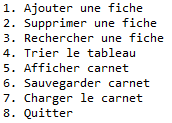
# Rapport L50/70 : Harmonisation Développement – Programmation Orientée Objet

## Partie 1 : Carnet d’adresse

### Java

L’application console que nous avons développé en java est un gestionnaire de carnet d’adresses (nom, prénom et numéro de téléphone des personnes). L’utilisateur doit pouvoir ajouter, supprimer ou bien rechercher une personne, mais aussi sauvegarder et charger le carnet dans un fichier. Le menu ressemble à ceci :

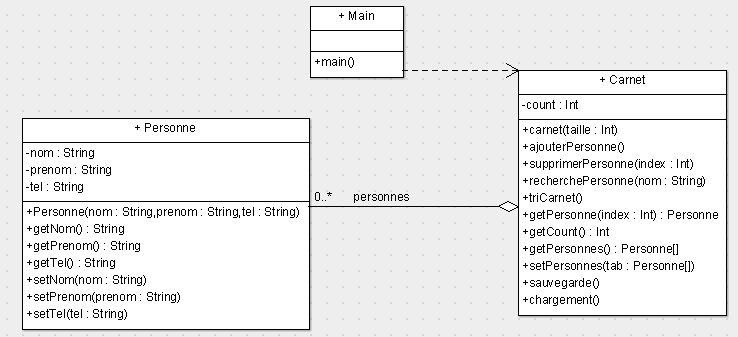


1. Menu de l’application

L’application est composée de 3 classes :

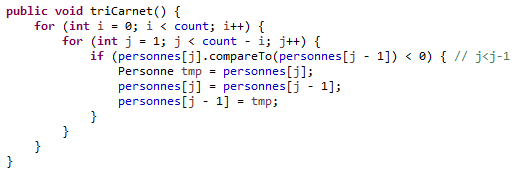
* La classe « Personne » qui définit simplement une personne par des abribus.
* La classe « Carnet » qui possède un tableau de personnes et les méthodes pour agir sur le carnet (supprimer, ajouter, trier …).
* La classe « Main » où se trouvent les utilisations des méthodes.

Voici le diagramme de classe :



1. Diagramme de classe

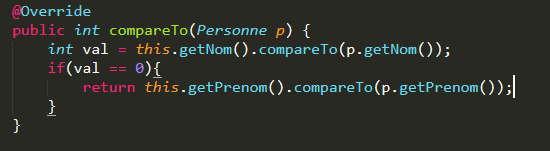
Parmi les méthodes certaines sont assez basiques, mais certaines méritent d’être étudiées, par exemple celle du tri où on a choisi un tri à bulle comme ceci :



1. Code du tri du carnet

L'algorithme parcourt le tableau et compare les éléments consécutifs. Lorsque deux éléments consécutifs ne sont pas dans l'ordre, ils sont échangés. Après un premier parcours complet du tableau, le plus grand élément est forcément en fin de tableau, à sa position définitive. On parcourt ensuite les (t-1) premiers éléments du tableau, avec t la taille du tableau. On reproduit ainsi ce traitement t fois, et le tableau est entièrement trié.

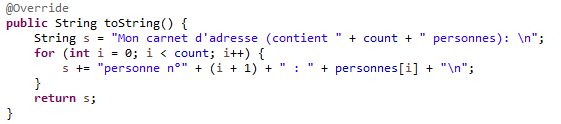
La méthode utilisé pour les comparaisons est « compareTo » qui est une méthode de la classe Object de java, et qu’on l’on redéfini de façon à comparer notre objet à un autre objet Personne, en comparant leur nom de famille par ordre alphabétique, d’abord sur le nom de famille puis par prénom en cas de redondance :



1. Méthode compareTo

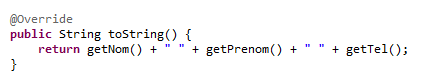
Il en a été fait de même pour la méthode « toString » qui permet de retourner un String représentant l’objet.

