Pratique de la programmation orientée-objet
Examen intermédiaire

19 avril 2024

Indications :

— l'examen dure de 10h15 à 12h00,

— indiquez votre nom, prénom et numéro SCIPER ci-dessous et sur toutes les éventuelles feuilles additionnelles que vous rendriez,

— placez votre carte d'étudiant·e sur la table.

Seuls les documents suivants sont autorisés, en version papier uniquement :

— les notes de cours,

— les énoncés et les corrigés des séries d'exercices,

— une feuille de résumé de format A4 au maximum, ne couvrant que le cours.

Aucun document concernant le projet n'est autorisé !

Bon travail !

Nom : _____

Prénom : _____

SCIPER : _____

1 Animations continues [24 points]

Comme nous l'avons vu lors des séries d'exercices sur les images continues, une image peut être représentée au moyen de l'interface fonctionnelle `Image` suivante :

```
@FunctionalInterface
interface Image<T> { T apply(double x, double y); }
```

où la méthode `apply` prend en arguments les coordonnées (x, y) d'un point du plan et retourne la valeur de l'image en ce point.

Par exemple, ce que nous avons appelée une « image colorée » est représentée par le type `Image<ColorRGB>`, où `ColorRGB` est un type représentant une couleur. Un exemple d'une telle image colorée est celle d'un disque rouge de rayon 1, centré à l'origine, sur fond blanc :

```
Image<ColorRGB> redDisk = (x, y) ->
  x * x + y * y <= 1 ? ColorRGB.RED : ColorRGB.WHITE;
```

Le but de cet exercice est d'étendre cette idée afin de représenter non plus une image statique, mais une animation, c.-à-d. une image qui varie au cours du temps. Pour ce faire, nous ajouterons à la méthode `apply` un argument nommé `t` et représentant le temps, pour obtenir l'interface fonctionnelle `Animation` suivante :

```
@FunctionalInterface
interface Animation<T> { T apply(double x, double y, double t); }
```

Tout comme pour les positions `x` et `y`, ni l'origine ni l'unité du temps `t` ne sont spécifiées. Toutefois, le temps est toujours positif ou nul. Et si cela vous aide à réfléchir, vous pouvez considérer que, lorsqu'une animation est affichée à l'écran dans un programme, il est exprimé en nombre de secondes écoulées depuis son démarrage.

Partie 1 [5 points] Ajoutez à l'interface `Animation` une méthode statique et *générique* nommée `ofImage`, prenant en argument une image et retournant l'animation « fixe » qui lui correspond, c.-à-d. l'animation qui est en tout temps identique à l'image donnée.

Cette méthode devrait par exemple permettre d'obtenir une animation visuellement identique à l'image du disque rouge plus haut, ainsi :

```
Animation<ColorRGB> nonMovingRedDisk = Animation.ofImage(redDisk);
```

Réponse :

Partie 2 [5 points] Ajoutez à l'interface `Animation` une méthode par défaut nommée `moving`, prenant en arguments deux vitesses et retournant une animation identique au récepteur, mais qui se déplace parallèlement à l'axe des x à la première vitesse donnée, et parallèlement à l'axe des y à la seconde vitesse donnée.

Cette méthode devrait par exemple permettre d'obtenir une animation du disque rouge se déplaçant vers la droite de 0.1 unités de distance par unité de temps, et vers le bas de 0.2 unités de distance par unité de temps, ainsi :

```
Animation<ColorRGB> movingRedDisk = nonMovingRedDisk.moving(0.1, -0.2);
```

Réponse :

Partie 3 [5 points] Ajoutez à l'interface `Animation` une méthode par défaut nommée `snapshot`, prenant en argument un instant — c.-à-d. une valeur de type **`double`** représentant le temps — et retournant l'image de l'animation à laquelle on l'applique, à l'instant donné.

Cette méthode devrait par exemple permettre d'obtenir une image du disque rouge centré en $(0.1, -0.2)$, ainsi :

```
Image<ColorRGB> offCenterRedDisk = movingRedDisk.snapshot(1);
```

Réponse :

Partie 4 [5 points] Ajoutez à l'interface `Animation` une méthode par défaut portant le nom de `alternatingWith`, prenant en argument une autre animation du même type que le récepteur, et retournant une animation dont l'image alterne entre celle du récepteur — visible lorsque la partie entière du temps est paire — et celle de l'animation passée en argument — visible le reste du temps.

Cette méthode devrait par exemple permettre d'obtenir une animation alternant entre le disque rouge centré à l'origine, et ce même disque se déplaçant vers le bas et la droite, ainsi :

```
Animation<ColorRGB> ambivalentRedDisk =
  nonMovingRedDisk.alternatingWith(movingRedDisk);
```

Par exemple, au temps 0.5 cette animation montre le disque rouge centré à l'origine, tandis qu'au temps 1.5, elle le montre centré en $(0.15, -0.3)$.

