Algorithmes et Pensée Computationnelle

Programmation orientée objet : Héritage et Polymorphisme - Exercices basiques

Le but de cette séance est d'approfondir les notions de programmation orientée objet vues précédemment. Nous commencerons par un rappel des notions de surcharge d'opérateurs avant d'introduire des exercices sur l'héritage et le polymorphisme. Au terme de cette séance, vous devez être en mesure de factoriser votre code afin de le rendre mieux structuré et plus lisible. À chaque exercice, le langage de programmation à utiliser sera spécifié.
Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier **Code**.

1 Rappel : Surcharge des opérateurs - Python

Dans cette section, vous manipulerez des fractions sous forme d'objets. Vous ferez des opérations de base sur ce nouveau type d'objets.

Question 1: (🕓 *5 minutes*) **Création de classe**
Dans un projet que vous aurez au préalable préparé, créez un fichier nommé **surcharge.py**. À l'intérieur de ce fichier, créez une classe **Fraction** qui aura comme attributs privés un numérateur et un dénominateur.

Question 2: (🕓 *5 minutes*) **Constructeur par défaut**
Définir un constructeur à votre classe. Assignez des valeurs par défaut à vos attributs.
Si un seul argument est passé à votre constructeur, la fraction devra être égale à l'entier correspondant.
Empêchez l'utilisateur d'assigner la valeur **zéro** au dénominateur.

> **Conseil**
>
> Les valeurs par défaut seront assignées à votre objet au cas où il est instancié sans valeurs. Ainsi en faisant **f = Fraction()**, on obtiendra un objet **Fraction** ayant pour valeurs un numérateur à 0 et un dénominateur à 1 soit $\frac{0}{1}$.
> En Python, vous pouvez donner des valeurs par défaut aux arguments de vos méthodes lors de leur définition. Par exemple, vous pouvez faire : **def add(self, x=10, y=5):...**

Question 3: (🕓 *5 minutes*) **Type casting**
Convertir les attributs en entier.

> **Conseil**
>
> Pensez à utiliser la fonction **int**.

Question 4: (🕓 *5 minutes*) **Redéfinition de méthodes**
Redéfinir la méthode **__str__()** pour produire une représentation textuelle de vos objets **Fraction**.

> **Conseil**
>
> Une fois la méthode **__str__()** redéfinie, lorsqu'on fera un **print()** sur une instance de votre classe **Fraction**, il affichera le message suivant : *Votre fraction a pour valeur* ***numérateur/dénominateur***. **Numérateur** et **dénominateur** étant les valeurs que vous passerez à votre objet **Fraction**.

Question 5: (🕓 *5 minutes*) **Accesseurs et mutateurs**
Créer des **getters** et **setters** pour chacun des attributs de votre classe **Fraction**.

Question 6: (🕓 *10 minutes*) **Simplification de fractions**
Définir une méthode **simplification** qui réduit la **Fraction**. Cette méthode ne renverra rien, elle modifiera simplement l'instance. Pour la suite des exercices, assurez-vous de toujours manipuler des fractions simplifiées.

Pour ce faire, vous pouvez faire appel à votre méthode **simplification** après chaque opération sur un objet de type **Fraction**.

Conseil

Afin de simplifier une fraction, vous devez diviser le numérateur et le dénominateur par leur plus grand diviseur commun. Pensez à utiliser la méthode **math.gcd** pour trouver le plus grand diviseur commun entre deux nombres.

Question 7: (*15 minutes*) **Redéfinition de méthodes - __eq__**
Redéfinir la méthode d'instance __eq__ qui prend en entrée un objet **Fraction** que vous nommerez *other* (en plus de **self**) et qui renvoie **True** si **self** et l'objet passé en argument ont la même valeur.

Conseil

Pour vérifier l'égalité entre a/b et c/d, tester que $a * d$ est égal à $b * c$.
Utilisez la méthode **isinstance()** afin de vérifier que *other* est bien de type **Fraction**. Dans le cas contraire, affichez un message d'erreur.
La fonction **isinstance** prend en entrée une valeur et un type. Elle vérifie que cette valeur est du type défini. Par exemple : **isinstance(nombre, int)** renverra **True** si la variable **nombre** est de type **int** et **False** dans le cas contraire.

