Java_keypoints_sol

December 17, 2019

1 Key concepts

1.1 Java Encapsulation

L'encapsulation sert à cacher les détails d'implémentation. L'encapsulation sert uniquement à montrer que les informations essentielles aux utilisateurs.

En Java, il est recommandé de déclarer les attributs des classes comme étant `private` et de mettre à disposition des utilisateurs des méthodes publiques d'accès afin qu'ils puissent accéder ou modifier la valeur des attributs privés.

Les méthodes publiques d'accès comme `getName`et `setName` doivent être nommées avec soit `get` soit `set` suivi du nom de l'attribut avec la 1ère lettre en majuscule (Java Naming convention)[https://www.oracle.com/technetwork/java/codeconventions-135099.html]).

Le mot clé `this` fait référence à l'objet en question.

1.1.1 Exercice

Complètez les deux cellules de code ci-dessous en vous aidant des commentaires.

[1]:
```
// déclarez une classe publique "Person"
public class Person {
    //déclarez une variable privée "name"
    private String name;

  // déclarez une méthode publique "getName"
    public String getName() {
        return name;
    }

  // déclarez une méthode publique void "setName"
    public void setName(String newName) {
        this.name = newName;
    }
}
```

[1]:
```
com.twosigma.beaker.javash.bkr326710a7.Person
```

[24]:
```
// déclarez une classe publique "MyClass"
public class MyClass {
    //déclarez une méthode publique statique "main"
   public static void main(String[] args) {
        // créez un nouvel objet "Person" nommé "myObj"
        Person myObj = new Person();
        // modifiez sa valeur de "name" avec la méthode "setName"
        myObj.setName("John");
        // affichez sa valeur avec la méthode "getName"
        System.out.println(myObj.getName());
   }
}

// Outputs "John"
```

[24]:
```
com.twosigma.beaker.javash.bkr326710a7.MyClass
```

[25]:
```
MyClass.main(null);
```

```
John
```

[25]:
```
null
```

1.2 Héritage en Java (sous-classe et super-classe)

Comme vous le savez, il est possible que des sous-classes héritent des attributs ou méthodes des super-classes. En Java, il faut utiliser le mot-clé `extends`. Ainsi, l'héritage permet la réutilisation des attributs et méthodes d'une classe existante.

1.2.1 Exercise

Complètez la cellule de code ci-dessous en vous aidant des commentaires. La classe `Car` est une sous-classe alors que la classe `Vehicle` est une super-classe.

[22]:
```
// déclarez une classe "Vehicle"
class Vehicle {
    //déclarez une variable protegée "brand" égale à "Ford"
    protected String brand = "Ford";
    //déclarez une méthode publique void "honk"
    public void honk() {
    // affichez "Tuut, tuut!"
        System.out.println("Tuut, tuut!");
  }
}

// déclarez une classe "Car" qui hérite de la classe "Vehicle"
class Car extends Vehicle {
    //déclarez une variable privée "modelName" égale à "Mustang"
```

```
    public String modelName = "Mustang";

    //déclarez une méthode publique statique "main"
    public static void main(String[] args) {

        // créez un nouvel objet "Car" nommé "myCar"
        Car myCar = new Car();

        //appelez la méthode "honk" de la classe "Vehicle" sur "myCar"
        myCar.honk();

        // affichez la "brand" et "modelName" de myCar
        System.out.println(myCar.brand + " " + myCar.modelName);
    }
}
```

[22]: com.twosigma.beaker.javash.bkr326710a7.Vehicle

[23]:
```
Car.main(null);
```

```
Tuut, tuut!
Ford Mustang
```

[23]: null

[19]:
```
//déclarez une méthode publique statique "main"
class Main{
    public static void main(String[] args) {

        // créez un nouvel objet "Car" nommé "myCar"
        Car myCar = new Car();

        //appelez la méthode "honk" de la classe "Vehicle" sur "myCar"
        myCar.honk();

        // affichez la "brand" et "modelName" de myCar
        System.out.println(myCar.brand + " " + myCar.modelName);
    }
}
```

[19]: com.twosigma.beaker.javash.bkr326710a7.Main

[20]:
```
Main.main(null);
```

```
Tuut, tuut!
Ford Mustang
```

[20]: `null`

1.3 Java Polymorphisme

Le polymorphisme apparaît quand plusieurs classes sont liées par héritage.

1.3.1 Exercise

Complètez la cellule de code ci-dessous en vous aidant des commentaires. Les sous-classes d'`Animal` comme `Pig, Cat, Dog ou Bird` de la super-classe `Animal` redéfinissent la méthode `animalSound`.

