Cours MOOC EPFL d'introduction à la programmation orientée objet, illustré en Java

Troisième devoir (noté)
Héritage

J. Sam & J.-C. Chappelier

du 12 novembre au 30 novembre 2015

Ce devoir comprend deux exercices à rendre.

1 Exercice 1 — Choc des titans

Dans cet exercice, vous allez faire s'affronter des dragons et des hydres.

1.1 Description

Télécharger le programme fourni sur le site du cours [1] et le compléter.

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :
   `Window > Preferences > Java > Editor > Save Actions`
   (et décocher l'option de reformatage si elle est cochée)
2. sauvegarder le fichier téléchargé sous le nom `Dragons.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/`;

1. `https://d396qusza40orc.cloudfront.net/intropoojava/assignments-data/Dragons.java`

3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;
4. écrire le code à fournir entre ces deux commentaires :

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier apres cette ligne.
 *****************************************/
```

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `Dragons.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

Le programme principal fourni met en scène un combat entre un dragon et une hydre. La hiérarchie de classes qui permet de modéliser les créatures de ce jeu manque et il vous est demandé de la compléter.

1.1.1 La classe `Creature`

Une créature est caractérisée par :
- son nom (`nom`, une chaîne de caractère constante) ;
- son niveau (`niveau`, un entier) ;
- des points de vie (`pointsDeVie`, un entier) ;
- sa force (`force`, un entier) ;
- et sa position (`position`, encore un entier ; nous supposons que notre jeu est en 1D pour simplifier).

Vous respecterez strictement le nommage des attributs. Ces attributs seront accessibles dans les sous-classes de `Creature`.

Les méthodes à prévoir pour cette classe sont :
- un constructeur permettant d'initialiser le nom, le niveau, les points de vie, la force et la position de le créature au moyen de valeurs passées en paramètre, dans cet ordre ; le constructeur acceptera zéro comme valeur par défaut pour la position ;
- une méthode `boolean vivant()` retournant `true` si la créature est vivante (nombre de points de vie supérieur à zéro) et `false` sinon ;
- une méthode `pointsAttaque` retournant les points d'attaque que la créature peut infliger ; il s'agit du niveau multiplié par la force si l'animal est vivant et zéro sinon ;

— une méthode `deplacer(int)`, ne retournant rien et ajoutant en entier passé en paramètre à la position de la créature ;
— une méthode `adieux()` ne retournant rien et affichant le message suivant : `<nom> n'est plus!`
en respectant strictement ce format. `<nom>` est le nom de la créature ;
— une méthode `faiblir`, ne retournant rien et retranchant au nombre de points de vie de la créature, si elle est vivante, un nombre de points passés en paramètre ; si la créature meurt, son nombre de points de vie sera mis à zéro et la méthode `adieux` invoquée ;
— la méthode `toString()` permettant de produire une représentation de la créature sous la forme d'une `String` respectant scrupuleusement le format suivant :
```
<nom>, niveau: <niveau>, points de vie: <points>, force: <force>, \
points d'attaque: <attaque>, position: <position>
```
Le caractère `'\'` ne fait pas partie de l'affichage et n'est pas suivi d'un saut de ligne. `<nom>` est le nom de la créature, `<niveau>` son niveau, `<points>` son nombre de points de vie, `<force>` sa force, `<attaque>` son nombre de points d'attaque et `<position>` sa position.

1.1.2 La classe `Dragon`

Un `Dragon` est une `Creature`. Il a pour caractéristique spécifique la portée de sa flamme (`porteeFlamme`, un entier). Ses méthodes spécifiques sont :

— un constructeur permettant d'initialiser le nom, le niveau, les points de vie, la force, la portée de la flamme et la position du dragon au moyen de valeurs passées en paramètre, dans cet ordre ; le constructeur acceptera zéro comme valeur par défaut pour la position ;
— une méthode `voler(int pos)` ne retournant rien et permettant au dragon de se déplacer à la position `pos` ;
— une méthode `souffleSur(Creature bete)` ne retournant rien et simulant ce qui se passe lorsque le dragon souffle sur une `Creature` :

1. si le dragon est vivant, que la créature l'est aussi et qu'elle est à portée de sa flamme, alors le dragon inflige ses points d'attaque à la créature ; cette dernière faiblit du nombre de points d'attaque ; Le dragon faiblit aussi ; il perd `d` points de vie où `d` est la distance le séparant de la créature (plus il lance sa flamme loin et plus il s'affaiblit) ;
2. si au terme de ce combat épique, le dragon est toujours en vie, il augmente son niveau d'une unité s'il a vaincu la créature (qu'elle n'est plus en vie) ;

La créature est à portée de flamme du dragon si la distance qui les sépare est inférieure ou égale à la portée de la flamme (utilisez la fonction

`Utility.distance` fournie).

1.1.3 La classe `Hydre`

Une `Hydre` est une `Creature`. Elle a pour caractéristiques spécifiques la longueur de son cou (`longueurCou`, un entier) ainsi que la dose de poison qu'elle peut injecter par attaque (`dosePoison`, un entier). Ses méthodes spécifiques sont :

- un constructeur permettant d'initialiser le nom, le niveau, les points de vie, la force, la longueur du cou, la dose de poison et la position de l'hydre au moyen de valeurs passées en paramètre, dans cet ordre ; le constructeur acceptera zéro comme valeur par défaut pour la position ;
- une méthode `empoisonne(Creature bete)` ne retournant rien et simulant ce qui se passe lorsque l'hydre empoisonne une `Creature` :
  1. si l'hydre est vivante, que la créature l'est aussi et qu'elle est à portée du cou, alors l'hydre inflige des dommages à la créature ; cette dernière faiblit du nombre de points d'attaque de l'hydre augmentés de la dose de poison ;
  2. si au terme de ce combat, la créature n'est plus en vie, l'hydre augmente son niveau d'une unité ;

  La créature est à « portée de cou » de l'hydre si la distance qui les sépare est inférieure ou égale à la longueur du cou (utilisez la fonction `Utility.distance` fournie).

La fonction `combat` prend en paramètre un dragon et une hydre. Elle fait en sorte que :

- l'hydre empoisonne le dragon,
- puis le dragon souffle sur l'hydre.

1.2 Exemples de déroulement

L'exemple de déroulement ci-dessous correspond au programme principal fourni.

