Programmation Java @ Et3
Polytech Paris-Saclay | 2024-25

**Nom :** BRACQ
**Prénom :** Paolo

## Exercice 3

On souhaite développer des classes Java permettant de faire certaines manipulations sur des nombres complexes.

1. Définissez une classe `Complexe` comprenant deux champs de type double, représentant respectivement la partie réelle et la partie imaginaire d'un nombre complexe, puis munissez-la d'un constructeur prenant une valeur pour la partie réelle et la partie imaginaire du nouveau nombre complexe.

#3.1.1 classe `Complexe`

```
/**
 * Constructeur d'une instance de la classe Complexe
 * @param real_part
 * @param imaginary_part
 */
public Complexe(double real_part,double imaginary_part){
    real=real_part;
    imaginary=imaginary_part;
}
```

2. Ajoutez à la classe `Complexe` le code nécessaire pour permettre un affichage approprié pour des objets de cette classe en redéfinissant la méthode `toString()` de la classe `Object`.

#3.2.1 méthode `toString`

```
/**
 * Affichage d'un nombre complexe
 * @return String qui représente la valeur complexe
 */
@Override
public String toString(){
    if(imaginary == 0 && real == 0){
        return "0";
    }
    if(real == 0){
        return imaginary+"* i ";
    }
    if(imaginary == 0){
        return ""+real;
```

```
    }

    return real+" + "+imaginary+"* i ";
}
```

3. On ne souhaite pas que les objets de la classe `Complexe` puissent être modifiés une fois créés (on les qualifiera donc d'*immuables* (*immutable*)). Comment peut-on garantir cela ? Implémentez la partie pertinente des méthodes d'accès.

#3.3.1 Approche pour garantir la non modification des objects `Complexe`

Puisque nous avons uniquement des types primitifs (double), il est coherent et efficace d'utiliser *final* .

```
public class Complexe {
    private final double real;
    private final double imaginary;
```

#3.3.2 Méthodes d'accès

```
/**
 * Méthode d'accès pour la partie imaginaire
 * @return double
 */
public double getImaginary() {
    return imaginary;
}

/**
 * Méthode d'accès pour la partie réelle
 * @return double
 */
public double getReal() {
    return real;
}
```

4. Ajoutez un constructeur de copie à votre classe prenant en unique paramètre une instance de la classe `Complexe`. Quelle peut être dans ce contexte l'utilité d'un tel constructeur ?

#3.4.1 Nouveau constructeur de `Complexe`

```
/**
 *Constructeur de copie
 * @param number il s'agit d'une instance de la classe Complexe
 */
public Complexe(Complexe number){
    real=number.real;
    imaginary=number.imaginary;
}
```

#3.4.2 Quelle est l'utilité d'un tel constructeur ?

Les constructeurs de copie peuvent être utilisés pour passer des objets en tant que paramètres à des méthodes, sans modifier l'objet original. Cela est particulièrement utile lorsque l'on souhaite assurer que la méthode ne modifie pas accidentellement l'objet original.

5. Ajoutez le code nécessaire à votre classe pour tester l'égalité d'état de deux objets de la classe `Complexe` par redéfinition de la méthode `equals` de la classe `Object`.

#3.5.1 redéfinition de `equals`

```
/**
 * Redéfinition de la fonction equals pour deux objets de la classe Complexe
 * @param autreObjet instance de la classe Complexe
 * @return booleen
 */
@Override
public boolean equals(Object autreObjet){
    Complexe number =  (Complexe) autreObjet;
    if (number.real != real){
        return false;
    }
    if (number.imaginary != imaginary){
        return false;
    }
    return true;
}
```

