
FICHE D'EXERCICES

POO (UN PEU PLUS) AVANCÉE

TP Java 10

Objectifs :

- écriture d'une classe
- usage de l'encapsulation

Au cours de ce TP, nous allons définir une classe `Complexe` permettant de représenter et de manipuler des nombres complexes, ainsi qu'une classe `TestComplexe`, qui contiendra la fonction principale `main` qui testera la classe `Complexe`.

Nous aurons donc deux fichiers différents nommés `Complexe.java` et `TestComplexe.java`.

Exercice(s) préparatoire(s) (à faire avant la séance)

(0.1) *En vous aidant des transparents de cours du CM6 (en particulier les derniers), prédire et expliquer l'affichage du programme suivant.*

```
Vecteur v1 = new Vecteur();
v1.x = 10;
Vecteur v2 = new Vecteur();
v2.x = 20;
v1 = v2;
System.out.println(v1.x);
System.out.println(v2.x);
v1.x = 30;
System.out.println(v1.x);
System.out.println(v2.x);
v2.x = 40;
System.out.println(v1.x);
System.out.println(v2.x);
```



Remarque : *Pour répondre à cette question, il est vivement conseillé de faire un schéma sur papier du contenu de la mémoire.* ┘

À faire en séance

Exercice 1 : Mise en place de la classe `Complexe`

À la fin de cet exercice, le diagramme UML de la classe `Complexe` devra être le suivant :

Complexe
+ re : double
+ im : double
+ estDansCadrant() : boolean
+ getTheta() : double
+ getRho() : double

1.1 En cartésien

Dans ce premier exercice, on considère la représentation cartésienne des nombres complexes. Dans ce cadre, un objet `Complexe` est caractérisé par une **partie réelle** et une **partie imaginaire**, toutes deux de type réel et notées respectivement `re` et `im`.

(1.1) Créer la classe `Complexe`, de manière à ce que l'exécution de la classe `TestComplexe` fournie sur le moodle crée deux instances $z1 = -1 - 2i$ et $z2 = 2 + 4i$.

(1.2) On souhaite que l'instruction ci-dessous (appartenant à la méthode `main`) affecte à `b` la valeur `true` si $z2$ est situé dans le premier quadrant (ce qui est le cas ici). Faites les modifications nécessaires et testez le résultat.

```
boolean b = z2.estDansCadrant();
```

1.2 Et en polaire ?

On désire pouvoir obtenir les coordonnées polaires d'un nombre complexe z , c'est-à-dire les valeurs du couple (ρ, θ) tel que $z = \rho e^{i\theta}$. Le passage d'un système de coordonnées à l'autre se fait à l'aide des formules du tableau 1.

TABLE 1 – Conversion cartésien/polaire

coordonnées polaires	coordonnées cartésiennes
$\rho = \sqrt{x^2 + y^2}$	$x = \rho \times \cos \theta$
$\theta = \begin{cases} \arctan(y/x), & \text{si } (x > 0) \\ \text{sgn}(y) \times \pi/2, & \text{si } (x = 0) \\ \text{sgn}(y) \times \pi + \arctan(y/x), & \text{si } (x < 0) \end{cases}$	$y = \rho \times \sin \theta$

Remarque : Comme le suggère le tableau 1, le calcul de l'argument θ de $z = x + iy$ dépend du sous-ensemble de \mathbb{C} dans laquelle se situe z . La méthode `atan2` tient compte de ces différentes situations de sorte que l'instruction

```
double theta = Math.atan2(y,x);
```

permet de calculer θ dans tous les cas de figure. ┘

(1.3) Dans la classe `Complexe`, créez (et testez) les méthodes `getRho` et `getTheta` renvoyant le module ρ et l'argument θ respectivement. Les signatures de ces méthodes sont les suivantes :

```
public double getTheta()  
public double getRho()
```

Remarque : vous pourrez vérifier les calculs de la question précédente grâce aux expressions suivantes :

$$z_1 \approx 2,24 e^{-2,03i}; \quad z_2 \approx 4,47 e^{1,11i}$$

-
- (1.4) Vérifiez que votre classe est conforme au diagramme UML donné au début de cet exercice.

Exercice 2 : Affichage

Dans la suite du TP, nous allons faire évoluer la classe `Complexe`. Après l'avoir recopié sur papier, enrichissez progressivement le diagramme UML précédent pour y incorporer ces modifications.

- (2.1) Observez le résultat de l'exécution des instructions suivantes de la méthode `main`.

```
System.out.println(z1);  
System.out.println(z2);
```

L'affichage devrait ressembler à quelque chose de la forme `Complexe@80cc9b4`. Pour obtenir un résultat différent, il faut créer une méthode `toString` (écrite exactement de cette façon). Cette dernière est en effet particulière puisque, par convention, l'instruction `System.out.println(z)` affiche :

- la valeur de `z` si cette variable est d'un type primitif (ex : `int`, `char`, `double`...);
- dans le cas contraire :
 - la chaîne de caractère renvoyée par `z.toString()`, s'il existe une méthode `public String toString()` dans la classe dont `z` est une instance;
 - quelque chose de la forme `Complexe@80cc9b4` (renseignant sur la valeur du pointeur) sinon.

