la racine – sont des composants intermédiaires,
- les feuilles sont des composants de base.

Le principal type de conteneur de niveau supérieur est la fenêtre, donc à chaque fenêtre affichée à l'écran par une application Swing correspond une hiérarchie.
A noter qu'un composant ne peut pas appartenir à plus d'une hiérarchie, ou apparaître plus d'une fois dans la même hiérarchie.

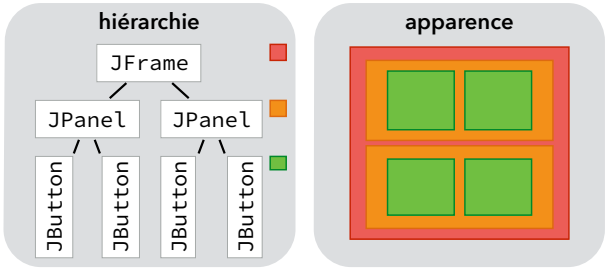
Exemple de hiérarchie



Note : cette hiérarchie a été légèrement simplifiée par rapport à la réalité.

Présentation à l'écran

A l'écran, tout composant occupe une zone rectangulaire.
Les fils d'un conteneur se partagent généralement sa zone à lui, et il n'y a pas de chevauchement entre composants frères.
La hiérarchie de composants est donc aplatie en un ensemble de zones rectangulaires imbriquées.



JComponent

La classe abstraite `JComponent` sert de classe-mère à tous les composants Swing, sauf ceux de niveau supérieur.
`JComponent` et ses sous-classes possèdent un grand nombre de **propriétés** (attributs) modifiables. A chacune d'entre-elles est associée une méthode de lecture nommée `get…` et une méthode d'écriture nommée `set…`.
Par exemple, la propriété `font`, qui contient la police utilisée pour le texte d'un composant, se lit avec la méthode `getFont` et s'écrit avec la méthode `setFont`.

Composants de base

Les composants de base offerts par Swing incluent :
- une étiquette, textuelle ou graphique (`JLabel`),
- différents boutons : à un état (`JButton`), à deux états (`JToggleButton`), « radio » (`JRadioButton`), à cocher (`JCheckBox`),
- un champ textuel (`JTextField`), également en version formatante (`JFormattedTextField`), ou masquée pour les mots de passe (`JPasswordField`),
- une liste de valeurs dont certaines peuvent être sélectionnées (`JList`),
- etc.

Conteneurs intermédiaires

Les conteneurs intermédiaires offerts par Swing incluent :
- un panneau séparé en deux parties redimensionnables, affichant chacune un composant (`JSplitPane`),
- un panneau à onglets (`JTabbedPane`),
- un panneau affichant une partie d'un composant trop grand pour être affiché en entier (`JScrollPane`),
- un panneau permettant de superposer plusieurs composants (`JLayeredPane`),
- un panneau sans représentation graphique (`JPanel`),
- etc.

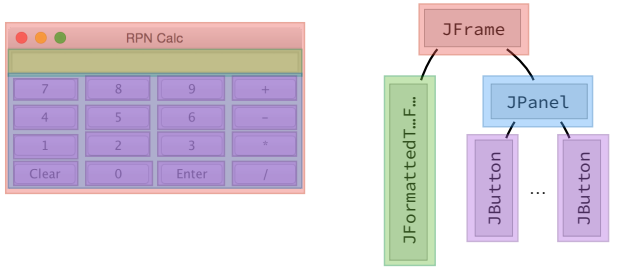
Conteneurs de niveau sup.

Les conteneurs de niveau supérieur offerts par Swing sont au nombre de trois :

- la fenêtre (`JFrame`),
- la boîte de dialogue (`JDialog`),
- l'« applet » (`JApplet`) destinée principalement à être utilisée dans un navigateur Web, rarement utilisée de nos jours.

Exemple

L'interface graphique d'une calculatrice à quatre opérations pourrait être obtenue au moyen de la hiérarchie suivante :



Conteneurs de niveau supérieur

Composants de niveau sup.

Les composants de niveau supérieur (*top-level components*) sont à la racine de toute hiérarchie visible à l'écran.
Contrairement aux autres types de composants, ceux de niveau supérieur ne peuvent pas être inclus dans d'autres composants.
Swing offre trois types de conteneurs de niveau supérieur : la fenêtre, la boîte de dialogue et l'applet. Seuls les deux premiers sont examinés ici.

JFrame

`JFrame` représente une fenêtre.
Ses principales propriétés sont : son titre (`title`), sa taille et sa position à l'écran (`bounds`), sa visibilité (`visible`, valant `false` par défaut), son comportement en cas de fermeture (`defaultCloseOperation`) et son panneau de contenu (`contentPane`).