Carnet :



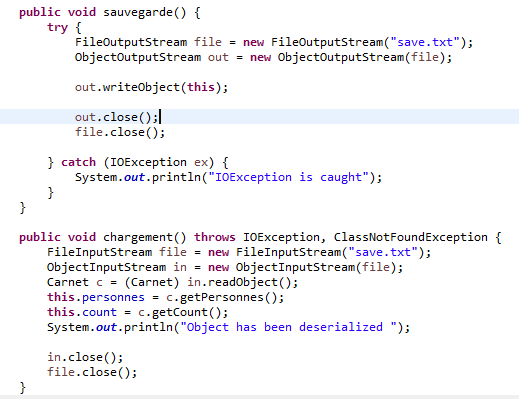
1. Méthode toString du carnet

Personne :



1. Méthode toString de Personne

Deux autres méthodes sont importantes, celle de sauvegarde et de chargement :

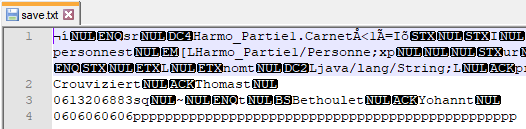


1. Code des fonctions de sauvegarde/chargement du fichier save.txt

Sauvegarde :

On passe par les méthodes de gestion de fichier de java pour ouvrir un fichier que l’on crée, écrire le carnet à l’intérieur et on le referme

Le carnet est sauvegardé dans un fichier « save.txt » (qui n’est pas vraiment human-friendly) :



1. Fichier save.txt

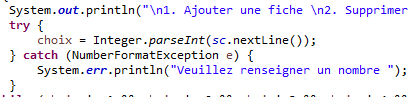
Chargement :

On remplit le carnet par les valeurs récupéré dans le fichier « save.txt » lu par le readObject(). Ces valeurs étant définie comme private dans le carnet, on conserve l’intégralité en ayant implémenté des getteurs/setteurs permettant l’accès au données.

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 gérées avec des « try/catch » de ce type :



1. Try/catch

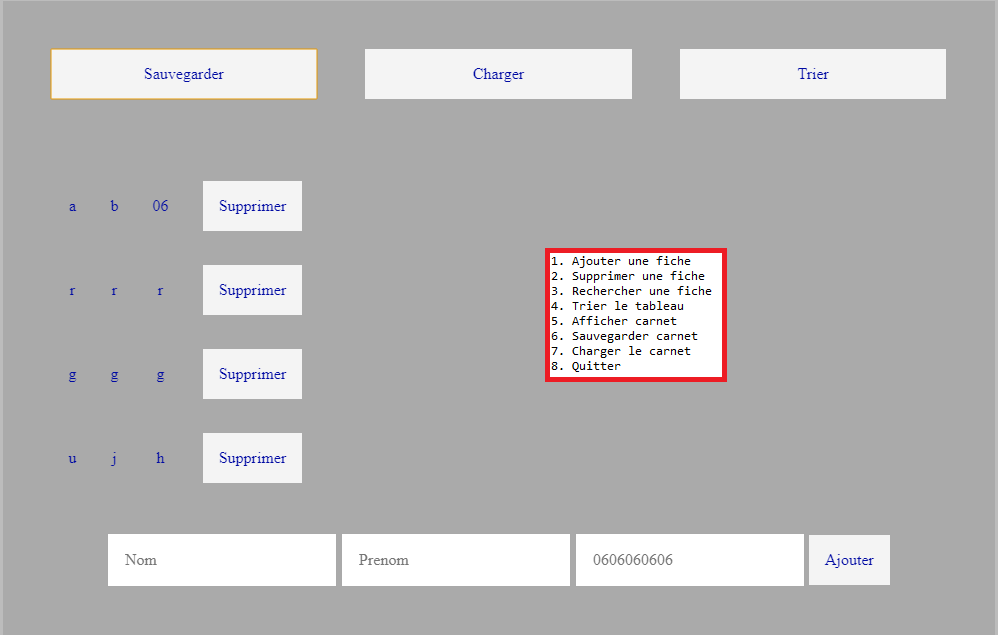


1. exception

Le principe est de tester un cas et selon le résultat potentiellement déclencher une exception. Dans l’exemple ci-dessus, si jamais le carnet est à son nombre maximum de personnes, il indiquera qu’il est plein.

