COLLÈGE DE
BOIS-DE-BOULOGNE

TP1 – Photorama

SIM – A25 – Développement d'applications dans un environnement graphique

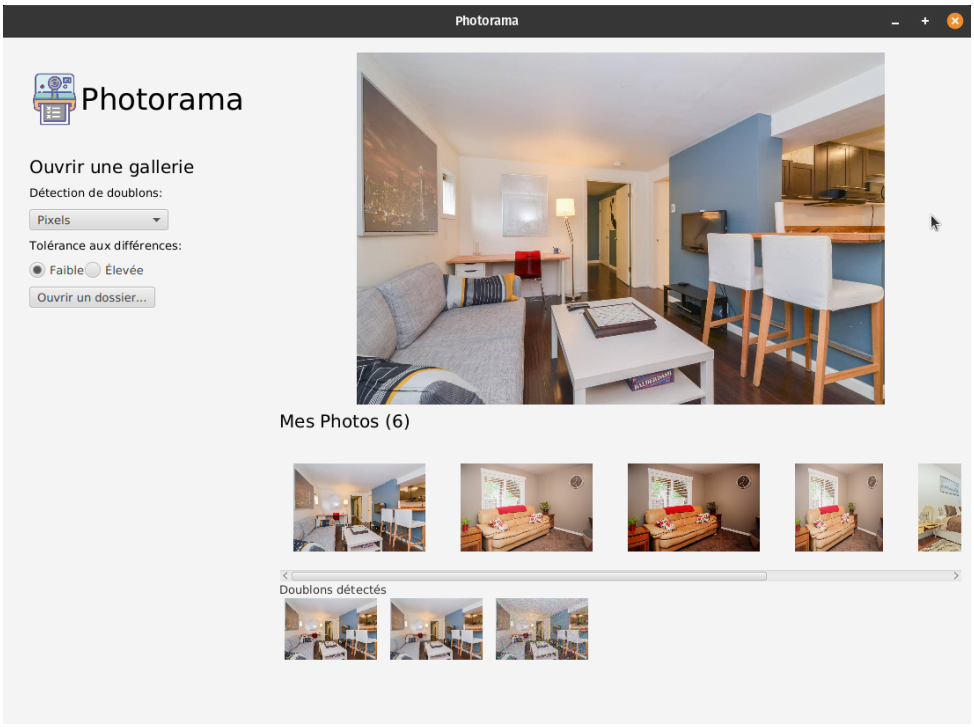
Nicolas Hurtubise et Raouf Babari

➢ Ne commencez pas le TP avant d'avoir lu toutes les sections, en particulier **Structure à suivre pour les classes** et **Gradle** à la fin!

Contexte

Pour votre premier gros programme JavaFX, votre tâche sera de coder une **gallerie d'images *intelligente***.

Vous aurez des dossiers de photos à afficher à l'écran, avec la particularité suivante : certaines photos sont des doublons (plus ou moins exacts) de d'autres photos. Votre application devra donc détecter ces doublons et les regrouper dans l'affichage.



Plus spécifiquement, vous devrez coder **deux versions** du programme : une interface graphique avec JavaFX, qui affichera les photos, et une autre en ligne de commande (la console), qui fera un petit rapport avec des informations de débogage.

Votre tâche

Dans le `.zip` disponible sur Léa, je vous fournis différents corpus de photos d'appartements tirées d'un jeu de données de photos d'appartements d'AirBnB. Ces photos contiennent des **doublons**. Vous devrez créer deux interfaces pour effectuer une recherche des doublons : une en **ligne de commande**, une autre avec **JavaFX**.

Je m'attends à ce que votre programme soit professionnel, au sens où il doit être **robuste**. Le programme ne doit pas planter et doit gérer ses exceptions correctement lorsqu'il manipule des fichiers.

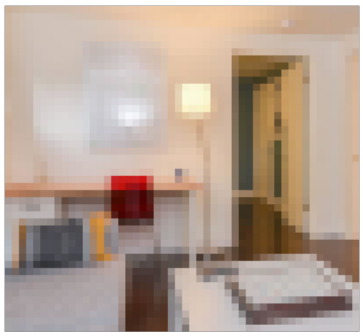
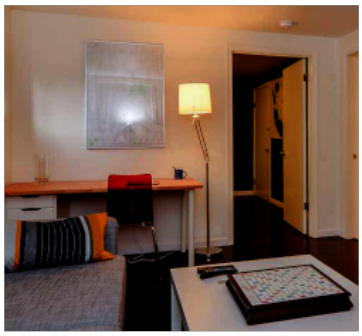
Détection de doublons

Le coeur de votre programme sera de répondre à la question :

> *Dans un tas de photos, quelles sont celles qui sont dupliquées de d'autres?*

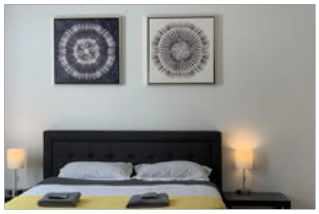
Le cas le plus simple serait celui où deux images comportent exactement les mêmes pixels (un même fichier se retrouve deux fois par erreur). On va cependant s'intéresser à des cas plus intéressants :

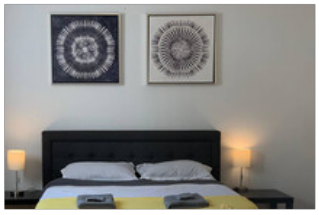
Pour une même image, des versions pixelisées, brouillées ou filtrées au niveau des couleurs devraient être considérées comme des duplications :


Une certaine altération des pixels, soit en redimensionnant l'image ou en appliquant un filigrane (*watermark*) sont également des dupliqués :

Plus généralement, deux images qui contiennent majoritairement la même chose devraient idéalement être détectées comme étant des duplications :





Ici, la personne qui a pris la photo a légèrement changé de place entre les deux photos.