[26]:
```
// déclarez une super-classe "Animal"
class Animal {
    //déclarez une méthode publique void "animalSound"
    public void animalSound() {
        // affichez "The animal makes a sound"
        System.out.println("The animal makes a sound");
    }
}
// déclarez une classe "Pig" qui hérite de "Animal"
class Pig extends Animal {
    //déclarez une méthode publique void "animalSound" qui redéfinie celle de␣
 ↪la super-classe "Animal"
    public void animalSound() {
        // affichez "The pig says: wee wee"
        System.out.println("The pig says: wee wee");
    }
}
// déclarez une classe "Dog" qui hérite de "Animal"
class Dog extends Animal {
    //déclarez une méthode publique void "animalSound" qui redéfinie celle de␣
 ↪la super-classe "Animal"
    public void animalSound() {
        // affichez "The dog says: bow wow"
        System.out.println("The dog says: bow wow");
    }
}
```

[26]: `com.twosigma.beaker.javash.bkr326710a7.Animal`

[27]:
```
// déclarez une classe "MyMainClass"
class MyMainClass {
    //déclarez une méthode publique statique "main"
    public static void main(String[] args) {
        // créez un nouvel objet "Animal" nommé "myAnimal"
        Animal myAnimal = new Animal();
        // créez un nouvel objet "Pig" nommé "myPig"
        Animal myPig = new Pig();
```

```
        // créez un nouvel objet "Dog" nommé "myDog"
        Animal myDog = new Dog();
        //appelez la méthode "animalSound" avec "myAnimal"
        myAnimal.animalSound();
        //appelez la méthode "animalSound" avec " myPig"
        myPig.animalSound();
        //appelez la méthode "animalSound" avec " myDog"
        myDog.animalSound();
    }
}
```

[27]: com.twosigma.beaker.javash.bkr326710a7.MyMainClass

[28]:
```
MyMainClass.main(null);
```

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

[28]: null

1.4 Java Classes et Methodes Abstraites

En Java, il suffit de déclarer des classes ou interfaces abstraites pour utiliser le concept d'abstraction. Les classes abstraites ne peuvent pas créer des objets sauf si elles héritent d'autres classes.

Les méthodes abstraites, ayant généralement pas de contenu car il est hérité de la sous-classe, peuvent uniquement être utilisées dans les classes abstraites.

[29]:
```
// déclarez une classe abstraite "Animal"
abstract class Animal {
    //déclarez une méthode  vide publique abstraite void "animalSound"
    public abstract void animalSound();
    // //déclarez une méthode publique void "sleep"
    public void sleep() {
        // affichez "Zzz"
        System.out.println("Zzz");
    }
}

// déclarez une classe "Pig" qui hérite de "Animal"
class Pig extends Animal {
    //déclarez une méthode publique void "animalSound" qui redéfinie celle de␣
 ↪la super-classe "Animal"
    public void animalSound() {
        // affichez "The pig says: wee wee"
        System.out.println("The pig says: wee wee");
    }
```

```
}
// déclarez une classe "MyMainClass"
class MyMainClass {
    //déclarez une méthode publique statique "main"
    public static void main(String[] args) {
        // créez un nouvel objet "Pig" nommé "myPig"
        Pig myPig = new Pig();
        //appelez la méthode "animalSound" avec " myPig"
        myPig.animalSound();
        //appelez la méthode "sleep" avec " myPig"
        myPig.sleep();
   }
}
```

[29]:
```
com.twosigma.beaker.javash.bkr326710a7.Animal
```

[30]:
```
MyMainClass.main(null);
```

```
The pig says: wee wee
Zzz
```

[30]:
```
null
```

[ ]:

abstract_class_and_interface_answers

December 17, 2019

1 Abstract Classes and Interfaces in Java

1.1 Abstract Classes [1]

Une classe abstraite est une classe dont l'implémentation n'est pas complète. Elle est déclarée en utilisant le mot-clé **abstract**. Elle peut inclure des méthodes abstraites ou non. Les classes abstraites ne peuvent pas être instanciées, mais servent de base à des sous-classes qui en sont dérivées.

Lorsqu'une sous-classe est dérivée d'une classe abstraite, elle complète généralement l'implémentation de toutes les méthodes abstraites de la classe mère. Si ce n'est pas le cas, la sous-classe doit également être déclarée comme abstraite.

```
// an abstract class declaration
public abstract class Animal {
    private int speed;
    // an abstract method declaration
    abstract void run();
}

public class Cat extends Animal {
    // implementation of abstract method
    void run() {
        speed += 10;
    }
}
```

[1]https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Implémentez une classe abstraite appelée `Item`.

1. Elle doit avoir 4 variables d'instance et une variable de classe, qui sont les suivantes:

   ```
   private int id;
   private static int count = 0;
   private String name;
   private double price;
   private ArrayList<String> ingredients;
   ```

2. Elle doit avoir un constructeur prenant les variables `name`,`price` et `ingredients` comme paramètres. Pour définir `id`, utilisez la ligne suivante: `this.id = ++count;`

3. Implémentez des méthodes d'accesseurs pour les variables `id`, `name`, `price` et `ingredient`.
4. Implémentez les méthodes `equals(Object o)` et `toString()`.

