TD 3 - POO : L'abstraction

Tokimahery Ramarozaka : Petit mot avant que vous ne commenciez... ce chapitre risque d'être assez "abstrait" comme son nom l'indique. Mais je vous promets, cela ne signifie pas qu'il est complexe, loin de là. L'idée de l'abstraction est une suite de l'héritage en fait : on peut spécifier des méthodes dans la classe mère, mais sans décrire le corps de la fonction immédiatement, plutôt le décrire dans les classes filles... Bizarre non ? Si vous vous posez la question "pourquoi faire", c'est normal ! J'étais passé par là aussi !

L'aspect important est de comprendre l'abstraction et comment **les classes abstraites peuvent être utilisées comme modèle commun pour des classes liées.**

Dans le milieu de l'enseignement, on vous fait souvent faire l'exercice sur les "Shapes" (forme), qui est une classe abstraite contenant la méthode abstraite "getSurface", qui est spécialisée par la suite par les classes "Circle", "Triangle", "Square", ou encore les classes "Animal", qui aura pour classes filles "Cat", "Dog", "Mouse" etc. Mais bon, j'ai essayé de trouver des exos plus concrets pour cette fois. Enjoy 😀

Exo 7 : Pourquoi des classes abstraites ?

Supposons que vous ayez une société qui propose de commander des produits venant d'Europe à Madagascar. Chaque produit (ou Product) aura notamment **son nom, son prix et sa description, et d'autres informations selon la catégorie du produit**. Vous souhaitez donc représenter chacune de ces catégories de produits et afficher sur votre site le prix TTC (livraison incluse) :

1) Dans un premier temps, implémentez comme vous le sentez les classes *Electronics (id, name, description, unitPrice, brand, weightInKg)*, et *Clothing (id, name, description, unitPrice, size, material)* pour les deux classes, créer une méthode pour calculer le prix TTC avec livraison :
    a) Pour les vêtements, la formule est : le prix unitaire + 5% du prix unitaire si la taille est 'L', 'XL', ou 'XXL' + 10% si le vêtement est en 'cotton';
    b) Pour les appareils électroniques, la formule est : le prix unitaire + 15000 par kilo;

2) On veut voir la liste des produits dans votre magasin. Représenter cela avec une classe *Shop* qui contient une liste de tous les produits, qu'ils soient *Electronics* ou *Clothing* avec leurs attributs respectifs. Ajoutez les méthodes suivantes dans la classe *Shop* :
    a) *addProduct* : pour ajouter un produit à votre magasin;
    b) *displayProduct* : pour afficher sur la console l'ensemble des produits déjà dans votre magasin.

Maintenant, parlons de la question concernant la structure de données :

3) N'y a-t-il pas comme un problème quand on définit cette structure de données? C'est une List oui, mais... List<QUOI?>, comment lui dire qu'il faut que ce soit une instance d'Electronics ou de Clothing ?
4) En se basant sur l'abstraction et l'héritage: proposer des solutions (à débattre en classe sur comment vous avez fait);
5) Si l'on venait à ajouter un autre type de produit plus tard : comparez la méthode sans abstraction et avec abstraction, doit-on ajouter du code dans Shop ? Donnez vos commentaires, analysez.

Conclusion : à travers le polymorphisme, c'est au niveau du runtime (execution) que les types des variables sont vraiment définis, même s'ils sont référencés en tant que leur classe mère. Ce sont des Product, mais une fois le programme lancé, ils pourront prendre leur "vraie forme" respective : *Electronics* ou *Clothing*, et exécuter le bon code en conséquence. C'est le concept de polymorphisme dynamique ou de "late binding".

Exo 8 : Un autre exemple de classe abstraite

On veut représenter différents types de réservation. Pour cela, définir la classe abstraite "Booking" :

- Ajouter des variables d'instance pour stocker les informations communes à toutes les réservations : l'identifiant de réservation, le nom du client, la date de réservation).
- Inclure une méthode abstraite appelée "*calculateTotalCost*()" pour calculer le coût total de la réservation (à implémenter par les sous-classes).

Parmi les classes spécifiques, définir la classe "RoomBooking" :

- Ajouter des variables d'instance pour stocker des informations spécifiques aux réservations de chambres : le numéro de chambre, le nombre de nuits, le type de chambre ('Simple', 'Double', 'VIP').
- Implémenter la méthode "calculateTotalCost()" pour les réservations de chambres en fonction du type de chambre et du nombre de nuits :

Définir la classe "ActivityBooking" :

- Ajouter des variables d'instance pour stocker des informations spécifiques aux réservations d'activités (par exemple, le nom de l'activité, le nombre de participants, la durée de l'activité).
- Implémenter la méthode "calculateTotalCost()" pour les réservations d'activités en fonction du nombre de participants et de la durée de l'activité :

Dans la méthode principale (ou dans une classe de test séparée) :

- Créer une liste unique contenant à la fois des "RoomBooking" et des "ActivityBooking",
- Calculer le prix total de toutes les réservations dans la liste.
- Si l'on ajoute des classes filles : FlightBooking, CarRentalBooking... a-t-on besoin de modifier quelque chose dans ce calcul du prix total ?

Moralité de l'histoire : ils sont des instances de Booking à la compilation, mais lors de l'exécution la méthode *calculateTotalCost()* appropriée est appelée, sans avoir à se dire : ***if objet instanceof classe*** et tout... Ils peuvent prendre plusieurs formes : c'est le polymorphisme dynamique, avec des classes abstraites.

Et le polymorphisme statique? Les interfaces ?... vous connaissez ?