Fermeture des fenêtres

Par défaut, lorsque l'utilisateur ferme une fenêtre, celle-ci est simplement rendue invisible.
Ce comportement est souvent inadéquat pour la fenêtre principale d'une application, car il est préférable que la fermeture de cette fenêtre provoque l'arrêt complet de l'application, p.ex. via un appel à `System.exit`.
Pour obtenir ce comportement, on peut changer la propriété `defaultCloseOperation` afin de lui donner la valeur `EXIT_ON_CLOSE` :

```
JFrame f = …;
frame.setDefaultCloseOperation(
  JFrame.EXIT_ON_CLOSE);
```


Agencement

Comme cela sera expliqué plus loin, l'agencement des composants à l'écran est généralement fait automatiquement dans une application Swing. Cet agencement se fait récursivement, en commençant par les feuilles puis en remontant jusqu'à la racine.
Les composants de niveau supérieur offrent une méthode `pack` permet d'agencer tous les composants de la hiérarchie dont ils constituent la racine.

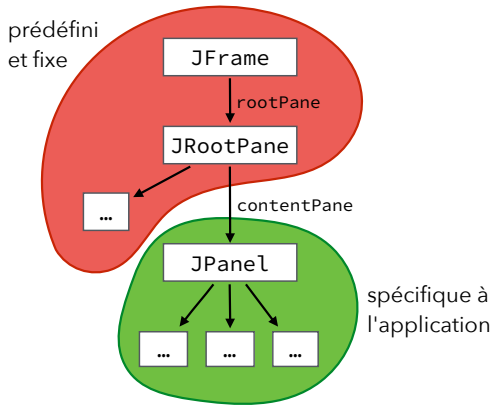
JDialog

`JDialog` représente une boîte de dialogue, visuellement similaire à une fenêtre mais se comportant différemment.
Une boîte de dialogue est toujours liée à une fenêtre existante, dans le sens où lorsque cette dernière est fermée ou iconifiée, il en va de même de la première.
Beaucoup de boîtes de dialogue sont **modales**, dans le sens où toute interaction avec le reste de l'application est impossible tant et aussi longtemps que la boîte n'est pas fermée.

Panneaux racine/contenu

Les conteneurs de niveau supérieur ne contiennent pas directement un certain nombre de composants fils. Au lieu de cela, ils possèdent un unique **panneau racine** (*root pane*), un type spécial de conteneur intermédiaire.
Ce panneau racine contient, entre autres, un **panneau de contenu** (*content pane*), dans lequel les composants spécifiques à l'application sont ajoutés. La plupart des applications ignorent donc le panneau racine et interagissent uniquement avec le panneau de contenu.
Le panneau de contenu est une propriété modifiable des conteneurs de niveau supérieur, nommée `contentPane`.

Panneaux racine/contenu



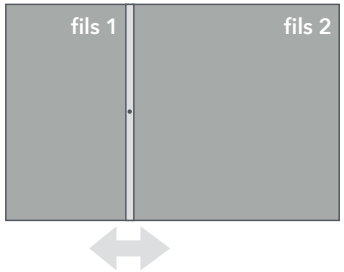
Conteneurs intermédiaires

Conteneur intermédiaire

Les conteneurs intermédiaires sont les nœuds – autres que la racine – de la hiérarchie.
Le but principal d'un conteneur intermédiaire est de grouper et d'organiser un certain nombre d'autres composants, nommé ses fils. Dès lors, sa représentation graphique propre est souvent minimale, voire inexistante.

JSplitPane

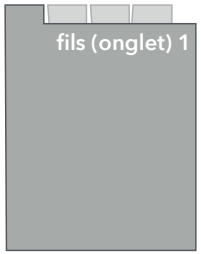
`JSplitPane` permet de diviser le composant en deux parties, chacune affichant un composant fils.
La division peut être verticale ou horizontale, et est redimensionnable, éventuellement par l'utilisateur.



25

JTabbedPane

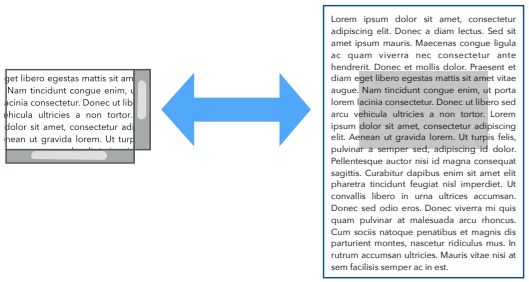
`JTabbedPane` est un panneau composé d'un certain nombre d'onglets affichant chacun un composant fils différent. Un seul onglet est visible à un instant donné.