Question 8: (*15 minutes*) **Addition et multiplication**
Redéfinir les méthodes __add__ et __mul__ afin d'effectuer des opérations d'addition et de multiplication sur vos objets de type **Fraction**. Attention, ces méthodes devront renvoyer des objets de type **Fraction**. Dans vos méthodes __add__ et __mul__, n'oubliez pas de simplifier ces fractions avant de les retourner.
Gérer le cas où l'élément passé en argument n'est ni une **Fraction**, ni un **int**.

Solution

```python
import math


# Question 1: Création de la classe Fraction
class Fraction:
    # Question 2: Déclaration du constructeur et initialisation des valeurs
    def __init__(self, numerateur=0, denominateur=1):
        # Question 3: type casting
        self.__num = int(numerateur)
        self.__den = int(denominateur)
        self.simplification()

    # Question 4: redéfinition de la méthode __str__()
    def __str__(self):
        return "Votre fraction a pour valeur {}/{}".format(self.__num, self.__den)

    # Question 5: Getters et Setters
    def get_num(self):
        return self.__num

    def get_den(self):
        return self.__den

    def set_num(self, n):
        self.__num = n

    def set_den(self, d):
        d = int(d)
        # Si on passe 0 au dénominateur, on lève une exception ce qui arrêtera le programme
        if d == 0:
            raise ZeroDivisionError
        self.__den = d

    # Question 6
    def simplification(self):
        if self.__num == 0:
            self.__den = 1
        if self.__den < 0:
            self.__num = -self.__num
            self.__den = -self.__den
        pgcd = math.gcd(self.__num, self.__den)
        self.__num = int(self.__num / pgcd)
        self.__den = int(self.__den / pgcd)

    # Question 7
    def __eq__(self, f):
        if isinstance(f, Fraction):
            # vu que les fractions sont toujours en représentation simplifiée, on pourrait se contenter de
            # self.__numerateur == f.__numerateur and self.__denominateur = f.denominateur
            return self.__num * f.__den == f.__num * self.__den
        # Au cas où on re coit un seul argument, on créé une fraction ayant pour numérateur l'argument et 1 comme
          dénominateur
        elif isinstance(f, int):
            return self.__eq__(Fraction(f))
        else:
            return False
```

Solution

```
    # Question 8
    def add(self, f):
        self.__num = self.__num * f.__den + f.__num * self.__den
        self.__den = self.__den * f.__den
        self.simplification()

    def plus(self, f):
        q = Fraction(self.__num, self.__den)
        q.add(f)
        return q

    def __add__(self, other):
        if isinstance(other, Fraction):
            return self.plus(other)
        elif isinstance(other, int):
            self.add__ = self.__add__(Fraction(other))
            return self.add__
        else:
            print("Unsupported operand types for +: '" + self.__class__.__name__ + "' and '" + other.__class__.__name__
             + "'")

    def __mul__(self, other):
        if isinstance(other, Fraction):
            return Fraction(self.__num * other.__num, self.__den * other.__den)
        elif isinstance(other, int):
            return Fraction(self.__num * other, self.__den)
        # On affiche un message d'erreur lorsque other n'est pas une Fraction
        else:
            print("Unsupported operand types for *: '" + self.__class__.__name__ + "' and '" + other.__class__.__name__
             + "'")

if __name__ == '__main__':
    f1 = Fraction()
    print(f1)
    f1 = Fraction(4)
    print(f1)
    f1 = Fraction(denominateur=5)
    print(f1)
    f1 = Fraction(4, -6)
    print(f1)
    f2 = Fraction(2, -8)
    print(f2)
    print(f1+f2)
    print(f1 == Fraction(22, -24))
    print(f1 == Fraction(1, 2))
```

2 Notions d'héritage - Java

Le but de cette partie est de mettre en pratique les notions liées à l'héritage. Nous allons créer une classe **Livre()** qui représentera notre classe-mère. Nous allons également créer deux classes filles, **Livre_Audio()** et **Livre_Illustre()**. Les classes filles hériteront des attributs et méthodes de la classe-mère.