### Electron

L’objectif était le même que dans la partie précédente, mais avec une application graphique codée en javascript avec Electron, un package node.js permettant de développer des applications Bureau en utilisant les technologies du web (HTML/CSS, javascript, node.js). Notre application ressemble à cela :



1. Menu graphique du carnet

Outre le code javascript qui écoute les évènements des boutons pour appeler les fonctions nécessaires, la majorité de notre application est contenu dans un objet nommé ‘carnet’, défini dans notre script.



3. Extrait de la définition de l'objet carnet

Contrairement au Java, nous ne retrouvons pas ici de classe bien définie, possédant des attributs et des méthodes typés et définissant explicitement leur visibilité. Ici, on utilise les particularités du langage javascript pour implémenter ce qui pourrait être défini comme un constructeur d’objet carnet. Cette façon de définir l’objet carnet permit de d’encapsuler correctement nos attributs et méthodes afin d’en protéger l’intégrité.

var carnet = function(){

//attributs privés

Return{

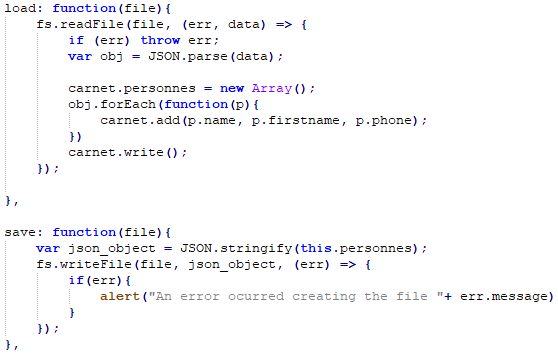
//methodes publics

}

}() ;

En définissant notre objet de cette façon, nous appelons directement le conducteur (avec les dernières parenthèses), instanciant donc le carnet à la volée.

On peut aussi noter qu’on a fait de même pour les objets personne dans la fonction ‘add’.



1. Fonctions de sauvegarde et chargement

On utilise le module node.js ‘fs’ de java script pour interagir avec le système de fichiers afin d’enregistrer ou lire nos fichiers, et le module ‘dialog’ pour ouvrir une boîte de dialogue permettant à l’utilisateur de parcourir ses fichiers et de sélectionner un emplacement.