```
Dragon rouge, niveau: 2, points de vie: 10, force: 3, \
points d'attaque: 6, position: 0
se prépare au combat avec :
Hydre maléfique, niveau: 2, points de vie: 10, force: 1, \
points d'attaque: 2, position: 42

1er combat :
```

```
 Les créatures ne sont pas à portée, donc ne peuvent pas s'attaquer.

Après le combat :
Dragon rouge, niveau: 2, points de vie: 10, force: 3, \
points d'attaque: 6, position: 0
Hydre maléfique, niveau: 2, points de vie: 10, force: 1, \
points d'attaque: 2, position: 42

Le dragon vole à proximité de l'hydre :
Dragon rouge, niveau: 2, points de vie: 10, force: 3, \
points d'attaque: 6, position: 41

L'hydre recule d'un pas :
Hydre maléfique, niveau: 2, points de vie: 10, force: 1, \
points d'attaque: 2, position: 43

2e combat :
+ l'hydre inflige au dragon une attaque de 3 points
 [ niveau (2) * force (1) + poison (1) = 3 ] ;
+  le dragon inflige à l'hydre une attaque de 6 points
 [ niveau (2) * force (3) = 6 ] ;
 + pendant son attaque, le dragon perd 2 points de vie supplémentaires
[ correspondant à la distance entre le dragon et l'hydre : 43 - 41 = 2 ].