26

JScrollPane

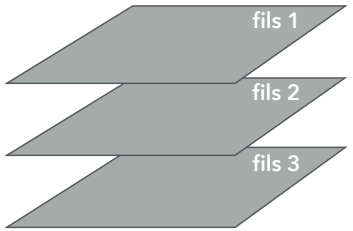
`JScrollPane` donne accès à une sous-partie d'un composant trop grand pour tenir à l'écran et permet de déplacer la zone visualisée de différentes manières, p.ex. au moyen de barres de défilement.



27

JLayeredPane

`JLayeredPane` permet de superposer plusieurs composants, ce qui peut être utile pour dessiner au-dessus de composants existants ou pour intercepter les clics de souris qui leur sont destinés.



28

Conteneur intermédiaire `JPanel`

`JPanel`

`JPanel` représente un panneau, un conteneur intermédiaire sans représentation graphique propre (exception faite de son éventuelle bordure, voir plus loin) et pouvant avoir un nombre quelconque de fils.
Malgré son invisibilité, `JPanel` joue un rôle fondamental dans l'organisation des interfaces, de par sa capacité à grouper et placer – via son gestionnaire d'agencement – d'autres composants.

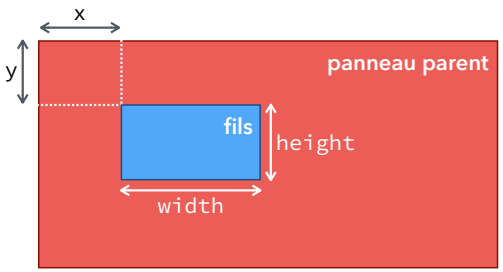
Gestion des fils

Les méthodes `add` et `remove` permettent de gérer les fils d'un conteneur `JPanel` :

- `void add(JComponent c, Object l, int i)` : insère le composant `c` à la position `i` dans les fils (-1 signifiant la fin) et lui associe l'information d'agencement `l` (voir plus loin),
- `void add(JComponent c, Object l)` ≡ `add(c, l, -1)`
- `void add(JComponent c)` ≡ `add(c, null)`
- `void remove(JComponent c)` : supprime le composant `c` des fils,
- `void remove(int i)` : supprime le fils d'index `i`.

Bornes des fils

Tout composant fils d'un panneau occupe un rectangle imbriqué dans celui de son parent. On nomme **bornes** (*bounds*) la position (`x`, `y`) et la taille (`width`, `height`) de ce rectangle, exprimés dans le repère du panneau.



Agencement des fils

L'agencement des fils – leur positionnement à l'intérieur du rectangle de leur parent – peut se faire de deux manières :

1. « manuellement », en changeant leurs bornes au moyen de la méthode `setBounds`,
2. via un **gestionnaire d'agencement** (*layout manager*) attaché au parent et responsable de l'agencement de ses fils et de son dimensionnement.

Dans la quasi-totalité des cas, la seconde solution est choisie.

Agencement

Les gestionnaires d'agencement peuvent utiliser plusieurs caractéristiques des composants fils pour les placer :

- leur index dans la séquence des fils,
- leur taille minimale (propriété `minimumSize`), préférée (`preferredSize`) et maximale (`maximumSize`),
- l'éventuelle information d'agencement qui leur a été associée au moment de leur ajout via la méthode `add`.

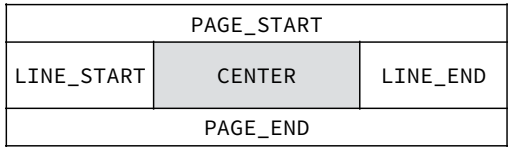
Attention : les gestionnaires ont le droit mais pas l'obligation d'utiliser ces caractéristiques. En particulier, plusieurs gestionnaires ignorent totalement les informations de taille.

LayoutManager

L'interface `LayoutManager` représente un gestionnaire d'agencement.
Un certain nombre de gestionnaires prédéfinis l'implémentant sont fournis avec Swing : `FlowLayout`, `BorderLayout`, `GridLayout`, `GridBagLayout`, etc. Ces gestionnaires agencent chacun les fils en fonction d'une technique qui leur est propre.

BorderLayout

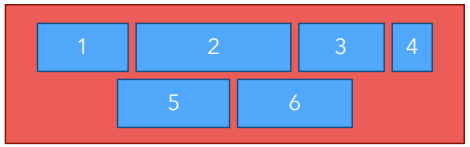
`BorderLayout` découpe le composant en cinq régions nommées, chacune d'entre-elles pouvant contenir au plus un composant.



Toutes les zones sauf la centrale sont dimensionnées en fonction de leur contenu – qui peut être inexistant. La totalité de l'espace restant est attribué à la zone centrale.

