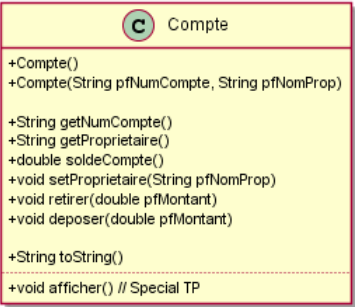
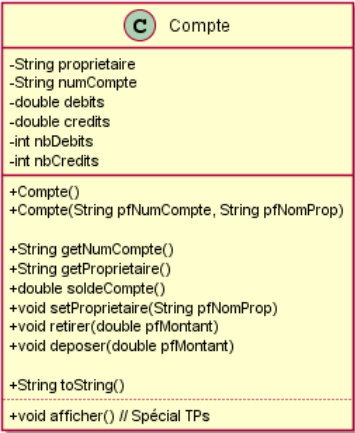


Table des matières

- 1. Classe à développer
- 2. Quelques explications
- 3. Version 1
- 4. Usage Version 1
- 5. Version 2
- 6. Usage Version 2
- 7. Conclusion

PreReq	Cours 1 : approche objet. TD2 objets. S1.
ObjTD	Ecrire une classe simple en 2 versions - Encapsulation.
Durée	1 séance de 1,5h

1. Classe à développer

Diagramme UML de l'interface de la classe Compte	Diagramme UML complet de la classe Compte
	

2. Quelques explications

Pour les méthodes :

- 1. `Compte()` : Constructeur non paramétré, appelé aussi constructeur "par défaut" : initialise un compte avec des débits/crédits à 0 euros et 0 opérations. Numéro compte : "Pas de numéro", propriétaire : "Pas de propriétaire".
- 2. `Compte(String pfNumCompte, String pfNomProp)` : Constructeur paramétré : initialise un compte avec des (débits/crédits) à 0 euros et 0 opérations. Numéro compte et propriétaires sont passés en paramètre
- 3. `String getNumCompte()` : permet de connaître le numéro de compte.
- 4. `String getProprietaire()` : permet de connaître (renvoyer) le nom du propriétaire.
- 5. `double soldeCompte()` : comme son nom l'indique : la somme restante sur le compte.
- 6. `void setProprietaire(String pfNomProp)` : modifie le propriétaire du compte.
- 7. `void retirer(double pfMontant)` : permet de retirer un montant du compte. Elle lève une exception `CompteException` si montant < 0.

8. `void deposer(double pfMontant)` : permet de déposer un montant sur le compte. Elle lève une exception `CompteException` si montant  $< 0$ .
9. `void afficher()` : affiche à l'écran `this.toString()` (redondante) **Spécial pour les TPS uniquement**
10. `String toString()` : Permet d'obtenir la représentation `String` d'un compte. Construit une chaine contenant le numéro de compte, le propriétaire, le total des débits, le total des crédits, le nombre d'opérations réalisées.  
" Num. : aaa - Prop. : bbb - Debit xxx E / Credit yyy E / NbOps zzz"

---

## 3. Version 1

---

1. Écrire la classe (le corps des méthodes) avec les attributs définis dans la page associée.

---

## 4. Usage Version 1

---

1. Ecrire un petit programme permettant de créer un `Compte` avec numéro et propriétaire, déposer 1000 €, retirer 500 €, changer le propriétaire, l'afficher, afficher le solde.

---

## 5. Version 2

---

Modifier d'une autre couleur le code précédent pour obtenir une nouvelle version de la classe en suivant les changements suivants :

1. Les débits et crédits restent stockés, mais on ajoute un attribut `solde` valant à tout moment le solde du compte. Ainsi le solde n'est pas à recalculer dans `soldeCompte()`.
2. le détail du nombre d'opérations n'est plus conservé en détail, seul le nombre total d'opérations est conservé.

```
package tps.banque;
import tps.banque.exception.CompteException;
public class Compte {
    // Nom du propriétaire
    private String proprietaire;
    // Numero de Compte
    private String numCompte;
    // Attributs debit et credit : additionner depots et retraits
    private double debits, credits;
    private double soldeCourant ;
    // Nb Opérations : compter depots et retraits
    private int nbOperations;
    // ...
}
```

---

## 6. Usage Version 2

---

Que pouvez-vous dire du code d'utilisation de la version précédente ? Fonctionnerait-il sans problème avec cette nouvelle version de classe ?

---

## 7. Conclusion

---

C'est cela que l'on appelle l'encapsulation : Tant que l'interface de la classe est respectée par l'implémenteur et le client/utilisateur :

- l'implémentation est complètement indépendante de l'utilisation,
- l'utilisation est complètement indépendante de l'implémentation.

Le mécanisme utilisé est :

- les attributs sont TOUJOURS privés
- l'interface (la liste des méthodes publiques) reste stable : identique en nombre de méthodes et signatures.

---

Dernière mise à jour 2017-02-16 12:38:09 CET