Après le combat :
Dragon rouge, niveau: 2, points de vie: 5, force: 3, \
points d'attaque: 6, position: 41
Hydre maléfique, niveau: 2, points de vie: 4, force: 1, \
points d'attaque: 2, position: 43

Le dragon avance d'un pas :
Dragon rouge, niveau: 2, points de vie: 5, force: 3, \
points d'attaque: 6, position: 42

3e combat :
+ l'hydre inflige au dragon une attaque de 3 points
[ niveau (2) * force (1) + poison (1) = 3 ] ;
+ le dragon inflige à l'hydre une attaque de 6 points
[ niveau (2) * force (3) = 6 ] ;
+ pendant son attaque, le dragon perd 1 point de vie supplémentaire
[ correspondant à la distance entre le dragon et l'hydre : 43 - 42 = 1 ] ;
+ l'hydre est vaincue et le dragon monte au niveau 3.
Hydre maléfique n'est plus !
```

```
Après le combat :
Dragon rouge, niveau: 3, points de vie: 1, force: 3, \
points d'attaque: 9, position: 42
Hydre maléfique, niveau: 2, points de vie: 0, force: 1, \
points d'attaque: 0, position: 43

4e Combat :
 quand une créature est vaincue, rien ne se passe.

Après le combat :
Dragon rouge, niveau: 3, points de vie: 1, force: 3, \
points d'attaque: 9, position: 42
Hydre maléfique, niveau: 2, points de vie: 0, force: 1, \
points d'attaque: 0, position: 43
```

Le caractère `'\'` ne fait pas partie de l'affichage et n'est pas suivi d'un saut de ligne.

2 Exercice 2 — Cuisine

Le but de cet exercice est de développer quelques fonctionnalités pour gérer des recettes de cuisine.

2.1 Description

Télécharger le programme fourni sur le site du cours [2] et le compléter.

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :
   `Window > Preferences > Java > Editor > Save Actions`
   (et décocher l'option de reformatage si elle est cochée)
2. sauvegarder le fichier téléchargé sous le nom `Restaurant.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/`;

2. `https://d396qusza40orc.cloudfront.net/intropoojava/assignments-data/Restaurant.java`

3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;
4. écrire le code à fournir entre ces deux commentaires :

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier apres cette ligne.
 *****************************************/
```

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `Restaurant.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

Le code fourni crée un recette sur la base de produits l'affiche, y trouve la quantité totale d'un produit donné et l'adapte à une quantité différente.

Un exemple de déroulement possible est fourni plus bas.

2.2 Classes à produire

Une *recette* est constituée d'une liste d'*ingrédients*. Elle a un nom et est aussi caractérisée par le nombre de fois que l'on souhaite la réaliser (un `double`, on peut faire une fois et demi la recette par exemple).

Un ingrédient est constitué :

— d'un *produit* ;
— d'une quantité (de ce produit, un `double`).

Un produit est caractérisé par :

— son nom (une chaîne de caractères, `"beurre"` par exemple) ;
— l'unité utilisée usuellement pour ce produit (une chaîne de caractères, `"grammes"` par exemple).

Il peut être *cuisiné* c'est-à-dire réalisé à partir d'une recette. Un produit cuisiné sera donc caractérisé par sa recette.

Il vous est demandé de coder les classes : `Produit`, `Ingredient`, `Recette` et `ProduitCuisine` correspondant à la description précédente.

Toutes les listes seront modélisées au moyen de `ArrayList`. Les ajouts dans ces listes se feront toujours **en fin de liste**.

La classe `Produit` Les méthodes publiques de la classe `Produit` seront pour commencer :

- un constructeur permettant d'initialiser le nom et l'unité du produit au moyen de valeurs passées en paramètre (dans cet ordre) ; par défaut l'unité est la chaîne de caractères vide ;
- les getters `getNom()` et `getUnite()` ;
- une méthode `toString()` retournant simplement le nom du `Produit`, sans aucun ajout ni modification.

La classe `Ingredient` Les méthodes publiques de la classe `Ingredient` seront :

- un constructeur permettant d'initialiser le produit et sa quantité au moyen de valeurs passées en paramètres dans cet ordre (le produit sera initialisé au moyen d'une référence sur le produit passée en paramètre, sans copie) ;
- les getters `getProduit()` et `getQuantite()` ;

  NOTE : vous ne ferez pas de « copie défensive » (protection contre les « fuites d'encapsulation ») du produit d'un ingredient. On veut en effet ici qu'un produit soit *partagé* par tous les ingrédients qui l'utilisent : si la recette d'un produit est modifiée, tous les ingrédients utilisant ce produit doivent avoir cette recette modifiée ; c'est donc bien – pour une fois – le même produit qui est partagé par tous (il s'agit en fait d'un *modèle* de produit) et ce n'est pas une « fuite d'encapsulation ».
- une méthode `toString()` produisant une représentation de l'`Ingredient` respectant ***strictement*** le format suivant :
  « `<quantite> <unite> de <representation_produit>` »
  où `<representation_produit>` est la chaîne de caractères représentant le produit [3] et `<unite>` est l'unité associée au produit.