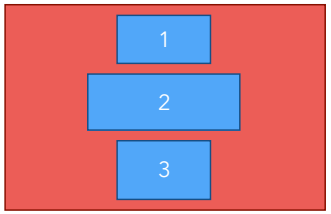
FlowLayout

`FlowLayout` place les fils de la même manière que les mots d'un texte sont arrangés en lignes successives.
Cet agencement peut se faire de gauche à droite ou de droite à gauche, et les lignes peuvent être alignées à gauche, à droite, ou centrées. Les fils sont séparés horizontalement et verticalement par un espacement configurable.
Par exemple, il peut agencer ainsi six fils d'un panneau en les plaçant de gauche à droite et en centrant chaque ligne :



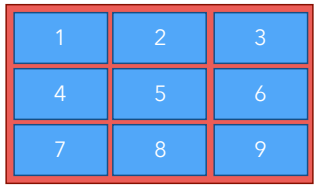
BoxLayout

`BoxLayout` place les fils dans l'ordre, soit alignés horizontalement, soit empilés verticalement.
Par exemple, il peut agencer ainsi trois fils d'un panneau en les empilant verticalement :



GridLayout

`GridLayout` place les fils dans une grille de n×m cellules de taille identique, en partant du coin haut-gauche et en suivant le sens de lecture.
Par exemple, il peut agencer ainsi neuf fils d'un panneau dans une grille de 3×3 cellules :



Autres gestionnaires

Swing offre encore d'autres gestionnaires d'agencement, plus puissants que ceux présentés jusqu'ici, parmi lesquels :

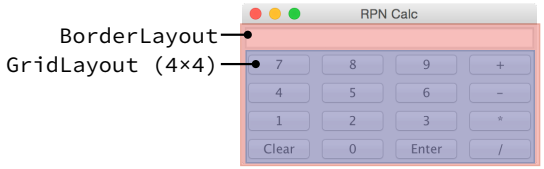
- `GridBagLayout`, version améliorée de `GridLayout` permettant de dimensionner les lignes et colonnes individuellement, et de combiner des cellules,
- `GroupLayout`, associant chaque composant à un groupe horizontal et un groupe vertical et alignant de manière configurable tous les composants d'un groupe.

Malheureusement, comme tous les gestionnaires d'agencement offerts par Swing, ils montrent assez vite leurs limites lors de la construction d'interfaces complexes.

Exemple

L'interface de la calculatrice peut être mise en page au moyen de deux gestionnaires :

1. le premier de type `BorderLayout`, attaché au panneau de contenu, permettant de placer l'affichage dans la zone `PAGE_START` et le panneau du clavier dans la zone `CENTER`,
2. le second de type `GridLayout` et de taille 4×4, attaché au panneau clavier.



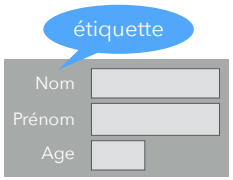
Exemple

```
JPanel p = new JPanel(new BorderLayout());
JFormattedTextField display = …;
// … configuration de display
p.add(display, BorderLayout.PAGE_START);

JPanel keyboard =
  new JPanel(new GridLayout(4, 4));
// … configuration de keyboard (boutons, …)
p.add(keyboard, BorderLayout.CENTER);

JFrame frame = new JFrame("RPN Calc");
frame.setContentPane(panel);
```


Composants de base

Etiquette

`JLabel` représente l'un des composants les plus simples qui soit, une étiquette textuelle et/ou graphique. Ses principales propriétés sont le texte (`text`) et l'image (`icon`) qu'elle affiche, pouvant les deux être vides (`null`).

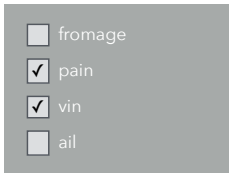


Bouton à un état

`JButton` représente un bouton à un état, destiné à être cliqué pour effectuer une action.
Sa principale propriété est le texte qu'il affiche (`text`). Des auditeurs peuvent lui être attachés pour gérer les clics de l'utilisateur – voir plus loin.

Case à cocher

`JCheckBox` représente une case à cocher, qui peut être sélectionnée ou non. Ses principales propriétés sont le texte qu'elle affiche (`text`) et son état (`selected`).



Bouton « radio »

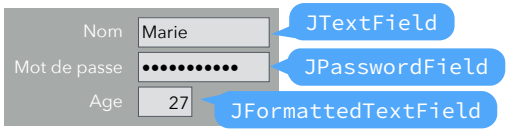
`JRadioButton` représente un bouton « radio », similaire à une case à cocher mais faisant partie d'un groupe (de type `ButtonGroup`) dont un seul peut être sélectionné.
Ses principales propriétés sont les mêmes que celles de `JCheckBox`.

Combo box

`JComboBox` représente une « *combo box* » permettant de choisir une valeur parmi un ensemble prédéfini et éventuellement d'en entrer une autre.

