TP3 : ZOO - Zoologie Orientée Objet

notions : Héritage, Polymorphisme, Abstraction, Liaison dynamique, Délégation, Collections

---

On veut créer une application permettant la gestion d'un zoo. Ce TP, en 4 parties, sera réalisé sur 3 séances.

1 Première séance : héritage

1.1 Les animaux

Pour les animaux, les fonctionnalités indispensables sont l'affichage de l'animal, le calcul du coût de nourriture par jour et la comparaison de deux animaux.

On considèrera des espèces différentes, avec les propriétés suivantes :
- Chimpanzé : Végétarien, 3kg de fruits/jour, cri : hurlement.
- Autruche : Végétarien, 5kg de fruits/jour, nombre de plumes (modifiable), cri : beuglement.
- Aigle : Carnivore, mange de la viande (quantité variable d'un aigle à l'autre, 1kg par défaut), envergure, cri : sifflement.
- Orque : Carnivore, 0,1 kg de viande par kilogramme, cri : sifflement.
- Tigre du Bengale : Carnivore, 4 kg de viande, nombre de rayures, cri : feulement.
- Tigre Blanc : Carnivore, 4 kg de viande, nombre de rayures, cri : feulement.

Chaque animal a un nom qui lui est propre, ainsi qu'un poids. Le calcul du coût de nourriture par jour est obtenu par :
- pour un carnivore : $(quantite * 10)^2 + 100$
- pour un végétarien : $1.2 * log((quantite + 5) * 2 + 1)$

1.1.1 Diagramme de classes

Représenter graphiquement par un diagramme de classes (le diagramme d'héritage) les différentes classes utiles pour représenter ces animaux. Précisez les champs et les prototypes de méthodes, en respectant l'encapsulation (accesseurs, mutateurs, protections des champs et des méthodes).

Quelques éléments à prendre en compte pour construire votre représentation.
- quels sont les propriétés et comportements sont communs à tous les types d'animaux ?
- quels sont les propriétés et comportements qui sont spécifiques à certains types d'animaux ?
- quels sont les propriétés qui sont des constantes et non des variables pour un type d'animal (*i.e.* qui sont toujours identiques / *v.s.* peuvent être différents / entre deux animaux de même race) ?
- quels sont les propriétés qui sont constantes (*v.s.* qui peuvent évoluer) au cours de la vie d'un objet animal ?
- quels sont les comportements ou méthodes que doivent posséder tous les animaux ?
- parmi ceux-ci, quels sont ceux qui sont parfaitement définis et ceux qui ne le sont pas au niveau des animaux ?

1.1.2 Classes en Java

Écrire en Java ces classes, en respectant les notions d'encapsulation (privé ou public) et la notion de réutilisabilité. Ces classes seront définies dans le package `zoo.animal` (File → new → package).

Elles comporteront au moins les méthodes publiques suivantes :

- Un constructeur adéquat permettant de construire un objet avec les paramètres nécessaires et suffisants. Utilisez correctement les constructeurs des classes mères.
- Une méthode permettant l'affichage du contenu de l'objet. La méthode `String toString()` sera utile à cet effet, et il sera approprié qu'elle soit définie à chaque niveau de la hiérarchie (e.g. : la méthode `String toString()` de la classe Animal traite entre autre le nom et la quantité de nourriture, la méthode `String toString()` de la classe Tigre blanc traite le nombre de rayure, etc).
- Le calcul du coût de nourriture, de signature `double getCout()` ;
- La méthode `boolean equals(Object)` qui indique si 2 animaux sont identiques ou non. On considèrera que deux animaux sont identiques si ils ont le même nom.

Ordre de développement : Commencez par implanter une seule classe d'animaux (par exemple les autruches) et les super-classes nécessaire à ce type, **tester et appelez l'enseignant pour qu'il regarde votre code**. Ne coder les autres classes que dans un second temps ; utiliser le copier-coller ! Si implanter tous les types d'animaux prend trop de temps, arrêtez-vous dès que vous avez bien compris le mécanisme d'héritage : il vaut mieux passer à la suite !

