Examen (rattrapage) Java

Juin 2018

Durée : 3 heures

Instructions

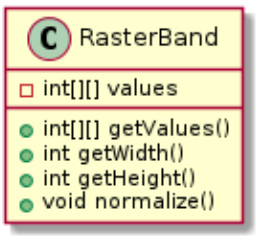
- L'examen sera réalisé sur papier libre et avec **Eclipse**.
- Créer un nouveau projet dans **Eclipse** que l'on nommera **Exam2-<votre_nom>**.
- Pour chaque exercice, créer un nouveau package portant le nom de l'exercice en minuscule (e.g. **exercice1**) et y inclure toutes les classes créées pour répondre à l'exercice.
- Sauf instruction contraire, lorsque des vérifications ou des tests sont demandés dans un exercice, ne **pas** utiliser de `main ...`
- À la fin de l'examen, créer une archive (*e.g.* `.tgz`, `.zip`) du répertoire correspondant au projet **Exam2-<votre_nom>**
- Les exercices sont classés par ordre de difficulté croissante. Il est donc fortement conseillé de les traiter dans l'ordre.

Exercice 1

(Barème 40%)
Configuration initiale

- Extraire le fichier `exercice1.zip` dans le répertoire `src` de votre projet **Eclipse**.
- Actualiser votre projet **Eclipse** en le sélectionnant dans **Eclipse** et en appuyant sur la touche *F5*, ou par un clic droit -> *Refresh*.

La classe `RasterBand` permet de modéliser une image 8 bits en niveaux de gris. Elle est définie par le diagramme ci-dessous :




```
/******** JAVADOC **************/
/**
* Returns the internal values.
*/
public int[][] getValues();

/**
* Returns the number of columns in values.
*/
public int getWidth();

/**
* Returns the number of rows in values.
*/
```

```java
public int getHeight();

/**
* Normalizes the values to [0, 255].
* Example, given i,j, values[i][j] = 255-values[i][j]
*/
public void normalize();

/**
* Computes the partial x gradient.
*/
public RasterBand getGradientx();

/**
* Computes the partial y gradient.
*/
public RasterBand getGradienty();

/**
* Computes the gradient.
*/
public RasterBand getGradient();
```

On définit l'opération `gradientx` sur une matrice $\mathbb{A}$ de n lignes et m colonnes comme suit :

$$\text{gradientx}(A_{i,j}) = \partial_x A_{i,j}$$

avec

$$\partial_x A_{i,j} \simeq \begin{cases} A_{i,j+1} - A_{i,j}, & \text{pour } j = 0 \\ \frac{A_{i,j+1} - A_{i,j-1}}{2}, & \text{pour } j = 1, \ldots, m-2 \\ A_{i,j} - A_{i,j-1}, & \text{pour } j = m-1 \end{cases}$$

On définit similairement l'opération `gradienty` sur une matrice $\mathbb{A}$ de n lignes et m colonnes comme suit :

$$\text{gradienty}(A_{i,j}) = \partial_y A_{i,j}$$

avec

$$\partial_y A_{i,j} \simeq \begin{cases} A_{i+1,j} - A_{i,j}, & \text{pour } i = 0 \\ \frac{A_{i+1,j} - A_{i-1,j}}{2}, & \text{pour } i = 1, \ldots, i = n-2 \\ A_{i,j} - A_{i-1,j}, & \text{pour } i = n-1 \end{cases}$$

Enfin, on définit l'opération `gradient` sur une matrice $\mathbb{A}$ de n lignes et m colonnes comme :