[1]:
```java
import java.util.*;

// VOTRE CODE ICI

public abstract class Item {

    private int id;
    private static int count = 0;
    private String name;
    private double price;
    private ArrayList<String> ingredients;

    public Item (String name, double price, ArrayList<String> ingredients) {
        this.id = ++count;
        this.name = name;
        this.price = price;
        this.ingredients = ingredients;
    }

    public int getID() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public double getPrice() {
        return this.price;
    }

    public ArrayList<String> getIngredients() {
        return this.ingredients;
    }

    public boolean equals(Object o) {
        if (o instanceof Item) {
            Item i = (Item) o;
            return i.getID() == this.getID();
        }
        return false;
    }

    public String toString() {
```

```
        return "*-*-*-*-*-*" +
            "\nID: " + this.getID() +
            "\nName: " + this.getName() +
            "\nPrice: " + this.getPrice() + " CHF" +
            "\nList of ingredients: " + this.getIngredients().toString() +
            "\n*-*-*-*-*-*";
    }
}
```

[1]: com.twosigma.beaker.javash.bkrd197fd95.Item

1.2 Interfaces [2]

Une déclaration d'interface comprend les modificateurs (public, etc.), le mot-clé **interface**, le nom de l'interface, une liste d'interfaces parentes séparées par des virgules et le corps de l'interface.

```
public interface IMakeSound {
    final double MY_DECIBEL_VALUE = 75;
    void makeSound();
}
```

Les méthodes déclarées dans une interface doivent être implémentées dans des sous-classes. Si on reprend l'exemple de la classe `Cat`:

```
public class Cat extends Animal implements IMakeSound {
    // implementation of the abstract method from the class Animal
    void run() {
        speed += 10;
    }
    // implementation of the method from the IMakeSound interface
    void makeSound() {
        System.out.println("I meow at " + MY_DECIBEL_VALUE + " decibel.");
    }
}
```

[2]https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html

Implémentez une interface `Edible` contenant une méthode appelée `eatMe` qui ne retourne aucune valeur.

[2]:
```
// VOTRE CODE ICI

public interface Edible {
    void eatMe();
}
```

[2]: com.twosigma.beaker.javash.bkrd197fd95.Edible

Implémentez une interface `Drinkable` contenant une méthode appelée `drinkMe` qui ne retourne aucune valeur.

[3]:
```java
// VOTRE CODE ICI

public interface Drinkable {
    void drinkMe();
}
```

[3]: com.twosigma.beaker.javash.bkrd197fd95.Drinkable

Implémentez une classe `Food` qui **extends** `Item` et **implements** `Edible`. Ensuite, implémentez à la fois un constructeur pour `Food` ainsi que la méthode `eatMe` de l'interface `Edible`.

Indice : Vous pouvez simplement mettre un `println` dans le corps de la méthode `eatMe()`.

[4]:
```java
import java.util.*;

// VOTRE CODE ICI

public class Food extends Item implements Edible {

    public Food(String name, double price, ArrayList<String> ingredients) {
        super(name, price, ingredients);
    }

    public void eatMe() {
        System.out.println("Eat me!\n" + toString());
    }
}
```

[4]: com.twosigma.beaker.javash.bkrd197fd95.Food

Implémentez une classe `Drink` qui **extends** `Item` et **implements** `Drinkable`. Ensuite, implémentez à la fois un constructeur pour `Drink` ainsi que la méthode `drinkMe` de l'interface `Drinkable`.

Indice : Vous pouvez simplement mettre un `println` dans le corps de la méthode `drinkMe()`.

[5]:
```java
import java.util.*;

// VOTRE CODE ICI

public class Drink extends Item implements Drinkable {

    public Drink(String name, double price, ArrayList<String> ingredients) {
        super(name, price, ingredients);
    }

    public void drinkMe() {
        System.out.println("Drink me!\n" + toString());
    }
}
```

[5]: `com.twosigma.beaker.javash.bkrd197fd95.Drink`

Certains aliments ne sont pas seulement `Edible`, mais aussi `Drinkable`, comme les soupes.

Implémentez une classe `Soup` qui **extends** `Food` et **implements** `Drinkable`. Ensuite, implémentez à la fois un constructeur pour `Soup` ainsi que la méthode `drinkMe` de l'interface `Drinkable`.

Indice : Vous pouvez simplement mettre un `println` dans le corps de la méthode `drinkMe()`.