1.1.3 Un programme de test

Ecrire une classe TestAnimaux contenant une méthode main() qui construit un objet aigle *Francois*, un chimpanzé *Florent* et une autruche *Nicolas*, affiche tous ces animaux et calcule le cout de nourriture par jour de chacun d'entre eux. Vous pouvez bien sûr utiliser d'autres types d'animaux si vous n'avez pas ceux-ci...

Créer une autre autruche de nom *Francois* et comparer les animaux entre eux, en utilisant la méthode `equals` puis avec l'opérateur `==`.

1.2 Le Zoo

1.2.1 La classe Zoo fournie

Le zoo est défini par son nom et par l'ensemble des animaux qui y vivent. Il faut pouvoir créer le zoo, ajouter un nouvel arrivant (animal) au zoo, afficher l'ensemble des animaux actuellement présents dans le zoo, calculer le coût de la nourriture du zoo.

Nous fournissons une classe `Zoo`, dans un paquetage zoo, répondant à ces spécifications. En voici le code (disponible également en `.java` sur le site) :

```
1 package zoo;
2 import java.util.LinkedList;
3
4 import zoo.animal.*;
5
6 /** Une classe représentant un Zoo :
7  * Un zoo a un nom, et peut contenir un nombre quelconque d'animaux.
8  * On peut : ajouter un nouvel animal, récupérer le nombre d'animaux, calculer
      le cout de nourriture
```

```java
 * total, récupérer une chaine de caractères représentant l'état du Zoo.
 */
public class Zoo {
    private String nom;
    private LinkedList<Animal> animaux;

    /** Construit le zoo de nom name, initialement vide */
    public Zoo(String name) {
        nom=name;
        animaux = new LinkedList<Animal>();
    }

    /** Ajoute l'animal a au zoo */
    public void ajoute(Animal a) {
        animaux.add(a);
    }

    /** Retourne le nombre d'animaux contenu dans le zoo */
    public int nbAnimaux() {
        return animaux.size();
    }

    /** Retourne le cout total du zoo par jour (cout de nourriture) */
    public double coutTotal() {
        double cout = 0;
        for(Animal a : animaux) {
            cout += a.getCout();
        }
        return cout;
    }

    /** Retourne une chaine de caractères repréentant l'état du zoo et de tous
        ces animaux */
    public String toString() {
        String s= "Zoo:"+nom+" avec "+nbAnimaux()+" animaux\n";
        for (Animal a : animaux) s += "     " + a + "\n";
        return s;
    }
}
```

Parcourir le code de la classe ; repérer les méthodes publiques (ou “méthodes de l'API de la classe”).

Remarquez qu'il n'est pas nécessaire de bien comprendre ce code pour utiliser le zoo : il suffit d'appeler les méthodes publiques de la classe.

Remarquez que la classe Zoo a en attribut une instance de la classe `LinkedList<Animal> animaux`. Une `LinkedList<>` est un type de ‘collection” java, qui implante, de façon générique, le type abstrait “liste chaînée”. Eh oui, en Java (...et en POO...), plus besoin de réimplanter les listes chaînées à chaque fois comme en C : il suffit d'utiliser la classe Java adéquate !

La classe `Zoo` utilise une `LinkedList<Animal>` pour stocker les animaux, plutôt qu'un tableau `Animal []`, car ainsi il n'est pas nécessaire de connaître le nombre d'animaux (la taille du tableau) à la création du zoo : on peut ajouter autant d'animaux que l'on veut !

Nous verrons bientôt en détail les collections Java. Pour le moment, contentez-vous de remarquer que le code de la classe Zoo est tout de même assez simple, de repérer les différences et similitudes avec les tableaux, et d'aller jeter un premier coup d'oeuil (5 min max par exemple) sur la Javadoc de la classe (dans votre outil de recherche Internet préféré, tapez “java 1.8 linkedlist”).

1.2.2 Un programme de test du Zoo

Récupérez le fichier `Zoo.java` sur le site ; ajoutez le à votre projet eclipse, dans le paquetage zoo.

