# Sokoban : étape 5

Dans ce TP, nous allons voir comment une interface graphique typique gère le dessin et le rafraîchissement dans sa zone d'affichage. Cela ne correspond pas à la manière usuelle d'aborder les interfaces graphiques, l'assemblage de composant graphiques étant généralement vu en premier. Ceci est justifié par l'objectif de rendre rapidement le Sokoban fonctionnel afin d'y implanter un jeu automatique, qui sera étudié en ALGO. Rassurez vous, l'assemblage de composants graphiques sera abordé ultérieurement.

### 1 - Un peu de culture : AWT, SWING et JavaFX

Swing est la bibliothèque graphique standard de java. Elle a remplacé AWT depuis la version 1.2 du langage. AWT, l'ancienne bibliothèque graphique, était construite au dessus d'une bibliothèque graphique native du système d'exploitation. Ce choix causait malheureusement des problèmes de portabilité d'un système à l'autre. C'est pourquoi, elle a rapidement été remplacée par Swing, qui propose *grosso modo* les mêmes fonctionnalités mais est entièrement écrite en java (donc portable) au dessus des couches les plus basses d'AWT (fenêtres et événements uniquement).

Plus récemment, un successeur à Swing a été proposé : JavaFX. Il s'agit d'une évolution de JavaFX script sorti en 2008. Depuis 2011, JavaFX est doté d'une API utilisable directement depuis un programme en Java. Par rapport à Swing, il apporte un certain nombre d'améliorations :

- une API plus moderne, générique avec un nommage plus uniforme (qui a malgré tout quelques petits défauts comme, par exemple, l'impossibilité pour une classe d'être, de manière simple, écouteur de plusieurs types d'évènements) ;
- un moteur conçu pour tirer parti de l'accélération matérielle disponible sur la machine hôte (même si ce support est encore partiel et si, en pratique, JavaFX n'est pas encore plus efficace que Swing) ;
- une gestion simplifiée du dessin, des animations et des effets (même si les grands principes de fonctionnement sont les même que ceux de Swing et si un programme utilisant l'un de ces deux toolkits aura à peu près la même architecture).

Malheureusement, après avoir intégré JavaFX dans le JDK 8 en 2014, Oracle annonce en 2018 que le JDK 11 ne l'intègrera plus et que la compagnie ne supportera plus cette technologie. Depuis, JavaFX est développé par le consortium OpenJFX comme un module intégrable au JDK et une version binaire peut être téléchargée auprès de Gluon (https://gluonhq.com). En conséquence, un programme JavaFX est aujourd'hui moins portable et un environnement de développement muni de JavaFX plus pénible à installer. Pour ces raisons, le Sokoban sera développé en Swing. Si vous souhaitez tout de même vous renseigner sur JavaFX, une version JavaFX du Sokoban est disponible sur demande auprès du responsable de l'UE (attention, les étapes ne suivent pas tout à fait le même découpage que la version Swing).

### 2 - Affichage et rafraîchissment

L'interface graphique d'une application est évidemment construite et affichée par l'application elle-même. Cependant, dans un système graphique classique, le contenu d'une fenêtre n'est pas sauvegardé par le système. Cela signifie que lorsqu'une fenêtre est redimensionnée, masquée/démasquée par une autre fenêtre ou iconifiée/déiconifiée, il faut que le système graphique puisse avertir l'application qu'elle doit rafraîchir son affichage.

Une application graphique bien construite DOIT donc définir, pour chaque élément qu'elle souhaite afficher, une méthode qui redessine TOUT l'affichage de cet élément et qui puisse être appelée par le système graphique en cas de besoin. En Swing, la construction d'une application graphique se fait en ajoutant des `JComponent` (Composant graphique duquel sont dérivés les autres) à une fenêtre. Lors d'un changement potentiel d'une partie de la fenêtre affichée, le système demande à la fenêtre de se redessiner. Pour cela, la fenêtre appelle la méthode `paintComponent` des `JComponent`s qu'elle contient. Pour les composants usuels (boutons, cases à cocher, boutons radio,

...) l'utilisateur n'a pas à s'en préoccuper : ces composants sont fournis avec une méthode `paintComponent` appropriée. En revanche si nous souhaitons dessiner librement dans une partie de la fenêtre, nous allons devoir gérer l'affichage nous mêmes : c'est l'objectif principal de ce TP.

Pour dessiner dans une fenêtre, il suffira donc d'y ajouter un composant qui pourra prendre tout l'espace disponible s'il est seul. Ce composant sera un objet d'une classe dérivée de `JComponent`, dans laquelle nous aurons redéfini la méthode `paintComponent`. Cette méthode prend en argument un objet de classe `Graphics` qui est une description de la portion d'écran sur laquelle notre composant peut dessiner et qui nous est fourni par le système.