## 

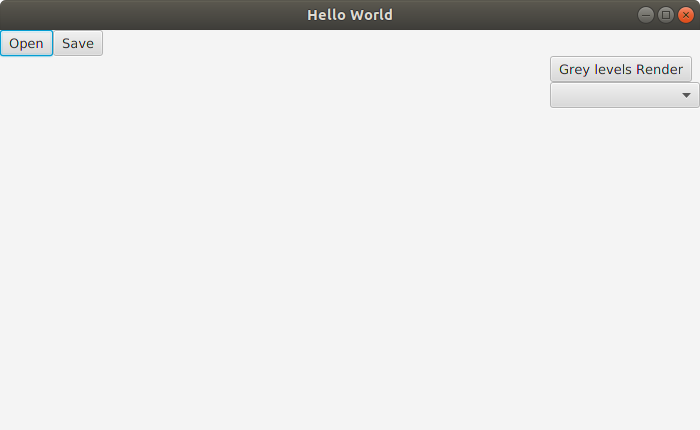
## Partie 2 : Visualiseur d’images

### Java Fx

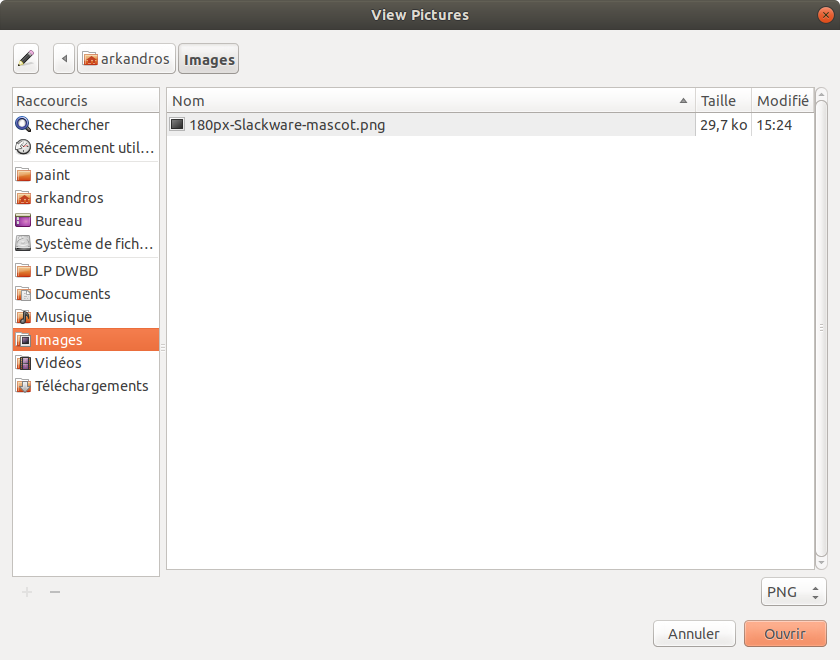
L’application que nous avons développée en Java Fx permet d’ouvrir un fichier image, d’en afficher le contenu et d’ensuite apporter des modifications à celle-ci, avant d’éventuellement la sauvegarder où il le souhaite dans son système de fichiers. L’utilisateur peut ainsi passer une image en niveaux 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 :

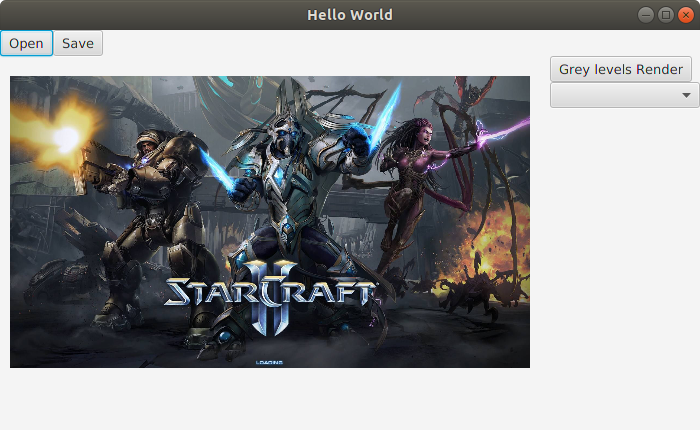
2. L’utilisateur lance l’application :



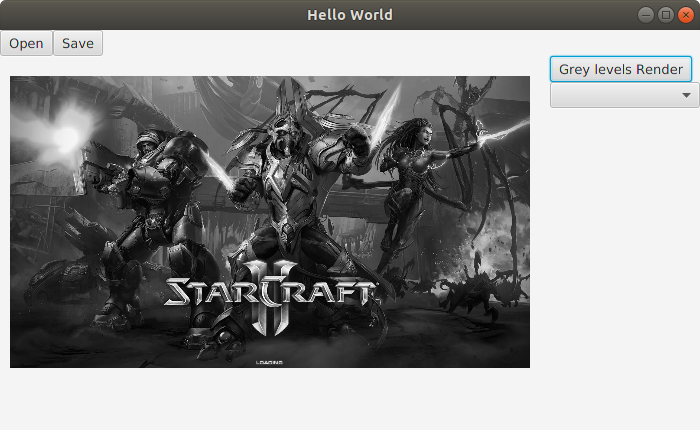
3. L’utilisateur appuie sur le bouton « Open », qui ouvre une fenêtre de dialogue permettant de sélectionner un fichier à ouvrir:



