Compte rendu des exercices sur la programmation orientée objet en Java

Par: RAHIOUI Youssef

Exercice 1 : Gestion des entiers naturels avec exceptions personnalisées

Dans cet exercice, nous avons développé une application Java permettant de gérer des entiers naturels (positifs ou nuls) avec une gestion appropriée des exceptions.

Classe NombreNegatifException

Cette classe personnalisée hérite de `Exception` et permet de signaler une tentative d'utilisation d'un nombre négatif:

- Elle stocke la valeur négative qui a causé l'erreur via l'attribut `valeurNegative`
- Elle fournit un accesseur `getValeurNegative()` pour récupérer cette valeur
- Elle transmet le message d'erreur à la classe parente via `super(message)`

Classe EntierNaturel

Cette classe encapsule un entier naturel avec les fonctionnalités suivantes:

- Un attribut privé `val` qui stocke la valeur de l'entier naturel
- Un constructeur qui vérifie si la valeur initiale est valide et lève une exception si nécessaire
- Un accesseur en lecture `getVal()` qui retourne la valeur actuelle
- Un accesseur en écriture `setVal()` qui vérifie la validité de la nouvelle valeur
- Une méthode `decrementer()` qui décrémente la valeur de 1 après vérification

Classe principale (Main)

La classe de test démontre l'utilisation de `EntierNaturel` avec plusieurs scénarios:

- Création d'un entier avec une valeur correcte (5)
- Tentative de création avec une valeur négative (-3)
- Modification de la valeur (10)
- Tentative de modification avec une valeur négative (-7)
- Décrémentation jusqu'à 0

- Tentative de décrémentation en-dessous de 0

Les exceptions sont correctement capturées et le programme affiche les messages d'erreur appropriés ainsi que les valeurs négatives qui ont causé les exceptions.

Exercice 2 : Gestion de comptes bancaires

Dans cet exercice, nous avons développé un système de gestion de comptes bancaires avec héritage et exceptions personnalisées.

Classes d'exceptions personnalisées

- `FondsInsuffisantsException` : Levée lors d'une tentative de retrait ou transfert avec solde insuffisant
  - Stocke le montant demandé et le solde disponible
  - Fournit des accesseurs pour ces informations
- `CompteInexistantException` : Levée lors d'une tentative de transfert vers un compte inexistant
  - Stocke le numéro du compte inexistant
  - Fournit un accesseur pour cette information

Classe CompteBancaire

Classe de base qui encapsule les fonctionnalités communes à tous les types de comptes:

- Attributs protégés: numéro de compte, solde, nom du titulaire
- Constructeur initialisant ces attributs
- Accesseurs pour consulter les attributs
- Méthodes pour effectuer des opérations:
  - `depot()` : Ajoute un montant au solde
  - `retrait()` : Soustrait un montant du solde avec vérification
  - `afficherSolde()` : Affiche les informations du compte
  - `transferer()` : Transfère un montant vers un autre compte

Classes dérivées

- `CompteCourant` : Étend `CompteBancaire` en ajoutant un découvert autorisé
  - Redéfinit la méthode `retrait()` pour prendre en compte le découvert
- `CompteEpargne` : Étend `CompteBancaire` en ajoutant un taux d'intérêt
  - Ajoute une méthode `calculerInterets()` pour générer des intérêts

Classe principale (Main)

La classe de test utilise une `ArrayList<CompteBancaire>` pour gérer une collection de comptes:

- Création et ajout de différents types de comptes (courants et épargne)
- Affichage de la liste des comptes créés
- Opérations diverses: dépôts, retraits, transferts, calcul d'intérêts
- Tests des situations d'erreur: retrait avec fonds insuffisants, transfert vers un compte inexistant
- Suppression et recherche de comptes

La méthode utilitaire `rechercherCompte()` permet de trouver un compte par son numéro dans la liste.

Conclusion

Ces exercices ont permis de mettre en pratique plusieurs concepts fondamentaux de la programmation orientée objet en Java:

1. **Encapsulation**: Les attributs sont déclarés privés ou protégés, avec des accesseurs contrôlant l'accès.
2. **Héritage**: Création de hiérarchies de classes (`CompteCourant` et `CompteEpargne` héritant de `CompteBancaire` , `NombreNegatifException` héritant de `Exception`).
3. **Polymorphisme**: Redéfinition de méthodes dans les classes dérivées (comme `retrait()` dans `CompteCourant` et `toString()` dans les deux classes dérivées).
4. **Gestion des exceptions**: Création et utilisation d'exceptions personnalisées pour signaler des erreurs spécifiques.
5. **Collections**: Utilisation d' `ArrayList` pour gérer un ensemble dynamique d'objets.