[6]:
```java
import java.util.*;

// VOTRE CODE ICI

public class Soup extends Food implements Drinkable {

    public Soup(String name, double price, ArrayList<String> ingredients) {
        super(name, price, ingredients);
    }

    public void drinkMe() {
        System.out.println("Drink the soup!\n" + toString());
    }
}
```

[6]: `com.twosigma.beaker.javash.bkrd197fd95.Soup`

Créez les 4 instances décrites ci-dessous; testez et observez leur fonctionnement:

1. Créez une instance `Soup`, puis appelez les méthodes `drinkMe()` et `eatMe()`.

   Vous pouvez utiliser la ligne suivante pour initialiser votre instance :
   `Soup("Analı kızlı soup", 7.5, new ArrayList<String>(Arrays.asList("chickpeas","bulgur", "meat", "tomato" )))`

2. Créez une instance `Food`, puis appelez la méthode `eatMe()`.

   Vous pouvez utiliser la ligne suivante pour initialiser votre instance : `Food("Stuffed peppers", 12, new ArrayList<String>(Arrays.asList("bell pepper", "rice", "tomato", "parsley", "onion")))`

3. Créez une instance `Drink`, puis appelez la méthode `drinkMe()`.

   Vous pouvez utiliser la ligne suivante pour initialiser votre instance : `Drink("Ayran", 3, new ArrayList<String>(Arrays.asList("yoghurt", "water", "mint")))`

4. Créez une instance `Food`, puis appelez la méthode `eatMe()`.

   Vous pouvez utiliser la ligne suivante pour initialiser votre instance : `Food("Cream of mushroom soup", 6, new ArrayList<String>(Arrays.asList("mushroom", "creme", "flour")))`

   Essayez d'appeler la méthode `drinkMe()` sur cette instance. Est-ce que ça marche ?

[7]:
```
import java.util.*;

// VOTRE CODE ICI

Soup s1 = new Soup("Analı kızlı soup", 7.5, new ArrayList<String>(Arrays.
 ↪asList("chickpeas","bulgur", "meat", "tomato" )));
s1.drinkMe();
s1.eatMe();

Food f = new Food("Stuffed peppers", 12, new ArrayList<String>(Arrays.
 ↪asList("bell pepper", "rice", "tomato", "parsley", "onion")));
f.eatMe();

Drink d = new Drink("Ayran", 3, new ArrayList<String>(Arrays.asList("yoghurt",␣
 ↪"water", "mint")));
d.drinkMe();

Food s2 = new Food("Cream of mushroom soup", 6, new ArrayList<String>(Arrays.
 ↪asList("mushroom", "creme", "flour")));
s2.eatMe();
// we cannot call this method
// s2.drinkMe();
```

```
Drink the soup!
*-*-*-*-*-*
ID: 1
Name: Analı kızlı soup
Price: 7.5 CHF
List of ingredients: [chickpeas, bulgur, meat, tomato]
*-*-*-*-*-*
Eat me!
*-*-*-*-*-*
ID: 1
Name: Analı kızlı soup
Price: 7.5 CHF
List of ingredients: [chickpeas, bulgur, meat, tomato]
*-*-*-*-*-*
Eat me!
*-*-*-*-*-*
ID: 2
Name: Stuffed peppers
Price: 12.0 CHF
List of ingredients: [bell pepper, rice, tomato, parsley, onion]
*-*-*-*-*-*
Drink me!
*-*-*-*-*-*
ID: 3
```

```
Name: Ayran
Price: 3.0 CHF
List of ingredients: [yoghurt, water, mint]
*-*-*-*-*-*
Eat me!
*-*-*-*-*-*
ID: 4
Name: Cream of mushroom soup
Price: 6.0 CHF
List of ingredients: [mushroom, creme, flour]
*-*-*-*-*-*
```

[7]: null

[0]:

[Solution] - classes_abstraites

December 17, 2019

1 Classes abstraites

Avant de commencer cette séance d'exercices, assurez-vous d'avoir bien compris la notion d'héritage.

Une classe abstraite est une classe qui ne peut être instanciée. Elle peut contenir une ou plusieurs méthodes abstraites.

L'abstraction sert uniquement à montrer que les informations essentielles aux utilisateurs. En Java, il suffit de déclarer des classes ou interfaces abstraites pour utiliser le concept d'abstraction. Les classes abstraites ne peuvent pas créer des objets sauf si elles héritent d'autres classes. source: https://www.w3schools.com/java/java_abstract.asp

Dans cet exercice, vous devez définir une classe abstraite `Polygone` contenant deux méthodes abstraites: - calculSurface(): effectue l'opération et retourne la surface du polygone - display(): Pour chaque polygone définit, cette methode affichera un message contenant tous les attributs de l'objet ainsi que le résultat de la méthode `calculSurface()`. Par exemple, dans le cas d'un rectangle (qui hérite de `Polygone`), `display()` retournera le message suivant: `Notre rectangle a pour dimensions L = 60.0 et l = 25.0. Sa superficie est de 1500.0 mètres carrés`

Créez des classes héritant de `Polygone` pour chaque type de figures listées dans l'image ci-dessous.