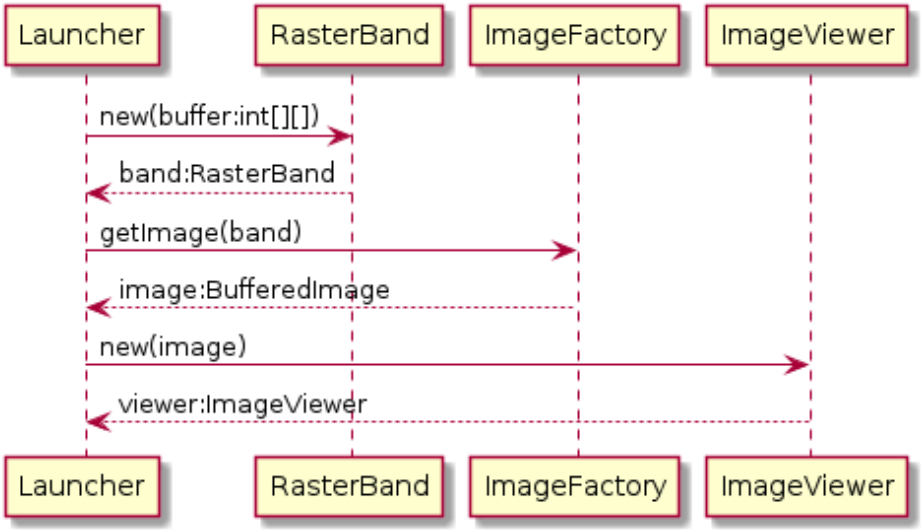
$$G(A_{i,j}) = \sqrt{(\partial_x A_{i,j})^2 + (\partial_y A_{i,j})^2}$$

1. Implémenter les méthodes `gradientx`, `gradienty` et `gradient` avec les informations fournies juste au dessus.
2. Tester l'implémentation en considérant la matrice :

$$\mathbb{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \end{pmatrix}$$

3. *Objectif bonus*
   Implémenter directement `gradient` (i.e. sans utiliser `gradientx` et `gradienty`)

La classe `ImageViewer`, fournie dans l'archive `exercice1.zip` permet de visualiser dans une fenêtre graphique un objet `RasterBand`. Le diagramme de séquence ci-dessous décrit les étapes nécessaires pour créer un objet `ImageViewer` à partir d'un objet `RasterBand` :



2. En utilisant le diagramme de séquence fourni, créer une classe de test `Launcher` avec une méthode `main` permettant de visualiser le buffer obtenu en appelant la méthode `Util.getLennaGreenBuffer()`.
   - Exécuter `Launcher` et vérifier qu'une fenêtre contenant une image apparait.
   - Que voyez-vous sur cette image ?
   - *Hint*

   ```
   ImageViewer viewer; // Initialize with something sensible
   viewer.pack();
   viewer.setVisible(true);
   ```

3. Créer une deuxième classe de test `Launcher2` avec une méthode `main` permettant de visualiser un objet `RasterBand` construit comme suit :

   a) `RasterBand` initialisé avec `Util.getLennaGreenBuffer()`

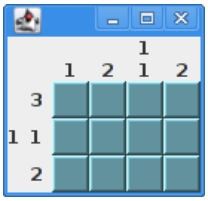
   b) Appel de `getGradient` sur l'object de l'étape précédente.

   c) Appel de `normalize` sur l'object de l'étape précédente.

   - Exécuter `Launcher2` et vérifier qu'une fenêtre contenant une image apparait.
   - Que voyez-vous sur cette image ? Décrire ce que fait concrètement l'opération `gradient` appliquée à une image.

Exercice 2

(Barème 60%)



Règles du picross

- Une cellule est dans un état rempli ou non-rempli.
- Les chiffres en haut d'une colonne (ou à gauche d'une ligne) constitue un "contrat" et indiquent le nombre et la taille de blocs de cellules devant être remplies pour cette ligne/colonne.
- Chaque ligne/colonne doit honorer son "contrat".
- Un bloc est constitué de cellules consécutives remplies.
- Deux blocs sont séparés par au moins une cellule non remplie.
- Exemple sur la figure :
  — La première colonne doit contenir une et une seule cellule remplie.
  — La première ligne doit contenir un bloc de 3 cellules consécutives remplies.

Configuration initiale

- Extraire le fichier `exercice2.zip` dans le répertoire `src` de votre projet **Eclipse**.
- Actualiser votre projet **Eclipse** en le sélectionnant dans **Eclipse** et en appuyant sur la touche *F5*, ou par un clic droit -> *Refresh*.

