Manipulation d'images[1]

Ce travail sera à mettre dans un dossier nommé `tp6/` de votre dépôt Git.

On s'intéresse à la représentation et la manipulation d'images. Ces images seront constituées de pixels caractérisés par une couleur représentant un niveau de gris.
Pour un aperçu du travail à réaliser :

```
java -jar image.jar
java -jar image.jar /images/storm.pgm 5 16
```

et vous pouvez remplacer `storm.pgm` par un des autres fichiers fourni dans le dossier `images` de l'archive.

Les couleurs Les niveaux de gris considérés seront codés par un entier sur 8 bits, et donc une valeur entre 0 et 255. De plus on peut gérer la transparence d'une couleur par la valeur de son « canal alpha ». Cette valeur est comprise entre 0 (transparence complète) et 1 (opacité complète).
De telles couleurs sont représentées par la classe `GrayColor` du paquetage `image.color` définie ainsi :

GrayColor
+ WHITE : GrayColor
+ BLACK : GrayColor
- grayLevel : int
- alpha : double
+ GrayColor(level : int)
+ getGrayLevel() : int
+ getAlpha() : double
+ setAlpha(a : double)
+ equals(o : Object) : boolean

où l'attribut `grayLevel` représente le niveau de gris (entre 0 et 255) de cette couleur ; l'attribut `alpha` représente le « canal alpha » dont la valeur est initialement à 1.
La niveau de gris (`grayLevel`) d'une couleur ne peut pas changer.
Les valeurs `BLACK` et `WHITE` sont des **constantes** qui correspondent à deux couleurs particulières dont le niveau de gris vaut respectivement 0 et 255 (et le canal alpha vaut 1).
Deux couleurs sont égales si elles ont les mêmes niveaux de gris et valeurs du canal alpha.

Q 1 . Comment faut-il définir les constantes `BLACK` et `WHITE` ? Comment les utilise-t-on dans un code ?

Q 2 . Codez la classe `GrayColor`.

Attention : dans la suite il ne faudra pas confondre les objets `GrayColor` et leur attribut `grayLevel`.

Les Pixels Les pixels sont modélisés par des objets de la classe `Pixel` du paquetage `image`. Les instances de cette classe dispose d'un attribut représentant leur couleur qui est une objet instance de la classe `GrayColor` définie précédemment.

Q 3 . Codez la classe `Pixel` sachant que :

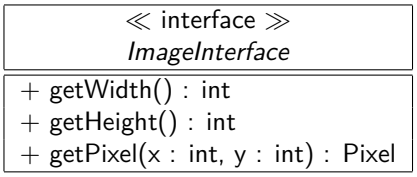
- la classe apppartient au paquetage `image`,
- elle définit un attribut `color` de type `image.color.GrayColor` et les modificateur (`setColor()`) et accesseur (`getColor()`) associés,
- elle offre un constructeur qui prend en paramètre la valeur d'initialisation de son attribut,
- la méthode `equals()` considère que deux pixels sont égaux si leurs couleurs sont égales,
- elle définit une méthode `colorLevelDifference()` qui a pour résultat un entier **positif** qui correspond à la différence des valeurs des niveaux de gris de ce pixel et d'un autre fourni en paramètre.
  Pour obtenir la valeur absolue d'un nombre vous pourrez utiliser la méthode **statique** `abs` de la classe `java.lang.Math`.

Attention : dans la suite il ne faudra pas confondre les objets `Pixel` et leur attribut `color`.

[1] Inspiré de « Introduction à la science informatique » dirigé par Gilles Dowek, p.113.

Les images Une image est modélisée par des instances de la classe `Image`.
Cette classe :

- appartient au paquetage `image`,
- implémente l'interface `ImageInterface`, dont le code est fourni :



  Dans les signatures des méthodes, x réprésente la coordonnée « horizontale » et y la coordonnée « verticale ». Le point de coordonnées $(0, 0)$ est le point situé en haut à gauche de l'image. L'axe horizontal est donc orienté vers la droite et l'axe vertical vers le bas.

- dispose d'un constructeur qui prend en paramètre la largeur et la hauteur de l'image en nombre de pixels.

  Les pixels qui composent l'image sont stockés dans un attribut qui est un **tableau à deux dimensions** d'objets `Pixel`.

  À la création de l'image, ces pixels sont tous de couleur blanche.

- définit une méthode :

```
public void changeColorPixel(int x, int y, GrayColor color)
```

  qui a pour effet de modifier la couleur du pixel de coordonnées (`x`,`y`) en lui attribuant la couleur `color`.

  Cette méthode déclenche une exception `UnknownPixelException` si le pixel de coordonnées (x, y) n'existe pas pour cette image (le code de la classe `UnknownPixelException` est fournie).