Classe abstraite et héritage

[5]:
```
// Définissez votre classe abstraite Polygone
abstract class Polygon {

    abstract public double calculateArea();
    abstract public String display();

}
```

[2]:
```
// Définissez les figures héritants de Polygone (recta)
// Rajoutez des attributs (par exemple: Longueur et largeur pour le rectangle)
// Créez des getters et setters pour accéder et modifier les valeurs de vos␣
 ↪attributs

public class Rectangle extends Polygon {

    private double longueur;
    private double largeur;
```

```
    public double getLongueur() {
        return longueur;
    }

    public void setLongueur(double longueur) {
        this.longueur = longueur;
    }

    public double getLargeur() {
        return largeur;
    }

    public void setLargeur(double largeur) {
        this.largeur = largeur;
    }

    public double calculateArea() {
        return this.longueur * this.largeur;
    }

    public String display() {
        return "Notre rectangle a pour dimensions L = "+this.longueur+" et l =␣
 ↪"+this.largeur+". Sa superficie est de "+calculateArea()+" mètres carrés";
    }

}
```

[3]:
```
// Instanciez vos objets et faire appel aux méthodes définies dans vos objets␣
 ↪(calculSurface() et display())
Rectangle r1 = new Rectangle();
r1.setLargeur(25);
r1.setLongueur(60);
System.out.println(r1.display());
```

[ ]:
```

```

RoleGame_sol

December 17, 2019

0.1 Exercise 2: Role Play

0.1.1 Les personnages

Cette semaine nous allons commencer un exercice consistant à implémenter un jeu de rôle avec des personnages.

Tout d'abord nous allons commencer par implémenter les classes de bases de nos personnages. Les différents personnages possibles dans notre jeu sont : le Guerrier (« Warrior »), le Paladin (« Paladin »), le magicien (« Magician ») et le chasseur (« Hunter »). Vous avez appris le concept d'héritage, dans notre situation celui-ci s'avère très utile car ces différentes classes de personnages posssèdent certains attributs ou actions similaires.

Ainsi il semble intéressant de construire une première classe "Character" (Personnage). Un personnage est un objet qui possède plusieurs arguments :

```
name (String) : le nom du personnage
level (int) : le niveau du personnage
pv (int) : les points de vie du personnage
vitality (int) : la vitaité du personnage
strength (int) : la force du personnage
dexterity (int) : la dextérité du personnage
endurance (int) : l'endurance du personnage
intelligence (int) : l'intelligence du personnage
```

• Ces attributs vont nous permmettre de complexifier les différentes actions du jeu. Pour initialiser notre objet, il est important de définir son constructeur. Implémenter le constructeur de la classe « Character » qui prend tous ces attributs en argument.

• Java utilise le principe d'encapsulation qui permet de contrôler l'accessibilité des méthodes, des attributs et des classes. Ainsi, les attributs de vos classes doivent être déclarés « private » pour éviter que n'importe qui puisse avoir accès ou modifier ces attributs. Cependant il est essentiel de pouvoir avoir accès à ces valeurs ou de devoir les modifier. Dans la classe « Character » implémenter les getters et les setters de ces attributs.

Ex : public String getName(){ return name //ou this.name }

• Enfin il peut être intéressant d'afficher les caractéristiques de votre personnage. Implémenter une méthode « getInfo » dans la classe « Character » qui affiche dans la console.

[1]:
```
import java.util.Random;
```

```java
public abstract class Character {

    private String name;
    private int level;
    private int pv;

    private int vitality;
    private int strength;
    private int dexterity;
    private int endurance;
    private int intelligence;

    private int posX;
    private int posY;


    public Character(String name, int level, int pv, int vitality, int␣
↪strength, int dexterity, int endurance, int intelligence, int posX, int␣
↪posY){
        this.name = name;
        this.level = level;
        this.pv = pv;

        this.vitality = vitality;
        this.strength = strength;
        this.dexterity = dexterity;
        this.endurance = endurance;
        this.intelligence = intelligence;

        this.posX = posX;
        this.posY = posY;



    }

    public void getInfo() {
        System.out.println("Name" + getName());
        System.out.println("Level" + getLevel());
        System.out.println("Vitality" + getVitality());
        System.out.println("Strength" + getStrength());
        System.out.println("Dexterity" + getDexterity());
        System.out.println("Endurance" + getEndurance());
        System.out.println("Intelligence" + getIntelligence());
```

```
    }


    public void act(Character other){
        basicAttack(other);
    }

    public boolean isInGame(){

        return pv > 0;
    }


    public abstract void basicAttack(Character other);
    public abstract void specialAttack(Character other);

    public String getName() {
        return name;
    }

    public int getIntelligence() {
        return intelligence;
    }

    public int getLevel() {
        return level;
    }

    public int getPv() {
        return pv;
    }

    public int getVitality() {
        return vitality;
    }

    public int getStrength() {
        return strength;
    }

    public int getDexterity() {
        return dexterity;
    }

    public int getEndurance() {
        return endurance;
    }
```

```
    public int getPosX() {
        return posX;
    }

    public int getPosY() {
        return posY;
    }

    public void setPv(int pv) {
        this.pv = pv > 0 ? pv : 0 ;
    }
}
```

[1]: com.twosigma.beaker.javash.bkrf8293482.Character

Maintenant, implémenter les classes Warrior, Paladin, Magician et Hunter qui héritent de Personnage en écrivant tout d'abord leurs constructeurs respectifs. Pour chacun des personnages, implémenter une méthode « basicAttack » qui prend un autre personnage en argument et ne retourne rien. Celle-ci crée une attaque de votre choix en fonction des caractéristiques des personnages (ex : l'attaque du guerrier dépendra de sa force, l'attaque du chasseur de son endurance etc..). Affichez le nom de celui que vous avez attaqué et ses points de vie restants.