Question 9: (*20 minutes*) **Création des différentes classes**
Créez la classe-mère **Livre** avec les caractéristiques suivantes :

- un attribut **privé String** nommé **titre**,
- un attribut **privé String** nommé **auteur**,
- un attribut **privé int** nommé **annee**,
- un attribut **privé int** nommé **note** (initialisé à **–1**),
- le **constructeur** de la classe qui prendra les trois premiers arguments cités ci-dessus,
- une méthode **setNote()** qui permet de définir l'attribut **note**,
- une méthode **getNote()** qui permet de retourner l'attribut **note**,
- une méthode **toString()** qui retournera le titre, l'auteur, l'année et la note d'un ouvrage **note** (réécrire cette méthode permettra d'afficher un objet **Livre** en utilisant **System.out.println()**).

Attention, si la **note** n'a pas été modifiée et qu'elle vaut toujours **–1**, affichez "Note : pas encore attribuée" au lieu de "Note : **note**" via la méthode **toString()**.

Créez les classes filles avec les caractéristiques suivantes :
class Livre_Audio extends Livre

- un attribut **privé String** nommé **narrateur**

class Livre_Illustre extends Livre

- un attribut **privé String** nommé **illustrateur**

Voici le squelette du programme à compléter :

```
1   class Livre {
2
3   }
4
5   class Livre_Audio extends Livre {
6
7   }
8
9   class Livre_Illustre extends Livre {
10
11  }
```

Conseil

En Java, lors de la déclaration d'une classe, le mot clé **extends** permet d'indiquer qu'il s'agit d'une classe fille de la classe indiquée.
Le mot clé **super** permet à la sous classe d'hériter d'éléments de la classe-mère. **super** peut être utilisé dans le constructeur de la classe-fill selon l'example suivant : **super(attribut_mère_1, attribut_mère_2, attribut_mère_3, etc.);**. Ainsi, il n'est pas nécessaire de redéfinir tous les attributs d'une classe fille !
L'instruction **super** doit toujours être la première instruction dans le constructeur d'une classe-fille.
Vous pouvez vous servir de **'\n'** dans une chaîne de caractères pour effectuer un retour à la ligne lors de l'affichage d'une chaîne de caractères.

Solution

```
1  public class Livre {
2
3     private String titre;
4     private String auteur;
5     private int annee;
6     private int note = -1;
7
8     public Livre(String titre, String auteur, int annee){
9     System.out.println("Création d'un livre");
10    this.titre = titre;
11    this.auteur = auteur;
12    this.annee = annee;
13 }
14
15 public int getNote(){
16    return this.note;
17 }
18
19 public void setNote(int note) {
20    this.note = note;
21 }
22
23 public String toString() {
24    if (note == -1){
25       return "A propos du livre \n---------- \nTitre : " +titre+ "\nAuteur : "+auteur+ "\nAnnée
          : "+annee+ "\nNote : non attribuée";
26    }
27    else{
28       return "A propos du livre \n---------- \nTitre : " +titre+ "\nAuteur : "+auteur+ "\nAnnée
          : "+annee+ "\nNote : "+note;
29    }
30 }
31 }
32
33 class Livre_Audio extends Livre {
34    private String narrateur;
35
36    public Livre_Audio(String titre, String auteur, int annee, String narrateur){
37    super(titre, auteur, annee);
38    System.out.println("Création d'un livre audio");
39    this.narrateur = narrateur;
40 }
41
42 }
43
44 class Livre_Illustre extends Livre {
45
46    private String illustrateur;
47
48    public Livre_Illustre(String titre, String auteur, int annee, String illustrateur) {
49    super(titre, auteur, annee);
50    System.out.println("Création d'un livre illustré");
51    this.illustrateur = illustrateur;
52 }
53
54 }
```

Question 10: (*5 minutes*) **Méthode et héritage**

Maintenant que vous avez créé la classe-mère et les classes filles correspondantes, vous pouvez créer un objet **Livre** à l'aide du constructeur de la classe **Livre_Audio** (et des arguments donnés lors de la création de l'objet).
En instanciant l'objet, vous pourriez utiliser les valeurs suivantes :
titre : "Hamlet", auteur : "Shakespeare", année : "1609" et le narrateur "William".

Une fois l'objet créé, attribuez-lui une note à l'aide de la méthode **setNote()** définie précédemment.

Finalement, utilisez la méthode **System.out.println()** pour afficher les informations du livre.