Le coeur du programme sera donc d'arriver à dire si deux images sont assez similaires pour être considérées comme la même chose.

On vous fournit trois corpus de photos pour tester votre programme :

1. `airbnb-mini`, qui ne contient que 9 photos
2. `airbnb-petit`, qui contient 70 photos en petit format (*rapide à tester*)
3. `airbnb-large`, qui contient les mêmes 70 photos mais en grand format (*les algos vont être **très** lents sur ça*)

Les photos ont été adaptées d'un jeu de données disponible sur Kaggle, qui contient des exemples de photos de logements AirBnB avec des duplications.

Vous avez également un corpus `airbnb-corrompu`, qui contient des images invalides (nécessaire pour tester la gestion des exceptions) et un dossier `debogage` qui contient des images simples de test.

Représentations pour une image

La classe `BufferedImage` est disponible par défaut dans la bibliothèque de Java. Elle permet de représenter une image avec un objet Java. La méthode `GestionnaireImages.lireImage()` permet de lire un fichier d'image JPG ou PNG et de construire un `BufferedImage`.

Pour travailler avec les pixels d'une image, on voudra plutôt passer par un tableau 2D `int[][]`. Pour garder les choses simples, les images seront toujours converties en noir et blanc.

La méthode `GestionnaireImages.toPixels(bufferedImage)` permet de convertir un `BufferedImage` en `int[][]`, où chaque case correspond à la couleur du pixel :

- 0 = noir
- 255 = blanc
- Les valeurs entre les deux sont des tons de gris

La hauteur de l'image (verticalement, le nombre total de lignes de pixels) est :

```
matrice.length
```

La largeur de l'image (horizontalement, le nombre de colonnes) est donc donné par :

```
matrice[0].length
```

Pour accéder à la valeur du pixel (x, y) lorsque l'image est au format matrice 2D, on utilisera :

```
matrice[y][x]
```

La méthode `GestionnaireImages.redimensionner(bufferedImage)` vous est également fournie, pour vous permettre de rétrécir une image (ça sera nécessaire pour compléter certains algorithmes).

Algorithmes de calcul de la similarité

Vous devrez implanter **trois algorithmes pour évaluer la similarité entre deux images**, qui vont utiliser des stratégies différentes :

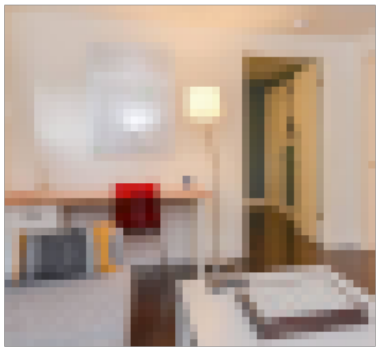
1. Comparer la valeurs des pixels directement
2. Hachage basé sur la moyenne de l'image
3. Hachage basé sur les différences de voisins

Ces trois algorithmes sont expliqués en détails dans les prochaines sections.

Algo 1: Comparaison des pixels

Une façon simple de détecter les cas faciles est de regarder les deux tableaux 2D, pixel par pixel, et de vérifier si les pixels sont suffisamment similaires entre les deux.

Si on prend le cas d'une image qui se fait brouiller, les pixels de la nouvelle image sont encore très proches de ceux de l'image originale :



Du noir pourrait peut-être devenir du gris foncé :

$$pixel_{original} = 0$$
$$pixel_{brouillé} = 6$$

Du blanc pourrait se faire mélanger à du noir et devenir un peu plus foncé :

$$pixel_{original} = 255$$
$$pixel_{brouillé} = 235$$

Dans le cas d'une image qui se fait appliquer un filigrane, certains pixels sont complètement détruits, mais la majorité de l'image reste de la même :

Ici, les pixels où le filigrane a été appliqué sont remplacés par des valeurs qui n'ont aucun lien avec les anciens pixels :

$$\begin{bmatrix} 114 & 156 & 110 & 120 & 167 & 114 & 129 & ... \\ 226 & 226 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 212 & 170 & ... \\ 175 & 172 & \mathbf{0} & \mathbf{0} & 158 & 154 & 152 & ... \\ 156 & 168 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 175 & 188 & ... \\ 216 & 209 & \mathbf{0} & \mathbf{0} & 215 & 218 & 210 & ... \end{bmatrix}$$

Une première façon de comparer deux images pourrait être : pour chaque pixel de l'image 1, regarde le pixel à la même position (x, y) dans l'image 2. Si la différence (en valeur absolue) est plus grande qu'un certain *seuil* (ex.: |valeur1 - valeur2| > 10), ce pixel est considéré comme différent d'une image à l'autre.

Si la proportion de pixels différents est au-dessus d'un *pourcentage de différence max accepté*, alors les deux images sont considérées comme différentes.

On pourra donc paramétrer l'algo en fonction du **seuil de différence** et du **max de pourcent de pixels différents**.

> *ex.: les pixels doivent respecter |valeur1 - valeur2| <= **15** pour être considérés comme similaires, et les deux images doivent avoir <= **20%** de pixels différents pour être considérées comme des doublons.*

On pourra modifier avec ces deux paramètres pour finir avec un algorithme plus ou moins strict ou plus ou moins flexible.