Cette méthode est commune à toutes les sous-classes, doit être déclarée abstraite dans la classe parente "Character". Changer la classe character pour qu'elle soit maintenant abstraite avec une méthode abstraite basicAttack():
Indication : utiliser le setteur pour réduire les pv de l'autre personnage.

[9]:
```
public class Warrior extends Character {


    private Axe axe;

    public Warrior(String name, int level, int pv,  int vitality, int strength,␣
↪int dexterity, int endurance, int intelligence, int posX, int posY, Axe axe)␣
↪{
        super(name,  level, pv, vitality,  strength,  dexterity,  endurance, ␣
↪intelligence, posX, posY);
        this.axe = axe;


    }


    public void basicAttack(Character other){
        int attack = (int) (getStrength()*0.5);
```

```
        other.setPv(other.getVitality()-attack);

        System.out.println(this.getName() + " attack " + other.getName());
        System.out.println("Basic attack og : " + attack);
        System.out.println(this.getName() + " PV:" + this.getPv());
        System.out.println(other.getName() + " PV:" + this.getPv());

    }


    public void specialAttack(Character other) {
        int attack = (int) (getStrength()*0.5 + 0.5*axe.
↪specialWeaponDamage(this));
        other.setPv(other.getVitality()-attack);

        System.out.println(this.getName() + " attack " + other.getName());
        System.out.println("Basic attack og : " + attack);
        System.out.println(this.getName() + " PV:" + this.getPv());
        System.out.println(other.getName() + " PV:" + this.getPv());
    }

}
```

[9]: com.twosigma.beaker.javash.bkrf8293482.Warrior

[10]:
```
public class Magician extends Character {

    private MagicStick stick;

    public Magician(String name, int level, int pv, int vitality, int strength,␣
↪int dexterity, int endurance, int intelligence, int posX, int posY,␣
↪MagicStick stick) {
        super(name,  level, pv, vitality,  strength,  dexterity,  endurance, ␣
↪intelligence, posX, posY);
        this.stick = stick;
    }
    @Override
    public void basicAttack(Character other){
        int attack = (int) (getIntelligence()*0.5);

        other.setPv(other.getVitality()-attack);

        System.out.println(this.getName() + " attack " + other.getName());
        System.out.println("Basic attack og : " + attack);
        System.out.println(this.getName() + " PV:" + this.getPv());
        System.out.println(other.getName() + " PV:" + this.getPv());
```

```
    }

    @Override
    public void specialAttack(Character other) {
        int attack = (int) (getIntelligence()*0.5 + 0.5*stick.
↪specialWeaponDamage(this));
        other.setPv(other.getVitality()-attack);

        System.out.println(this.getName() + " attack " + other.getName());
        System.out.println("Basic attack og : " + attack);
        System.out.println(this.getName() + " PV:" + this.getPv());
        System.out.println(other.getName() + " PV:" + this.getPv());

    }

}
```

[10]: com.twosigma.beaker.javash.bkrf8293482.Magician

[11]:
```
public class Paladin extends Character {

    private Sword sword;

    public Paladin(String name, int level, int pv, int vitality, int strength,␣
↪int dexterity, int endurance, int intelligence, int posX, int posY, Sword␣
↪sword) {
        super(name, level, pv, vitality, strength, dexterity, endurance, ␣
↪intelligence, posX, posY);
        this.sword = sword;
    }


    @Override
    public void basicAttack(Character other){
        int attack = (int) (getEndurance()*0.7);

        other.setPv(other.getVitality()-attack);

        System.out.println(this.getName() + " attack " + other.getName());
        System.out.println("Basic attack og : " + attack);
        System.out.println(this.getName() + " PV:" + this.getPv());
        System.out.println(other.getName() + " PV:" + this.getPv());

    }

    @Override
    public void specialAttack(Character other) {
```

```
        int attack = (int) (getEndurance()*0.5 + 0.5*sword.
↪specialWeaponDamage(this));
        other.setPv(other.getVitality()-attack);

        System.out.println(this.getName() + " attack " + other.getName());
        System.out.println("Basic attack og : " + attack);
        System.out.println(this.getName() + " PV:" + this.getPv());
        System.out.println(other.getName() + " PV:" + this.getPv());

    }

}
```

