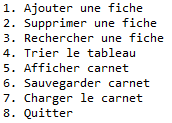
# Rapport L50/70 : Harmonisation Développement – Programmation Orientée Objet

## Partie 1 : Carnet d’adresse

### Eclipse

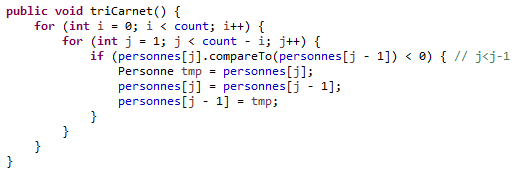
L’application que nous avons développés sur Eclipse en java est un gestionnaire de carnet d’adresse (nom , prénom et numéro de telephone des personnes). Le but étant déjà de pouvoir ajouter, supprimer ou bien rechercher une personne par exemple mais surtout de sauvegarder et charger le carnet. Le menu ressemble à ceci :



Niveau programation on a 3 classes :

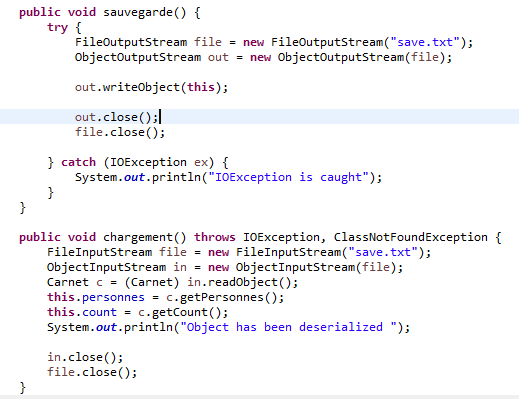
* La classe « personne » qui définit simplement une personne par des abribus.
* La classe « carnet » qui possède un tableau de personnes et les méthodes pour agir sur le carnet (supprimer, ajouter, trier …).
* La classe « main » où se trouvent les utilisations des méthodes.

Pour les méthodes certaines sont assez basique, mais certaines mérite d’être étudiée, par exemple celle du tri où on a choisi un tri à bulle comme ceci :

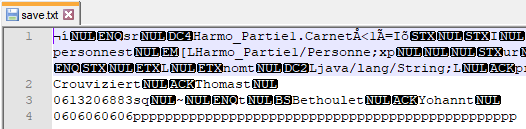


On compare chaque éléments du tableau 1 à 1 et en les échangeant si besoin.

Deux autres fonctions sont importante, celle de sauvegarde et de chargement :



Le carnet est sauvegardé dans un fichier « save.txt » (que si l’on ouvre avec un éditeur de texte est très illisible) :

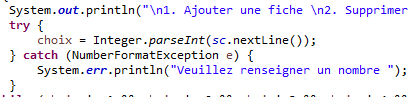


Mais le chargement fonctionne correctement et permet de récupérer tous le carnet d’adresse n’importe quand.

Pour terminer il faut savoir que la gestion d’erreur a été prise en compte pour :

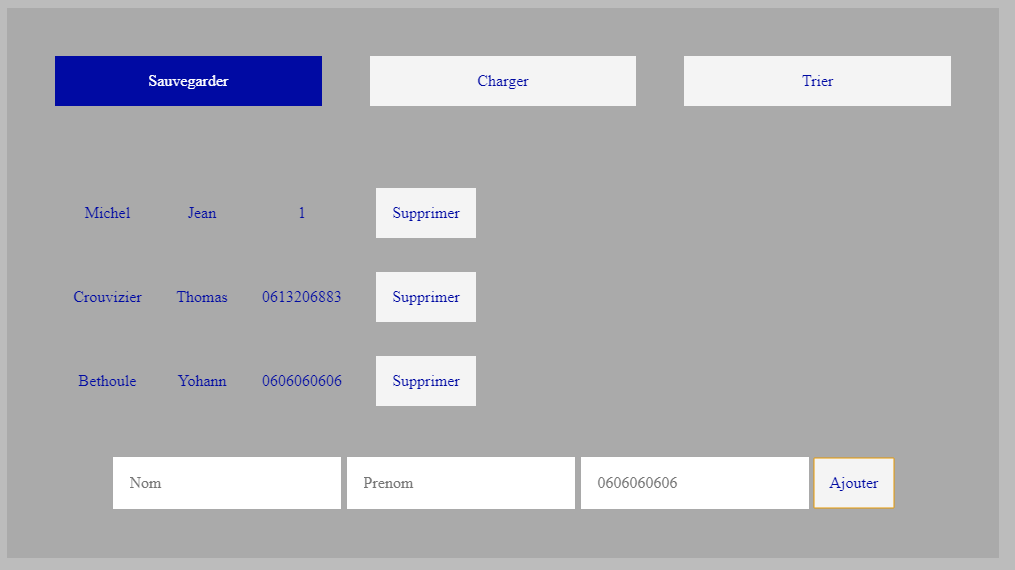
* Le choix de l’option du menu (forcément un chiffre et forcement entre 1 et 8, sinon redemande un choix)
* Lors de la suppression demande un chiffre compris dans le nombre de personne présent dans le carnet
* Lors de l’ajout si quelqu’un rentre des valeurs qui ne marcheront pas
* Lors de la sauvegarde en cas d’erreur