Composant textuel simple

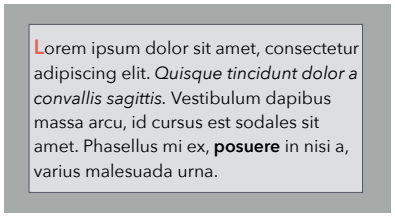
`JTextField` et ses sous-classes `JFormattedTextField` et `JPasswordField` représentent des champs textuels à une ligne et à présentation uniforme (police unique, etc.).
Leur principale propriété est leur texte (`text`), une simple chaîne de caractères de type `String`.



`JFormattedTextField` stocke une valeur non textuelle dans sa propriété `value` et la transforme en texte (et inversement) au moyen d'un formateur qui lui est attaché.

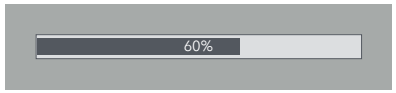
Composant textuel évolué

`JTextArea` est similaire à `JTextField` mais permet les lignes multiples, tandis que `JTextPane` et `JEditorPane` permettent l'affichage – et éventuellement l'édition – de texte mis en page.
Leur principale propriété est le texte qu'ils contiennent (`text`), également disponible en version mise en page (`document`).



Barre de progression

`JProgressBar` représente une barre de progression, permettant de visualiser l'état d'avancement d'une opération de longue durée.
Sa principale propriété est sa valeur (`value`), un entier compris entre 0 et une valeur maximale choisie, indiquant l'état d'avancement actuel.



Macro-composants

En plus des composants de base « simples » décrits précédemment, Swing offre des macro-composants souvent utiles, construits à partir des composants de base :

- `JColorChooser`, qui permet de choisir une couleur de différentes manières (via une palette, en choisissant ses composantes dans différents modèles, etc.),
- `JFileChooser`, qui permet de choisir un ou plusieurs fichiers ou répertoires, existants ou non, en naviguant dans le système de fichiers.

Ces deux macro-composants peuvent s'utiliser comme des composants normaux ou dans des boîtes de dialogue.

Composants structurés

Finalement, Swing offre des composants permettant d'afficher des données structurées : listes, tableaux, hiérarchie arborescente.
Ces composants sont plus complexes à utiliser que les autres étant donné la nature des données qu'ils affichent.
Pour les obtenir, ces composants utilisent un **modèle** qu'on leur fournit.

Modèles

Composants complexes

Plusieurs composants de base permettent d'afficher et de manipuler des données complexes, par exemple :

- `JList` affiche une liste d'éléments,
- `JTree` affiche une hiérarchie arborescente,
- `JTable` affiche une table bidimensionnelle de cellules,
- etc.

Où stocker les données affichées par ces composants et quel type leur attribuer ?

Modèles

Une solution serait de stocker les données directement dans les composants. Par exemple, le composant `JList` pourrait stocker la liste des éléments qu'il affiche.
Cette solution est mauvaise car elle va à l'encontre de la séparation conseillée par le patron *MVC* : les données à afficher font partie du modèle et ne doivent donc pas être stockée dans la vue.
Une meilleure solution est de faire en sorte que les composants puissent directement utiliser les données du modèle.

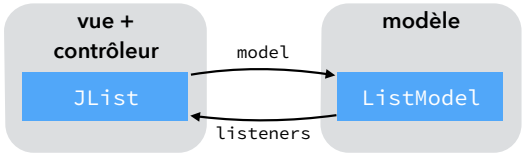
Modèles

Swing offre la possibilité d'attacher à chacun des composants complexes précités un objet appelé **modèle** (*model*) et représentant les données sous-jacentes. Ce modèle doit être observable au sens du patron *Observer* pour permettre au composant de se mettre à jour automatiquement.
Le type de ces modèles est spécifié par des interfaces fournies par Swing, et il est donc généralement nécessaire d'écrire un adaptateur pour adapter les données du programme aux types attendus par Swing.

Exemple : `JList`

Le composant `JList` affiche une liste d'éléments et permet leur sélection. Il doit donc pouvoir effectuer les opérations suivantes sur la liste à afficher :
- obtenir sa taille,
- obtenir l'élément d'indice donné,
- observer la liste pour être informé des changements.

Ces opérations sont regroupées dans l'interface `ListModel`, représentant un modèle de liste.



`ListModel`

L'interface `ListModel` offre d'une part des méthodes permettant de gérer l'ensemble des observateurs – appelés ici auditeurs (*listeners*) – et d'autre part des méthodes de consultation de la liste de valeurs.

```
public interface ListModel<E> {
  void addListDataListener(LDL l);
  void removeListDataListener(LDL l);

  int getSize();
  E getElementAt(int i);
}
```

(Le type `ListDataListener` a été abrégé `LDL` pour des questions de présentation).