[11]: com.twosigma.beaker.javash.bkrf8293482.Paladin

[12]:
```
public class Hunter extends Character {

    private Bow bow;

    public Hunter(String name, int level, int pv, int vitality, int strength,␣
↪int dexterity, int endurance, int intelligence, int posX, int posY, Bow bow)␣
↪{
        super(name, level, pv, vitality, strength, dexterity, endurance, ␣
↪intelligence, posX, posY);
        this.bow = bow;
    }

    @Override
    public void basicAttack(Character other){
        int attack = (int) (getDexterity()*0.5);

        other.setPv(other.getVitality()-attack);

        System.out.println(this.getName() + " attack " + other.getName());
        System.out.println("Basic attack og : " + attack);
        System.out.println(this.getName() + " PV:" + this.getPv());
        System.out.println(other.getName() + " PV:" + this.getPv());

    }

    @Override
    public void specialAttack(Character other) {

        int attack = (int) (getDexterity()*0.5 + 0.5*bow.
↪specialWeaponDamage(this));
        other.setPv(other.getVitality()-attack);
```

```
        System.out.println(this.getName() + " attack " + other.getName());
        System.out.println("Basic attack og : " + attack);
        System.out.println(this.getName() + " PV:" + this.getPv());
        System.out.println(other.getName() + " PV:" + this.getPv());
    }

}
```

[12]: com.twosigma.beaker.javash.bkrf8293482.Hunter

0.1.2 Les armes

Chaque type de personnage possède également une arme. Le chasseur un arc ("Bow"), le magicien un baton ("MagicStick"), le paladin une épée ("Sword") et le guerrier une hache ("Axe"). Implémentez d'abord une classe parente "Weapon" ayant comme attributs:

```
- damage (int): le dommage causé par l'arme
- robustness (int): état de détèrioration de l'arme
- levelPower (int): niveau necessaire pour pouvoir utiliser le pouvoir de l'arme
```

Implémentez le constructeur de Weapon et les getters de ces attributs.

Rendre la classe Weapon abstraite et définir la méthode abstraite specialWeaponDamage(Character owner). Cette méthode donne au détenteur de l'arme, la capacité d'utiliser son attaque spéciale. Cependant le personnage ne pourra l'utiliser qu'à partir d'un certain niveau. Ajoutez un attribut "levelPower", initilalisez le dans le constructeur.

Enfin implémentez les 4 sous-classes de Weapon. Chacune d'entre elle devra implémenter la méthode specialWeaponDamage(). Cette méthode vérifie si le niveau du propriétaire est suffisant et si l'état de l'arme l'est aussi. Dans ce cas, l'arme perd en robustesse (5: axe, 2: sword, 1: magicStick, 8: bow)

[4]:
```
public abstract class Weapon {

    private int damage;
    private int robustness;
    private int levelPower;

    public Weapon(int damage, int robustness, int levelPower){
        this.damage = damage;
        this.robustness = robustness;
        this.levelPower = levelPower;

    }

    public int getLevelPower() {
        return levelPower;
    }
```

```
    public int getDamage() {
        return damage;
    }

    public int getRobustness() {
        return robustness;
    }

    public void reduceRobustness(int value){
        this.robustness -= value;
    }

    public abstract int specialWeaponDamage( Character owner);



}
```

[4]: com.twosigma.beaker.javash.bkrf8293482.Weapon

[20]:
```
public class Axe extends Weapon {

    public Axe(int damage, int robustness, int levelPower){
        super(damage, robustness, levelPower);

    }
    /**
     * Confer a special dammage attack to the owner if it
     * @param owner the weapon owner
     * @return  the dammage caused
     */
    public int specialWeaponDamage(Character owner){
        if (getRobustness() > 0 && (owner.getLevel() > getLevelPower())){
            reduceRobustness(5);
            return getDamage();
        } else {
            System.out.println("Ton hache est trop abimé ou ton niveau est trop␣
 ↪bas");
            return 0;
        }
    }
}
```

[20]: com.twosigma.beaker.javash.bkrf8293482.Axe

[21]:
```
public class Bow extends Weapon{

```

```
    public Bow(int damage, int robustness, int levelPower){
        super(damage, robustness, levelPower);

    }

    /**
     * Confer a special dammage attack to the owner
     * @param owner the weapon owner
     * @return the dammage caused
     */
    public int specialWeaponDamage(Character owner){
        if (getRobustness() > 0 && (owner.getLevel() > getLevelPower())){
            reduceRobustness(8);
            return getDamage();
        } else {
            System.out.println("Ton arc est trop abimé ou ton niveau est trop␣
 ↪bas");
            return 0;
        }
    }
}
```

[21]: com.twosigma.beaker.javash.bkrf8293482.Bow

[22]:
```
public class MagicStick extends Weapon{

    public MagicStick(int damage, int robustness, int levelPower){
        super(damage, robustness, levelPower);

    }
    /**
     * Confer a special dammage attack to the owner
     * @param owner the weapon owner
     * @return  the dammage caused
     */
    public int specialWeaponDamage(Character owner){
        if (getRobustness() > 0 && (owner.getLevel() > getLevelPower())){
            reduceRobustness(1);
            return getDamage();
        } else {
            System.out.println("Ton baton est trop abimé ou ton niveau est trop␣
 ↪bas");
            return 0;
        }
    }
}
```