La classe `Recette` Les méthodes publiques de la classe `Recette` seront pour commencer :

- un constructeur initialisant le nom et le nombre de fois qu'est réalisée la recette au moyen de valeurs passées en paramètre (dans cet ordre) ; par défaut, le nombre de fois vaut `1.` ;
- une méthode `void ajouter(Produit p, double quantite)` ajoutant un ingrédient construit au moyen des paramètres à la liste des in-

3. Voir la classe `Produit` et sa méthode `toString()`.

grédients de la recette. La quantité de l'ingrédient vaudra le nombre de fois que la recette est réalisée multiplié par la quantité passée en paramètre ;

— une méthode `Recette adapter(double n)` retournant une nouvelle recette correspondant à la recette courante réalisée `n` fois de plus (donc `n*nbFois` au total où `nbFois` est le nombre de fois qu'est réalisée la recette courante).
Notez ici que les ingrédients de la recette courante ont déjà vu leur quantité multipliée par le nombre de fois que la recette est à réaliser (voir méthode précédente) ; il faudra donc tenir compte de ce facteur dans le calcul des nouvelles quantités ;

— une méthode `toString()` produisant une représentation d'une `Recette` respectant ***strictement*** le format suivant :

```
Recette "<nom>" x <nb_fois>:
1. <ingredient_1>
2. <ingredient_2>
...
```

où `<nb_fois>` est le nombre de fois que la recette sera réalisée et `<ingredient_i>` est la chaîne de caractères représentant le i[e] ingrédient de la recette [4]. Voir l'exemple de déroulement plus bas pour plus de détails.
Attention : le dernier ingrédient **ne** devra **pas** être suivi d'un retour à la ligne ! On suppose que toutes les recettes ont au moins un ingrédient.

La classe `ProduitCuisine` Un `ProduitCuisine` est un `Produit`.

Les méthodes publiques de la classe `ProduitCuisine` seront :

— un constructeur conforme à la méthode `main()` fournie et permettant d'initialiser le nom du produit cuisiné au moyen d'une valeur passée en paramètre ; l'unité vaudra toujours `"portion(s)"` ; le nom de la recette du produit cuisiné sera le même que celui du produit ;

— une méthode `void ajouterARecette(Produit produit, double quantite)` permettant d'ajouter un ingrédient à la recette du produit (vous utiliserez la méthode `ajouter` de la classe `Recette`) ;

— une méthode `ProduitCuisine adapter(double n)` retournant un nouveau produit cuisiné correspondant au produit courant dont la recette est réalisée `n` fois ;

— une méthode `toString()` produisant une représentation d'un `ProduitCuisine`

4. voir la classe `Ingredient` et sa méthode `toString()`

en respectant ***strictement*** le format suivant :

`<produit>`
`<recette>`
où `<produit>` correspond à la représentation traditionnelle d'un produit[5] et `<recette>` correspond à la représentation de la recette du produit[6]. Voir l'exemple de déroulement plus bas pour plus de détails.
Attention : la représentation du produit cuisiné **ne** devra **pas** être suivie d'un retour à la ligne !

Le produit d'un ingrédient peut désormais être soit un produit (de base) soit un produit cuisiné. La méthode `toString()` de l'ingrédient devra utiliser en guise de `<representation_produit>` celle du produit dont la recette a été adaptée à la quantité.

La méthode `adapter` se doit donc d'être *polymorphique* pour les produits.

Retouchez donc la classe `Produit` en y ajoutant la méthode `adapter` nécessaire. Comme il n'y a rien à adapter pour un produit de base (pas de recette pour le produit dont il faudrait adapter les quantités) ; on se contentera de retourner l'objet courant.