- (2.2) Faire les modifications permettant au code précédent d'afficher à l'écran le message suivant.

```
-1.0 - i*2.0  
2.0 + i*4.0
```

Exercice 3 : Encapsulation

Dans l'état actuel, chaque fois que l'on souhaite accéder aux coordonnées polaires d'un complexe `z`, il est nécessaire de faire le calcul de ρ et θ via les méthodes `getRho` et `getTheta`. Pourtant, si `z` n'a pas changé, ces calculs conduisent toujours au même résultat (et sont donc redondants).

Pour alléger la charge de calcul, on se propose dans cet exercice de stocker en mémoire ρ et θ sous la forme de nouveaux attributs de la classe `Complexe`.

Nous allons mettre en œuvre cette solution pas à pas dans la suite de cet exercice. Certaines des questions suivantes nécessitent donc individuellement peu de modification de votre code.

- (3.1) Ajouter les attributs `rho` et `theta` à la classe `Complexe`.

Dans l'état actuel, la solution à quatre attributs est dangereuse. En effet, les attributs d'un `Complexe` sont publiques (par défaut), c'est-à-dire que l'utilisateur de la classe (écrivant `TestComplexe`) a un accès en lecture et en écriture à tous les attributs. Or dans notre cas, il existe une relation bien

particulière entre les attributs, de sorte qu'ils ne peuvent pas être modifiés sans préserver leur cohérence, c'est-à-dire que les coordonnées cartésiennes et polaires doivent correspondre au même point dans le plan complexe.

Pour apporter une solution à cette situation, on se conforme au principe d'encapsulation (via les mots-clés `public/private`) de façon à protéger la cohérence de l'objet. Nous allons restreindre les accès aux attributs (principe d'encapsulation) de la façon suivante :

- tous les attributs sont privés et on ajoute des accesseurs (méthodes `set` et `get`) pour chacun d'eux, lorsque ces méthodes n'existent pas déjà ;
- à la fin de l'exécution de chaque méthode publique et de chaque constructeur, les attributs doivent être cohérents.

Remarque : *N'oubliez pas que les attributs privés sont accessibles dans les méthodes de la classe. Dans les méthodes du fichier `Complexe.java`, les accesseurs ne sont donc pas nécessaire pour accéder aux attributs privés. En revanche, il n'est plus possible d'accéder à ces derniers dans `TestComplexe.java`, ce qui risque d'invalider certaines instructions de la méthode `main` actuelle. On mettra ces instructions en commentaire.* ┘

(3.2) *Imposer à tous les attributs de `Complexe` d'être privés.*

(3.3) *Vérifier qu'une instruction du type `double x = z.re` dans la méthode `main` interdit la compilation. Lire le message de l'erreur apparaissant dans cette situation.*

À l'extérieur de `Complexe`, les accès en lecture aux attributs doivent maintenant être effectués par l'intermédiaire de méthodes publiques `getRe`, `getIm`, `getRho` et `getTheta`. On donne le code d'une d'entre elles :

```
public double getRe() {  
    return this.re;  
}
```

(3.4) *Ajoutez `getRe` et `getIm` et modifiez `getRho` et `getTheta` sur le modèle de la méthode `getRe`, puis testez ces méthodes dans la méthode `main` (ne pas supprimer le code maintenant superflu, mais le mettre en commentaire pour réutilisation ultérieure).*

Remarque : *L'usage des méthodes `getRe` et `getIm` pour les instances `z1` et `z2` devrait donner des résultats satisfaisants. Ce n'est pas le cas pour `getRho` et `getTheta` qui doivent, à ce stade, renvoyer toujours zéro. Pourquoi ? La suite de l'exercice va remédier à ce problème.* ┘

Nous allons maintenant garantir la cohérence entre les attributs en modifiant les constructeurs et en écrivant les accesseurs d'écritures.

(3.5) *Concevez les méthodes instrumentales `majPolaire` et `majCart` mettant à jour les coordonnées polaires à partir des coordonnées cartésiennes, et inversement, et dont les signatures sont les suivantes :*

```
private void majPolaire()  
private void majCartesien()
```

(3.6) Pourquoi les méthodes précédentes sont-elles privées ?

(3.7) En faisant usage de ces méthodes, modifiez le constructeur et créez les accesseurs d'écritures `setRe`, `setIm`, `setRho` et `setTheta` de la classe `Complexe`. Attention, à la fin de l'exécution de chacune de ces méthodes, la cohérence entre les attributs doit être assurée.

(3.8) Existe-t-il des cas d'utilisation de la classe `Complexe` pour lesquels la solution antérieure à deux attributs est préférable à celle à quatre ?

