TP n°1

Rappels Java et Introduction à la POO (programmation orientée objet)

Remarque : pensez à créer au début de chaque TP un nouveau répertoire, par exemple `TP1` pour le premier, puis pendant tout le TP, ne travaillez que dans ce dossier.

Rappels de syntaxe

La définition d'une classe en java a la forme suivante : la définition des champs/propriétés, la définition des constructeurs puis celle des méthodes.

```
class NomClasse {
    acces type nom_champ_1
    acces type nom_champ_2
    ...
    public NomClasse(type nom_arg, ...){
    ...
    }
    public NomClasse(type nom_arg, ...){
    ...
    }
    ...
    acces type_retour nomMethode(type nom_arg, ...){
    ...
    }
    ...
}
```

L'accès est `public` ou `private` selon que la propriété ou la méthode sera visible ou non par les autres classes. Une classe peut avoir un ou plusieurs constructeurs. Par défaut, la classe ne contient que le constructeur sans argument (qui peut être redéfini dans la définition de la classe).

Une fois la classe Toto définie, on peut déclarer des références à des objets de type Toto grâce à l'instruction :
`Toto nom ;`

La création de l'objet proprement dit se fait à l'aide d'un constructeur avec l'instruction :
`nom = new Toto() ;`

Lors de la définition des constructeurs et des méthodes, on fait référence à l'instance courante à l'aide du mot clé `this`.

Exercice 1 *Passage par référence - passage par valeur*

```
$ cat MaClasse.java
```

```
public class MaClasse {
    private boolean mon_booleen;

    public MaClasse(boolean b) {
        this.mon_booleen = b;
    }

    public void invValue() {
        mon_booleen = !mon_booleen;
    }

    public boolean getMonBooleen() {
        return mon_booleen;
    }

    public String toString() {
        return "" + this.getMonBooleen();
    }
}

$ cat Tests.java
public class Tests {
    public static void swap(int i, String s, short[] t1, double[] t2, MaClasse c){
        i = -42;
        s = "ahah";
        t1[1]=-4;
        c.invValue();
        t2 = new double [3];
        t2[0] = 4.4; t2[1] = 5.5; t2[2] = 6.6;
    }

    public static void main(String[] args) {
        int entier = 123;
        String str = "456";
        short [] tab1 = {7, 8, 9};
        double [] tab2 = {1.1, 2.2, 3.3};
        MaClasse cl = new MaClasse(false);

        System.out.println(entier + " ; " + str + " ; " + tab1[0] + " " + tab1[1] +
                           " " + tab1[2] + " ; " + tab2[0] + " " + tab2[1] + " " +
                           tab2[2] + " ; " + cl);

        swap(entier, str, tab1, tab2, cl);

        System.out.println(entier + " ; " + str + " ; " + tab1[0] + " " + tab1[1] +
                           " " + tab1[2] + " ; " + tab2[0] + " " + tab2[1] + " " +
                           tab2[2] + " ; " + cl);
    }
```

```
}
```

1. À quoi sert la méthode `toString()` définie dans la classe `MaClasse` ?
2. Que fait ce programme ?

En général, en Java, les arguments sont passés par valeur. Pour les types simples, cela signifie que leurs valeurs sont copiées lors du passage des arguments à la méthode, ce qui explique le comportement de ... et de .... Pour les objets, seule la référence est copiée, ce qui entraîne la modification de ... et de ... tout en laissant inchangé .... On appelle, par abus de langage, ce dernier mode de passage de paramètres le passage par référence.

Exercice 2 *Équations bicarrées*

Nous traiterons dans cet exercice uniquement les équations bicarrées à solutions dans $\mathbb{R}$.

Un polynôme est dit bicarré lorsqu'il est de la forme $ax^4 + bx^2 + c$, $x \in \mathbb{R}$, $(a, b, c) \in \mathbb{R}^3$. En posant $z = x^2$, les solutions de l'équation du second degré $az^2 + bz + c = 0$ correspondent aux carrés des solutions de l'équation bicarrée $ax^4 + bx^2 + c = 0$.

1. Résoudre, sur papier, l'équation bicarrée $x^4 - 13x^2 + 36 = 0$.
2. Écrire une classe `Equation` avec les propriétés coefficients (`a`, `b`, `c`) et racines (`r1`, `r2`, `r3` et `r4`).
3. Écrire un constructeur qui permette d'initialiser les coefficients.
4. On veut ajouter le comportement `calculerRacines` aux instances de la classe `Equation`. Cette méthode doit calculer les racines de l'équation bicarrée et affecter leurs valeurs aux propriétés `r1`, `r2`, `r3` et `r4`.

   Notons tout de suite que l'état d'une instance de la classe `Equation` change lorsqu'on lui fait exécuter sa méthode `calculerRacines`.
   - Avant : les valeurs de `r1`, `r2`, `r3` et `r4` ne sont pas calculées donc elles ne sont pas valides.
   - Après : les valeurs de `r1`, `r2`, `r3` et `r4` sont calculées et sont donc valides.