#3.5.2 tests sur plusieurs valeur d'objets `Complexe`

```
package Complx_number;

public class TestComplx {
    public static void main(String[] args) {
        Complexe test1 = new Complexe(2, 3);
        System.out.println("test1 = "+test1);
        Complexe test2 = new Complexe(0, 3);
        System.out.println("test2 = "+test2);
        Complexe test3 = new Complexe(2, 0);
        System.out.println("test3 = "+test3);
        Complexe test4 = new Complexe(0, 0);
        System.out.println("test4 = "+test4);
        Complexe test5 = new Complexe(2, 3);
        System.out.println("test5 = "+test5);
        System.out.println("Est-ce que test1=test2 : "+test1.equals(test2));
        System.out.println("Est-ce que test1=test5 : "+test1.equals(test5));
        System.out.println("Est-ce que test1=test3 : "+test1.equals(test3));
        System.out.println("Est-ce que test1=test4 : "+test1.equals(test4));

    }
}
```

```
Avec comme résultat:

test1 = 2.0 + 3.0 * i
test2 = 3.0 * i
test3 = 2.0
```

```
test4 = 0
test5 = 2.0 + 3.0 * i
Est-ce que test1=test2 : false
Est-ce que test1=test5 : true
Est-ce que test1=test3 : false
Est-ce que test1=test4 : false

Process finished with exit code 0
```

6. Ajoutez des méthodes d'instance pour le calcul du module et de l'argument d'un complexe. On rappelle :

— module = sqrt($re^2$ + $im^2$)
— argument = acos( re / module )

#3.6.1 nouvelles méthodes

```java
/**
 * Méthode d'instance pour le calcul du module
 * @return double qui représente le module
 */
public double module(){
    return Math.sqrt(Math.pow(real,2)+Math.pow(imaginary,2));
}

/**
 * Méthode d'instance pour le calcul de l'argument
 * @return double qui représente l'argument
 */
public double argument(){
    return Math.acos(real/module());
}
```

#3.6.2 Est-il plus pertinent d'ajouter de nouveaux champs à la classe ou bien de faire des calculs de ces valeurs à la volée ?

Je pense qu'il est plus pertinent de faire les calculs car on n'a pas toujours besoin de ces valeurs.

7. Définissez des méthodes pour l'addition et la multiplication de deux nombres complexes. Ces méthodes prendront un paramètre implicite et un paramètre explicite, et retourneront une nouvelle instance correspondant au résultat de l'opération. On rappelle :

— ($re_1$ + $im_1$ i) + ($re_2$ + $im_2$ i) = ($re_1$ + $re_2$ + ($im_1$ + $im_2$)i)
— ($re_1$ + $im_1$ i) x ($re_2$ + $im_2$ i) = ($re_1$ x $re_2$ − $im_1$ x $im_2$ + ($re_1$ x $im_2$ + $im_1$ x $re_2$)i)

#3.7.1 nouvelles méthodes

```java
/**
 * Méthodes pour l'addition de deux nombres complexes
 * @param autreNombre instance de la classe Complexe
 * @return une nouvelle instance correspondant au résultat de l'addition
 */
public Complexe addition(Complexe autreNombre){
```

```
    return new Complexe(real+autreNombre.real,imaginary+ autreNombre.imaginary);
}

/**
 * Méthodes pour la multipication de deux nombres complexes
 * @param autreNombre instance de la classe Complexe
 * @return une nouvelle instance correspondant au résultat de la multiplication
 */
public Complexe multiplication(Complexe autreNombre){
    return new Complexe((real*autreNombre.real)-(imaginary*autreNombre.imaginary),(real*autreNombre.imaginary)+(imaginary* autreNombre.real));
}
```

Avec des tests:

```
System.out.println("test1 + test2 : "+test1.addition(test2));
System.out.println("test1 * test5 : "+test1.multiplication(test5));
System.out.println("test1+test4 : "+test1.addition(test4));
System.out.println("test1*test4 : "+test1.multiplication(test4));
```

```
test1 + test2 : 2.0 + 6.0 * i
test1 * test5 : -5.0 + 12.0 * i
test1+test4 : 2.0 + 3.0 * i
test1*test4 : 0
```

8. On souhaite à présent bénéficier de la classe `Complexe` et définir une classe permettant de représenter un nombre complexe telle que l'historique des opérations dans lesquelles ce complexe aura servi d'opérande est conservé. Définissez une nouvelle classe `ComplexeMemoire`, dans un autre package que `Complexe`, permettant de réaliser cela. Pour l'historique des opérations, on veut garder la trace des opérations subies (addition ou multiplication), de la valeur des autres opérandes et des résultats obtenus (une simple chaîne de caractères pourra convenir ici). Proposez une implémentation appropriée, et ajoutez un constructeur prenant une partie réelle et une partie imaginaire en paramètres.