Redéfinir la méthode **toString()** de la classe **Livre_Audio** afin que la valeur de l'attribut **narrateur** soit affichée. Faites pareil avec la classe **Livre_Illustre** et son attribut **Illustrateur**

Conseil

Attention, vous devez créer un objet **Livre** et non **Livre_Audio**.
Le mot-clé **super** peut être utilisé dans la redéfinition d'une méthode selon l'exemple suivant : **super.nom_de_la_methode();**. Le mot clé **super** représente la classe parent, tout comme le mot clé **this** fait référence à l'instance avec laquelle la méthode est appelée.
L'instruction **super** doit toujours être la première instruction dans le redéfinition d'une méthode dans une classe fille.

Solution

```
1  class Livre_Audio extends Livre {
2
3     private String narrateur;
4
5     public Livre_Audio(String titre, String auteur, int annee, String narrateur){
6        super(titre, auteur, annee);
7        System.out.println("Création d'un livre audio");
8        this.narrateur = narrateur;
9  }
10
11    // redéfinition de la fonction toString dans la classe fille Livre_Audio
12    public String toString() {
13       return super.toString() + "\nNarrateur: "+ narrateur+"\n"; //Ajoute narrateur à la chaine de caractère
          crée par la classe mère (super)
14    }
15 }
16
17 class Livre_Illustre extends Livre {
18
19    private String illustrateur;
20
21    public Livre_Illustre(String titre, String auteur, int annee, String illustrateur) {
22    super(titre, auteur, annee);
23    System.out.println("Création d'un livre illustré");
24    this.illustrateur = illustrateur;
25 }
26    public String toString() {
27       return super.toString() + "\nIllustrateur: "+ illustrateur + "\n"; //Ajoute illustrateur à la chaine de
          caractère crée par la classe mère (super)
28    }
29 }
```

```
1  public class Main {
2
3     public static void main(String[] args) {
4     Livre Livre1 = new Livre_Audio("Hamlet", "Shakespeare", 1609,"William");
5     Livre1.setNote(5);
6     System.out.println(Livre1);
7  }
8
9  }
```

Lorsque toutes les étapes auront été effectuées, placez le code suivant dans votre **main** et exécutez votre programme :

```
public class Main {

    public static void main(String[] args) {
    Livre Livre1 = new Livre_Audio("Hamlet", "Shakespeare", 1609,"William");
    Livre1.setNote(5);
    System.out.println(Livre1);
    Livre Livre2 = new Livre("Les Misérables","Hugo",1862);
    System.out.println(Livre2);

}

}
```

Vous devriez obtenir :

```
Création d'un livre
Création d'un livre audio
Création d'un livre
A propos du livre
--------------------
Titre : Hamlet
Auteur : Shakespeare
Année : 1609
Note : 5
Narrateur: William
A propos du livre
--------------------
Titre : Les Misérables
Auteur : Hugo
Année : 1862
Note : non attribuée
Process finished with exit code 0
```

3 Polymorphisme - Java

Les exercices de cette section sont la suite des exercices de la section 2 de la semaine passée. N'hésitez pas à réutiliser les solutions disponibles sur Moodle.
Dans cette partie, vous devez créer 2 nouvelles classe-filles de la classe-mère **Fighter**. La première classe représentera un **Soigneur**, qui, lorsqu'il "attaquera" un **Fighter**, le soignera au lieu de le blesser. La deuxième classe représentera un Fighter spécialisé dans l'attaque **Attaquant**, qui aura la capacité d'attaquer un certain nombre de fois (ce nombre sera défini lors de l'instanciation).

Avant d'effectuer la série de questions suivantes, téléchargez le fichier **fighter–squelette.java** disponible sur Moodle.

Question 11: (*5 minutes*) classe-fille **Soigneur**
Commencez par déclarer une nouvelle classe-fille **Soigneur**. Cette classe-fille aura un nouvel attribut privé nommé **résurrection** qui vaudra 1 lors de l'instanciation de la classe.

Créez le constructeur de cette classe ainsi que les **getter** et **setter** permettant d'interagir avec le nouvel attribut (**résurrection**).

Conseil

Pensez à utiliser le constructeur de votre classe-mère **Fighter**

Solution

```
1  class Soigneur extends Fighter {
2
3      private int résurrection;
4
5      public Soigneur(String name, int health, int attack, int defense, int soin){
6          super(name,health,attack,defense);
7          résurrection = 1;
8      }
9
10     public int getRésurrection(){
11         return this.résurrection;
12     }
13
14     public void setRésurrection(int etat){
15         this.résurrection = etat;
16     }
17 }
```

Question 12: (*15 minutes*) Méthode **résurrection(Fighter other)** de la classe-fille **Soigneur**
Créez une méthode **résurrection(Fighter other)** qui permettra de faire revenir un **Fighter** à la vie, mais le **Soigneur** ne pourra le faire qu'une seule fois.