- définit une méthode

```
public void fillRectangle(int x, int y, int width, int height, GrayColor color)
```

  qui « dessine » dans l'image un rectangle « plein » de couleur `color`.

  En effet, dans l'objet `Image` concerné, cette méthode définit un rectangle dont tous les pixels sont de la couleur `color`. Ce rectangle a pour largeur `width` et hauteur `height`. Son coin supérieur gauche correspond au pixel de coordonnées `(x,y)` dans l'image.

Q 4 . Après avoir lu la documentation des méthodes de `ImageInterface`,

- définissez des méthodes de tests correspondant à ce cahier des charges de la classe `Image`,
- codez la classe `Image` respectant ces contraintes.

Q 5 . Consultez les documentations des classes `image.util.ImageLoader` et `image.util.ImageDisplayer`, puis étudiez le code de la classe `ImageExampleMain` fournie.

Compilez cette classe et copiez le dossier `images/` et son contenu dans votre dossier `classes`.

Exécutez la méthode `main()` de la classe `ImageExampleMain`.

Vous devriez voir s'afficher deux images, dont celle de gauche de la figure 1.

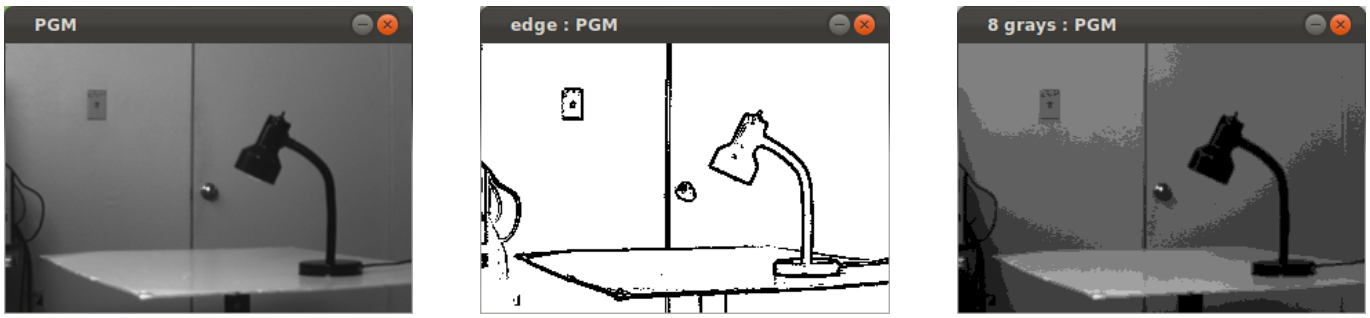


Figure 1: A gauche l'image initiale, au centre l'image résultat obtenue par extraction de contours avec un seuil de 10 et à droite l'image obtenue en diminuant à 8 le nombre de niveaux de gris.

Q 6 . Définissez, puis exécutez, une méthode `main` d'une classe `ImageMain` qui :

- crée une image $\mathcal{I}$ (blanche) de largeur 200 et hauteur 150 pixels.
- dessine dans cette image des rectangles :
  - noir de taille 30×50 à partir du point (20,30)
  - gris, niveau 128, de taille 40×40 à partir du point (50,100),
  - gris, niveau 200, de taille (70×50) à partir du point (90,20).
- affiche cette image en utilisant un objet de la classe `ImageDisplayer` et sa méthode `display`.

Q 7 . Ajoutez à la classe `Image` une méthode

```
public Image negative()
```

qui permet de créer une nouvelle image obtenue à partir de l'image initiale en remplaçant la couleur de niveau l de chacun des pixels par une couleur dont le niveau de gris est $255 - l$.

Remarque : pensez à utiliser la méthode `changeColorPixel()` de `Image`.

Q 8 . En vous inspirant de ce qui est proposé dans `ImageExampleMain`, complétez la méthode `main()` de classe `ImageMain` pour

- créer et initialiser un objet `Image` correspondant à un fichier dont le nom est fourni en paramètre de la ligne de commande,
- définir l'image obtenue en appliquant la méthode `negative()` à l'image précédente.
- afficher les deux images (pensez à décaler les coordonnées d'affichage des images pour qu'elles ne se superposent pas)

Vous devriez obtenir quelque chose similaire à la figure 2.

Figure 2: A gauche l'image initiale, à droite l'image en négatif.

Q 9 . Ajoutez à la classe `Image` une méthode

```
public Image edgeExtraction(int threshold)
```

qui permet de créer une nouvelle image obtenue à partir de l'image initiale par *extraction de contours*. Un exemple du résultat souhaité est présenté dans l'image du milieu de la figure 1.

Il n'est pas très compliqué d'obtenir l'image résultat (l'image des « *contours* ») à partir de l'image initiale :

En effet, une manière de procéder[2] est la suivante :

On commence par créer une nouvelle image de même taille que l'image initiale, cet objet contiendra l'image résultat. L'image résultat est donc initialement blanche.

