TP de PPO en JAVA

Polytech Lille GIS4 2018-2019

Objectifs : hiérarchie de classes, héritage et redéfinition de méthode, typage statique et dynamique, modularité.

1 Hiérarchie de classes de compte vue en cours

Travailler dans un répertoire ``comptes''.
Programmer la hiérarchie de classes `Compte` et `CompteEpargne` vue en cours :

- `Compte : crediter(double x), debiter(double x), solde(), toString()` (pour afficher credit, debit et solde), contructeur paramétré par un crédit initial.
- sous-classe `CompteEpargne` :
  - `interets()` et `echeance()`
  - redéfinition de la méthode `debiter()` (pas de solde negatif permis) et de `toString()` (pour afficher en plus les intérêts)
  - un constructeur paramétré par un crédit initial et un taux d'intérets.

2 Héritage et typage(s)

Ecrire un programme de test dans une classe principale (`main`) `TestComptes` qui manipule 2 variables : `unCompte` et `unCE` respectivement de type `Compte` et `CompteEpargne`.

Typage statique

Vérifier (à la compilation) que :

- les méthodes de `Compte` sont applicables sur les 2 variables (héritage)
- les méthodes de `CompteEpargne` ne sont applicables que sur `unCE`
- l'affectation : ``unCompte = unCE;'' est valide (sous-typage) et non l'inverse : ``unCE = unCompte;''

Redéfinition de méthode

Créer une instance de `Compte` dans `unCompte` et une instance de `CompteEpargne` dans `unCE`. Vérifier que les méthodes redéfinies `debiter()` et `toString()` s'exécutent différemment selon qu'elles sont appliquées à `unCompte` ou à `unCE`.

Typage dynamique

Dans `TestComptes` programmer la procédure suivante :

```
static void debitInteractif(Compte c) {
  // demander a l'utilisateur un montant a crediter et un montant a debiter
  // crediter et debiter c
  // afficher le nouvel etat de c
}
```

Dans le `main` tester la portion de code suivante pour vérifier la liaison dynamique des méthodes `debiter()` et `toString()` dans `debitInteractif(c)` en fonction du type dynamique de `c` :

```
debitInteractif(unCompte); // => type dynamique de c = Compte
debitInteractif(unCE); // => type dynamique de c = CompteEpargne
unCompte = unCE ;
debitInteractif(unCompte); // => type dynamique de c = ?
```

3 Opérations historisées (supplément)

Créer un répertoire ``operations_historisees'' et y programmer une nouvelle version de la classe `Compte` comme suit. Au lieu de cumuler les montants crédités et débités dans les variables `credit/debit`, l'historique de ces montants est mémorisé. Les variables d'instance `credit/debit` ne sont plus nécessaires et sont remplacées par deux tableaux de `double` de taille `MAX_OPERATIONS` (constante à définir dans la classe `Compte`) `credits` et `debits`, munis de leur indice respectif `dernierCredit` et `dernierDebit`, sur le dernier montant crédité/débité. Les opérations deviennent :

- `crediter(double x)` : range x en fin du tableau `credits`, quand `MAX_OPERATIONS` est atteint, le tableau est réinitialisé avec le cumul des crédits dans `credits[0]`
- `debiter(double x)` : range x en fin du tableau `debits`, quand `MAX_OPERATIONS` est atteint, le tableau est réinitialisé avec le cumul des débits dans `debits[0]`
- `solde()` = $\sum credits[i] - \sum debits[j]$
- `toString()` affiche les historiques `credits` et `debits` et le solde.

4 Modularité

Le protocole de la classe `Compte` n'a pas changé, seule son implantation interne a été modifiée. Les autres classes `CompteEpargne` et `TestComptes` n'ont donc pas à être recompilées. Vérifier cela en copiant simplement leur `.class` (et non leur source) du répertoire ``comptes'' dans ``operations_historisees'' et ré-exécuter directement `TestComptes`.