Java DUT 1 - TP noté - 2015/2016 - Université Paris-Est Marne-la-Vallée

Durée: 2 heures - Documents interdits

On souhaite représenter des magiciens qui peuvent collecter des ingrédients et des potions et combiner les ingrédients entre eux, en suivant une recette, pour fabriquer de nouvelles potions.

Tous les tests nécessaires pour vérifier le bon fonctionnement des classes et méthodes demandées devront être écrits dans une classe `Test` faisant partie du travail à rendre. Vous pouvez utiliser les tests présents dans ce sujet. Cela ne signifie pas que vous ne devez pas y ajouter vos propres tests.

Exercice 1.—
On commence par écrire les classes représentant les ingrédients et les potions.

1. On définit une classe `Ingredient` pour représenter des ingrédients. Chaque ingrédient a un nom (`name`) et on utilisera un booléen `heal` pour représenter le fait qu'un ingrédient peut être utilisé pour soigner (`heal = true`) ou pour provoquer des dégâts (`heal = false`).

   Écrire une classe `Ingredient` avec les champs nécessaires, ainsi que le constructeur qui prend en argument tous les champs.

2. On définit une classe `Potion` pour représenter des potions. Chaque potion a une couleur (`color`), et une puissance (`power`) représentée par un entier. S'il est positif, c'est une potion de soin, s'il est négatif, c'est une potion qui fait des dégâts (et s'il est nul, c'est une potion sans effet).

   Écrire une classe `Potion` avec les champs nécessaires, ainsi que le constructeur qui prend en argument tous les champs.

3. Ajouter aux classes `Ingredient` et `Potion` ce qu'il faut pour que l'on puisse les afficher de la façon suivante (le résultat de l'affichage est dans le commentaire) :

```
Potion p1 = new Potion("Bleu", 4);
System.out.println(p1);
// Potion Bleu (4)
Ingredient i1 = new Ingredient("Pissenlit", true);
System.out.println(i1);
// Pissenlit (Soin)
Ingredient i2 = new Ingredient("Bave de crapaud", false);
System.out.println(i2);
// Bave de crapaud (Dégâts)
```

4. Redéfinir les méthodes `equals` des classes `Ingredient` et `Potion`.

5. Redéfinir les méthodes `hashCode` des classes `Ingredient` et `Potion` en utilisant respectivement (et uniquement) les `hashCode` des champs `name` et `color`.

6. Les magiciens souhaitent pouvoir regrouper les ingrédients et les potions dans une même collection. Pour pouvoir faire cela, écrire une **interface** `WizardItem` avec les 2 méthodes suivantes:

```
public int weight();
public int power();
```

   La méthode `weight` renvoie le poids d'un item (1 gramme pour un ingrédient et 3 grammes pour une potion) et la méthode `power` renvoie sa puissance. Dans le cas d'un ingrédient, la puissance est -1 s'il fait des dégâts et $+1$ s'il permet de soigner.

7. Rajouter ce qu'il faut pour que les classes `Ingredient` et `Potion` implémentent l'interface `WizardItem`.

Exercice 2.—

Nous allons maintenant créer de magiciens qui peuvent collecter des ingrédients et des potions.

1. On définit une classe `Wizard` pour représenter des magiciens. Chaque magicien a un nom (`name`) et un sac à dos (`backpack`) de type `HashMap<WizardItem, Integer>` pour porter ses ingrédients et ses potions. Dans le sac à dos, chaque item a une multiplicité (entière) qui indique la quantité de cet item dans le sac. Le sac à dos a également un poids maximal (`maxWeight`, que la somme des poids des items dans le sac ne peut pas dépasser), mais pas de limite de volume.

   Écrire une classe `Wizard` avec les champs nécessaires, ainsi que le constructeur qui prend en argument le nom du magicien et le poids maximal du sac à dos.

2. Rajouter un constructeur qui appelle le précédent mais fixe un poids maximal par défaut de 10.

3. Ajouter un champs `weight` qui va permettre de conserver le poids actuel (la somme des poids des items) du sac. Penser à l'initialiser.

4. Ajouter ce qu'il faut pour que l'on puisse afficher un magicien de la façon suivante:

```java
Wizard harry = new Wizard("Harry Potter");
System.out.println(harry);
// Harry Potter (0/10) : {}
```

   Entre parenthèses, on indique le poids actuel du sac par rapport au poids maximal, et l'affichage du sac à dos est effectué grâce à l'affichage par défaut de `HashMap` (ici, le sac à dos est vide, donc l'affichage est `{}`.

5. Écrire une méthode `add` qui prend en paramètre un `WizardItem` et l'ajoute dans le sac à dos. Si l'item que l'on veut ajouter est trop lourd, il n'est pas ajouté et la méthode affiche simplement "Le sac est trop lourd!".

   Attention à gérer correctement les multiplicités des items. Exemple:

```java
harry.add(new Potion("Bleu", 4));
harry.add(new Ingredient("Pissenlit", true));
harry.add(new Ingredient("Pissenlit", true));
harry.add(new Ingredient("Pissenlit", true));
harry.add(new Ingredient("Bave de crapaud", false));
harry.add(new Ingredient("Bave de crapaud", false));

System.out.println(harry);
// Harry Potter (8/10) : {Potion Bleu (4)=1, Bave de crapaud (Dégâts)=2, Pissenlit (Soin)=3}
```

6. Écrire une méthode `remove` qui prend en paramètre un `WizardItem` et le supprime du sac à dos. Attention l'item ne doit être supprimé qu'une seule fois et s'il n'est pas dans le sac à dos, il ne se passe rien.

```java
harry.remove(new Ingredient("Bave de crapaud", false));
System.out.println(harry);
// Harry Potter (7/10) : {Potion Bleu (4)=1, Bave de crapaud (Dégâts)=1, Pissenlit (Soin)=3}
harry.remove(new Ingredient("Bave de crapaud", false));
System.out.println(harry);
// Harry Potter (6/10) : {Potion Bleu (4)=1, Pissenlit (Soin)=3}
harry.remove(new Ingredient("Bave de crapaud", false));
System.out.println(harry);
// Harry Potter (6/10) : {Potion Bleu (4)=1, Pissenlit (Soin)=3}
```

Exercice 3.—

On veut finalement qu'un magicien puisse créer ses propres potions en suivant des recettes.