   Pour décrire ces deux états d'une instance de la classe `Equation`, on ajoute la propriété d'instance `racinesCalculees` de type booléen. `racinesCalculees` est initialisée à faux lors de la création de l'instance, puis, lorsque cette dernière exécute sa méthode `calculerRacines`, `racinesCalculees` devient vraie.

   Modifier en conséquence le constructeur de la classe `Equation` et écrire la méthode d'instance `calculerRacines`.

5. Écrire une méthode `toString` qui convertit une instance en chaîne de caractères. Cette méthode doit avoir le comportement suivant :
   - Si les racines n'ont pas été calculées, `toString` renvoie par exemple la chaîne de caratères :
     `Equation 3*x^4 - 123*x^2 + 1200 = 0.`
   - Si les racines ont été calculées, `toString` renvoie par exemple la chaîne de caractères :
     `Equation 3*x^4 - 123*x^2 + 1200 = 0. Racines 4, -4, 5 et -5.`

***Remarque** : on appelle toujours* `toString` *la méthode qui convertit une instance en chaîne de caractères.*

6. Écrire une classe `Probleme` contenant la méthode `main`. Dans cette méthode, créer une instance `monEquation` de la classe `Equation` puis, en utilisant les méthodes de cette instance, afficher l'équation, calculer ses racines et afficher les.

7. Dans la méthode `main`, on insère l'instruction `monEquation.a=0;` après le calcul des racines de `monEquation`. Le résultat affiché est-il juste ?

   ***Qu'est-ce que l'encapsulation ?** Pour empêcher le type de problème que l'on vient de rencontrer, on rend systématiquement inaccessible les propriétés des instances d'une classe par les instances des autres classes. Plutôt que de spécifier l'accès* `public` *lors de la définition d'une propriété, on spécifie toujours l'accès* `private`*. On dit que les propriétés sont encapsulées, comme si elles étaient protégées par une capsule.*

8. Modifier en `private` l'accès aux propriétés de la classe `Equation`.

9. On veut néanmoins pouvoir lire et modifier (accès en lecture/écriture) chacun des coefficients d'une équation et on veut pouvoir lire le discriminant et les racines.

   ***Comment accéder aux propriétés ?** Comme il est maintenant impossible de lire ou de modifier les propriétés directement, on ajoute des méthodes d'instances qui ont uniquement pour rôle de permettre de lire ou de modifier certaines des propriétés.*

   - *On appelle **accesseur** une méthode qui permet de lire une propriété. L'identificateur d'un accesseur commence toujours par* `get`*. Par exemple, l'accesseur de* `a` *se nommera* `getA`.
   - *On appelle **modifieur** une méthode qui permet de modifier une propriété. L'identificateur d'un modifieur commence toujours par* `set`*. Par exemple, le modifieur de* `a` *se nommera* `setA`.

   Écrire les modifieurs et accesseurs nécessaires. Quelle instruction faut-il ajouter dans les modifieurs des coefficients pour éviter le bug de la question 6 ?

***Interface d'un objet :** on appelle interface d'un objet tout ce qui permet d'interagir avec celui-ci. Comme les propriétés doivent toujours être spécifiées avec l'accès* `private`*, seules les méthodes avec accès public permettent d'interagir avec l'objet. L'interface d'un objet est donc définie par ses méthodes publiques.*

Exercice 3 *Nombres complexes*
L'idée de cet exercice est de définir une classe correspondant aux nombres complexes. Vous allez définir la classe `Complexe` dans un fichier `Complexe.java`. De plus, tout au long de l'exercice, vous testerez votre classe à l'aide d'un programme de test (contenu dans le fichier `TestComplexe.java` par exemple).

1. Définir dans le fichier `Complexe.java` la classe `Complexe` qui contiendra deux propriétés (de type `double`) `re` et `im` correspondant respectivement à la partie réelle et imaginaire d'un nombre.

2. Définir deux constructeurs : le constructeur par défaut (sans arguments) initialisera le nombre complexe à 0.
3. Définir un second contructeur qui prendra en arguments deux nombres correspondant aux parties réelle et imaginaire du nombre à créer.
4. Définir les méthodes accesseurs `getRe`, `getIm` qui renvoient respectivement la partie réelle et imaginaire du nombre.
5. De la meme manière, définir les méthodes mofifieurs `setRe` et `setIm` qui permettent d'affecter à `re` ou à `im` une valeur donnée en argument. On peut maintenant modifier l'accès aux propriétés de la classe en `private`.
6. Définir la méthode `toString` qui renvoie la chaîne de caractères suivante "`<re> + i <im>`" où `<re>` et `<im>` sont bien sûr remplacés par les valeurs correspondantes.
7. Définir une méthode `additionner` qui additionne à l'objet courant un objet de type `Complexe` passé en argument.
8. De la même manière `multiplier` qui multiplie l'objet courant par un objet de type `Complexe` passé en argument.
9. Définir une méthode `rotation` qui fait subir à l'objet courant une rotation d'un angle dont la valeur en degrés est donnée en argument.(Vous pourrez utiliser la classe `Math`)