Méthode `quantiteTotale` On souhaite finalement permettre de rechercher la quantité totale d'un produit de nom donné dans une recette. La méthode publique `double quantiteTotale(String nomProduit)` de la classe `Recette` cherchera le produit en question dans la liste de ses ingrédients (lequels peuvent être des produits cuisinés contenant aussi le produit recherché). Vous programmerez pour réaliser ce traitement des méthodes `double quantiteTotale(String nomProduit)` dans :

- la classe `Recette` : où elle retournera la somme des quantités totales de ce produit trouvées dans chaque ingrédient ;
- la classe `Produit` : où elle retournera `1.` si le produit a pour nom `nomProduit` et `0.` sinon ;
- la classe `ProduitCuisine` : où elle retournera `1.` si le produit a pour nom `nomProduit` et la quantité totale de ce produit dans la recette du produit cuisiné sinon ;
- la classe `Ingredient` : où elle retournera la quantité de l'ingrédient multipliée par la quantité totale du produit recherché dans le produit de l'ingrédient.

5. voir la classe `Produit` et sa méthode `toString()`
6. voir la classe `Recette` et sa méthode `toString()`

Il vous est donc demandé de coder la hiérarchie de classes et les différentes fonctionnalités découlant de cette description. Vous éviterez la duplication de code, le masquage d'attributs et nommerez les classes tel qu'il vous est suggéré de le faire dans l'énoncé.

2.3 Exemple de déroulement

```
glaçage au chocolat
  Recette "glaçage au chocolat" x 1.0:
  1. 200.0 grammes de chocolat noir
  2. 25.0 grammes de beurre
  3. 100.0 grammes de sucre glacé

glaçage au chocolat parfumé
  Recette "glaçage au chocolat parfumé" x 1.0:
  1. 2.0 gouttes de extrait d'amandes
  2. 1.0 portion(s) de glaçage au chocolat
  Recette "glaçage au chocolat" x 1.0:
  1. 200.0 grammes de chocolat noir
  2. 25.0 grammes de beurre
  3. 100.0 grammes de sucre glacé

===  Recette finale  ======

  Recette "tourte glacée au chocolat" x 1.0:
  1. 5.0  de oeufs
  2. 150.0 grammes de farine
  3. 100.0 grammes de beurre
  4. 50.0 grammes de amandes moulues
  5. 2.0 portion(s) de glaçage au chocolat parfumé
  Recette "glaçage au chocolat parfumé" x 2.0:
  1. 4.0 gouttes de extrait d'amandes
  2. 2.0 portion(s) de glaçage au chocolat
  Recette "glaçage au chocolat" x 2.0:
  1. 400.0 grammes de chocolat noir
  2. 50.0 grammes de beurre
  3. 200.0 grammes de sucre glacé

Cette recette contient 150.0 grammes de beurre

===  Recette finale x 2 ===

  Recette "tourte glacée au chocolat" x 2.0:
  1. 10.0  de oeufs
  2. 300.0 grammes de farine
  3. 200.0 grammes de beurre
  4. 100.0 grammes de amandes moulues
```

```
  5. 4.0 portion(s) de glaçage au chocolat parfumé
  Recette "glaçage au chocolat parfumé" x 4.0:
  1. 8.0 gouttes de extrait d'amandes
  2. 4.0 portion(s) de glaçage au chocolat
  Recette "glaçage au chocolat" x 4.0:
  1. 800.0 grammes de chocolat noir
  2. 100.0 grammes de beurre
  3. 400.0 grammes de sucre glacé

Cette recette contient 300.0 grammes de beurre
Cette recette contient 10.0  de oeufs
Cette recette contient 8.0 gouttes de extrait d'amandes
Cette recette contient 4.0 portion(s) de glaçage au chocolat

==========================

Vérification que le glaçage n'a pas été modifié :

glaçage au chocolat
  Recette "glaçage au chocolat" x 1.0:
  1. 200.0 grammes de chocolat noir
  2. 25.0 grammes de beurre
  3. 100.0 grammes de sucre glacé
```