Faculté des Sciences
Aix*Marseille Université

Année universitaire 2021-2022

Site : ☒ Luminy ☐ St-Charles ☐ St-Jérôme ☐ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☒ 1er semestre ☐ 2ème semestre ☐ Session 2 Durée de l'épreuve : 2h
Examen de : L2 Nom du diplôme : Licence d'informatique
Code du module : SIN3U02 Libellé du module : Programmation 2
Calculatrices autorisées : NON Documents autorisés : OUI

---

1 Gestion de comptes en banque

On s'intéresse à un système qui gère des comptes bancaires. Dans ce système, les transferts d'argent entre comptes sont représentés par des objets de la classe `Transfer`. La banque propose des comptes chèques définis par la classe `CheckingAccount`. Cette classe contient des attributs et les accesseurs et mutateurs (*getters* et *setters*) décrit dans le code en annexe (page 3 du sujet).
Cette classe contient aussi une méthode `makeTransfer`. Cette méthode crée un objet `Transfer` (représentant un transfert d'argent entre deux comptes) à partir d'un numéro de compte vers lequel transférer de l'argent (de type `String`) et un montant d'argent (de type `Money`). Elle vérifie que les propriétés suivantes soient satisfaites (levant une exception de type `BusinessException` le cas échéant avec un message spécifique) :

1. le montant à transférer ne dépasse pas une certaine limite
2. le numéro de compte est composé d'exactement neufs chiffres (pas d'autres caractères)
3. le numéro de compte respecte la somme de contrôle 11-test, c'est-à-dire que pour un compte $c_0c_1c_2c_3c_4c_5c_6c_7c_8$ (c_i représente le i-ème chiffre du compte), la somme $\sum_{i=0}^{8}(9-i)c_i$ doit être congru à 0 modulo 11.

La classe `CheckingAccount` utilise aussi `Accounts` qui est une classe donnant accès aux :

— comptes via une méthode `public static CheckingAccount findAccountByNumber(String accountNumber)`
— utilisateurs via une méthode `public static boolean containsUser(String name)`.

Le code de la classe `CheckingAccount` est disponible en annexe à la page 3 du sujet.

— **Question 1 (1 point) :** *Donnez le code de la classe `BusinessException` qui étend la classe `Exception`.*
— **Question 2 (3 points) :** *Quels sont le ou les problème(s) avec le code de la classe `CheckingAccount` ?* On ne vous demande pas de les résoudre pour le moment car c'est en partie l'objet de la dernière question de l'examen.

Supposons maintenant que la banque introduise un nouveau type de compte, appelé compte d'épargne dont les les objets correspondants sont définis par une classe `SavingsAccount` dont le code est donné en annexe (page 4 du sujet). Un compte d'épargne n'a pas de limite de transfert, mais il a une restriction : l'argent ne peut être transféré que vers un compte chèque (fixe) particulier. L'idée est que le titulaire du compte choisit de coupler un compte chèque particulier avec un compte d'épargne.

— **Question 3 (1 point) :** *Quel est l'effet du mot-clé `final` dans la première ligne de la classe (`public final class SavingsAccount`) ?*
— **Question 4 (1 point) :** *Quel est l'effet du mot-clé `final` dans la deuxième ligne de la classe (`private final CheckingAccount registeredCounterAccount;`) ?*
— **Question 5 (2 points) :** *Définissez une interface `Account` qui sera implémenté par `SavingsAccount` et `CheckingAccount` et qui contient le plus de méthodes possibles sans changer ces deux classes.*

On considérera par la suite qu'une instance de la classe `Transfer` a trois attributs : `Account senderAccount`, `Account receiverAccount` et `Money amount` qui lui permettent de représenter un transfert de `senderAccount` à `receiverAccount` d'un montant `amount`. Elle contient aussi un constructeur permettant d'instancier un `Transfer` avec des valeurs pour ces trois attributs ainsi que des accesseurs.