#3.8.1 classe `ComplexeMemoire`

```
package Mémoire;
import Complx_number.Complexe;



public class ComplexeMemoire extends Complexe {
    private String history;

    public ComplexeMemoire(double partieReelle, double partieImaginaire) {
        super(partieReelle, partieImaginaire);
        this.history = "";
    }
}
```

9. Ajoutez un constructeur par copie à la classe `ComplexeMemoire`.

#3.9.1 constructeur par copie de `ComplexeMemoire`

```
    ComplexeMemoire(ComplexeMemoire autreComplxe){
        super(autreComplxe.getReal(), autreComplxe.getImaginary());
        history= autreComplxe.history;
    }
}
```

10. Serait-il possible d'ajouter simplement un constructeur sans paramètres à la classe ComplexeMemoire ?

#3.10.1 réponse et solution(s) possible(s)

Oui, il est tout à fait possible d'ajouter un constructeur sans paramètres à la classe ComplexeMemoire

```
/**
 * Constructeur sans paramètres de ComplexeMemoire
 * Initialise un complexe nul avec une mémoire vide
 */
public ComplexeMemoire() {
   super(0, 0); // Appelle le constructeur de Complexe avec 0 + 0i
   this.history = "";
}
```

11. Ajoutez à la classe `ComplexeMemoire` les méthodes nécessaires pour pouvoir ajouter des messages à la mémoire des opérations d'un objet de la classe et consulter cette mémoire.

#3.11.1 nouvelles méthodes

```
/**
 * Ajouter des messages à la mémoire des opérations d'un objet de la classe
 * @param message
 */
public void addMessage(String message) {
    history+="\n " +message;
}

/**
 * Consulter la mémoire
 * @return
 */
public String consultMemoire() {
    return history;
}
```

12. Proposez à présent une redéfinition adaptée dans la classe `ComplexeMemoire` des méthodes d'addition et de multiplication de complexes définies dans la classe `Complexe`.

#3.12.1 Est-il possible et utile d'adapter le type de retour (covariance) ?

Cela permet d'enchaîner les opérations spécifiques à `ComplexeMemoire`, comme l'enregistrement dans la mémoire, sans avoir besoin de caster le résultat.

#3.12.2 redéfinition des méthodes d'addition et de multiplication dans `ComplexeMemoire`

```
/**
 * Redéfinition des méthodes d'addition
 * @param autre instance de la classe Complexe
 * @return
 */
@Override
public ComplexeMemoire addition(Complexe autre) {
    Complexe resultat = super.addition(autre);

    if(autre instanceof ComplexeMemoire nouveau){
        nouveau.addMessage("Addition avec " + autre + " résultat: " + resultat);
    }

    addMessage("Addition avec " + autre + " résultat: " + resultat);

    return new ComplexeMemoire(resultat.getReal(), resultat.getImaginary());
}

/**
 * Redéfinition des méthodes de la multiplication
 * @param autre instance de la classe Complexe
 * @return
 */
@Override
public ComplexeMemoire multiplication(Complexe autre) {
    Complexe resultat = super.multiplication(autre);
    if(autre instanceof ComplexeMemoire nouveau){
        nouveau.addMessage("Multiplication avec " + autre + " résultat: " + re-
sultat);
    }
    addMessage("Multiplication avec " + autre + " résultat: " + resultat);
    return new ComplexeMemoire(resultat.getReal(), resultat.getImaginary());
}
```

13. On souhaite à présent mettre en place une mémoire collective où apparaît une seule fois chaque opération effectuée sur des instances de `ComplexeMemoire`. Adaptez votre classe afin de permettre cela.