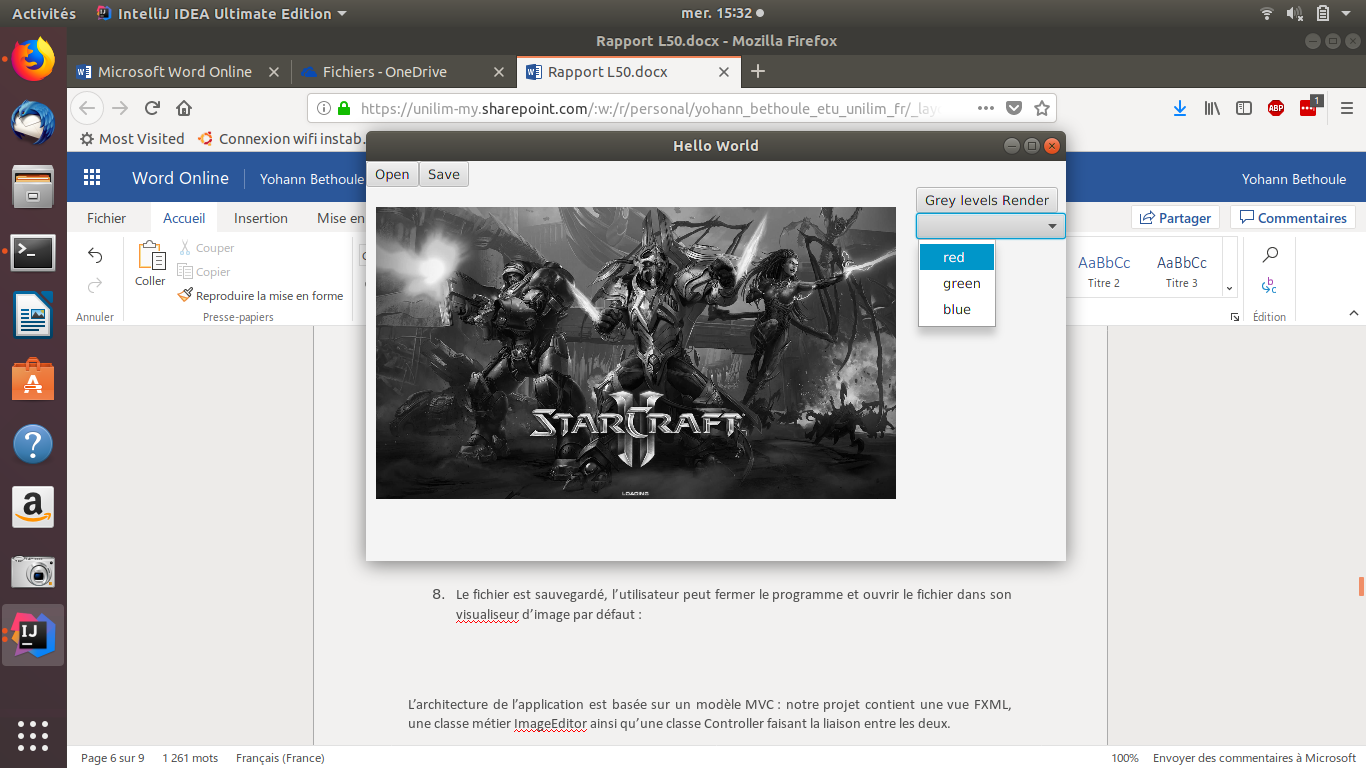
4. L’image s’affiche dans le programme :



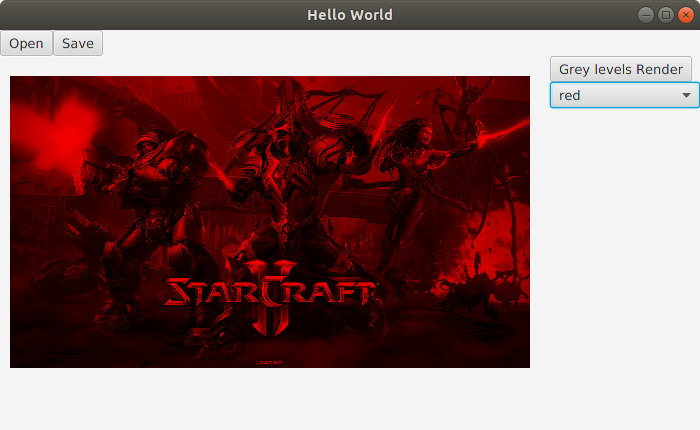
5. L’utilisateur appuie sur le bouton pour transformer l’image en niveaux de gris, après un court temps de chargement signalé par une barre de progression, l’image transformée est affichée :