`AbstractListModel`

Pour faciliter l'écriture de modèles de listes, Swing offre la classe `AbstractListModel` qui met en œuvre les méthodes gérant les auditeurs, laissant uniquement les méthodes `getSize` et `getElementAtIndex` abstraites.
Cette classe fournit de plus des méthodes dont le nom commence par `fire…` et permettant d'informer les auditeurs de changements dans le modèle – similaires à la méthode `notifyObservers` des sujets du patron *Observer*.

ListDataListener

L'interface `ListDataListener` représente un observateur de modèle de liste, appelé ici auditeur.

```
public interface ListDataListener {
  void intervalAdded(ListDataEvent e);
  void intervalRemoved(ListDataEvent e);
  void contentsChanged(ListDataEvent e);
}
```

Plutôt qu'une unique méthode de mise à jour (`update`), il en possède trois. La dernière est générale, tandis que les deux premières représentent des cas particuliers – ajout et suppression d'éléments – qui ont été jugés assez fréquents pour valoir la peine d'être traités séparément.

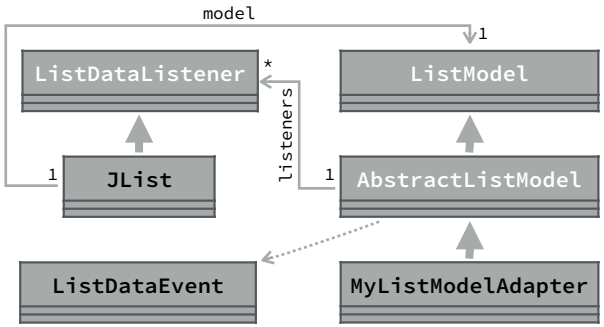
ListDataEvent

La classe `ListDataEvent` contient les informations liées au changement du contenu de la liste observée, à savoir :

- le type de changement – ajout d'un intervalle, suppression d'un intervalle, changement général,
- l'index du premier élément affecté par le changement,
- l'index du dernier élément affecté par le changement.

Diagramme de classes

Le diagramme de classes ci-dessous illustre – de manière légèrement simplifiée – les classes qui seraient utilisées par une application incluant un composant `JList`.



Composants personnalisés

Composants personnalisés

Même si les composants prédéfinis couvrent un grand nombre de besoins, ils ne sauraient les couvrir tous.
Lorsqu'une application nécessite un composant ne figurant pas parmi les prédéfinis, il est possible de le définir sous la forme d'une nouvelle sous-classe de `JComponent`.
Les principales méthodes à redéfinir dans ce cas sont :

- `getPreferredSize`, `getMinimumSize` et `getMaximumSize`, qui permettent aux gestionnaires d'agencement de le dimensionner correctement,
- `paintComponent`, qui dessine le composant.

get…Size

Les méthodes `getPreferredSize`, `getMinimumSize` et `getMaximumSize` sont utilisées par certains gestionnaires de mise en page – mais pas tous – pour déterminer la taille du composant.
Il peut donc être utile de les redéfinir lorsque la taille du composant ne peut pas être arbitraire.

paintComponent

La méthode `paintComponent` est automatiquement appelée par Swing chaque fois que le composant doit être (re)dessiné, par exemple lorsqu'il apparaît pour la première fois à l'écran.
Cette méthode reçoit en argument le contexte graphique à utiliser pour effectuer le dessin. Le repère qui lui est associé est celui du composant.

Exemple

Le composant ci-dessous affiche un rectangle rouge centré sur fond noir :

```
public final class RonB extends JComponent {
  @Override
  protected void paintComponent(Graphics g) {
    int w = (int)getBounds().getWidth();
    int h = (int)getBounds().getHeight();
    g.setColor(Color.BLACK);
    g.fillRect(0, 0, w, h);
    g.setColor(Color.RED);
    g.fillRect(w/4, h/4, w/2, h/2);
  }
}
```

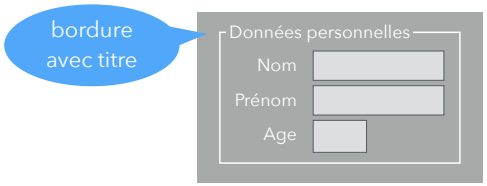
repaint

Il peut arriver qu'un composant doive être redessiné, p.ex. suite à un changement des données du modèle qu'il représente.
Cela ne doit en aucun cas être fait en appelant directement `paintComponent`, cette méthode étant uniquement destinée à être appelée par Swing.
Au lieu de cela, il convient d'appeler la méthode `repaint` du composant, afin de demander à Swing de le redessiner (via sa méthode `paintComponent`) aussi vite que possible.
A noter que Swing redessine automatiquement les composants dont la taille a changé, sans qu'un appel à `repaint` ne soit nécessaire.

