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 Correction

— **Question 1 :** *Donnez le code de la classe* `BusinessException` *qui étend la classe* `Exception`.

```
public class BusinessException extends Exception{
  public BusinessException(String message) {
    super(message);
  }
}
```

— **Question 2 :** *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.

Le code a les problèmes suivants :

— Un problème à la compilation lié à la signature du constructeur `CheckingAccount`. Un constructeur doit préciser dans sa signature toutes les exceptions qu'il peut lever et qu'il n'a pas traitées avec un bloc `try`/`catch`. Puisque le constructeur `CheckingAccount` appelle la méthode `setName` pouvant lever une `BusinessException`, il devrait avoir dans sa signature `throws BusinessException` puisqu'il ne traite pas cette exception dans un bloc `try`/`catch`.

— Un problème lié aux attributs `name` et `transferLimit`. Ces deux attributs sont publiques et il est donc possible de les modifier sans passer par les mutateurs (*setters*). Il faut modifications directes pour éviter que les attributs prennent des valeurs non-voulues (comme une limite de transfert négative).

— Le code de la méthode `makeTransfer` est trop long et nécessite des commentaires. Il faudrait donc décomposer la méthode et extraire les parties du code dans des méthodes ayant le même niveau d'abstraction.

— **Question 3 :** *Quel est l'effet du mot-clé* `final` *dans la première ligne de la classe (*`public final class SavingsAccount`*) ?*

Le mot-clé `final` dans la déclaration d'une classe interdit l'extension de la classe

— **Question 4 :** *Quel est l'effet du mot-clé* `final` *dans la deuxième ligne de la classe (*`private final CheckingAccount registeredCounterAccount;`*) ?*

Le mot-clé `final` dans la déclaration d'un attribut interdit la modification de la valeur de l'attribut après la construction de l'objet.

— **Question 5 :** *Définissez une interface* `Account` *qui sera implémentée par* `SavingsAccount` *et* `CheckingAccount` *et qui contient le plus de méthodes possibles sans changer des deux classes.*

```
public interface Account {
  void setName(String name) throws BusinessException;
  String getName();
  Transfer makeTransfer(String counterAccountNumber, Money amount)
          throws BusinessException;
}
```

— **Question 6 :** *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 ?*

Il faudrait changer le type de retour de la méthode `findAccountByNumber` de `Accounts` afin qu'elle retourne un `Account` et changer le type de la variable `account` en `Account` dans les méthodes `makeTransfer` de `CheckingAccount` et `SavingsAccount`.

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

La signature de `addAll` pourrait être la suivante :
`public static void addAll(List<? extends Accounts> accounts)`

— **Question 8 :** *Donnez le code des méthodes de test qui permettent de vérifier le bon comportement des méthodes* ***`greaterThan`*** *et* ***`add`*** *de la classe* ***`Money`***.

```
import correction.Money;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.*;

public class MoneyTest {
  @Test
  void testAdd(){
    Money money = new Money(100);
    money.add(100);
    assertThat(money.getAmount()).isEqualTo(200);
    money.add(0);
    assertThat(money.getAmount()).isEqualTo(200);
  }
  @Test
  void testGreaterThan(){
    Money money = new Money(100);
    assertThat(money.isGreaterThan(10)).isTrue();
    assertThat(money.isGreaterThan(200)).isFalse();
    assertThat(money.isGreaterThan(100)).isFalse();
  }
}
```

— **Question 9 :** *Donnez le code de la classe* ***`AccountNumberValidator`***.

```
public class AccountNumberValidator {
  private final int numberOfDigits;
  private final int modulo;

  public AccountNumberValidator(int numberOfDigits, int modulo) {
    this.numberOfDigits = numberOfDigits;
    this.modulo = modulo;
  }

  void requiresValidAccountNumber(String accountNumber)
          throws BusinessException {
    requiresNumberOfDigitsNumber(accountNumber);
    requiresValidatingModuloTest(accountNumber);
  }

  private void requiresValidatingModuloTest(String accountNumber)
          throws BusinessException {
    int sum = computeSum(accountNumber);
    if(sum % modulo != 0 ){
      throw new BusinessException("Invalid account number: not validating "
              + modulo +"-sum");
    }
```

```
  }

  private int computeSum(String accountNumber) {
    int sum = 0;
    for (int i = 0; i < accountNumber.length(); i++) {
      sum = sum + (9 - i) * Character.getNumericValue(accountNumber.charAt(i));
    }
    return sum;
  }

  private void requiresNumberOfDigitsNumber(String accountNumber)
          throws BusinessException {
    if (accountNumber.length() != numberOfDigits
            || !accountNumber.matches("[0-9]*")) {
      throw new BusinessException("Invalid account number: not a "
              + numberOfDigits + " digits number");
    }
  }
}
```

— **Question 10 :** *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).*

```
public abstract class AbstractAccount implements Account {
  protected String name;

  public AbstractAccount(String name) throws BusinessException{
    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 counterAccountNumber, Money amount)
          throws BusinessException {
    new AccountNumberValidator(9,11)
            .requiresValidAccountNumber(counterAccountNumber);
    CheckingAccount account = Accounts.findAccountByNumber(counterAccountNumber);
    return new Transfer(this, account, amount);
  }
}

public class CheckingAccount extends AbstractAccount{
  private int transferLimit = 100;

  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) throws BusinessException {
    super(name);
  }

  public Transfer makeTransfer(String accountNumber, Money amount)
          throws BusinessException {
    requiresTransferLimitGreaterThan(amount);
    return super.makeTransfer(accountNumber, amount);
  }

  private void requiresTransferLimitGreaterThan(Money amount) throws BusinessException {
    if (amount.isGreaterThan(this.transferLimit)) {
      throw new BusinessException("Transfer limit exceeded");
    }
  }
}

public final class SavingsAccount extends AbstractAccount {
  private final CheckingAccount registeredCounterAccount;


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


  public Transfer makeTransfer(String accountNumber, Money amount)
          throws BusinessException {
    Transfer result = super.makeTransfer(accountNumber, amount);
    requiresRegisteredCounterAccountEqualsTo(result.getReceiverAccount());
    return result;
  }

  private void requiresRegisteredCounterAccountEqualsTo(Account account)
          throws BusinessException {
    if (!account.equals(this.registeredCounterAccount)) {
      throw new BusinessException("Counter-account not registered");
    }
  }
}
```