Contrainte importante de cet algorithme : si les images n'ont pas la même largeur et la même hauteur, cet algorithme ne s'applique pas. On doit dire que les images sont nécessairement différentes si elles n'ont pas la même taille.

Algo 2: Hachage basé sur la moyenne

Un **hachage** est un "condensé" d'une valeur. Dans notre cas pour les algos 2 et 3, on va "condenser" les images en **tableaux 2D binaires** plus petits, qui seront plus faciles à comparer entre eux.

Voici ce qu'on va faire pour calculer un hachage d'une image en se basant sur la valeur moyenne :

1. Redimensionner l'image originale au format 8x8
2. Calculer la *valeur moyenne* des pixels de l'image réduite
3. Créer une matrice de taille 8x8, qui sera le hachage de l'image originale
4. Assigner à chaque case de cette nouvelle matrice soit un 0, soit un 1 :
   - 0 si le pixel de l'image réduite a une valeur plus foncée (ou égale) à la moyenne de l'image
   - 1 sinon (donc si le pixel de l'image est plus pâle que la moyenne)

Exemple avec des données inventées :

$$\text{image originale}_{\text{taille=très grosse}} = \begin{bmatrix} 114 & 156 & 110 & 120 & 167 & 114 & 129 & 135 & ... \\ 226 & 226 & 120 & 126 & 160 & 212 & 170 & 180 & ... \\ 175 & 172 & 156 & 154 & 158 & 154 & 152 & 162 & ... \\ 226 & 120 & 126 & 160 & 291 & 212 & 170 & 180 & ... \\ 156 & 168 & 135 & 144 & 163 & 175 & 188 & 190 & ... \\ ... & & & & & & & & \\ 216 & 209 & 201 & 206 & 215 & 218 & 210 & 215 & ... \end{bmatrix}$$

$$\text{image réduite}_{\text{taille=8x8}} = \begin{bmatrix} 15 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\ 20 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\ 30 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\ 40 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\ 50 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\ 60 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\ 70 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\ 80 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \end{bmatrix}$$

$$moyenne = \frac{\text{somme de tous les pixels}}{\text{nombre de pixels}} = \frac{3165}{64} \approx 49.45$$

Tous les pixels <= 49.45 donneront un 0 dans la matrice de hachage

Tous les pixels > 49.45 donneront un 1 dans la matrice de hachage

$$\text{hachage} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Pour décider si deux images sont identiques ou non, on pourra: hacher chacune des deux images avec cet algo, puis compter le nombre de cases différentes :

$$\text{hachage1} = \begin{bmatrix} 0 & 0 & 0 & \underline{\mathbf{0}} & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & \underline{\mathbf{0}} & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & \underline{\mathbf{1}} & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & \underline{\mathbf{1}} \end{bmatrix}, \text{hachage2} = \begin{bmatrix} 0 & 0 & 0 & \underline{\mathbf{1}} & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & \underline{\mathbf{1}} & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & \underline{\mathbf{0}} & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & \underline{\mathbf{0}} \end{bmatrix}$$

différences = 4 cases différentes

Si le nombre de cases est *plus grand que le nombre de différences accepté*, alors les images seront considérées comme différentes.

Cet algo sera paramétré en fonction du **nombre max de cases différentes acceptées entre les hachages**.

> *ex.: il doit y avoir <= **10** cases différentes pour considérer que les deux images sont similaires.*

Algo 3: Hachage basé sur les différences de voisins

Ce dernier algorithme ressemble beaucoup à l'algorithme 2. La seule différence est dans la méthode utilisée pour calculer le hachage.

Plutôt que de considérer chaque pixel de l'image par rapport à la *moyenne de l'image*, on considère chaque pixel *par rapport à son voisin d'en-dessous*.

Pour finir avec un hachage qui est un tableau de taille 8x8, on devra commencer par redimensionner l'image au format **8x9 (hauteur de 9)** :

$$\text{image réduite}_{\text{taille=9 lignes, 8 colonnes}} = \begin{bmatrix} 10 & 99 & 30 & 99 & 50 & 99 & 70 & 99 \\ 20 & 20 & 90 & 40 & 90 & 60 & 90 & 80 \\ 30 & 98 & 30 & 98 & 50 & 98 & 70 & 98 \\ 40 & 20 & 97 & 40 & 97 & 60 & 97 & 80 \\ 50 & 96 & 30 & 96 & 50 & 96 & 70 & 96 \\ 60 & 20 & 95 & 40 & 95 & 60 & 95 & 80 \\ 70 & 94 & 30 & 94 & 50 & 94 & 70 & 94 \\ 80 & 20 & 93 & 40 & 93 & 60 & 93 & 80 \\ 90 & 92 & 30 & 92 & 50 & 92 & 70 & 92 \end{bmatrix}$$

La matrice 2D de hachage est encore au format 8x8. Le pixel (x, y) de l'image réduite est comparé à son voisin d'en-dessous (la case $(x, y + 1)$). Si le pixel (x, y) est plus foncé ou égal, on met un 0 à la case correspondante de la matrice de hachage. Sinon, on y met un 1.

Dans l'exemple ci-haut, si on prend la première colonne verticale, la toute première case de la matrice de hachage serait **0** (car 10 <= 20). La case juste en-dessous serait également 0 (car 20 <= 30), et ainsi de suite. Dans la deuxième colonne, la première case serait à 1 (car 99 > 20), puis la deuxième case serait à 0 (car 20 <= 98), etc. La matrice de hachage pour cet exemple inventé serait :

$$\text{hachage}_{\text{taille=8x8}} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Une fois deux images hachées, on utilise **exactement la même méthode que pour l'Algo 2** lorsque vient le temps de déterminer si deux images sont des doublons ou non.

