L3 2025-2026

Programmation orientée objet - Java - Examen Novembre 2025. Durée 2 heures.

Pour cet examen, votre fond d'écran doit être vert clair. Vous devez absolument mettre tous les fichiers que vous souhaitez rendre dans le répertoire `EXAM`, qui est déjà présent sur votre compte à l'ouverture de la session. Tout ce qui ne se trouve pas dans ce dossier sera perdu lorsque vous vous déconnecterez. Ne créez pas vous-même de répertoire `EXAM`. Mettez tous vos fichiers dans celui qui existe.

Sous Eclipse, votre workspace doit être dans `EXAM`. Sinon configurez le workspace d'Eclipse (File > Switch WorkSpace) pour qu'il corresponde à un répertoire dans `EXAM`). Attention, les noms des classes, des champs et des méthodes que l'on vous demande doivent impérativement être respectés. De plus, le code doit être indenté correctement.

Commencez par vérifier la configuration d'Eclipse.

- Vérifiez que le JRE est /usr/local/apps/java25.
- Vérifez que le compilateur est java 25
- Créez ensuite votre Java project qui porte votre nom : `NOM_PRENOM`.

La javadoc 25 et les slides du cours (seuls documents autorisés) sont accessibles ici :
`http://igm.univ-mlv.fr/~juge/javadoc-25/`
`https://igm.univ-mlv.fr/~beal/JavaL3.html`

Les tests des exemples donnés dans chaque question doivent être faits (les mettre en commentaire s'ils ne marchent pas). Ces tests sont pris en compte dans la notation.

Toutes les classes seront écrites dans un package `fr.uge.bank`. On écrira une méthode `main` dans une classe `Main` dans le package pour faire les tests.

On souhaite modéliser une banque avec ses comptes clients, le solde de chaque compte (la somme d'argent positive ou négative pour chaque compte) et des ordres d'écriture qui créditent ou débitent un compte. Dans notre design, le solde d'un compte est associé à un compte dans une structure de données annexe et n'est pas stocké dans le compte.

On souhaite pouvoir créer des comptes de deux types, les comptes chèques (`CheckingAccount`) et les comptes épargne (`SavingsAccount`) et les ajouter à une banque.

Les deux types de comptes ont un numéro de compte (`accountId`) qui est un entier `long` positif ou nul, un nom de client (`name`), une date de naissance du client (`dateOfBirth`) entre 0 et 3000.

Attention, le solde du compte n'est pas stocké dans le compte.

Une `Bank` contiendra une liste `accounts` de comptes et les comptes seront stockés dans la liste par ordre d'insertion. On ne s'occupera pas pour l'instant de vérifier si un même compte figure plusieurs fois dans la liste.

Voici un exemple de code qui doit fonctionner.

```
var bank = new Bank();
var account1 = new CheckingAccount(1L, "Joe First", 1999);
var account2 = new CheckingAccount(3L, "Jane Second", 2001);
var account3 = new SavingsAccount(12L, "James Bond", 1960);
var account4 = new CheckingAccount(7L, "James Bond", 1960);
bank.addAccount(account1);
bank.addAccount(account2);
bank.addAccount(account3);
bank.addAccount(account4);
```

1. Écrire les codes de `CheckingAccount` et `SavingsAccount` en pensant à vérifier les préconditions.
2. Écrire le code de la classe `Bank`.

3. On souhaite maintenant afficher tous les comptes d'une banque avec le code suivant :

```
IO.println("test 1");
IO.println(bank);
```

qui donne en sortie

```
test 1
Checking account 1: Joe First (1999)
Checking account 3: Jane Second (2001)
Savings account 12: James Bond (1960)
Checking account 7: James Bond (1960)
```

Le format d'affichage pour un compte est le type du compte, le numéro du compte suivi d'un « : », puis le nom du client puis la date de naissance entre parenthèses. Chaque compte doit être affiché sur sa propre ligne, il n'y a pas de ligne vide à la fin.

4. On souhaite qu'il ne soit pas possible de créer d'autres types de comptes que `CheckingAccount` et `SavingsAccount`. Modifier le code pour cela.

5. Ajouter une méthode `checkingAccounts` renvoyant une liste non modifiable des comptes chèques de la banque. Le type de retour pourra être `List<Account>`, où `Account` est un type commun pour tous les types de comptes, ou bien `List<CheckingAccount>`.

   Écrire `checkingAccounts`.

```
IO.println("test 2");
var checkingAccounts = bank.checkingAccounts();
IO.println(checkingAccounts);
```

donne

```
test 2
[Checking account 1: Joe First (1999),
 Checking account 3: Jane Second (2001)
 Checking account 7: James Bond (1960)]
```

Sur l'exemple ci-dessus, la sortie est donnée sur plusieurs lignes uniquement pour la présentation du sujet.

6. Écrire une méthode statique `sameAccounts(Bank bank1, Bank bank2)` qui renvoie `true` si les banques `bank1` et `bank2` contiennent les mêmes comptes sans tenir compte ni des répétitions éventuelles, ni de leur ordre dans les listes. La vérification doit être en temps linéaire.