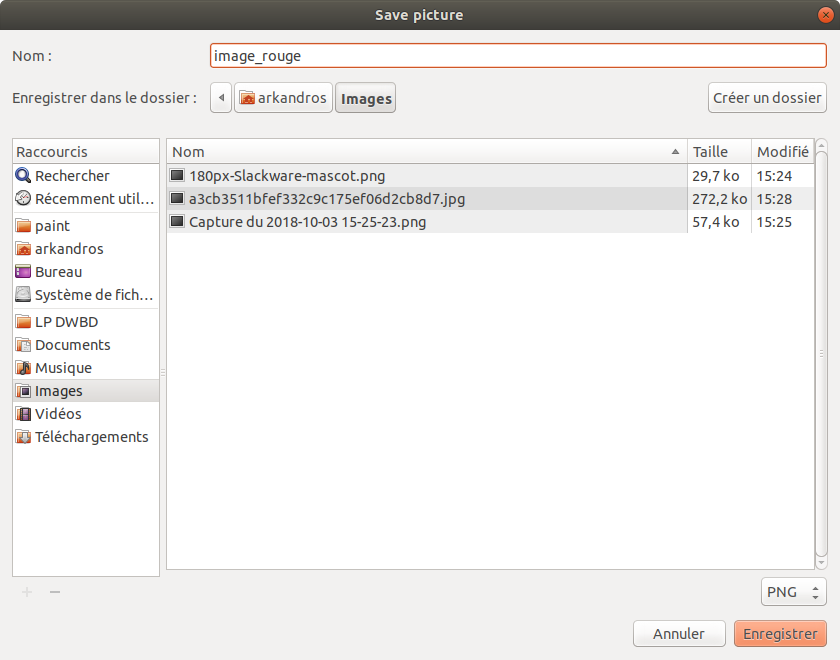
6. L’utilisateur ouvre le menu des rendus RGB, et sélectionne une couleur dans le menu :



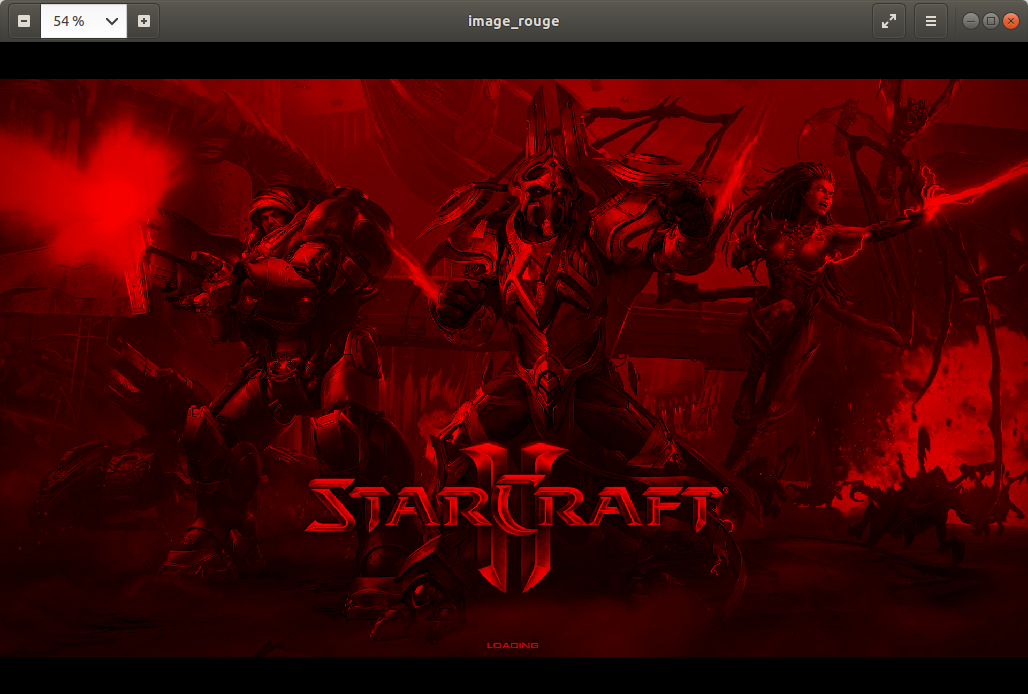
7. Comme pour la transformation en niveaux de gris, l’image s’affiche après l’opération :