Commencez par contrôler que l'instance depuis laquelle la méthode est appelée soit toujours en vie. Si ce n'est pas le cas, indiquez : **nom_instance** est mort et ne peut plus rien faire.

Contrôlez ensuite que l'instance **other** soit vraiment morte. Si ce n'est pas le cas, indiquez le en affichant le texte suivant : "**nom_other** est toujours en vie."

Pour finir, contrôlez que l'attribut **résurrection** de l'instance depuis laquelle la méthode est appelée est égale à 1. Si ce n'est pas le cas, indiquez : **nom_instance** ne peut plus ressusciter personne.

Si tous ces éléments sont réunis, faites revenir le **Fighter other** à la vie en lui remettant 10 points de vie et en l'ajoutant à la liste **instances** de la classe **Fighter**. Pensez aussi à :

— Mettre l'attribut **résurrection** de l'instance appelée à 0 afin de l'empêcher de réutiliser ce pouvoir,

— appeler la méthode **checkHealth**(), et à indiquer : "**nom_other** est revenu à la vie !"

Conseil

Utilisez un branchement conditionnel pour les contrôles.

Une nouvelle méthode nommée **addInstances(Fighter other)** a été créée dans la classe **Fighter**. Regardez à quoi elle sert et utilisez la.

Pour les indications en fonction des différentes conditions, imprimez simplement la phrase en question.

Solution

```
1  public void résurrection(Fighter other){
2    if(!this.isAlive()) {
3      System.out.println(this.getName() + " est mort et ne peut plus rien faire");
4    }
5    else{
6      if (other.isAlive()) {
7        System.out.println(other.getName() + " est toujours en vie !");
8      } else {
9        if (this.getRésurrection() == 0) {
10         System.out.println(this.getName() + " ne peut plus ressuciter personne");
11       } else {
12         other.setHealth(10);
13         Fighter.addInstances(other);
14         this.setRésurrection(0);
15         System.out.println(other.getName() + " vient de revenir à la vie");
16         Fighter.checkHealth();
17       }
18     }
19   }
20 }
```

Question 13: (*10 minutes*) Méthode **attack** de la classe-fille **Soigneur**

Réécrivez la méthode **attack** de la classe-fille **Soigneur** afin d'ajouter des points de vie à **other** au lieu de lui en retirer.

Le seul argument nécessaire pour cette méthode sera une autre instance de **Fighter** qu'on pourrait nommer **other**.

Commencez par contrôler que le **Soigneur** depuis lequel la méthode est appelée est encore en vie. Si ce n'est pas le cas, affichez : "**nom_instance** est mort et ne peut plus rien faire."
Contrôlez ensuite que le **Fighter other** est toujours en vie. Si ce n'est pas le cas indiquez : "**nom_other** est déjà mort, ressuscitez-le afin de pouvoir le soigner." Contrôlez également qu'il ait moins de 10 points de vie. Si ce n'est pas le cas, indiquez le via le texte suivant : "**nom_other** a déjà le maximum de points de vie."

Si toutes ces conditions sont réunies, ajoutez la valeur de l'attaque de l'instance qui appelle la méthode aux points de vie de **other**, puis appelez la méthode de classe **checkHealth**().

Conseil

Pensez à utiliser un branchement conditionnel pour les contrôles.

Le nombre de points de vie à ajouter est simplement égal à l'attaque de l'instance depuis laquelle la méthode est appelée. Ajoutez la valeur de cet attribut **attack** au **Fighter other**.

Solution

```
public void attack(Fighter other) {
    if(!this.isAlive()) {
        System.out.println(this.getName() + " est mort et ne peut plus rien faire");
    }
    else{
        if (other.getHealth() >= 10) {
            System.out.println(other.getName() + " a déjà le maximum de points de vie");
        }
        if (!other.isAlive()) {
            System.out.println(other.getName() + " est déjà mort, ressuscitez–le pour pouvoir le soigner");
        } else {
            other.setHealth(other.getHealth() + this.getAttack());
            Fighter.checkHealth();
        }
    }
}
```

Question 14: (*5 minutes*) Classe-fille **Attaquant**

Commencez par déclarer une nouvelle classe-fille **Attaquant** héritant de la classe **Fighter**. Cette classe-fille prendra un nouvel attribut privé nommé **multiplicateur** de type **int**, qui sera passé comme argument du **constructeur** de la classe-fille.

