TD 2 - POO : la notion d'héritage et de généralisation

Tokimahery Ramarozaka : nous avons vu comment nous pouvons créer des classes pour représenter des choses de la vraie vie dans un programme, et comment nous pouvions composer nos classes pour en créer de nouvelles. Nous avons également appris comment contrôler l'accès à certains attributs ou méthodes avec l'encapsulation, à travers les visibilités public et private ainsi que les getters et les setters. Supposons que l'on veuille gérer l'école et qu'on ait des utilisateurs qui peuvent être des étudiants, des enseignants, ou des administrateurs.

Dans ces TD, nous allons comprendre un autre pilier de la POO : l'héritage, qui nous permettra entre autres de décrire par exemple qu'une classe Student "hérite" des attributs et méthodes de la classe User (car Student est un User, avec quelques particularités en plus).

Ce mécanisme aura deux avantages : 1) on n'a pas à se répéter que la classe User et Student ont tous les deux un nom, un prénom, un mail et un mot de passe par exemple; et 2) les Student compte comme des User également, et peuvent faire tout ce qu'un User peut faire.

Pas très convaincu ? c'est normal ! voyons ce que l'héritage est en pratique, et pourquoi c'est si génial ! Let's go !

Exo 5 : C'est quoi l'intérêt de l'héritage ?

Suite à l'exercice 4 sur la classe Account(compte normal), on souhaite maintenant créer deux autres types de comptes : les comptes épargnes (SavingsAccount), et les comptes courants (CurrentAccount):

- Un compte épargne est **pareil à un compte normal** (les mêmes attributs et les mêmes méthodes) excepté qu'il comporte en plus un taux d'intérêt (nommé *interest*), il dispose aussi d'une méthode *applyInterest* qui ajoute le taux d'intérêt au solde du compte;
- Un **compte courant est pareil à un compte normal**, excepté qu'il y a un taux de découvert autorisé (solde négatif autorisé), notamment l'utilisateur n'est pas permis de dépasser ce solde négatif quand il va retirer ou transférer l'argent de son compte.

1) Sans utiliser la notion d'héritage, écrire les classes *SavingsAccount* et *CurrentAccount*. Testez que tout fonctionne si nécessaire. Que remarquez-vous en observant le code de Account, *SavingsAccount*, et *CurrentAccount* ? N'y a-t-il pas comme un problème ?
2) Maintenant que vous avez vu le problème (supposément), utilisez l'héritage pour écrire ces deux classes. N'oubliez pas de tester le tout dans une classe OtherAccountTest les nouvelles classes *SavingsAccount* et *CurrentAccount*.

Vous le remarquerez mais, retirer de l'argent d'un compte courant n'est pas la même chose que retirer d'un compte normal ou d'un compte épargne : en effet, il peut y avoir un solde négatif, jusqu'à une certaine limite. Voyez également qu'on peut utiliser @*Override* pour redéfinir certaines méthodes au niveau de *CurrentAccount*.

Exo 6 : Ok, mais c'est quoi la généralisation ?

Maintenant on souhaite créer une classe *Bank* qui regroupe un ensemble de comptes ouverts et fermés. Mais quel type serait susceptible d'accepter à la fois des *Account*, des *SavingsAccount*, et des *CurrentAccount* ? Hum, si seulement on pouvait les "généraliser"... À vous de jouer !

Pour information, les comptes de la classe Bank peuvent être des instances de *Account*, *SavingsAccount*, ou de *CurrentAccount*.

La classe Bank aura une méthode update qui servira à mettre à jour la liste des comptes qui mettra à jour les comptes en fonction de leur types :

- Pour les comptes épargnes, le taux d'intérêt sera ajouté au solde du compte en question (méthode que vous avez déjà écrite);
- Pour les comptes courants, si leur solde est négatif, il basculera dans la liste des comptes fermés.

Testez que chaque méthode de la classe Bank fonctionne correctement.

Ce fut très court, mais vous avez pu comprendre le concept. Alors... convaincu(e)des avantages de l'héritage ? ou il vous faut d'autres TDs encore ? nous allons ajouter un sujet de plus... en présentiel. À très bientôt. 😉