8. 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 :



9. 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 :

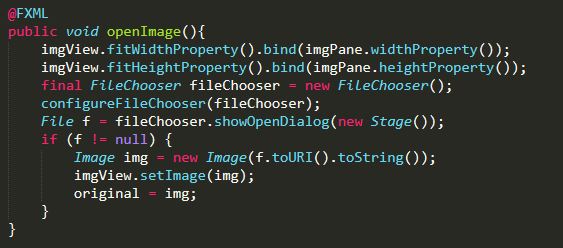


Figure 10: Méthode openImage() du Controller

L’objet imgView auquel je fais référence dans cette méthode est en fait un objet ImageView représentant l’élément HTML éponyme 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 à 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.*



Figure 11: Méthode createTask() de la classe ImageEditor

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, une Image et un String représentant la couleur obtenue souhaitée

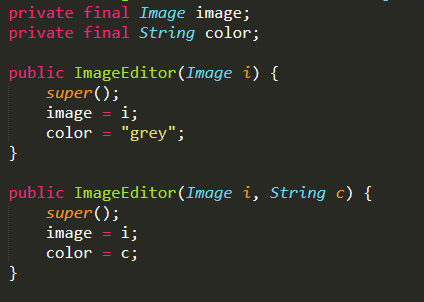


Figure 12: Extrait de la classe ImageEditor

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 modifier servira de canevas au programme pour dessiner une nouvelle image 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. Ensuite, pour chaque pixel un à un :

1. On lit la couleur du pixel, c’est-à-dire le pourcentage de bleu, de rouge et de vert affiché par le pixel. Si (r, g, b)=(1,1,1), l’image est blanche; si (r, g, b)=(0,0,0), l’image est noire.
2. On écrit sur la WritableImage un pixel à la même position que le pixel actuellement traité, en fonction de l’attribut ‘color’ de la classe.

* En rouge, bleu ou vert: on met les valeurs des deux couleurs non voulues à 0, et on garde la même valeur pour la couleur d’image que l’on souhaite obtenir.
* En niveaux de gris: On pourrait se contenter de diviser chaque valeur RGB par 3 pour obtenir une image en gris, mais on a choisi d’utiliser un algorithme permettant d’obtenir un meilleur résultat, prenant en compte la différence de perception de l’intensité de chaque couleur par l’œil humain:

0.3 \* R + 0.59 \* G + 0.11 \* B

1. On incrémente le compteur de 1 et on met à jour la propriété de progression du service, ce qui va mettre à jour l’affichage de la barre de progression pour afficher en temps réel l’état d’avancement du traitement.

Quand le parcours est fini, on retourne l’objet WritableImage qui contient maintenant l’image transformée. Comme la fonction déclare Image comme type de retour, elle sera considérée comme telle quand elle sera récupérée.

### Electron

Nous avons ensuite développé une application similaire en utilisant Electron. Le fonctionnement en étant très semblable, nous éviterons de surcharger le rapport en omettant les captures d’écran de l’application, pour nous concentrer sur le code.

Pour commencer, l’architecture de l’application est très différente. Electron étant basé sur le principe du développement web classique (HTML/CSS, Javascript), on a une architecture semblable à celle d’une page web classique :

Une page ‘index.html’ liée à une feuille de style css et un fichier ‘script.js’.