   Par exemple

```
IO.println("test 3");
var bank2 = new Bank();
bank2.addAccount(account4);
bank2.addAccount(account2);
bank2.addAccount(account2);
bank2.addAccount(account1);
bank2.addAccount(account3);
bank2.addAccount(account4);
IO.println(Bank.sameAccounts(bank, bank2));
var bank3 = new Bank();
bank3.addAccount(account4);
bank3.addAccount(account2);
bank3.addAccount(account1);
bank3.addAccount(account3);
IO.println(Bank.sameAccounts(bank, bank3));
```

```
var bank4 = new Bank();
bank4.addAccount(account2);
bank4.addAccount(account3);
bank4.addAccount(account1);
IO.println(Bank.sameAccounts(bank, bank4));
```

donne

```
test 3
true
true
false
```

7. Aouter une méthode `accountsByName` renvoyant une map avec les clés non modifiables associant à chaque nom de propriétaire la liste de ses comptes.

```
IO.println("test 4");
var map = bank.accountsByName();
IO.println(map);
```

donne

```
{Joe First=[Checking account 1: Joe First (1999)],
 James Bond=[Savings account 12: James Bond (1960),
             Checking account 7: James Bond (1960)],
 Jane Second=[Checking account 3: Jane Second (2001)]}
```

La sortie est donnée ici sur plusieurs lignes uniquement pour la présentation du sujet.

8. Mettre en commentaire la méthode `sameAccounts(Bank bank1, Bank bank2)` et le test 3 ci-dessus dans le `main`.

9. On souhaite maintenant ajouter un solde (balance en anglais) à un compte client. Pour cela, nous allons utiliser une `HashMap<Long, Long>` qui associe, **à chaque numéro de compte**, le solde (un entier positif, négatif ou nul) de ce compte. Cela va aussi permettre de lever une exception si on essaie d'ajouter un compte déjà présent dans la banque.

   Mettez en commentaire votre ancien code de `addAccount` et écrivez le nouveau code de `addAccount` qui vérifie en plus que l'on n'ajoute pas deux comptes ayant le même numéro et qui associe au compte que l'on crée un solde égal à zéro.

10. Écrire une méthode `credit(long amount, Account account)` qui augmente le solde d'un compte client d'un certain montant positif ou nul. Une exception sera levée si le compte n'est pas présent dans la banque ou si le montant est négatif ou nul.

11. On souhaite ajouter une méthode pour retirer de l'argent d'un compte. Un retrait est possible si le compte est présent dans la banque, si la somme à retirer est un entier positif et si le solde du compte après le retrait est suffisant. Pour un compte épargne aucun découvert n'est autorisé et pour un compte chèque un découvert de 100 est autorisé. Écrire une méthode privée `checkWithdraw(long amount, Account account)` qui renvoie `true` si le retrait est possible.

12. Écrire une méthode `withdraw(long amount, Account account)` qui diminue le solde d'un compte client d'un certain montant positif ou nul. Une exception sera levée si le retrait n'est pas possible.

13. Écrire une méthode `transfer(long amount, Account from, Account to)` qui permet de faire un virement du compte `from` vers le compte `to` d'un certain montant positif ou nul. Une exception sera levée si le virement n'est pas possible.

14. Modifier l'affichage d'une banque pour afficher en plus le solde des comptes comme ci-dessous. (Mettez en commentaire votre ancien code de `toString`.)

    Tester le code suivant :

    ```
    IO.println("test 5");
    bank.credit(100L, account3);
    bank.credit(1_000L, account4);
    bank.withdraw(500L, account4);
    bank.transfer(200L, account4, account1);
    IO.println(bank);
    ```

    qui doit donner

    ```
    test 5
    Checking account 1: Joe First (1999)200
    Checking account 3: Jane Second (2001)0
    Savings account 12: James Bond (1960)100
    Checking account 7: James Bond (1960)300
    ```

15. Écrire une méthode `get(long accountId)` qui renvoie le solde d'un compte en temps presque constant à partir de son numéro. Si le compte n'est pas dans la banque, une exception sera levée.

    Tester le code suivant :

    ```
    IO.println("test 6");
    IO.println(bank.get(7L));
    ```

    qui doit donner

    ```
    test 6
    300
    ```

16. Écrire une méthode `balanceByName` renvoyant une map non modifiable associant à chaque nom de propriétaire le montant total de tous les comptes de cette personne.

    Tester le code suivant :

    ```
    IO.println("test 7");
    IO.println(bank.balanceByName());
    ```

    qui doit donner

    ```
    test 7
    {James Bond=400, Jane Second=0, Joe First=200}
    ```

17. Pour finir, ajouter une méthode `wealthiestClient()` renvoyant le nom du client le plus riche de la banque (ou l'un des plus riches s'il y en a plusieurs). La méthode lèvera une exception si la banque est vide (ne possède aucun compte). Tester le code suivant :

    ```
    IO.println("test 8");
    IO.println(bank.wealthiestClient());
    ```

    qui doit donner

    ```
    test 8
    James Bond
    ```