Regroupement des images similaires

Une fois qu'on a une fonction qui nous permet de dire si deux images sont similaires ou non, on peut les regrouper en paquets d'images similaires avec un algo simple :

1. Trier l'ensemble des images à utiliser par nom de fichier
   - Utilisez `Arrays.sort()`, pas besoin de coder un algo de tri par vous-mêmes
2. Former chaque groupe d'images similaires en prenant la première image + toutes les images qui sont identifiées comme similaires à celle-ci
3. Retirer de l'ensemble d'images toutes celles qui ont été choisies
4. Reprendre depuis l'étape 2 jusqu'à ce que toutes les images aient été regroupées

Je vous recommande fortement d'utiliser comme structure de données un `ArrayList<ArrayList<String>>`, *ie*, une liste de "groupes de noms d'images dupliquées"

Si on a les 6 images :

```
images = [f.png a.png c.png b.png e.png d.png]
```

On devrait d'abord trier ces images par ordre alphabétique :

```
images = [a.png b.png c.png d.png e.png f.png]
```

Puis partir de la première image pour former le premier groupe. Toutes les images similaires à `a.png` seront regroupées avec elle, puis retirées des images restantes :

```
groupes = [ [a.png b.png e.png] ]
images = [c.png d.png f.png]
```

On repasse à travers la liste d'images restantes jusqu'à ce que toutes les images aient été attribuées à un groupe (quitte à ce que certains "groupes" ne contiennent qu'une seule image, lorsqu'elle n'a pas de doublons)

```
groupes = [ [a.png b.png e.png] [c.png] [d.png f.png] ]
images = []
```

Version 1 : en ligne de commande

Vous devrez faire une première version du programme qui ne fait que produire un petit rapport en ligne de commande, sans interactions (**n'utilisez pas de `Scanner` ici**).

Votre rapport va contenir :

1. Des informations de débogage pour vérifier les résultats des différents algorithmes
2. Une liste textuelle des doublons trouvés **pour la gallerie `airbnb-petit` seulement**

Dans les informations de débogage, vous devez afficher les résultats suivants :

1. Le résultat de la détection de doublons avec l'Algo 1, pour les paramètres et paires d'images suivants :
   - seuil=5, max pourcent=0.2, `debogage/pixels1.png` vs `debogage/pixels2.png`
   - seuil=5, max pourcent=0.2, `debogage/pixels1.png` vs `debogage/pixels3.png`
   - seuil=5, max pourcent=0.2, `debogage/pixels1.png` vs `debogage/pixels4.png`
   - seuil=5, max pourcent=0.5, `debogage/pixels1.png` vs `debogage/pixels4.png`
   - seuil=128, max pourcent=0.49, `debogage/pixels1.png` vs `debogage/pixels4.png`
   - seuil=128, max pourcent=0.50, `debogage/pixels1.png` vs `debogage/pixels4.png`
   - seuil=128, max pourcent=0.51, `debogage/pixels1.png` vs `debogage/pixels4.png`
2. Un affichage textuel de la matrice 8x8 de hachage pour l'algo basé sur la moyenne (algo 2) pour les fichiers :
   - `debogage/moyenne-test1.png`
   - `debogage/moyenne-test2.png`
   - `debogage/moyenne-test3.png`
3. Un affichage textuel de la matrice 8x8 de hachage pour l'algo basé sur les différences (algo 3) pour les fichiers :
   - `debogage/diff-test1.png`
   - `debogage/diff-test2.png`
   - `debogage/diff-test3.png`

Pour les tests de comparaisons de pixels, utilisez le format suivant :

```
seuil=5, max pourcent=0.2: debogage/pixels1.png et debogage/pixels2.png SIMILAIRE
seuil=5, max pourcent=0.2: debogage/pixels1.png et debogage/pixels3.png DIFFÉRENT
...
```

Pour afficher textuellement une matrice 8x8 de hachage, vous devez utiliser de `1` là où il y a des uns, et des espaces là où il y a des zéros, question de visualiser facilement le résultat, par exemple :

```
Pour debogage/moyenne-test1.png:
             1111
             1111
             1111
             1111
         1111
         1111
         1111
         1111
```

```
Pour debogage/diff-test1.png:
          1 1 1 1
         1 1 1 1
          1 1 1 1
         1 1 1 1
          1 1 1 1
         1 1 1 1
          1 1 1 1
         1 1 1 1
```

Pour la liste textuelle des doublons trouvés, vous devez afficher chaque $i^{\text{ème}}$ groupe de doublons sur sa propre ligne, numéroté avec `[i]`. Les noms des fichiers sont ensuite listés un après l'autre, séparés par des espaces.