Exercice 4 : Surcharge


Pour ajouter de la flexibilité dans l'utilisation de la classe `Complexe`, on souhaite écrire plusieurs versions du constructeur. On parle de surcharge (ici du constructeur), qui permet au compilateur de sélectionner parmi différentes versions d'une même méthode se distinguant (uniquement et nécessairement) par le nombre et le type des paramètres requis.

(4.1) Sans remettre en cause le bon fonctionnement de la méthode `main` existante, modifiez le fichier `Complexe.java` pour permettre l'exécution des instructions suivantes de la méthode `main`.

```
Complexe z3 = new Complexe(); // donne z3 = 0
Complexe z4 = new Complexe(3); // donne z4 = 3 + y*i ou y est un reel aleatoire dans [0,10[
```

(4.2) Toujours sans remettre en cause le bon fonctionnement de la méthode `main` existante, faites les modifications autorisant la méthode `estDansCadrant` à accepter un paramètre correspondant au numéro du quadrant choisi.

On souhaite disposer d'un constructeur permettant d'initialiser un complexe à partir de ses coordonnées polaires.

(4.3) Pourquoi l'ajout d'un nouveau constructeur dont la signature est `public Complexe(double rhoInit, double thetaInit)` n'est pas possible ? 

On va enrichir le constructeur afin de pouvoir spécifier si la représentation utilisée pour la création d'un nombre complexe est de type polaire ou cartésienne. On introduit donc un troisième paramètre d'entrée au constructeur, qui spécifiera la représentation utilisée pour les deux premiers paramètres.

(4.4) Quel est le type de ce troisième paramètre ? Faites les modifications nécessaires. 

Exercice 5 : Opérations sur les complexes

Dans la suite de cet exercice, on souhaite concevoir des méthodes permettant d'effectuer les opérations suivantes sur les nombres complexes : somme et multiplication. Pour ce faire, plusieurs solutions sont possibles :

1. la méthode modifie l'objet courant, ce qui conduit à des instructions de `main` du type :

```
z1.sommeV1(z2); // les attributs de z1 sont modifiés apres cette instruction
```

-
2. le résultat de l'opération est renvoyé par la méthode, sous la forme d'une nouvelle instance de la classe `Complexe`, ce qui conduit à des instructions de `main` du type :

```
Complexe z3 = z1.sommeV2(z2); // ni z1, ni z2 ne sont modifiés après cette instruction
```

(5.1) *Créez ces deux versions de la méthode `somme` dans la classe `Complexe` puis testez leur bon fonctionnement via `main`.*

Remarque : À nouveau, n'oubliez pas qu'à la fin de l'exécution de chacune de ces méthodes, la cohérence entre les attributs doit être assurée. ┘

(5.2) *Sur le même principe, créez et testez deux versions de la méthode `multiplication`.*

Remarque : Notez que le produit de deux complexes est plus simple à écrire en utilisant les coordonnées polaires :

$$\rho(c1 \times c2) = \rho(c1) \times \rho(c2)$$

$$\theta(c1 \times c2) = \theta(c1) + \theta(c2)$$

┘

Exercice 6 : Représentation mémoire des instances

Grâce aux exercices préparatoires, vous savez que `(z1==z2)` ne permet pas de tester si `z1` et `z2` correspondent au même point de \mathbb{C} . Pour faire ce type de test, il faut en effet comparer les *attributs* associés à ces deux instances de `Complexe` et non pas leurs *valeurs* (qui sont en fait des adresses mémoires).

Pour donner accès à cette fonctionnalité (le test d'égalité des attributs), on crée typiquement la méthode suivante

```
public boolean equals(Complexe z)
```

dans la classe `Complexe` (on pourra notamment vérifier que la classe `String` possède, elle-aussi, une méthode `equals`, dont la signature est analogue).

(6.1) *Écrire cette méthode `equals` et tester son bon fonctionnement en enrichissant la méthode `main`.*

Bonus (pour vous entraîner)

Exercice 7 : Transformations géométriques

Il est possible de modéliser une transformation géométrique du plan (translation, rotation, homothétie) par une opération avec un nombre complexe. En effet, si on modélise un point A de coordonnées (x, y) par le nombre complexe $z = x + i \times y$, on peut alors lui appliquer les transformations suivantes :

-
- translation d'un vecteur (u_x, u_y) :

$$z' = z + b, \quad \text{où } b = u_x + i \times u_y$$

- rotation de centre (x_B, y_B) et d'angle ω :

$$z' = e^{i\omega} \times (z - c) + c, \quad \text{où } c = x_B + i \times y_B$$

En utilisant ce principe, on souhaite effectuer les opérations suivantes :

- définir un point initial de coordonnées $(1, 1)$,
- appliquer à ce point une translation de vecteur $(2, 3)$,
- appliquer au résultat une rotation de centre $(-1, 1)$ et d'angle $\pi/4$.

Remarque : En Java, π est obtenu via `Math.PI`.

┘

(7.1) *Enrichir la méthode `main` afin de réaliser les opérations précédentes (conduisant au point $(-0.29, 5.95)$). On affichera le résultat de chaque étape intermédiaire.*