Aix*Marseille université
Initiative d'excellence
Année 2018-19

Domaine Sciences et Technologies
Licence informatique

Programmation 2 : Corrigé du TD 3
Code UE : SIN3U02

Bonnes pratiques de programmation

1 BankAccount

1. L'attribut `balance` correspond au crédit du compte. C'est un attribut d'instance car chaque compte a son propre crédit. On utilise le mot-clé `private` car on ne souhaite pas que l'utilisateur de la classe puisse augmenter ou diminuer le crédit d'un compte sans passer par les méthodes `deposit` et `withdraw`. En effet, La valeur de crédit est quelque chose que l'on souhaite contrôler et ne doit pas être accessible directement par l'utilisateur de la classe.
2. Le mot-clé `this` sert à indiquer qu'on manipule des attributs d'instance. Il n'est pas indispensable car il n'y a pas de variables ou arguments dans les méthodes ayant le même nom que les attributs manipulés. Il n'y a donc pas d'ambiguïté dans le code et le mot-clé `this` peut être enlevé sans que cela change le comportement de la classe.
3. Le mot-clé `static` sert à indiquer que `lastAccountNumber` est un attribut de classe et non d'instance. Le dernier numéro du compte créé est une notion qui ne dépend pas de l'instance.
4. Les rôles des méthodes sont les suivants :
   — `deposit` : sert à déposer un montant d'argent égal à `depositAmount` en augmentant le crédit du compte d'autant.
   — `withdraw` : sert à retirer un montant d'argent égal à `withdrawAmount` en crédit le crédit du compte d'autant.
   — `getNumber` : permet d'accéder au numéro du compte.
   — `getBalance` : permet d'accéder au crédit du compte.
5. Dans le code écrit par le programmeur, le compte est débiter d'un montant négatif ce qui revient à augmenter son crédit et donc est de facto un dépôt. Ce n'est pas une utilisation normale de la classe. Il y a plusieurs façons de régler le problème :
   — Rajouter une assertion en changeant la méthode `deposit` en :
   ```
   public void deposit(double depositAmount)
   {
       assert depositAmmount => 0;
       this.balance += depositAmount;
   }
   ```
   — Rajouter un test ne faisant le dépôt que s'il est positif. La nouvelle méthode `deposit` renvoie un booléen afin d'indiquer le succès ou non du dépôt :
   ```
   public boolean deposit(double depositAmount)
   {
       if (depositAmmount < 0)
           return false;
       this.balance += depositAmount;
       return true;
   }
   ```
   — Rajouter une exception qu'on verra dans un chapitre ultérieur du cours.
6. Il suffit de rajouter la méthode suivante dans le code de la classe :

```
public String toString(){
    return "Account number : " + this.number
```

```
                + " Balance : " + this.balance;
    }
```

7. Il suffit de changer le code de la méthode `withdraw` en :

```
public boolean withdraw(double withdrawAmount)
{
    if (this.balance - withdrawAmount < 0)
        return false;
    this.balance -= withdrawAmount;
    return true;
}
```

8. Il suffit de rajouter un attribut à la classe. Cet attribut peut être une chaîne de caractère `String ownerName` représentant le nom de la personne ou bien directement un objet `Person owner` de type `Person` définie par exemple par la classe suivante :

```
public class Person {
    String firstName;
    String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String toString() {
        return firstName + " " + lastName;
    }
}
```

2 Application de la classe `Point`

```
public class PolygonalChain {
    private final Point[] vertices;

    public PolygonalChain(int size){
        this.vertices = new Point[size];
    }

    public PolygonalChain(Point[] vertices) {
        this.vertices = vertices.clone();
    }

    Point getVertex(int index){
        return this.vertices[index];
    }

    void setVertex(int index, Point p){
        this.vertices[index] = p;
    }
```

```
    boolean isClosed(){
        return vertices[0].
                equals(vertices[vertices.length-1]);
    }

    void translate(int dx, int dy){
        for(int index = 0; index<vertices.length; index++){
            this.setVertex(index,
                getVertex(index).translate(dx,dy));
        }
    }

    double length(){
        double sumDistances = 0;
        for(int index = 0; index<vertices.length-1; index++){
            sumDistances += vertices[index]
                        .distanceTo(vertices[index+1]);
        }
        return sumDistances;
    }

    public String toString(){
        StringBuilder stringBuilder = new StringBuilder();

        stringBuilder.append("[");
        for(int index=0; index<vertices.length; index++){
            stringBuilder.append(vertices[index]);
            if (index != vertices.length-1) {
                stringBuilder.append(", ");
            }
        }
        stringBuilder.append("]");
        return stringBuilder.toString();
    }
}
```

3 Vecteur d'entiers

```
import java.util.Arrays;

/**
 * La classe <code>Vector</code> implémente un tableau d'entiers
 * de taille dynamique. Les éléments du vecteur sont stockés dans un tableau.
 * La taille de ce tableau est au minimum doublée à chaque fois qu'il est
 * nécessaire de le faire grossir.
 */
public class Vector {

    /**
```

```java
 * Tableau permettant de stocker les éléments du vecteur.
 * Seuls les <code>size</code> premiers éléments font partie du vecteur.
 * La taille de ce tableau est égale à la capacité du vecteur, c'est-à-dire,
 * au nombre d'éléments maximum que le vecteur peut contenir sans
 * avoir besoin d'allouer un nouveau tableau.
 */
private int[] elements;

/**
 * Nombre d'éléments présents dans le vecteur.
 */
private int size;

/**
 * Construit un vecteur de capacité initiale <code>initialCapacity</code>.
 *
 * @param initialCapacity Capacité initiale du vecteur
 */
public Vector(int initialCapacity) {
    this.elements = new int[initialCapacity];
    this.size = 0;
}
/**
 * Construit un vecteur de capacité initiale <code>10</code>.
 */
public Vector() {
    this(10);
}

/**
 * Augmente la capacité du vecteur si nécessaire de façon
 * à permettre le stockage d'au moins <code>minCapacity</code>
 * éléments. S'il est nécessaire d'augmenter la capacité du vecteur,
 * elle est au minimum doublée.
 *
 * @param minCapacity Capacité minimale à assurer
 */
public void ensureCapacity(int minCapacity) {
    int oldCapacity = elements.length;
    if (oldCapacity >= minCapacity) return;
    int newCapacity = Math.max(oldCapacity * 2, minCapacity);
    elements = Arrays.copyOf(elements, newCapacity);
}

public void resize(int newSize) {
    ensureCapacity(newSize);
    this.size = newSize;
    for (int index = size; index <capacity(); index++){
        elements[index] = 0;
    }
```

```
    }

    /**
     * Retourne la capacité du vecteur.
     *
     * @return Capacité du vecteur.
     */
    public int capacity() {
        return elements.length;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void add(int element) {
        resize(size()+1)
        set(size()-1, element);
    }

    public void set(int index, int element) {
        if (!indexIsInBounds(index))
            return;
        elements[index] = element;
    }

    public int get(int index) {
        if (!indexIsInBounds(index))
            return 0;
        return elements[index];
    }

    private boolean indexIsInBounds(int index){
        return index >= 0 && index < this.size;
    }

}
```