Bordures

Bordures

Il est possible d'ajouter une **bordure** (*border*) à n'importe quel composant au moyen de la méthode `setBorder` de `JComponent`.
Il est assez fréquent d'ajouter une bordure à un panneau (`JPanel`) afin de grouper visuellement un certain nombre de composants liés logiquement. Par exemple :



BorderFactory

Contrairement à d'autres objets Swing, les bordures ne doivent pas être créées directement au moyen de l'énoncé `new`. Au lieu de cela, il convient d'appeler l'une des méthodes fabriques de la classe `BorderFactory`, par exemple :

- `createLineBorder` prend une couleur et une épaisseur de trait et retourne une bordure composée d'une simple ligne,
- `createTitledBorder` prend une bordure existante et une chaîne, qu'elle lui ajoute en titre,
- etc.

Gestion des événements

Dire que l'interface est bloquée tant que le gestionnaire ne retourne pas

Prog. événementielle

Un programme doté d'une interface graphique ne fait généralement qu'attendre que l'utilisateur interagisse avec celle-ci, réagit en conséquence, puis se remet à attendre.
Chaque fois que le programme est forcé de réagir, on dit qu'un **événement** (*event*) s'est produit.
Cette caractéristique des programmes graphiques induit un style de programmation particulier nommé **programmation événementielle** (*event-driven programming*).
Ce style se retrouve dans toutes les applications dont le but principal est de réagir à des événements externes, p.ex. les serveurs (Web et autres).

Boucle événementielle

Au cœur de tout programme événementiel se trouve une boucle traitant successivement les événements dans leur ordre d'arrivée. Cette boucle, nommée **boucle événementielle** (*event loop*), pourrait s'exprimer ainsi en pseudo-code :

```
tant que le programme n'est pas terminé
  attendre le prochain événement
  traiter cet événement
```

Cette boucle est souvent fournie par une bibliothèque et ne doit dans ce cas pas être écrite explicitement. Il en va ainsi de la plupart des bibliothèques de gestion d'interfaces graphiques, dont Swing.

Gestion des événements

Si la boucle événementielle peut être identique pour tous les programmes, la manière dont les différents événements sont gérés est bien entendu spécifique à chaque programme.
Dès lors, il doit être possible de définir la manière de traiter chaque événement. Cela se fait généralement en écrivant, pour chaque événement important, un morceau de code le gérant, appelé le **gestionnaire d'événement** (*event handler*).
Ce gestionnaire est associé, d'une manière ou d'une autre, à l'événement.

Gestion des événements

La boucle événementielle est séquentielle, dans le sens où un événement ne peut être traité que lorsque le traitement de l'événement précédent est terminé.
Dans le cas d'une interface graphique, cela signifie que celle-ci est totalement bloquée tant et aussi longtemps qu'un événement est en cours de traitement.
Pour cette raison, les gestionnaires d'événements doivent autant que possible s'exécuter rapidement. Si cela n'est pas possible, il faut utiliser la concurrence pour effectuer les longs traitements en tâche de fond – ce qui sort du cadre de ce cours.

Boucle événementielle Swing

Dans une application graphique Swing, la boucle événementielle s'exécute dans un fil d'exécution (*thread*) séparé, nommé le **fil de gestion des événements** (*event dispatching thread* ou *EDT*).
Ce fil d'exécution est démarré automatiquement lors de l'utilisation de Swing, rendant la boucle événementielle particulièrement invisible au programmeur. Elle n'en existe pas moins !
De manière générale, tous les appels à des méthodes de Swing doit se faire depuis le fil de gestion des événements. En particulier, toute la création de l'interface doit s'y faire.

Auditeurs

Dans Swing, les gestionnaires d'événements sont des objets, appelés **auditeurs** (*listeners*), implémentant une interface qui définit une ou plusieurs méthodes de gestion d'événement.
Les auditeurs sont attachés à une source d'événements – généralement un composant – et leurs méthodes sont appelées chaque fois qu'un événement se produit.
(Les auditeurs sont assez similaires aux observateurs du patron *Observer*, la méthode `update` de ces derniers gérant l'événement « l'état du sujet a changé »).