Les erreurs sont toutes gérées avec des « try/catch » de ce type :



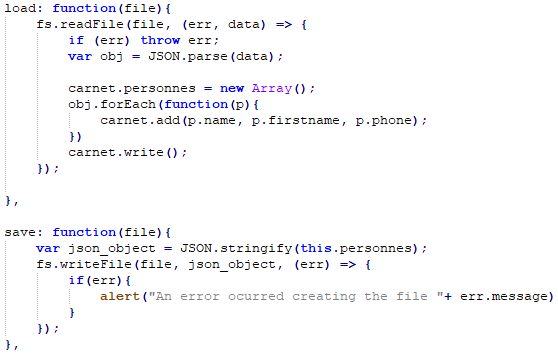
### electron

Même objectif que la partie d’avant, avec une base graphique maintenant ressemblant à ceci :



Le but graphiquement parlant était d’être le plus simple possible pour l’utilisateur, on retrouve toutes les fonctions que possédais la version Eclipse avec l’avantage de pouvoir nommé et placé les fichiers sauvegarder comme on le souhaite.

Niveau code c’est très ressemblant de l’application précédente, sauf pour la sauvegarde et chargement fait avec du Json :



## Partie 2 : Visualiseur d’images

### JavaFx

L’application que nous avons développée en JavaFx permet d’ouvrir un fichier image, d’en afficher le contenu et d’ensuite apporter des modifications à celle-ci, avant d’éventuellement la sauvegarder où il le souhaite dans son système de fichiers. L’utilisateur peut ainsi passer une image en niveaux de gris, ou en extraire un rendu rouge, vert ou bleu. Pour sélectionner le fichier à ouvrir ainsi que l’emplacement de sauvegarde, notre programme ouvre une fenêtre de sélection de fichiers.

Exemple type d’utilisation :

1. L’utilisateur lance l’application :
2. L’utilisateur appuie sur le bouton « Open », qui ouvre une fenêtre de dialogue permettant de sélectionner un fichier à ouvrir :
3. L’image s’affiche dans le programme :
4. L’utilisateur appuie sur le bouton pour transformer l’image en niveaux de gris, après un court temps de chargement signalé par une barre de progression, l’image transformée est affichée :
5. L’utilisateur ouvre le menu des rendus RGB, et sélectionne une couleur dans le menu :
6. Comme pour la transformation en niveaux de gris, l’image s’affiche après l’opération :
7. Enfin, l’utilisateur est satisfait et souhaite enregistrer son dur labeur. Il clique pour cela sur le bouton save, qui ouvre une nouvelle fois une fenêtre de dialogue pour sélectionner l’emplacement auquel sauvegarder le fichier image :
8. Le fichier est sauvegardé, l’utilisateur peut fermer le programme et ouvrir le fichier dans son visualiseur d’image par défaut :

L’architecture de l’application est basée sur un modèle MVC : notre projet contient une vue FXML, une classe métier ImageEditor ainsi qu’une classe Controller faisant la liaison entre les deux.

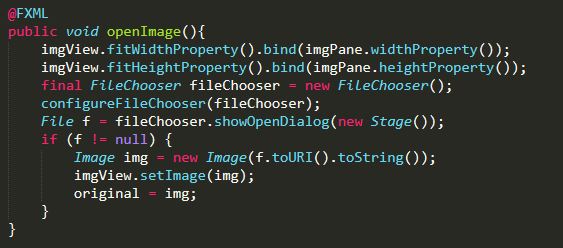
Le Controller peut agir sur la vue, ainsi que réagir à des évènements provenant de celle-ci.

