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

Nom : BENCHIKHI
Prénom : Anas

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`

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

    public Complexe(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    public double getReal() {
        return real;
    }

    public double getImaginary() {
        return imaginary;
    }
}
```

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`

```
@Override
    public String toString() {
        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`

Il faut faire en sorte que le mot clé "final" apparaissent devant la partie réelle et la partie imaginaire pour qu'ils soient immuables.

#3.3.2 Méthodes d'accès

```
    private final double real;
    private  final double imaginary;
```

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`

```
  public Complexe(Complexe other) {
        this.real = other.real;
        this.imaginary = other.imaginary;
    }
```

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

C'est utile pour créer des copies indépendantes d'objets complexes, pour les manipuler sans toucher à l'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`

```
@Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Complexe complexe = (Complexe) obj;
        return Double.compare(complexe.real, real) == 0 &&
               Double.compare(complexe.imaginary, imaginary) == 0;
    }
```

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

```
public class Mainframe {
    public static void main(String[] args) {
        Complexe c1 = new Complexe(1, 2);
        Complexe c2 = new Complexe(1, 2);
        Complexe c3 = new Complexe(3, 4);

        System.out.println(c1.equals(c2));
        System.out.println("c1: " + c1);
        System.out.println("c2: " + c2);
        System.out.println("c3: " + c3);

        System.out.println("c1.equals(c2): " + c1.equals(c2)); //
true
        System.out.println("c1.equals(c3): " + c1.equals(c3)); //
false

    }
}
```

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

```
 public double module() {
        return Math.sqrt(real * real + imaginary * imaginary);
    }

    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 ?

Il est plus pertinent de faire des calculs de ces valeurs à la volée.

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 :

— $(re_1 + im_1\ i) + (re_2 + im_2\ i) = (re_1 + re_2 + (im_1 + im_2)i)$
— $(re_1 + im_1\ i) \times (re_2 + im_2\ i) = (re_1 \times re_2 - im_1 \times im_2 + (re_1 \times im_2 + im_1 \times re_2)i)$

#3.7.1 nouvelles méthodes

```
public Complexe add(Complexe other) {
        return new Complexe(this.real + other.real, this.imaginary
+ other.imaginary);
    }

    public Complexe multiply(Complexe other) {
        double newReal = this.real * other.real - this.imaginary *
other.imaginary;
        double newImaginary = this.real * other.imaginary +
this.imaginary * other.real;
        return new Complexe(newReal, newImaginary);
    }
```

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`

```
import com.example.Complexe;
import java.util.ArrayList;
import java.util.List;

public class ComplexeMemoire extends Complexe {
    private final List<String> historique;
    private static final List<String> historiqueCollectif = new
ArrayList<>();


    public ComplexeMemoire(double real, double imaginary) {
        this.complexe = new Complexe(real, imaginary);
        this.historique = new ArrayList<>();
    }

    public ComplexeMemoire add(ComplexeMemoire other) {
        Complexe result = this.complexe.add(other.complexe);
        String operation = this.complexe + " + " + other.complexe
+ " = " + result;
        historique.add(operation);
        other.historique.add(operation);
```

```
        return new ComplexeMemoire(result.getReal(),
result.getImaginary());
    }

    public ComplexeMemoire multiply(ComplexeMemoire other) {
        Complexe result = this.complexe.multiply(other.complexe);
        String operation = this.complexe + " * " + other.complexe
+ " = " + result;
        historique.add(operation);
        other.historique.add(operation);
        return new ComplexeMemoire(result.getReal(),
result.getImaginary());
    }

    public List<String> getHistorique() {
        return new ArrayList<>(historique);
    }

    @Override
    public String toString() {
        return complexe.toString();
    }
}
```

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

#3.9.1 constructeur par copie de `ComplexeMemoire`

```
    public ComplexeMemoire(ComplexeMemoire other) {
        this.complexe = new Complexe(other.complexe);
        this.historique = new ArrayList<>(other.historique);
    }
```

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 possible d'ajouter un constructeur. Ce constructeur initialisera un nombre complexe avec des valeurs par défaut (par exemple, 0 pour la partie réelle et 0 pour la partie imaginaire) et un historique vide.

Code :

```
  public ComplexeMemoire() {
    this.complexe = new Complexe(0, 0);
    this.historique = new ArrayList<>();
  }
```

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

```
 public void ajouterMessage(String message) {
        historique.add(message);

    }

  public List<String> getHistorique() {
        return new ArrayList<>(historique);
    }
```

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) ?

Oui c'est possible et ça permet de retourner des instances de ComplexeMemoire au lieu de Complexe dans les methodes redéfinies, ce qui est plus utile dans ce context.

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