Exemple

La classe ci-dessous illustre la structure typique d'un programme Swing. L'interface graphique est créée par la méthode `createUI`, appelée indirectement via `invokeLater` afin de garantir qu'elle s'exécute sur le fil de gestion des événements.

```
public final class Main {
  private static void createUI()
    { /* … voir page suivante */ }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(
      () -> createUI());
  }
}
```

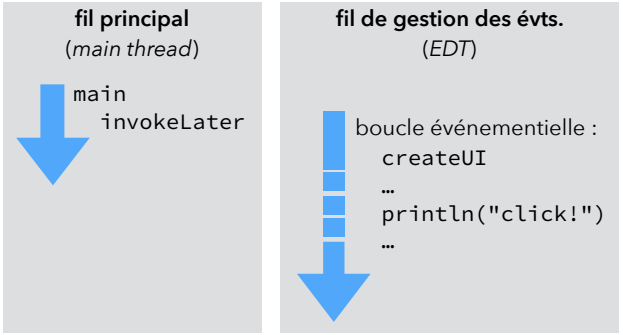
Exemple

La méthode `createUI` créée une fenêtre et y ajoute un bouton dont l'auditeur affiche un message à l'écran lorsqu'on lui clique dessus.

```
private static void createUI() {
  JFrame w = new JFrame();
  w.setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE);
  JButton b = new JButton("click me!");
  b.addActionListener(
    e -> System.out.println("click!"));
  w.getContentPane().add(b);
  w.pack();
  w.setVisible(true);
}
```


Exemple

La figure ci-dessous illustre l'exécution des différentes parties du programme sur les deux fils existants.



Objets événements

Lorsqu'un événement se produit, Swing collecte les informations à son sujet dans un objet passé à l'auditeur. Par un léger abus de langage, cet objet lui-même est nommé événement (*event*).

Généralement, à chaque type d'événement correspond un type d'auditeur et un type d'objet-événement.

Types d'auditeurs

Swing définit un grand nombre de types d'auditeurs, en fonction du type d'événement qu'ils écoutent. Les principaux types d'auditeurs sont :

- les auditeurs d'action (*action listeners*),
- les auditeurs de souris (*mouse listeners*),
- les auditeurs de menu (*menu listeners*),
- les auditeurs de clavier (*key listeners*),
- les auditeurs de cible de saisie (*focus listeners*).

Seuls les deux premiers sont examinés ici, les autres ayant un fonctionnement similaire.

ActionListener

L'interface (fonctionnelle) `ActionListener` représente un auditeur à l'écoute des événements de type « action ». Ces événements se produisent p.ex. lorsque l'utilisateur clique sur un bouton, sélectionne un menu ou presse *Entrée* dans un champ textuel.

```
public interface ActionListener {
  void actionPerformed(ActionEvent e);
}
```

La classe `ActionEvent` contient différentes informations au sujet de l'événement, p.ex. les touches de modifications (*Control*, *Shift*, …) pressées au moment du déclenchement de l'action.

ActionListener

Un auditeur de type `ActionListener` peut être attaché à plusieurs types de composants, principalement les boutons, les champs textuels et les entrées de menus.
Chacun de ces types de composants offre les méthodes `addActionListener` et `removeActionListener` pour ajouter/supprimer un auditeur d'action.

MouseListener

L'interface `MouseListener` représente un auditeur à l'écoute de certains événements liés à la souris : entrée et sortie des bornes du composant, pression et relâchement des boutons :

```
public interface MouseListener {
  void mouseEntered(MouseEvent e);
  void mouseExited(MouseEvent e);
  void mouseClicked(MouseEvent e);
  void mousePressed(MouseEvent e);
  void mouseReleased(MouseEvent e);
}
```

La classe `MouseEvent` contient différentes informations concernant l'événement, p.ex. la position de la souris.

MouseMotionListener

L'interface `MouseMotionListener` représente un auditeur à l'écoute des événements liés au déplacement de la souris, avec un bouton pressé (`mouseDragged`) ou non (`mouseMoved`) :

```
public interface MouseMotionListener {
  void mouseDragged(MouseEvent e);
  void mouseMoved(MouseEvent e);
}
```

Chaque mouvement de la souris constituant un nouvel événement, ceux-ci peuvent être très nombreux. Les auditeurs de type `MouseMotionListener` se doivent donc de traiter rapidement les événements.

MouseWheelListener

L'interface (fonctionnelle) `MouseWheelListener` représente un auditeur à l'écoute des événements liés à la molette de la souris :

```
public interface MouseWheelListener {
  void mouseWheelMoved(MouseWheelEvent e);
}
```

La classe `MouseWheelEvent` étend `MouseEvent` en lui ajoutant des informations liées à la molette : distance et sens de rotation, etc.

Mouse…Listener

Les trois types d'auditeurs liés à la souris peuvent être attachés à (et détachés de) n'importe quel composant au moyen des méthodes `addMouse…Listener` et `removeMouse…Listener`.

MouseAdapter

La classe héritable `MouseAdapter` fournit une mise en œuvre par défaut des interfaces `MouseListener`, `MouseMotionListener` et `MouseWheelListener`. Elle implémente la totalité des méthodes de ces interfaces, en ne faisant rien dans tous les cas.
Attention : malgré son nom, cette classe n'est pas un adaptateur au sens du patron *Adapter*. Un nom plus conforme aux conventions généralement utilisées dans la bibliothèque Java serait `AbstractMouseListener`.