Écrire un programme de test qui effectue les actions suivantes :
— Création du zoo « Minatec ».
— Ajout du tigre du Bengale « Fantôme » , de poids 120kg, 40 rayures.
— Ajout de l'autruche « Ann », de poids 50kg.
— Ajout du chimpanzé « Chita » de poids 30kg.
— Ajout de l'aigle « Roquette », d'envergure 200cm, de poids 5kg, mange 2 kg.
— Ajout de l'orque « Azog », de poids 9000kg.
— Affichage du coût de nourriture du zoo.
Testez.

2 Séance 2 : abstraction et délégation

Durant la seconde séance, terminer le travail de la séance 1 puis traiter les nouvelles questions qui suivent.

2.1 Abstraction

A votre avis, est-il logique de pouvoir créer (`new`) un `Animal` qui n'est ne serait ni un Tigre du Bengale, ni une Autruche, ni un Chimpanzé, ni .... ? Quel serait, par exemple, le coût de nourriture par jour d'un tel animal ?

Transformez vos classes d'animaux en rendant les classes et les méthodes appropriées abstraites.

Remarque : Le travail à faire est en fait très simple et ne devrait prendre que quelques minutes !

2.2 Approche par délégation pour le régime

2.2.1 Pourquoi une classe Regime et des classes dérivées ?

Le régime alimentaire est géré pour le moment directement dans la classe `Animal` et ses sous-classes, par un type de nourriture, une quantité par jour, et une méthode de calcul du cout.

C'est un peu simple... Dans un vrai programme, un "régime" serait défini par de nombreuses autres propriétés (attributs ; l'heure du repas par exemple) et comportements (méthodes)... De plus, classifier les animaux (distinguer les types des animaux) en fonction de leur régime, au moyen des sous-classes `Carnivore` et `Herbivore` n'est pas très approprié.

Il serait plus adapté que les régimes soient implantés au moyen de classes spécifiques : une classe abstraite `Regime` et des sous-classes concrètes, une par type de régime.

Avec cette nouvelle architecture logicielle, un animal possède un objet instance de l'une des classes définissant un régime. Le calcul du cout de nourriture, actuellement réalisé directement dans la classe `Animal`, est alors *réalisé par délégation* sur l'objet régime possédé par l'animal : la classe `Animal` a toujours une méthode qui retourne le coût de nourriture par jour, mais le code de cette méthode se contente d'interroger l'objet régime de l'animal.

2.2.2 Travail à réaliser

— Créer la classe Regime abstraite dans le package `nourriture` et deux sous classes Legume et Viande. Toutes ces classes ont une méthode de calcul de cout et une méthode `toString()`.
— Modifier la classe `Animal` : elle possède maintenant un attribut `regime` adapté à chaque animal. Modifier en particulier les constructeurs des animaux.
— Modifier la méthode `double getCout()` de la classe `Animal`. Est-elle toujours abstraite ?
— Les classe abstraites `Carnivore` et `Herbivore` sont-elles toujours utiles ?

3 Séance 3 : collections

Le directeur du zoo souhaite faire des améliorations au système gestion du zoo fait lors des séances précédentes. Voici la liste de ses exigences.

— Un Zoo ne doit jamais avoir deux animaux qui ont le même nom.
— Dans la classe Zoo, il faut maintenant une méthode permettant de récupérer un animal à partir de son nom. Comme cette méthode sera beaucoup utilisée, elle doit être aussi rapide que possible...
— Il faut également une méthode permettant de supprimer un animal à partir de son nom.
— Dans un zoo, il peut y avoir des naissances. Un animal doit donc connaître ses enfants. De plus, le constructeur de la classe Animal doit prendre le père et la mère de cet animal en paramètre (s'ils sont connus).

3.1 Utilisation de collections pour stocker les animaux

Transformer la classe `Zoo` pour répondre, aussi efficacement que possible, aux nouvelles spécifications sur cette classe.

On considèrera que l'ordre d'arrivée des animaux dans le zoo est important. La méthode `String toString()` devra, par exemple, le respecter.