Réponse :

Partie 5 [4 points] Les méthodes `ofImage`, `moving`, `snapshot` et `alternatingWith` utilisent chacune l'un des trois patrons de conception ci-dessous. À côté de chacun de ces patrons, notez le nom de la ou les méthodes qui l'utilisent :

1. *Decorator* _____
2. *Composite* _____
3. *Adapter* _____

2 Tableau d'affichage [25 points]

Dans le projet ChaCuN, l'enregistrement `MessageBoard` représente ce que nous avons appelé le tableau d'affichage. Il stocke la liste de tous les messages affichés à l'écran lors d'une partie. Ces messages ont principalement pour but d'informer les joueurs que certains d'entre eux ont remporté des points.

La version de `MessageBoard` utilisée dans le projet stocke dans chaque message les éventuels points qui lui sont associés, ainsi que l'ensemble des joueurs qui les ont remportés. Le but de cet exercice est de compléter la définition de l'enregistrement immuable `MessageBoard2` ci-dessous, qui utilise une représentation alternative dans laquelle les points des joueurs sont stockés dans la table associative `points`, et les messages sont de simples chaînes de caractères.

```
record MessageBoard2(Map<PlayerColor, Integer> points,
                     List<String> messages) { /* ... */ }
```

Pour mémoire, `PlayerColor` est un type énuméré représentant les couleurs des différents joueurs d'une partie. Sa définition simplifiée est :

```
enum PlayerColor { RED, BLUE, GREEN, YELLOW, PURPLE }
```

Partie 1 [4 points] Ajoutez à `MessageBoard2` un constructeur compact qui

1. valide la table `points` en levant une exception de type `IllegalArgumentException` si elle associe un nombre de points négatif ou nul à au moins un joueur, et
2. garantisse l'immuabilité de la classe.

Réponse :

Partie 2 [9 points] Ajoutez à `MessageBoard2` une méthode nommée `leaders` retournant l'ensemble des couleurs des joueurs ayant le plus grand nombre de points, qui est vide si et seulement si la table `points` l'est également.

Attention : votre méthode ne doit parcourir *qu'une seule fois* la table `points` pour déterminer son résultat !

Réponse :

Partie 3 [12 points] Ajoutez à `MessageBoard2` un bâtisseur sous la forme d'une classe imbriquée statiquement, nommée `Builder` et offrant :

— Un constructeur prenant en argument une valeur de type `MessageBoard2` et utilisant ses attributs comme valeurs initiales pour ceux du bâtisseur.

— Une première méthode nommée `addMessage`, prenant en argument un message (de type `String`) et l'ajoutant au tableau d'affichage en cours de construction, sans modifier les points. Cette méthode ne retourne aucune valeur.

— Une seconde méthode nommée `addMessage` prenant en arguments un message, un nombre de points et un ensemble de couleurs de joueurs. Elle ajoute le message au tableau d'affichage en cours de construction, et les points aux joueurs donnés. Cette méthode ne retourne aucune valeur, mais lève une `IllegalArgumentException` si le nombre de points donné n'est pas supérieur à zéro, ou si l'ensemble des joueurs est vide.

— Une méthode nommée `build` retournant le tableau d'affichage en cours de construction.

Réponse :

3 Plateau de jeu [26 points]

Une partie de ChaCuN se joue sur un plateau de 25×25 cases, indexées par une paire d'index (x, y) qui augmentent de gauche à droite et de haut en bas. La case centrale du plateau a pour index $(0, 0)$, et chacun des deux index est donc compris entre -12 (inclus) et $+12$ (inclus). La valeur 12 est ce que nous avons appelé la *portée* (*reach* en anglais) du plateau.

La classe `Board` du projet représente le plateau de jeu au moyen de deux tableaux. Le premier, de type `PlacedTile[]`, contient les tuiles, stockées dans l'ordre dit *row-major* en anglais, c.-à-d. que son premier élément correspond à l'index $(-12, -12)$, le second à l'index $(-11, -12)$, etc. Le second tableau, de type `int[]`, contient les index, dans le premier tableau, de toutes les tuiles posées, dans l'ordre dans lequel elles l'ont été.

Le but de cet exercice est de compléter la définition de la classe immuable `Board2` ci-dessous, qui est une représentation alternative du plateau de jeu de ChaCuN. Dans un souci de simplicité, cette version ne contient ni les partitions de zones, ni l'ensemble des animaux annulés.

```
public final class Board2 {
  public static final Board2 EMPTY = /* ... */;

  private final int reach;
  private final PlacedTile[] tiles;
  private final int[] indices;

  private static int sideLength(int reach) { return 2 * reach + 1; }
  private static int indexFor(int reach, int x, int y) { /* ... */ }

  private Board2(int reach, PlacedTile[] tiles, int[] indices) {
    this.reach = reach; this.tiles = tiles; this.indices = indices;
  }

  public Board2 withNewTile(PlacedTile t, int x, int y) { /* ... */ }
}
```

La différence principale entre cette classe et celle du projet est que la portée du plateau n'est plus une constante mais un attribut de la classe (`reach`). En conséquence, le tableau `tiles` contenant les tuiles n'a plus toujours une taille de 625 élément. Au lieu de cela, sa taille dépend de la portée r et est égale à $(2r + 1)^2$.