— **Question 6 (1 point) :** *Il n'est pas possible pour le moment de faire un transfert vers un `SavingsAccount`. Que faudrait-il changer dans le code pour que cela soit possible ?*

On souhaiterait rajouter une méthode statique `addAll` dans la classe `Accounts` qui permettrait de rajouter une liste de comptes. On souhaite pouvoir ajouter des listes de comptes chèque et de comptes d'épargne en ne définissant qu'une méthode `addAll`. Le code ci-dessous vous donne un exemple d'utilisation de la méthode `addAll`.

```
public class App {
  public static void main(String[] args) throws BusinessException{
    List<CheckingAccount> checkingAccounts = new ArrayList<>();
    List<SavingsAccount> savingsAccounts = new ArrayList<>();
    CheckingAccount checkingAccount = new CheckingAccount("Arnaud");
    checkingAccounts.add(checkingAccount);
    SavingsAccount savingsAccount = new SavingsAccount(checkingAccount, "Arnaud");
    savingsAccounts.add(savingsAccount);
    Accounts.addAll(savingsAccounts);
    Accounts.addAll(checkingAccounts);
  }
}
```

— **Question 7 (1 point) :** *Donner la signature (pas le code) de la méthode* ***`addAll`***.

La classe `Money` a les éléments suivants :

— un constructeur `public Money(int amount)`
— un attribut `private int amount`
— un accesseur `public int getAmount()`
— une méthode `public boolean greaterThan(int amount)` qui retourne `true` si l'attribut `amount` est strictement supérieur à l'argument `amount` et `false` sinon.
— une méthode `public void add(int amount)` qui ajoute l'argument `amount` à l'attribut `amount` de l'objet avec lequel `add` est appelé.

Il ne vous est pas demandé de donner le code de cette classe.

— **Question 8 (2 points) :** *Donnez le code des méthodes de test qui permettent de vérifier le bon comportement des méthodes* ***`greaterThan`*** *et* ***`add`*** *de la classe* ***`Money`***. Vous devez fournir uniquement le code de ces méthodes de test, le code du reste de la classe de test (l'entête avec les imports) n'est pas demandé. Un rappel des méthodes de test est donné en annexe (page 4 du sujet).

La banque souhaiterait pouvoir dans le futur modifier la méthode de validation des comptes en changeant le nombre de chiffres des comptes et l'entier utilisé par le modulo. Pour cela, on souhaite construire des objets `AccountNumberValidator` avec un entier `n` correspondant aux nombres de chiffre et un entier `m` correspondant à l'entier utilisé par le modulo. On souhaiterait que l'instruction `new AccountNumberValidator(n,m).requiresValidAccount(accountNumber)` vérifie que les propriétés suivantes soient satisfaites (levant une exception de type `BusinessException` le cas échéant avec un message spécifique) :

1. le numéro de compte (`accountNumber`de type `String`) est composé d'exactement n chiffres (pas d'autres caractères)
2. le numéro de compte respecte la somme de contrôle m-test, c'est-à-dire que pour un compte $c_0c_1 \dots c_{n-1}$ (c_i représente le i-ème chiffre du compte), la somme $\sum_{i=0}^{n-1}(n-i)c_i$ doit être congru à 0 modulo m.

L'instruction `new AccountNumberValidator(9,11).requiresValidAccountNumber(accountNumber);` doit pouvoir remplacer le code des étape `2.` et `3.` de la méthode `makeTransfer` de `CheckingAccount`.

— **Question 9 (3 points) :** *Donnez le code de la classe* ***`AccountNumberValidator`***.

Vous pouvez remarquer que la classe `SavingsAccount` a beaucoup de points communs avec la classe `CheckingAccount`. Afin d'éviter la duplication de code, il est possible de créer une classe abstraite `AbstractAccount` qui contiendra le code en commun des deux classes et qui sera étendue par les deux classes `SavingsAccount` et `CheckingAccount`.