Vous devez afficher les doublons détectés pour les 6 comparateurs suivants :

1. Comparateur Pixels (seuil différences=8, pourcentage différences max=0.2)
2. Comparateur Pixels (seuil différences=8, pourcentage différences max=0.2)
3. Comparateur Hachage Moyenne (cases différentes max=8)
4. Comparateur Hachage Moyenne (cases différentes max=16)
5. Comparateur Hachage Différences (cases différentes max=8)
6. Comparateur Hachage Différences (cases différentes max=16)

Je vous fais confiance ici pour ne pas faire des copiers-collers pour les 6 cas `:^)`

Utilisez le format suivant pour afficher les doublons tels que trouvés par chaque algorithme :

```
Algo: Comparateur Pixels (seuil différences=8, pourcentage différences max=0.2)
[1] 1. a1.jpg
[2] 1. a2.jpg
[3] 1. a3.jpg
[4] 1. b1.jpg 2. b2.jpg
[5] 1. b3.jpg
[6] 1. b4.jpg
...
Algo: Comparateur Hachage Moyenne (cases différentes max=16)
[1] a1.jpg a2.jpg a3.jpg
[2] b1.jpg b2.jpg b3.jpg b4.jpg c1.jpg zz11.jpg zz27.jpg zz52.jpg
[3] c2.jpg zz36.jpg
[4] d1.jpg d2.jpg
...
Algo: Comparateur Hachage Différences (cases différentes max=16)
[1] a1.jpg a2.jpg a3.jpg
[2] b1.jpg b2.jpg b3.jpg b4.jpg
[3] c1.jpg c2.jpg
[4] d1.jpg
...
```

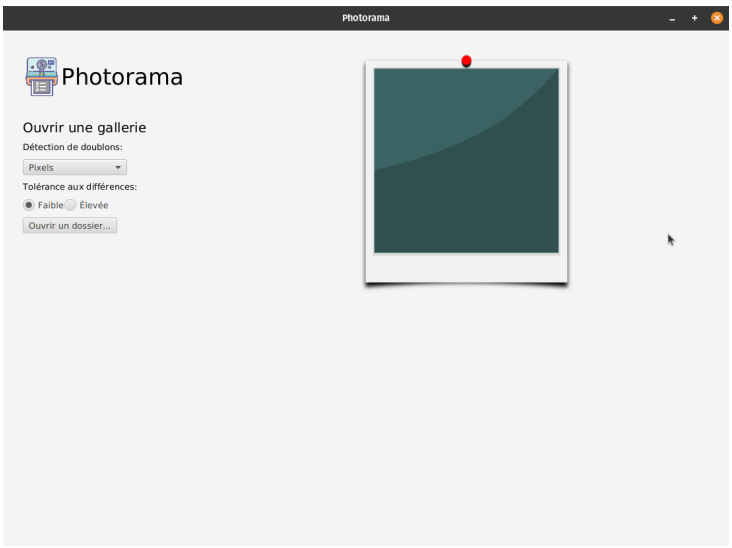
Assurez-vous d'afficher seulement les noms des fichiers (sans le chemin complet du dossier), question que ça soit facile à lire

Le résultat attendu lorsqu'on exécute la version en ligne de commande vous est fourni dans les fichiers avec l'énoncé, dans `resultat-cmd.txt`

Version 2 : JavaFX

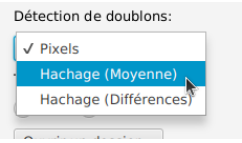
La version JavaFX est un programme de visualisation d'une gallerie d'images.

Initialement, lorsqu'on ouvre le programme, aucune gallerie n'est chargée: il y a une image temporaire qui occupe l'espace central (`polaroid.png`)



On peut utiliser le menu dans la colonne de gauche pour ouvrir un dossier.

On peut sélectionner l'algorithme de *Détection des doublons* parmi les trois algos :



Dans la capture d'écran, on utilise un objet `ChoiceBox<>` pour faire cette sélection.

La *Tolérance aux différences* offre seulement deux choix (*Faible* ou *Élevée*), qui iront définir les paramètres qu'on va utiliser pour les algorithmes de comparaison d'images.

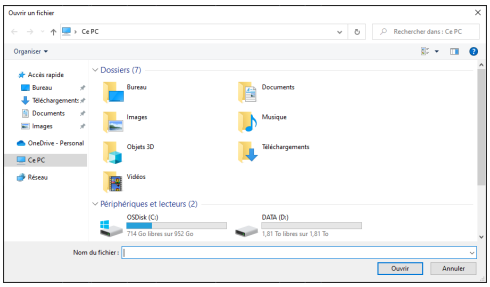
Dans la capture d'écran, on utilise deux objets `RadioButton`, qui sont connectés ensemble via un objet `ToggleGroup`. Fouillez dans la documentation pour trouver comment on utilise ça.

Les paramètres à utiliser selon le choix fait sont les suivants :

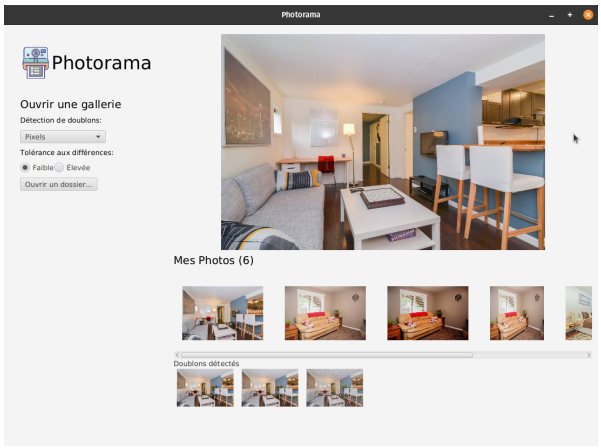
	Faible	**Élevée**
Pixel	Seuil de différence: 20 Max pourcent différences: 10%	Seuil de différence: 30 Max pourcent différences: 40%
Hachage Moyenne	Max cases différentes: 10	Max cases différentes: 15
Hachage Différences	Max cases différentes: 10	Max cases différentes: 15

*Notez bien ici : les paramètres à utiliser ne sont **pas** les mêmes que dans la version en ligne de commande*

Lorsqu'on clique sur le bouton *Ouvrir un dossier...*, une boîte de dialogue s'ouvre pour demander quel dossier sélectionner (voir le code fourni plus loin pour vous aider).