Le fichier ‘index.js’ est le point d’entrée pour lancer l’application. Il contient les instructions permettant son lancement :



Figure 13: Le fichier 'index.js'



Figure 14 : Extrait de 'script.js'

Dans l’extrait ci-dessus, on peut voir les lignes du script qui gère les interactions avec le DOM. Dans l’ordre :

1. On crée des attributs DOMElement pour manipuler les éléments HTML dont a besoin
2. On écoute l’évènement ‘submit’ du filechooser, c’est-à-dire lorsque l’utilisateur a choisi un fichier à ouvrir, et on lit ensuite le fichier.
3. Lorsque l’image est chargée, on l’affiche.
4. On ajoute des event listerners sur les boutons pour transformer l’image, qui appellent la méthode appropriée.
5. On ajoute un event listenener sur le bouton de sauvegarde, qui ouvre une boîte de dialogue pour sélectionner l’emplacement de la sauvegarde et écrit l’image dans le fichier.

Jusqu’ici, notre fichier semble remplir un rôle similaire à celui du Controller en JavaFX.

La différence avec notre application JavaFX est qu’ici, la fonction métier de notre application est codée directement dans le même fichier que les actions du controller. La méthode changeColor(), qui permet de transformer l’image, est donc déclarée directement dans le fichier ‘script.js’.

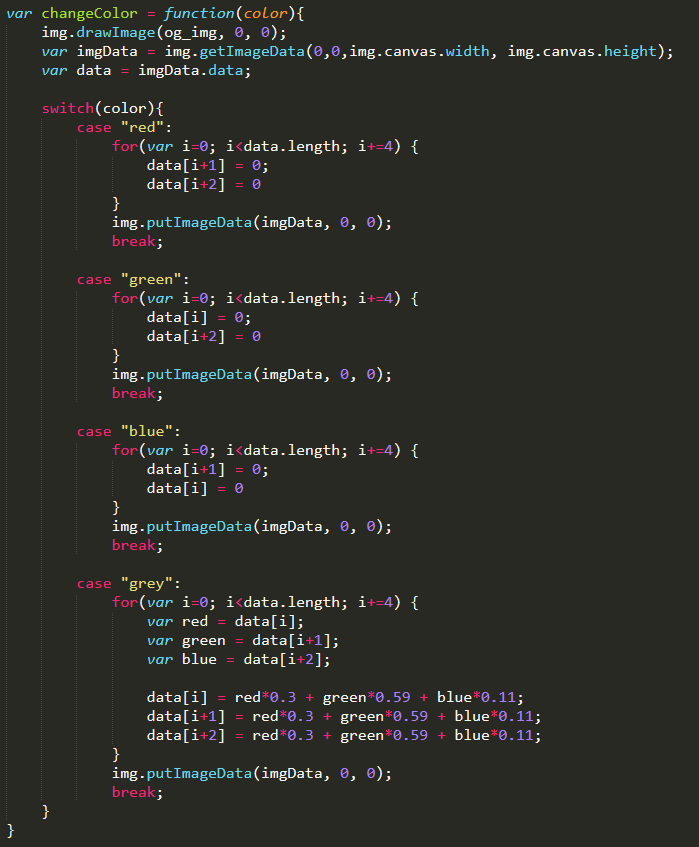


Figure 15 : La fonction changeColor() du ‘script.js’

On peut voir que nous avons fait le choix d’imbriquer nos boucles ‘for’ dans un ‘switch’, ce qui crée de la redondance de code (étant donné qu’on décrit 4 fois une opération très semblable). Néanmoins, il nous a semblé judicieux de procéder ainsi plutôt que l’inverse (c’est-à-dire imbriquer le ‘switch’ dans un seul ‘for’) pour des raisons de performance. Nous pensons en effet qu’exécuter une seule fois la vérification des conditions du switch est moins lourd pour le processeur que de vérifier ces conditions à chaque itération. Ces itérations sont en effet très nombreuses, puisqu’une image contient en général un grand nombre de pixels, et leur traitement rapide.

## 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.