Fiche 07 : Héritage

Objectifs

- Découvrir le concept d'héritage.

Vocabulaire

héritage	redéfinition	Chaîne de constructeurs	polymorphisme	lien dynamique

Exercices

1. Bar

Un bar désire avoir une application pour gérer les boissons qui se trouvent sur sa carte. Pour chaque boisson, il faut garder son nom, sa contenance (donnée en cl) et son prix. Le nom de la boisson avec la contenance permet d'identifier la boisson. Le prix d'une boisson doit être modifiable.

En plus des caractéristiques de toutes les boissons, les boissons alcoolisées doivent aussi garder un degré d'alcool. Parmi les boissons alcoolisées, on trouve les bières pour lesquelles il faut préciser si elles sont à la pression ou en bouteille et les vins pour lesquels il faut préciser le cépage, la couleur (rouge, blanc ou rosé), la région et le pays d'origine.

Toute boisson doit aussi pouvoir renvoyer toutes les informations la concernant sous forme de chaîne de caractères.

a) Dans un premier temps, faites le diagramme de classes UML correspondant à la hiérarchie des boissons sachant que :
   - toutes les classes possèderont un constructeur recevant en paramètre toutes les informations **nécessaires** afin d'initialiser ses attributs ;
   - il y aura des getters pour tous les attributs.

b) Une fois le diagramme valide, implémentez ces classes en java et testez-là à l'aide de la classe `TestBoisson` (disponible sur moodle).
Remarques :
   - Pour commencer, ne traiter que l'exception en cas de couleur invalide pour le vin. Gérez les autres cas d'exception que si vous avez fini tout le reste de la fiche.
   - Pour le format des chaînes de caractères renvoyées, basez-vous sur le fichier affichage_TestBoisson.txt (disponible sur moodle) fournissant la sortie attendue lors de l'exécution de la classe `TestBoisson`.

c) Il faut maintenant pouvoir gérer la carte des boissons du bar. Cette carte garde une liste de boissons. Plus précisément, il faut pouvoir :
   - ajouter une boisson (il ne faut pas de doublon) ;
   - retirer une boisson ;
   - voir si une boisson est présente ou non sur la carte ;
   - donner le nombre de boissons de la carte ;
   - récupérer les informations de la carte sous forme textuelle.

Complétez votre diagramme de classe UML

d) Complétez l'implémentation java et récupérez la classe `TestBar` sur moodle en l'adaptant, si nécessaire, afin qu'elle compile. Exécutez `TestBar` et vérifiez que vous obtenez bien l'affichage attendu (fourni dans le fichier affichage_TestBar).

e) Représentez les objets qui se trouvent en mémoire à la fin de l'exécution du programme `TestBar`.

2. Mag… IPL !

Dans cet exercice, nous allons décrire les données d'un jeu simulant des combats de magiciens.

Dans ce jeu, il existe trois types de cartes : les terrains, les créatures et les sortilèges.

Les terrains possèdent une couleur (parmi 5 : blanc('B'), bleu ('b'), noir ('n'), rouge ('r') et vert ('v').). Utilisez un tableau de `char` pour vérifier ces valeurs à l'instanciation ; si la couleur transmise n'est pas l'une permise alors on retient la valeur 'i'.

Les créatures possèdent un nom, un nombre de points de dégâts et un nombre de points de vie. Soit les points de dégâts et de vie sont tous deux fournis et doivent être des entiers strictement positifs, soit aucun des deux n'est fourni et ils valent 0.

Les sortilèges possèdent un nom et une explication sous forme de texte.

Toute carte possède un coût qui est un entier. Celui d'un terrain est 0.

Chaque classe aura un constructeur permettant de spécifier la/les valeurs de ses attributs. Attention, pour les créatures, il existe plusieurs constructeurs (voir ci-dessus).

Il existe des accesseurs pour tous les attributs mais aucun setter.

Il faut aussi définir, pour chaque carte, une méthode `fournirDetail()` qui, selon le type de carte, renvoie une chaîne de caractères répondant aux conformités suivantes :

```
Terrain - coût : 0 couleur : b
Créature - coût : 6 nom : Golem(4/6)
Sortilège - coût : 1 nom : Croissance Gigantesque
```

a) Dans un premier temps, proposez une hiérarchie de classes en UML qui permet de représenter les différents types de cartes.

b) Ensuite, lorsque ce diagramme est valide implémentez-le.

Le programme Java doit utiliser la conception orientée Objet (polymorphisme) et ne doit pas comporter de duplication de code (pas de copier-coller).

c) Il faut maintenant pouvoir garder la main d'un joueur c.-à-d. les cartes dont disposent le joueur. Dans ce jeu, une main est composée de maximum 10 cartes.

Il existe une méthode `piocher` permettant d'ajouter une carte à la main si le nombre de cartes déjà présentes dans la main le permet sinon elle lance une `TropDeCartesException` (récupérez cette classe sur moodle).

La main comporte également une méthode `jouer` permettant de jouer une carte. Cette méthode reçoit comme donnée le numéro de la carte dans la collection. Elle renvoie la carte qui vient d'être supprimée. Elle lance une `IllegalArgumentException` en cas de numéro invalide.

La main fournit aussi une méthode qui renvoie sous forme textuelle toutes les cartes de la main.

Complétez votre UML et implémentez-le.

d) Implémentez un programme de test `Magipl.java` constitué d'une main contenant divers types de cartes dont voici le résultat de l'exécution :

```
Là, j'ai en stock :
Terrain - coût : 0 couleur : b
Créature - coût : 6 nom : Golem(4/6)
Sortilège - coût : 1 nom : Croissance Gigantesque

Carte jouée Sortilège - coût : 1 nom : Croissance Gigantesque

Maintenant, il me reste :
Terrain - coût : 0 couleur : b
Créature - coût : 6 nom : Golem(4/6)
```

e) Poussez le bouchon encore plus loin … imaginez deux mains en cours. Une carte ne peut être présente que dans une seule main à la fois. Il faut donc savoir si une carte est dans une main. Il faut donc vérifier lorsqu'on pioche une carte qu'elle n'est pas déjà dans une autre main. À vous de trouver une manière de résoudre ceci. N'oubliez pas de tester votre code !