Lorsque le dossier est choisi, on crée un nouvel objet `Gallerie`, qui sera utilisé pour remplir l'interface graphique avec les images trouvées.



Le titre sous la grande image affiche *Mes Photos (N)*, où N est le nombre de photos **sans compter les doublons**, soit simplement le nombre de groupes de doublons trouvés.

Pour l'exemple donné plus haut, avec :

```
groupes = [ [a.png b.png e.png] [c.png] [d.png f.png] ]
```

On aurait $N = 3$ images. La première rangée affiche la première image de chaque groupe de doublons. Toujours avec ce même exemple, ça afficherait les images :

```
a.png c.png d.png
```

Lorsqu'on clique sur une image la première rangée :

1. L'image sur laquelle on a cliqué s'affiche en gros juste au-dessus
2. La rangée du bas se met à afficher les différents doublons de cette image (toutes les images de son groupe)

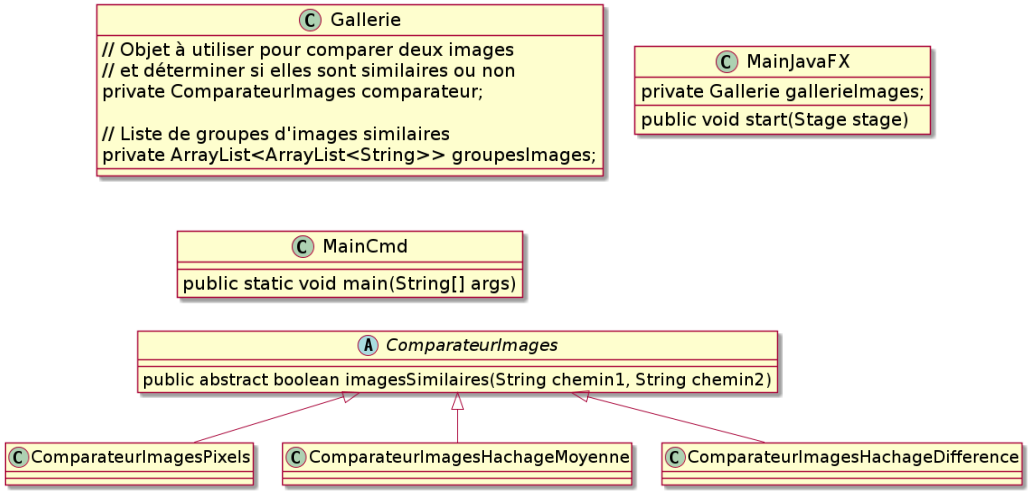
On peut cliquer sur un des doublons dans la rangée du bas pour l'afficher en plus gros dans l'emplacement centré au-dessus.

À tout moment, on doit pouvoir **appuyer sur `Escape` pour fermer le programme**.
Utilisez `Platform.exit();` pour quitter l'application JavaFX quand on appuie sur Escape.

Notez: **vous avez le droit de modifier l'interface proposée** si vous avez une meilleure idée que moi. Je ne vous évalue pas sur le fait d'avoir reproduit au pixel près la même chose que moi. Votre programme JavaFX doit cependant **offrir toutes les fonctionnalités demandées** et **être un gros minimum joli** (si vous faites quelque chose de trop simple, vous allez perdre des points).

Structure à suivre pour les classes

Voici une ébauche de structure pour vos classes :



Classe `Gallerie`

Cette classe est responsable de construire une gallerie d'images avec les doublons regroupés.

Lorsqu'on crée une nouvelle gallerie, on devrait lui fournir un chemin vers un dossier à ouvrir. Cet objet doit alors lister le contenu du dossier et trouver tous les fichiers `.png` ou `.jpg` à charger.

Vous pouvez lister les fichiers d'un dossier donné avec :

```
var repertoire = new File(dossier);

if (repertoire.listFiles() == null) {
    throw new FileNotFoundException(dossier + " n'est pas un répertoire");
}

for (var fichier : repertoire.listFiles()) {
    var nomFichier = fichier.getName(); // Exemple: a.png

    // TODO : Garder seulement les images .png ou .jpg
    // Ignorer le fichier si le nom ne termine pas par .png ou .jpg

    // Exemple: airbnb-petit/a.png
    var cheminFichier = dossier + "/" + nomFichier;

    // On pourra lire l'image en fournissant cheminFichier
    // à GestionnaireImages.lireImage()
}
```

L'objet gallerie devra ensuite regrouper les images en groupes de doublons. La structure de données attendue pour noter quelle image est dans quel groupe est :

```
private ArrayList<ArrayList<String>> groupesImages;
```

`groupesImages.get(i)` va donner accès à un `ArrayList<String>` qui devrait contenir les noms des images dans le $i^{ème}$ groupe de doublons. Les images qui sont uniques (ie, qui n'ont pas de doublons) auront donc leur propre `ArrayList<String>` à elles seules, avec un seul élément dedans.

Hiérarchie de classes pour les `ComparateurImages`

L'objet `Gallerie` va posséder un objet `ComparateurImages` (qui lui sera fourni dans son constructeur) pour décider comment regrouper les images.

Pour lui permettre de choisir et un des trois algos avec ses paramètres, la gallerie va utiliser un **objet algorithme**, qui possède une méthode :

```
public boolean imagesSimilaires(String chemin1, String chemin2)
```

Cet objet pourra avoir en *attributs* les paramètres nécessaires pour chaque algorithme.