La portée est choisie de manière à être positive ou nulle, et aussi petite que possible pour inclure la position de toutes les tuiles posées. Ainsi, le plateau vide a une portée de 0, de même que le plateau contenant uniquement la tuile de départ à la position $(0, 0)$. Par contre, dès que la première tuile est placée, par exemple à la position $(1, 0)$, la portée du plateau passe à 1. Ensuite, si une deuxième tuile est placée à la position $(-1, 0)$, la portée du plateau reste à 1 ; par contre, si cette tuile est placée à la position $(2, 0)$, alors la portée passe à 2 ; et ainsi de suite.

Partie 1 [3 points] Écrivez la définition de l'attribut `EMPTY` de `Board2`, qui représente un plateau vide.

Réponse :

Partie 2 [8 points] Écrivez le corps de la méthode statique `indexFor` qui, étant donné une portée `reach` et une paire d'index (`x`, `y`), retourne l'index, dans le tableau des tuiles d'un plateau de la portée donnée, correspondant à (`x`, `y`), ou `-1` si cet index n'existe pas.

Note : la méthode `sideLength` peut simplifier votre travail.

Réponse :

Partie 3 [15 points] Écrivez le corps de la méthode `withNewTile`, qui retourne un nouveau plateau identique au récepteur (**`this`**), mais avec la tuile `t` dans la case d'index (`x`, `y`).

Votre méthode peut faire l'hypothèse que les arguments qu'elle reçoit sont valides selon les règles du jeu, sans devoir le vérifier explicitement. Elle peut donc supposer que la case d'index (`x`, `y`) n'est pas occupée au moment de l'appel ; que le plateau est vide ou que l'une des cases voisines de celle d'index (`x`, `y`) est occupée ; et que les bords des éventuelles tuiles voisines sont compatibles avec ceux de la nouvelle tuile.

Réponse :

Réponse :

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque Java nécessaires à cet examen. De nombreuses méthodes ont été omises et les types parfois simplifiés.

Classe `Math`

La classe `java.lang.Math`, non instanciable, contient des méthodes statiques permettant d'effectuer différents calculs.

```
public class Math {
  // Retourne le maximum de x et y.
  static int max(int x, int y);

  // Retourne la valeur absolue de x.
  static int abs(int x);

  // Retourne la partie entière de x, c.-à-d. le plus grand entier inférieur ou égal à x.
  static double floor(double x);
}
```

Interface `List`

L'interface `java.util.List` représente les listes. Elle est entre autres implémentée par la classe `ArrayList`, qui possède un constructeur de copie prenant une liste en argument.

```
public interface List<E> {
  // Retourne une copie immuable de l.
  static <E> List<E> copyOf(List<E> l);

  // Ajoute l'élément e à la fin de la liste et retourne true.
  boolean add(E e);
}
```

Interface `Set`

L'interface `java.util.Set` représente un ensemble. Elle est implémentée, entre autres, par la classe `HashSet`, qui possède un constructeur de copie prenant un ensemble en argument.

```
public interface Set<E> {
  // Retourne vrai ssi l'ensemble est vide.
  boolean isEmpty();

  // Ajoute l'élément e à l'ensemble et retourne true ssi il a été modifié en conséquence.
  boolean add(E e);

  // Efface la totalité des éléments de l'ensemble.
  void clear();
}
```

Interface `Map`

L'interface `java.util.Map` représente les tables associatives. Elle est implémentée, entre autres, par la classe `HashMap`, qui possède un constructeur de copie prenant une table associative en argument.

```
public interface Map<K, V> {
  // Retourne une copie immuable de m.
  static <K, V> Map<K, V> copyOf(Map<K, V> m);

  // Retourne vrai ssi la table contient la clef k.
  boolean containsKey(K k);

  // Associe la valeur v à la clef k.
  V put(K k, V v);

  // Retourne la valeur associée à k, ou null s'il n'y en a aucune.
  V get(K k);

  // Retourne la valeur associée à k, ou d s'il n'y en a aucune.
  V getOrDefault(K k, V d);

  // Retourne une vue sur l'ensemble des paires clef/valeur.
  Set<Map.Entry<K, V>> entrySet();
}
```

Interface `Map.Entry`

L'interface `java.util.Map.Entry` représente une paire clef/valeur.

```
public interface Map.Entry<K, V> {
  // Retourne la clef de la paire.
  K getKey();

  // Retourne la valeur de la paire.
  V getValue();
}
```

Classe `Arrays`

La classe `java.util.Arrays`, non instanciable, contient des méthodes statiques de manipulation de tableaux.

```
public class Arrays {
  // Retourne une copie du tableau original, de longeur newLength.
  // Les éventuels nouveaux éléments valent 0.
  static int[] copyOf(int[] original, int newLength);
}
```