Par ailleurs, on rappèle que deux animaux ne peuvent avoir le même nom, qu'il faut pouvoir retrouver l'animal par son nom (très rapidement, sachant que le zoo contiendra un très grand nombre d'animaux...), et qu'on veut désormais pouvoir enlever un animal du Zoo à partir de son nom.

3.2 Les enfants

— Dans quelle classe peut on définir les enfants ? Quelle type de collection est appropriée ?
— Modifier les classes et écrire le constructeur créant un animal avec un père et une mère.
— Écrire dans la classe Animal une méthode qui retourne le nombre de descendants (enfants, petits enfants, etc...).

Pour la dernière méthode, attention : deux animaux d'une même fratrie (frère et soeur) peuvent se reproduire entre eux ! Eh oui, ces petites bêtes n'ont aucune morale... Il faut faut faire attention à ne compter qu'une seule fois leur progéniture. Construire un `HashSet<Animal>` (ou un `HashSet<String>` contenant les noms) et le remplir au fur et à mesure du parcours des enfants, petits enfants (etc.) pourrait être une bonne idée...

3.3 Test...

Tester les classes, en utilisant le main suivant :
— Création du zoo « Manotec »
— Ajout du tigre du Bengale « Fantôme », de poids 120kg, 40 rayures
— Ajout du tigre du Bengale « Fantômette », de poids 100kg, 20 rayures
— Ajout du tigre du Bengale « Fantômas », de poids 80kg, 20 rayures, fils de Fantômette et Fantôme
— Ajout du tigre du Bengale « Mimi Geignarde », de poids 80kg, 20 rayures, fils de Fantômette et Fantôme
— Ajout du tigre du Bengale « Casper », de poids 80kg, 15 rayures, fils de Fantômas et Mimi Geignarde
— Affichage du nombre de descendants de Fantôme (ce devrait être 3...)
— Vérifier que l'orque « Paul », de poids 8000kg ne peut avoir Fantômas comme parent

4 Encore plus fort !

4.1 Optimiser la méthode `getCout()`

La méthode `getCout()` du zoo est de complexité $O(nbanimaux)$. Comment pourrait-on l'implanter en $O(1)$, grâce au principe d'encapsulation, en ajoutant un attribut à la classe `Zoo` ? Est-ce que cela changerait l'API de la classe Zoo ?

4.2 Les animaux triés du moins cher au plus cher ? Facile !

Ajouter à la classe Zoo une méthode `TreeSet<Animal> getAnimauxTriesParCout()` qui retourne une collection d'animaux ordonnée en fonction de leur coût journalier.

4.3 Interface CoutEvaluable

Dans la vraie vie, le coût de fonctionnement d'un Zoo par jour ne se limite pas au coût de la nourriture des animaux. S'y ajoute le coût journalier des employés, du chauffage, de l'eau, etc, etc. Tous ces éléments sont très différents, mais ils ont tous un coût par jour : ils possèdent tous une méthode `double getCout()`.

Votre chef de projet logiciel décide alors de doter la classe Zoo d'une collection qui regroupera tous les objets du zoo qui ont un coût journalier. Ainsi, dit-il, la méthode `double getCout()` du Zoo sera très simple à écrire [1].

— Créer une interface `CoutEvaluable`, qui ne définit qu'une méthode : `double getCout()`.
— Assurez vous que la classe `Animal` réalise cette interface.
— Doter le zoo d'une collection de `CoutEvaluable`.
— Assurez vous d'ajouter l'animal à cette collection quand il est ajouté au zoo.
— Créez une nouvelle classe, qui n'est pas un animal et qui réalise cette interface - par exemple, une classe `Employe` (nom, coût par jour), ou si vous préférez une classe `Ampoule`. Doter la classe `Zoo` d'une méthode permettant d'ajouter au zoo une instance de cette nouvelle classe (par exemple : ajouter un employé).
— Modifiez la méthode `getCout()` de la classe `Zoo` : c'est désormais la somme de tous les objets `CoutEvaluable` du Zoo.
— Tester le calcul du coût total de votre zoo.

1. ce choix est en fait discutable... mais puisque le chef vous le demande, eh bien... soit !