Exemple : ouvrir une image

Dans ma vue, je crée un bouton pour permettre à l’utilisateur d’ouvrir une image :



On voit que j’y associe une méthode en lui donnant l’attribut « onAction= ‘#openImage’ ». Je déclare cette méthode avec le préfixe @FXML dans mon Controller :



L’objet imgView auquel je fais référence dans cette méthode est en fait un objet ImageView représentant l’élément ImageView qui affichera l’image dans la vue. Cet attribut est déclaré plus haut dans le Controller :

La méthode openImage() du Controller est donc exécutée chaque fois que l’utilisateur clique sur le bouton, et agit sur la vue de la manière suivante :

* Elle définit la hauteur et la largeur de imgView pour l’adapter à la taille de l’image.
* Elle crée une instance de fileChooser qu’elle configure par une autre méthode définie dans le Controller, et cet objet FileChooser va se charger d’ouvrir une fenêtre de dialogue à l’utilisateur pour lui permettre d’ouvrir le fichier souhaité.
* Lorsque la fenêtre de dialogue se ferme, la méthode showOpenDialog() renvoie un objet File si un fichier a été sélectionné, ou null si l’utilisateur a fermé la fenêtre sans sélectionner de fichier. Si un fichier a bien été retourné, on instancie un objet Image a partir de l’image contenue dans le fichier et on l’affiche en l’assignant à imgView.

La partie la plus importante et la plus difficile de notre application se trouve dans le package model, il s’agit de la classe ImageEditor, qui dérive de la classe Service<Image>.

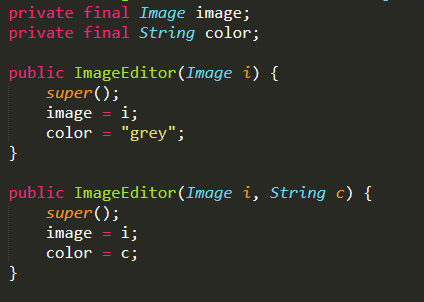
La classe Service<V> est une classe abstraite permettant de gérer un thread. Elle possède une méthode abstraite createTask(), que toute classe fille se doit d’implémenter : cette méthode permet d’instancier un objet Task<V> qui va exécuter une tâche de fond sur un thread secondaire.

*Note : dans les classes Task<V> et Service<V>, V est le type de retour du thread, qui sera déclaré à l’instanciation de l’objet. Ainsi, la méthode Task<Image> createTask() crée et retourne un objet Task qui retourne un objet Image.*



La méthode retourne donc un objet Task<Image>, qui redéfinit sa méthode call(). Le contenu de cette méthode sera le traitement exécuté en arrière-plan par le programme, ici créer et retourner une nouvelle image transformée par les filtres choisis par l’utilisateur.

Afin de pouvoir y accéder dans la méthode call(), la classe ImageEditor possède deux attributs :



C’est à partir de ces deux attributs que les traitements sont exécutés :

Un objet PixelReader va s’occuper de lire l’image pixel par pixel.

Un objet WritableImage aux mêmes proportions que l’image à modifer servira de canevas au programme pour dessiner une nouvelle pixel par pixel à partir de l’ancienne transformée.

Un objet PixelWriter écrire les pixels dans l’instance de WritableImage.

On crée un compteur à 0, et on lit la couleur de chaque pixel de l’image un à un. Selon la couleur choisie par l’utilisateur, un filtre particulier est appliqué à chaque pixel :

* Si l’utilisateur choisit une couleur rouge, bleue ou verte :
* Si l’utilisateur choisir de transformer l’image en niveau de gris :

### Electron

Nous avons ensuite développé une application similaire en utilisant Electron.

## Conclusion

Bien que leurs noms soient semblables et que tous deux soient basés sur le paradigme de la Programmation Orientée Objet, les langages Java et Javascript possèdent des différences marquées. En effet, bien qu’en JS, tout soit objet, cela n’apparaît pas aussi clairement dans le code qu’en Java.

Là où un programme Java est composé uniquement de classes bien définies, organisées en packages explicites et possédant un typage statique et une encapsulation voyante et restrictive mais sécuritaire, il est rare en Javascript natif de définir une classe, les typages dynamiques rendent le polymorphisme beaucoup plus transparent qu’en Java.