#3.13.1 nouvelle définition de `ComplexeMemoire`

```
package Mémoire;
import Complx_number.Complexe;


public class ComplexeMemoire extends Complexe {
    private String history;
    private static String memoireCollective;

    /**
     * Constructeur d'une instance de ComplexeMemoire
     *
     * @param partieReelle
```

```
     * @param partieImaginaire
     */
    public ComplexeMemoire(double partieReelle, double
partieImaginaire) {
        super(partieReelle, partieImaginaire);
        this.history = "";
    }

    /**
     * Constructeur par copie
     *
     * @param autreComplxe
     */
    ComplexeMemoire(ComplexeMemoire autreComplxe) {
        super(autreComplxe.getReal(),
autreComplxe.getImaginary());
        history = autreComplxe.history;
    }

    /**
     * Ajouter un message à l'historique des opérations
     *
     * @param message texte à ajouter
     */
    public void addMessage(String message) {
        history += message + "\n";
        memoireCollective += message; // Ajout unique à la mémoire
collective
    }

    /**
     * Consulter la mémoire individuelle des opérations
     *
     * @return l'historique des opérations
     */
    public String consultMemoire() {
        return history;
    }

    /**
     * Consulter la mémoire collective des opérations
     *
     * @return l'historique global des opérations effectuées sur
toutes les instances
     */
    public static String consultMemoireCollective() {
        return String.join("\n", memoireCollective);
    }

    /**
     * Redéfinition de la méthode d'addition
     *
     * @param autre instance de la classe Complexe
```

```
     * @return un nouvel objet ComplexeMemoire représentant le
résultat
     */
    @Override
    public ComplexeMemoire addition(Complexe autre) {
        Complexe resultat = super.addition(autre);

        String message = "Addition avec " + autre + " résultat: "
+ resultat;
        addMessage(message);

        if (autre instanceof ComplexeMemoire) {
            ((ComplexeMemoire) autre).addMessage(message);
        }

        ComplexeMemoire nouveauResultat = new
ComplexeMemoire(resultat.getReal(), resultat.getImaginary());
        nouveauResultat.history = this.history;

        return nouveauResultat;
    }

    /**
     * Redéfinition de la méthode de multiplication
     *
     * @param autre instance de la classe Complexe
     * @return un nouvel objet ComplexeMemoire représentant le
résultat
     */
    @Override
    public ComplexeMemoire multiplication(Complexe autre) {
        Complexe resultat = super.multiplication(autre);

        String message = "Multiplication avec " + autre + "
résultat: " + resultat;
        addMessage(message);

        if (autre instanceof ComplexeMemoire) {
            ((ComplexeMemoire) autre).addMessage(message);
        }

        ComplexeMemoire nouveauResultat = new
ComplexeMemoire(resultat.getReal(), resultat.getImaginary());
        nouveauResultat.history = this.history;

        return nouveauResultat;
    }
}
```

14. Sans modifier le code développé jusqu'à présent, que se passera-t-il si l'on invoque la méthode `equals` sur deux instances de la classe `ComplexeMemoire` ? Sur une instance de la classe `Complexe` et une instance de la classe `ComplexeMemoire` ?

#3.14.1 réponse et tests

```
Par principe je dirais que pour le premier cas, même si c1 et c2
ont les mêmes valeurs, ils ne sont pas considérés égaux car ils
sont stockés à des adresses mémoire différentes.

Et que pour le second cas, si Complexe ne redéfinit pas equals(),
le comportement est le même. Même si c3 et c4 représentent le même
nombre complexe, l'implémentation de equals de Object fait que ce
test retourne false.

Or ComplexeMemoire est un enfant de Complexe, donc nous n'avons
pas besoin de réfinir equals().

package Mémoire;

import Complx_number.Complexe;

public class Test_Memoire {
    public static void main(String[] args) {
        ComplexeMemoire c1 = new ComplexeMemoire(1, 2);
        ComplexeMemoire c2 = new ComplexeMemoire(1, 2);

        System.out.println(c1.equals(c2));

        Complexe c3 = new Complexe(1, 2);
        ComplexeMemoire c4 = new ComplexeMemoire(1, 2);

        System.out.println(c3.equals(c4));

    }
}

Résultat :

true
true
```