— **Question 10 (5 points) :** *Écrivez le code des classes* ***`AbstractAccount`***, ***`SavingsAccount`*** *et* ***`CheckingAccount`*** *de façon à éviter la duplication de code et prenant en compte vos réponses aux questions précédentes (c'est-à-dire corrigeant les problèmes soulevés dans vos réponses).*

2 Annexe

2.1 Classe CheckingAccount

```
public class CheckingAccount {
  public int transferLimit = 100;
  public String name;

  public void setTransferLimit(int transferLimit)
          throws BusinessException {
    if (transferLimit < 0) {
      throw new BusinessException("Negative transfer limit not permitted");
    }
    this.transferLimit = transferLimit;
  }

  public CheckingAccount(String name) {
    setName(name);
  }

  public String getName() {
    return name;
  }

  public void setName(String name) throws BusinessException {
    if (!Accounts.containsUser(name))
      throw new BusinessException("Unregistered user");
    this.name = name;
  }

  public Transfer makeTransfer(String accountNumber, Money amount)
          throws BusinessException {
    // 1. Check withdrawal limit:
    if (amount.greaterThan(this.transferLimit)) {
      throw new BusinessException("Transfer limit exceeded");
    }
    // 2. Check if the account to transfer is a nine digits string
    if(accountNumber.length()!=9 || !accountNumber.matches("[0-9]*")) {
      throw new BusinessException("Invalid account number: not a 9 digits number");
    }
    // 3. Check if bank account number validates 11-test:
    int sum = 0;
    for (int i = 0; i < accountNumber.length(); i++) {
      sum = sum + (9 - i) * Character.getNumericValue(
              accountNumber.charAt(i));
    }
    if (sum % 11 != 0) {
      throw new BusinessException("Invalid account number: not validating 11-sum");
    }
    // 4. Look up counter account:
    CheckingAccount account = Accounts.findAccountByNumber(accountNumber);
    // 5. Make transfer object:
    return new Transfer(this, account, amount);
  }
}
```

2.2 Classe SavingsAccount

```
public final class SavingsAccount {
  private final CheckingAccount registeredCounterAccount;
  private String name;

  public SavingsAccount(CheckingAccount registeredCounterAccount, String name)
          throws BusinessException{
    this.registeredCounterAccount = registeredCounterAccount;
    setName(name);
  }

  public String getName() {
    return name;
  }

  public void setName(String name) throws BusinessException {
    if (!Accounts.containsUser(name))
      throw new BusinessException("Unregistered user");
    this.name = name;
  }

  public Transfer makeTransfer(String accountNumber, Money amount)
          throws BusinessException {
    // 1. Check if the account to transfer is a nine digits string
    if(accountNumber.length()!=9 || !accountNumber.matches("[0-9]*")){
      throw new BusinessException("Invalid account number: not a 9 digits number");
    }
    // 2. Check if bank account number validates 11-test:
    int sum = 0;
    for (int i = 0; i < accountNumber.length(); i++) {
      sum = sum + (9 - i) * Character.getNumericValue(
              accountNumber.charAt(i));
    }
    if (sum % 11 != 0) {
      throw new BusinessException("Invalid account number: not validating 11-sum");
    }
    // 3. Look up counter account:
    CheckingAccount account = Accounts.findAccountByNumber(accountNumber);
    // 4. Make transfer object:
    Transfer result = new Transfer(this, account, amount);
    // 5. Check whether withdrawal is to registered counter account:
    if (!result.getReceiverAccount().equals(this.registeredCounterAccount)) {
      throw new BusinessException("Counter-account not registered");
    }
    return result;
  }
}
```

2.3 Rappel des méthodes de test

- `assertThat(actual).isCloseTo(expected, within(delta))` : vérifie que $|expected - actual| \leq delta$.
- `assertThat(object).isNotNull()` : vérifie que la référence **n'**est **pas** `null`
- `assertThat(actual).isSameAs(expected)` : vérifie que les deux objets sont les mêmes (même référence).
- `assertThat(condition).isTrue()` : vérifie que `condition` est vraie.
- `assertThat(condition).isFalse()` : vérifie que `condition` est faux.
- `assertThat(actual).isEqualTo(expected)` : vérifie que `expected` est égal à `actual`.