```
public ComplexeMemoire add(ComplexeMemoire other) {
        Complexe result = this.complexe.add(other.complexe);
        String operation = this.complexe + " + " + other.complexe
+ " = " + result;
        historique.add(operation);
        other.historique.add(operation);
        ComplexeMemoire resultMemoire = new
ComplexeMemoire(result.getReal(), result.getImaginary());
        resultMemoire.historique.addAll(this.historique);
        resultMemoire.historique.addAll(other.historique);
        resultMemoire.historique.add(operation);
        return resultMemoire;
    }

    public ComplexeMemoire multiply(ComplexeMemoire other) {
        Complexe result = this.complexe.multiply(other.complexe);
        String operation = this.complexe + " * " + other.complexe
+ " = " + result;
        historique.add(operation);
        other.historique.add(operation);
        ComplexeMemoire resultMemoire = new
ComplexeMemoire(result.getReal(), result.getImaginary());
        resultMemoire.historique.addAll(this.historique);
```

```
        resultMemoire.historique.addAll(other.historique);
        resultMemoire.historique.add(operation);
        return resultMemoire;
    }
```

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 com.example.memoirec;

import com.example.Complexe;
import java.util.ArrayList;
import java.util.List;

public class ComplexeMemoire extends Complexe {
    private final List<String> historique;
    private static final List<String> historiqueCollectif = new
ArrayList<>();

    // Constructeur sans paramètres
    public ComplexeMemoire() {
        super(0, 0);
        this.historique = new ArrayList<>();
    }

    public ComplexeMemoire(double real, double imaginary) {
        super(real, imaginary);
        this.historique = new ArrayList<>();
    }

    // Constructeur par copie
    public ComplexeMemoire(ComplexeMemoire other) {
        super(other.getReal(), other.getImaginary());
        this.historique = new ArrayList<>(other.historique);
    }

    @Override
    public ComplexeMemoire add(Complexe other) {
        Complexe result = super.add(other);
        String operation = this + " + " + other + " = " + result;
        historique.add(operation);
        historiqueCollectif.add(operation);
        return new ComplexeMemoire(result.getReal(),
result.getImaginary());
    }

    @Override
    public ComplexeMemoire multiply(Complexe other) {
```

```
        Complexe result = super.multiply(other);
        String operation = this + " * " + other + " = " + result;
        historique.add(operation);
        historiqueCollectif.add(operation);
        return new ComplexeMemoire(result.getReal(),
result.getImaginary());
    }

    // Méthode pour ajouter un message à l'historique
    public void ajouterMessage(String message) {
        historique.add(message);
        historiqueCollectif.add(message);
    }

    // Méthode pour consulter l'historique
    public List<String> getHistorique() {
        return new ArrayList<>(historique);
    }

    // Méthode pour consulter l'historique collectif
    public static List<String> getHistoriqueCollectif() {
        return new ArrayList<>(historiqueCollectif);
    }

    @Override
    public String toString() {
        return super.toString();
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        ComplexeMemoire that = (ComplexeMemoire) obj;
        return super.equals(that);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }
}
```

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

```
Si l'on invoque la méthode equals sur deux instances de la
classe ComplexeMémoire, la méthode equals de la classe Object sera
utilisée par défaut, car la classe ComplexeMémoire n'a pas
redéfini cette méthode. La méthode equals de la
classe Object compare les références des objets, ce qui signifie
qu'elle retournera true uniquement si les deux références pointent
vers le même objet en mémoire.

Pour une instance de la classe Complexe et une instance de la
classe ComplexeMemoire, la méthode equals de la
classe Complexe sera utilisée si l'on invoque equals sur
l'instance de Complexe avec l'instance de ComplexeMemoire comme
argument. La méthode equals de Complexe retournera false car les
deux objets ne sont pas de la même classe.


Code de tests:

 // Test equals sur deux instances de ComplexeMemoire
        System.out.println("cm1.equals(cm2): " +
cm1.equals(cm2)); // false, car ce sont des objets différents
   System.out.println("cm1.equals(cm3): " + cm1.equals(cm3));
// true, car cm3 référence le même objet que cm1

        // Test equals sur une instance de Complexe et une
instance de ComplexeMemoire
   System.out.println("c1.equals(cm1): " + c1.equals(cm1));
// false, car ce ne sont pas des objets de la même classe
   System.out.println("cm1.equals(c1): " + cm1.equals(c1));
// false, car ce ne sont pas des objets de la même classe
```