1. On définit une classe `Recipe` pour représenter des recettes. Chaque recette a une couleur (`potionColor`) pour indiquer la couleur de la potion qu'elle permet de préparer et un ensemble d'ingrédients (`HashSet<Ingredient> ingredients`) qui indique les ingrédients à utiliser pour la potion. Dans une recette, chaque ingrédient apparaît au plus une fois (d'où l'utilisation d'un `HashSet`).

   Écrire une classe `Recipe` avec les champs nécessaires, ainsi que le constructeur qui prend en argument tous les champs. Ajouter également les getteurs (attention, pour les ingredients, il faut réfléchir un peu...).

2. Ajouter ce qu'il faut pour pouvoir afficher une recette:

```
HashSet<Ingredient> ingredients = new HashSet<Ingredient>();
ingredients.add(new Ingredient("Pissenlit", true));
ingredients.add(new Ingredient("Bave de crapaud", false));
Recipe recipe = new Recipe("Vert", ingredients);
System.out.println(recipe);
// Recette pour une potion de couleur Vert:[Bave de crapaud (Dégâts), Pissenlit (Soin)]
```

3. Pour fabriquer une potion à l'aide d'une recette, il suffit de suivre la liste des ingrédients et d'additionner leurs puissances pour connaître la puissance de la potion créée. Dans la classe `Potion`, écrire une méthode statique `makePotion` qui prend en paramètre une recette et crée (et renvoie) une potion de la couleur correspondante et avec la bonne puissance.

```
System.out.println(Potion.makePotion(recipe));  // Vert (0)
```

4. Avant de se lancer dans une recette, un magicien doit vérifier qu'il a les ingrédients nécessaires. Dans la classe `Wizard`, écrire une méthode `tryPotion` qui prend en paramètre une recette et vérifie que le magicien a tous les ingrédients de la recette dans son sac à dos. Si c'est le cas, la méthode ne fait rien, sinon elle lance une exception de type `IllegalArgumentException`.

5. Dans la classe `Wizard`, écrire une méthode `tryAndMakePotion` qui prend en paramètre une recette et, après avoir vérifié qu'ils étaient présents, retire du sac à dos les ingrédients de la recette, fabrique la potion indiquée et l'ajoute au sac à dos.

   Remarque: si après avoir fabriqué sa potion, le magicien s'aperçoit que son sac à dos est trop lourd, la potion est perdue, et les ingrédients utilisés aussi.

```
harry.add(new Ingredient("Bave de crapaud", false));
System.out.println(harry);
// Harry Potter (7/10) : {Potion Bleu (4)=1, Bave de crapaud (Dégâts)=1, Pissenlit (Soin)=3}
harry.tryAndMakePotion(recipe);
System.out.println(harry);
// Harry Potter (8/10) : {Potion Bleu (4)=1, Potion Vert (0)=1, Pissenlit (Soin)=2}
harry.tryAndMakePotion(recipe);
// Exception in thread "main" java.lang.IllegalArgumentException: Harry Potter ne peut pas ...
```

6. Rajouter ce qu'il faut dans `Test` pour que la dernière instruction n'interrompe pas l'exécution.

7. Enfin, on souhaite pouvoir créer des recettes à partir de vieux manuscrits que l'on a recopiés dans des fichiers sous la forme suivante: la couleur sur la première ligne, puis une *, puis les ingrédients de type Soin sur les lignes suivantes, puis une *, puis les ingrédients de type Dégâts (voir exemples).

   Dans la classe `Recipe`, écrire une méthode statique `makeRecipe` qui prend en paramètre un nom de fichier et fabrique (et renvoie) une nouvelle recette créée à partir des informations contenues dans le fichier. Il n'est pas nécessaire de tester que le fichier est bien formé.

Exemple (`recette1.txt`):

```
Rose
*
Tomate
Lilas
Pissenlit
*
Bave de Crapaud
```

```java
Recipe r1 = Recipe.makeRecipe("recette1.txt");
System.out.println(r1);
// Recette pour une potion de couleur Rose:
// [Lilas (Soin), Bave de Crapaud (Dégâts), Tomate (Soin), Pissenlit (Soin)]
```

Exemple (`recette2.txt`):

```
Noir
*
Aubergine
*
Bave de Crapaud
Charbon
Poudre à Canon
```

```java
Recipe r1 = Recipe.makeRecipe("recette2.txt");
System.out.println(r3);
// Recette pour une potion de couleur Noir :
// [Bave de Crapaud (Dégâts), Aubergine (Soin), Poudre à Canon (Dégâts), Charbon (Dégâts)]
```

Exemple (`recette3.txt`):

```
Arc en ciel
*
Tomate
Lilas
Pissenlit
Aubergine
*
```

```java
Recipe r1 = Recipe.makeRecipe("recette3.txt");
System.out.println(r3);
// Recette pour une potion de couleur Arc en ciel :
// [Lilas (Soin), Tomate (Soin), Aubergine (Soin), Pissenlit (Soin)]
```