Au moment de regrouper les images selon par doublons, la `Gallerie` va utiliser la méthode de son objet :

```
boolean similaire = objetAlgorithme.imagesSimilaires(img1, img2);
```

De cette façon, la classe `Gallerie` n'aura qu'à utiliser l'objet qui lui est fourni, sans avoir à savoir lequel des trois algorithmes est utilisé :

```
var comparateur = ...
       // Choix parmi:
       new ComparateurImagesPixels(...);
       new ComparateurImagesHachageMoyenne(...);
       new ComparateurImagesHachageDifference(...);

var gallerie = new Gallerie("airbnb-mini", comparateur);

// => la gallerie va utiliser l'objet comparateur qu'on lui aura donné
```

Classes `MainJavaFX` et `MainCmd`

Ces deux classes devraient **uniquement** contenir :

- De l'affichage (respectivement, avec des composantes de JavaFX ou des `System.out.println()`)
- Des interactions (événements en JavaFX, rien à faire en console)

Aucune logique de regroupement d'images, de détection de similarité, de hachage, etc ne devrait s'y retrouver. Ces deux classes vont plutôt *utiliser* la classe `Gallerie` et les classes `ComparateurImages`.

Le code de la classe `MainCmd` devrait être tout petit (moins de 100 lignes dans mon cas), puisqu'il ne devrait contenir aucune logique, **seulement de l'affichage de choses que le modèle calcule**. Mis à part utiliser les objets du modèle et faire des `System.out.println`, le code de cette classe ne fera pas grand chose.

Le code de la classe `MainJavaFX` sera un peu plus gros car il y a plus de choses à faire (mon code est quelque part entre 200 et 400 lignes). **Utilisez des méthodes, limitez-vous à maximum 2 lignes dans les événements**.

Autres précisions

Vous **pouvez** ajouter des classes, des attributs et des méthodes. Cette ébauche n'est qu'un point de départ. **Il va vous manquer des choses si vous n'ajoutez rien** à ce qui est proposé dans le diagramme.

Vous serez évalués sur la qualité de votre code final, **arrangez-vous pour éviter les copiers-collers**.

Vous **devez** utiliser une hiérarchie de classes `ComparateurImages` pour les différents algorithmes à coder.

Choisir un dossier sur l'ordinateur avec JavaFX

Voici une méthode que vous pouvez intégrer à votre code `MainJavaFX` pour sélectionner un dossier sur l'ordinateur :

```
/**
 * Ouvre une boîte de dialogue pour sélectionner un dossier à
 * ouvrir dans l'application
 *
 * @param stage Le stage qui contrôle la fenêtre de dialogue
 *              (l'objet passé en paramètre à la fonction start())
 * @return Le chemin du dossier choisi, ou null si l'ouverture a été annulée
 */
private String choisirDossier(Stage stage) {
    DirectoryChooser selecteur = new DirectoryChooser();
    selecteur.setTitle("Sélectionnez un dossier d'images");
    selecteur.setInitialDirectory(new File("."));

    File dossierChoisi = selecteur.showDialog(stage);
    if (dossierChoisi != null) {
        String chemin = dossierChoisi.getAbsolutePath();

        return chemin;
    }

    return null;
}
```

Gestion des Exceptions

Le corpus `airbnb-corrompu` contient des images invalides. Si on essaie d'ouvrir ce dossier, une `IOException` sera levée au moment de lire un des fichiers.

L'application ne doit pas planter, même si on charge ce dossier.

Plutôt, on doit *gérer correctement l'exception.*

- En ligne de commande, on devrait afficher un message d'erreur propre (en français) et fermer le programme
- En JavaFX, on devrait afficher une barre d'erreur en rouge tout en bas de la fenêtre, avec un message qui explique le problème

Petit truc : ce n'est **PAS** la responsabilité des objets du *Modèle* de faire un `catch()` en cas de `IOException`. Le `catch()` pour ces erreurs devrait se retrouver dans les deux classes `MainJavaFX` et `MainCmd`.

Dans les classes du modèle, mettez un `throws IOException` après le nom des méthodes qui auraient besoin de relancer ces exceptions en cas de problème.

Référez-vous aux notes de cour sur les exceptions au besoin.

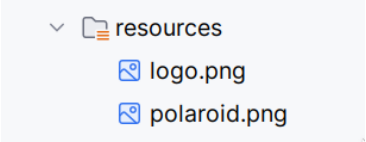
Notes additionnelles

Où placer les images?

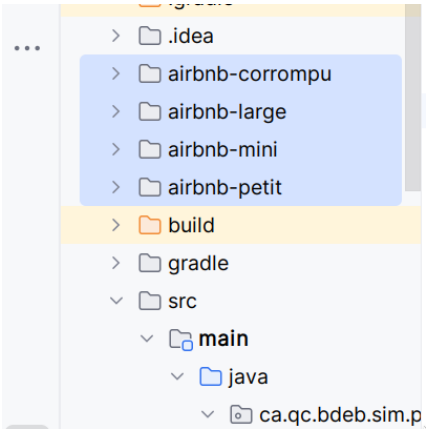
Vous avez deux sortes d'images fournies avec l'énoncé :

- Des ressources nécessaires au fonctionnement du programme : l'icône du programme et l'image *"temporaire"* de photo polaroid (qui est là pour occuper de l'espace avant d'avoir chargé une gallerie d'images)
- Des photos à afficher, qui pourraient être amenées à changer d'une personne à l'autre (on suppose que notre programme ne servira *pas uniquement* à analyser des photos d'appartements!)