Etudier attentivement le code fourni dans l'archive `exercice2.zip`. Ce code correspond à un projet d'application graphique permettant de jouer à un picross dans une version à l'état de prototype.
On souhaite faire 2 **évolutions** au projet :

- Actuellement, chaque fois que l'utilisateur clic avec le bouton gauche sur une cellule, la cellule change d'un état "non-rempli" à un état "rempli", et vice-versa.
  On souhaite avoir le comportement suivant :
  — Si une cellule est dans un état "non-rempli", sur un clic gauche elle bascule dans un état "rempli" (comportement initial).
  — Si une cellule est dans un état "rempli", sur un clic gauche elle bascule dans un état "non-rempli" (comportement initial).
  — Si une cellule est dans un état "non-rempli", sur un clic droit elle bascule dans un état "barré".
  — Si une cellule est dans un état "barré", sur un clic droit elle bascule dans un état "non-rempli".
  — Si une cellule est dans un état "rempli", sur un clic droit rien ne se passe.
  — Si une cellule est dans un état "barré", sur un clic gauche rien ne se passe.
  On souhaite que l'état barré soit représenté comme l'état non rempli mais avec un "X" dans la cellule.
- On souhaite ajouter une aide de jeu permettant au joueur de connaître l'état courant de son picross, i.e. avoir une indication lui signalant si une ligne/colonne semble remplie correctement ou non.
  L'aide de jeu que l'on souhaite ajouter consiste à colorier les chiffres correspondant aux lignes/colonnes en **rouge** lorsque l'on "détecte" une erreur, en noir sinon.
  Dans une première version simpliste, la détection consistera à comparer la somme des chiffres d'une ligne/colonne avec le nombre de cellule remplie de cette ligne/colonne.

1. Dessiner le diagramme de classe correspondant au code fourni, en fournissant tout commentaire ou remarque que vous jugez utile à la compréhension du code fourni.
   - On pourra exécuter la classe "principale" `Launcher` et tester l'application pour aider à en comprendre le fonctionnement.
2. Objet *Cellule*
   - Quelles classes permettent de modéliser une *Cellule* ?
   - Quels sont les différents *états* dans lesquels une *Cellule* peut être ?
   - Expliquer ce qui se passe concrètement lorsqu'on clique sur une *Cellule*. Quelles sont les interactions entre les différentes classes mises en jeu (e.g méthodes appelées, impacts sur les objets...) ?
   - Les classes mises en oeuvre pour une *Cellule* utilise un *pattern* pour communiquer/intéragir. Lequel ?
3. Objet *Grille*
   - Expliquer simplement ce que représente la classe `Grid`.
   - Expliquer simplement ce que représente la classe `CellGroup`

- Expliquer à quoi correspondent concrètement les attributs `lines` et `columns` de la classe `Grid`.

Evolution 1 : Ajout d'un nouvel état "barré".

4. Sachant que la conception actuelle du code utilise fortement le pattern MVC, qu'est-ce que l'évolution implique comme modifications au niveau :
   - du modèle ?
   - de la vue ?
   - du controlleur ?
5. En déduire quelle(s) classe(s) il faut modifier pour réaliser l'évolution.
6. Faire toutes les modifications nécessaires pour réaliser l'évolution.
   Hint

```
CellWidget w;
// Draw an X in a cell
w.setText("X");
```

7. Ecrire une fiche de test permettant de valider l'évolution.

Evolution 2 : Ajout d'une aide de jeu.

8. Sachant que la conception actuelle du code utilise fortement le pattern MVC, qu'est-ce que l'évolution implique comme modifications au niveau de(s) :
   - modèle(s) ?
   - vue(s) ?
   - controlleur(s) ?
9. Quels composants doivent collaborer pour réaliser l'évolution ? Décrire par quel moyen ces composants communiquent.
   Hint : On pourra s'inspirer de ce qui est fait pour `Cell` et `CellGroup`.
10. En déduire quelle(s) classe(s) il faut modifier pour réaliser l'évolution.
11. Faire toutes les modifications nécessaires pour réaliser l'évolution.
    Hint

```
SpecWidget w;
// Set the color of the numbers to red
for (JLabel label:w.labels) {
  label.setForeground(Color.RED);
}
```

12. Ecrire une fiche de test permettant de valider l'évolution.