Déclarez le **constructeur** de cette classe ainsi que les **getter** et **setter** permettant d'interagir avec ce nouvel attribut **multiplicateur**.

Conseil

Pensez à utiliser le **constructeur** de votre classe-mère **Fighter**.

Solution

```
class Attaquant extends Fighter{

    private int multiplicateur;

    public Attaquant(String name, int health, int attack, int defense, int multiplicateur){
    super(name,health,attack,defense);
    this.multiplicateur = multiplicateur;
    }

    public int getMultiplicateur() {
        return multiplicateur;
    }

    public void setMultiplicateur(int multiplicateur){
        this.multiplicateur = multiplicateur;
    }
}
```

Question 15: (*10 minutes*) Méthode **attack** de la classe-fille **Attaquant**

Réécrivez la méthode **attack** de la classe-fille **Attaquant** afin d'effectuer plusieurs attaques sur **other** en fonction de l'attribut **multiplicateur**.

Assurez-vous de toujours indiquer le type de l'attaque lorsque vous faites appel à la méthode **attack**().

Conseil

Pensez à contrôler que l'instance depuis laquelle la méthode est appelée est encore en vie. Aidez-vous de la méthode **attack** de la classe-mère **Fighter**.

Pour effectuer plusieurs attaques, vous pouvez utiliser une boucle allant de 0 à **multiplicateur**.

Solution

```
1  public void attack(String type, Fighter other) {
2      for (int i = 0; i < this.getMultiplicateur(); i++) {
3          System.out.println("Attaque n " + (i+1));
4          super.attack(type, other);
5      }
6  }
```

Si tout est correct, en utilisant ce main :

```
1  public class Main {
2      public static void main(String[] args) {
3      Fighter P1 = new Fighter("P1", 10, 2, 2);
4      Attaquant P2 = new Attaquant("P2", 10, 2, 2,2);
5      Soigneur P3 = new Soigneur("P3",10,4,2,4);
6      P1.attack("pied",P2);
7      P1.attack("poing",P2);
8      P1.attack("tete",P2);
9      P1.attack("tete",P2);
10     P3.résurrection(P2);
11     P1.attack("pied",P2);
12     P1.attack("poing",P2);
13     P1.attack("tete",P2);
14     P3.attack(P2);
15     P2.attack("tete",P1);
16 }
17 }
```

Vous devriez obtenir :

```
1  P1 a encore 10 points de vie
2  P2 a encore 8 points de vie
3  P3 a encore 10 points de vie
4  ------------------------------
5  P1 a encore 10 points de vie
6  P2 a encore 6 points de vie
7  P3 a encore 10 points de vie
8  ------------------------------
9  P1 a encore 10 points de vie
10 P2 a encore 2 points de vie
11 P3 a encore 10 points de vie
12 ------------------------------
13 P2 est mort
14 P1 a encore 10 points de vie
15 P3 a encore 10 points de vie
16 ------------------------------
17 P2 vient de revenir à la vie
18 P1 a encore 10 points de vie
19 P3 a encore 10 points de vie
20 P2 a encore 10 points de vie
21 ------------------------------
22 P1 a encore 10 points de vie
23 P3 a encore 10 points de vie
24 P2 a encore 8 points de vie
25 ------------------------------
26 P1 a encore 10 points de vie
27 P3 a encore 10 points de vie
28 P2 a encore 6 points de vie
```

```
--------------------------------
P1 a encore 10 points de vie
P3 a encore 10 points de vie
P2 a encore 2 points de vie
--------------------------------
P1 a encore 10 points de vie
P3 a encore 10 points de vie
P2 a encore 6 points de vie
--------------------------------
Attaque n 1
P1 a encore 6 points de vie
P3 a encore 10 points de vie
P2 a encore 6 points de vie
--------------------------------
Attaque n 2
P1 a encore 2 points de vie
P3 a encore 10 points de vie
P2 a encore 6 points de vie
--------------------------------
```