Le dossier `ressources` ne devrait contenir que les images qui *font partie de l'application*



Les différents dossiers d'images de logements AirBnB fournies devraient être placées à la racine du projet :



Performance du code

L'algorithmie telle qu'elle est décrite dans cet énoncé est **lente**. La détection de doublons sur `airbnb-large` va prendre un temps considérable pour terminer. **C'est normal**.

Dans le cas de l'algorithme `Pixels`, il n'y a pas grand chose à faire, mais dans le cas des deux algos de hachage, il y a moyen d'améliorer la performance.

Si vous voulez que votre programme fonctionne raisonnablement rapidement même quand on a beaucoup de grosses photos à classer, trouvez une solution pour **ne pas recalculer les mêmes hachages plusieurs fois**. Cet élément n'est **pas évalué** : pour les fins du TP, vous ne perdrez pas de points si votre code ne roule rapidement que les dossiers `airbnb-mini`, `airbnb-petit` et qu'il est très lent sur les images de `airbnb-large`.

Gradle

➢ N'EXÉCUTEZ **PAS** VOTRE CODE SANS PASSER PAR GRADLE

Gradle est un outil très chouette, qui permet entre autres de définir *plusieurs façons d'exécuter le projet.*

Au moment de créer votre projet, nommez-le **TP1-Photorama**

Utilisez le **nom de groupe** `ca.qc.bdeb.sim`

Supprimez les fichiers générés pour JavaFX (voir le tout premier chapitre de notes de cours **Intro à JavaFX** à ce sujet) et assurez-vous que votre classe principale commence bien avec la ligne :

```
package ca.qc.bdeb.sim.tp1photorama;
```

Et mettez-y **deux** classes différentes qui vont chacune avoir un `main()` :

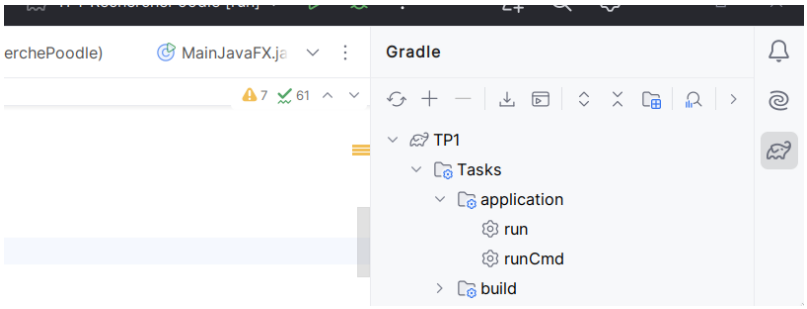
- `MainCmd.java`, qui va démarrer votre projet en ligne de commande
  - Cette classe contient un `public static void main(String[] args)` comme vous êtes habitués de faire
- `MainJavaFX.java`, qui va contenir votre code pour démarrer le projet en JavaFX
  - Cette classe doit `extends Application` et définir votre méthode `start()` pour afficher la fenêtre

Pour expliquer à Gradle les deux façons de lancer votre programme, **vous DEVEZ ajouter ces lignes de configuration À LA FIN DU FICHIER `build.gradle.kts` de votre projet :**

```
tasks.register<JavaExec>("runCmd") {
    group = "application"
    mainClass.set("ca.qc.bdeb.sim.tp1photorama.MainCmd")
    classpath = sourceSets["main"].runtimeClasspath
    standardInput = System.`in`
}
```

Passez par le menu `Gradle` à droite dans IntelliJ pour lancer votre code. Avec ce fichier Gradle, vous aurez deux configurations :

- `Tasks > Application > run` pour lancer la version JavaFX
- `Tasks > Application > runCmd` pour lancer la version en ligne de commande



Remise

Vous devez remettre sur Léa votre projet IntelliJ (avec la configuration Gradle, le code, les images, etc.) **dans un fichier .zip**

La date de remise est spécifiée sur Léa.

Barème

- 70% : Fonctionnalités demandées implantées correctement
  - (35%) Algorithmes
    - (5%) Utilisation correcte de la hiérarchie de `ComparateurImages`
    - (5%) Algorithme 1: Pixels
    - (15%) Algorithme 2 et 3: Hachage
    - (10%) Regroupement des images par similarité
  - (25%) Interface JavaFX : définition de l'interface graphique et des événements
  - (5%) Interface en ligne de commande
  - (5%) **Gestion correcte des exceptions, de la façon demandée**
- 30% : Qualité du code
  - 10% Respect de la séparation du *Modèle* et des interfaces
    - Pas de composantes JavaFX dans le code qui fait la logique
    - Pas de `System.out.println()` dans le code qui fait la logique
    - Pas de logique de regroupement de doublons dans le `MainCmd` ni dans le `MainJavaFX`
  - 20% Qualité générale
    - **Événements de max 2 lignes dans `MainJavaFX`**
    - Code bien commenté
    - Respect du minusculeCamelCase pour les variables/méthodes, MajusculeCamelCase pour les noms de classes
    - Bon découpage en méthodes
    - Encapsulation : attributs `private`, avec getters/setters au besoin
    - **Pas de variables globales!**
    - **Pas de copier-coller de code!**

Note sur le plagiat

Le travail est à faire **individuellement**. Ne *partagez pas de code*, même pas "juste pour aider un ami", ça constituerait un **plagiat**, et les deux personnes auraient la note de *zéro*.

Si jamais vous utilisez *chatGPT*, vous **devez** me citer quels bouts de code ont été générés avec quels prompts. Mieux vaut me donner plus de détails que pas assez.

Références

Les images fournies pour ce TP proviennent de openclipart.org

Les photos de logements sont tirés de [ce jeu de données sur Kaggle].