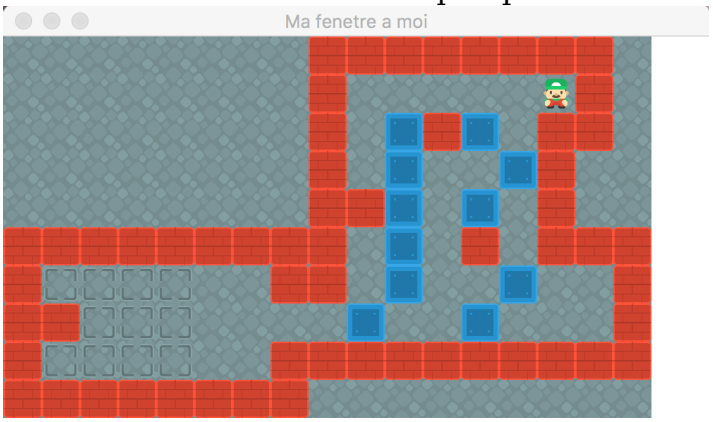
**Remarque culturelle :** avec les progrès technologiques, les cartes graphiques modernes disposent de suffisamment de mémoire pour stocker en permanence le contenu de plusieurs fenêtres (par exemple sur une carte bas de gamme disposant de 1Go de mémoire, on peut stocker environ 128 fenêtres de 1920x1080 pixels). Cette mémoire est exploitée par les systèmes modernes et permet à la fois d'accélérer le rendu et d'ajouter des effets comme la transparence au contenu des fenêtres. Cependant cela ne change rien au problème qui nous intéresse aujourd'hui : des opérations comme le redimensionnement ou l'évolution du contenu de l'affichage résultant d'animations nous obligent à programmer comme si le contenu d'une fenêtre n'était pas sauvegardé.

### Exercices

- Récupérez les fichiers DemoFenetre.java et AireDeDessin.java et lisez les pour comprendre leur contenu. Ils constituent un petit exemple permettant d'ouvrir une fenêtre graphique et d'y dessiner une image que vous pourrez retrouver dans Images.zip (ces images sont libres de droit, tirées de ce site). Attention, si vous avez oublié comment organiser vos fichiers, reportez vous à la section 2bis de la première APNEE ;
- En vous inspirant de l'exemple précédent, écrivez un programme permettant d'afficher dans une fenêtre graphique le niveau dont le numéro est donné en argument. Voici quelques éléments qui pourront vous aider à atteindre cet objectif :
  - Encapsulez votre lecteur de niveau dans une classe `Jeu` qui nous servira, par la suite, à orchestrer le déroulement du jeu. Pour l'instant implémentez y les méthodes :
    - Niveau niveau() : permettant de récupérer le niveau en cours ;
    - boolean prochainNiveau() : permettant d'avancer d'un niveau.

    Utilisez cette classe depuis le programme principal pour vous placer sur le bon niveau ;
  - En vous inspirant d'`AireDeDessin`, écrivez une classe `NiveauGraphique` qui, étant donné un `Jeu`, dessine le niveau courant dans sa méthode `paintComponent` ;
  - En vous inspirant de `DemoFenetre`, écrivez une classe `InterfaceGraphique` qui sera utilisée par le programme principal pour ouvrir une nouvelle fenetre et y tracer le niveau d'un `Jeu` donné.

  Testez votre implémentation en redimensionnant le fenêtre et assurez vous que le niveau se redimensionne correctement. Vous devriez obtenir quelque chose comme :




- En vous inspirant de la méthode :

```
public void toggleFullscreen() {
        GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
        GraphicsDevice device = env.getDefaultScreenDevice();
        if (maximized) {
                device.setFullScreenWindow(null);
                maximized = false;
        } else {
```

```
                    device.setFullScreenWindow(frame);
                    maximized = true;
                }
        }
```

faites basculer votre application en plein écran.

### 3 - Un peu de factorisation

Nous avons maintenant des chargements de fichiers à plusieurs endroits dans le code : le chargement des niveaux dans le programme principal et le chargement des images dans l'affichage d'un niveau. Les lectures de ces différents fichiers sont très différentes (notre lecteur de niveaux d'un part et `javax.imageio.ImageIO.read` d'autre part), mais leur ouverture est similaire (nouveau `FileInputStream`). Il n'y a aucun point commun entre les parties du code ouvrant des fichiers, mais le système de fichier, lui, est global et donc commun.

**Questions :**

- créez un paquetage nommé `Global` et créez dans ce paquetage une classe `Configuration` qui nous servira à placer nos méthodes globales.
- si ce n'est pas déjà le cas, créez un répertoire `res` à la racine de votre projet de Sokoban dans lequel vous placerez tous les fichiers de données du jeu. Ce répertoire nous permettra, plus tard, de produire facilement une archive exécutable à partir de notre code.
- créez une méthode nommée `ouvre` dans `Configuration` permettant d'ouvrir un fichier de chemin relatif dans `res` donné. Cette méthode devra renvoyer un `InputStream` ou afficher un message et mettre fin au programme en cas d'erreur.
- remplacez toutes les ouvertures de fichier par un appel à `ouvre`
- de manière analogue, créez deux méthodes dans `Configuration` permettant de gérer les messages d'avertissement et d'erreur et remplacez tous les affichages de votre programme par un appel à une de ces deux méthodes.
- pour finir, ajoutez un réglage dans `Configuration` permettant de désactiver les messages d'avertissement ou bien de désactiver tous les messages.