Ensuite, on parcourt l'image initiale pour en examiner les pixels un a à un et décider si on modifie ou non le pixel correspondant dans l'image résultat. Un pixel de l'image résultat doit être changé en noir s'il appartient au contour de l'image initiale. Les autres pixels de l'image résultat (ce qui ne sont pas identifiés comme appartenant à un contour) restent blancs.

On dira qu'un pixel appartient à un contour s'il est *très différent* du pixel situé à sa droite ou s'il est *très différent* du pixel situé en-dessous de lui dans l'image initiale. On n'examinera donc pas les pixels de la dernière colonne ni ceux de la dernière ligne.

On considère que deux pixels sont « *très différents* » si la différence entre les niveaux de gris de leurs couleurs est supérieure à un *seuil* fixé. Le seuil à prendre en compte pour l'extraction des contours est la valeur du paramètre `threshold`[3] de la méthode `edgeExtraction`.

[2] imparfaite mais simple à mettre en œuvre
[3] threshold=seuil

Remarque : pensez à utiliser les méthodes `colorLevelDifference()` de `Pixel` et `changeColorPixel()` de `Image`.

Q 10 . Complétez la méthode `main()` de classe `ImageMain` pour

- définir l'image obtenue en appliquant la méthode `edgeExtraction()` à l'image initiale.
  La valeur du paramètre de `edgeExtraction()` peut elle aussi être fournie en argument du `main` (il est possible d'attribuer une valeur par défaut pour traiter le cas où aucune valeur n'est fournie sur la ligne de commande).
- afficher l'image obtenue, en plus des images déjà affichées, (à nouveau, pensez à décaler les coordonnées d'affichage des images pour qu'elles ne se superposent pas).

Q 11 . Ajoutez à la classe `Image` une méthode

```
public Image decreaseNbGrayLevels(int nbGrayLevels)
```

qui produit une nouvelle image obtenue à partir de l'image initiale en utilisant un nombre de niveaux de gris limité, déterminé par `nbGrayLevels`. On supposera, sans le vérifier dans le code, que `nbGrayLevels` est toujours une puissance de 2 comprise entre 2 et 128.

Un exemple du résultat recherché peut être observé à droite dans la figure 1.

On peut à nouveau travailler de manière assez simple[4] :

On commence par créer une nouvelle image résultat de même dimension que l'image initiale.

On va considérer que l'on décompose l'intervalle [0,255] en `nbGrayLevels` sous-intervalles de taille $t = \frac{256}{nbGrayLevels}$, et pour chaque pixel qui a une couleur dont le niveau de gris est dans l'intervalle $[k \times t, (k+1) \times t[$, alors on veut un pixel d'une couleur de niveau $k \times t$.

Pour produire l'image résultat, on parcourt l'image initiale, pixel par pixel. Et, pour chaque pixel, on considère l le niveau de gris de sa couleur, le pixel correspondant dans l'image résultat aura donc une couleur dont le niveau de gris est $(l/t) \times t$ (où « / » représente bien sûr la division entière).

Par exemple, si on `nbGrayLevels = 8`, alors $t = 32$. Si, dans l'image originale, le pixel de coordonnées $(23, 83)$ a comme niveau de gris de sa couleur $l = 102$ ($l \in [96, 128[$) alors dans l'image résultat le pixel de coordonnées $(23, 83)$ aura une couleur dont le niveau de gris est 96 ($= 3 \times 32$ car le quotient de 102 par $32 = 102/32 = 3$).

Q 12 . Complétez la méthode `main()` de la classe `ImageMain` pour afficher, en plus des autres images, l'image obtenue après application de la méthode `decreaseNbGrayLevels()` sur l'image initiale.

Vous modifierez cette méthode `main()` pour que le nom du fichier image, le seuil d'extraction de contours et le nombre de niveaux de gris à utiliser puissent être fournis comme arguments de la ligne de commande d'exécution du programme.

Par exemple :

```
···/image/classes> java image.ImageMain /images/fruit.pgm 15 16
```

Q 13 . Rendez votre travail en le déposant sur GitLab dans un dossier nommé `tp6/`. Vous ajouterez le dossier `images` dans votre dépôt.

Vous appliquerez les consignes de rendu pour la structure du dossier `tp6/` et contenu du fichier `readme.md`.

Dans ce fichier vous indiquerez notamment comment exécuter la méthode `main()` de `ImageMain` et comment produire un `jar` qui exécutera cette même méthode.

Le dossier `images` peut être ajouté à la racine de l'archive exécutable, il suffit de l'ajouter à la commande de création du jar, comme un paquetage de classes, comme ceci :

```
jar cvfe image.jar image.ImageMain -C classes image images
```

[4] Avec cette méthode une image à 2 niveaux de gris ne sera pas en noir et blanc, mais en noir et gris (128)