[22]: com.twosigma.beaker.javash.bkrf8293482.MagicStick

[23]:
```
public class Sword extends Weapon{

    public Sword(int damage, int robustness, int levelPower){
        super(damage, robustness, levelPower);

    }

    /**
     * Confer a special dammage attack to the owner
     * @param owner the weapon owner
     * @return  the dammage caused
     */
    public int specialWeaponDamage(Character owner){
        if (getRobustness() > 0 && (owner.getLevel() > getLevelPower())){
            reduceRobustness(2);
            return getDamage();
        } else {
            System.out.println("Ton épée est trop abimé ou ton niveau est trop␣
 ↪bas");
            return 0;
        }
    }
}
```

[23]: com.twosigma.beaker.javash.bkrf8293482.Sword

0.1.3 Incorporation de l'arme au personnage

Ajouter un attribut dans chacunes des classes de personnages avec leur arme respectif (warrior -> axe, hunter -> bow etc...). Initialiser l'arme grace au constructeur prenant l'arme en argument.

Maintenant nous pouvons implémenter une méthode abstraite de Weapon, specialWeaponAttack(Character other). Cette méthode calcule la valeur des dommages d'une attaque en fonction de la compétence principale du personnage et de l'attaque spéciale de l'arme en fonction des sous-classes.

0.1.4 Retour sur character

Nous allons rajouter des fonctionnalités aux personnages:

- le personnage va aussi avoir la possibilité de bouger donc on doit lui assisgner une positio

- implémenter une méthode act(), qui appelle soit la basicAttack, soit la specialAttack.
- une méthode isInGame() qui return True si le nombre de pv est positif.

Grace au programme suivant vous avez à disposition une liste de personnages. Tester vos fonctions grâce à cette liste.

[24]:
```
import java.util.Random;
import java.util.ArrayList;

ArrayList<Character> players = new ArrayList<>();
for (int i = 0; i < 10; i++){
    Random r = new Random();
    int posX = r.nextInt(900 ) - 1;
    int posY = r.nextInt(900 ) - 1;
    int characterType =  r.nextInt(3);

    switch (characterType){
        //Hunter
        case 0:
            Bow bow = new Bow(50, 100, 1);
            players.add( new Hunter("Legolas", 1, 100, 50, 80, 50, 70, 70,␣
 ↪posX, posY, bow));
            break;
        case 1:
            MagicStick stick = new MagicStick(50, 100, 1);
            players.add( new Magician("Gandalf", 1, 100, 50, 80, 50, 70, 30,␣
 ↪posX, posY, stick));
            break;
        case 2:
            Sword sword = new Sword(50, 100, 1);

            players.add(new Paladin("Aragorn", 1, 100, 50, 80, 50, 70, 30,␣
 ↪posX, posY, sword));
            break;
        default:
            Axe axe = new Axe(50, 100, 1);

            players.add(new Warrior("Gimli", 1, 100, 50, 80, 50, 70, 30, posX,␣
 ↪posY, axe));
            break;

    }
}
```

[24]:
```
null
```

[ ]:
```

```

PokerGame_sol

December 17, 2019

0.1 Exercise 1: Poker Game

Cette semaine nous allons commencer un exercise consistant à implémenter un jeu de poker. Tout d'abord nous allons construire deux classes de base.

La classe Carte (« Card ») qui réprésente une carte. Elle possède deux attributs :

`rank (int) : la valeur de la carte (ex : as, 10 etc.)`

`suit (int) : la couleur (ou enseigne pour les puristes) de la carte.`

- `Implémenter le constructeur de la classe « Card » qui prend en argument une « value » et une`
- `Implémenter une méthode «getInfo» qui affiche dans la console la valeur et la couleur de la`

Nous choisissons ici de représenter les couleurs comme des entiers de 0 à 3 à la place de String. Ceci nous permettra de manipuler plus facilemet le jeu de carte par la suite. Pour cela, définir 4 variables statiques et finales (ex: public static final int HEART = 0;) pour définir les indices des enseignes.

[3]:
```
public class Card {

    private int rank;
    private int suit;

    public static final int HEART = 0;
    public static final int DIAMOND = 1;
    public static final int SPADE = 2;
    public static final int CLUB = 3;


    public Card(int value, int suit){
        this.rank = value;
        this.suit = suit;
    }

    public void getInfo(){
        System.out.println("Carte: " + getRank() + " de " + getSuit());
    }

    public String getRank() {
```

```
        String res = "";
        switch (rank) {
            case 11:
                res = "Valet";
                break;
            case 12:
                res = "Dame";
                break;
            case 13:
                res = "King";
                break;
            default:
                res = Integer.toString(rank);
                break;

        }
        return res;
    }

    public String getSuit() {
        String res = "";
        switch (suit) {
            case 0:
                res = "Coeur";
                break;
            case 1:
                res = "Carreau";
                break;
            case 2:
                res = "Pic";
                break;
            default:
                res = "Trèfle";
                break;

        }
        return res;
    }


}
```

[3]: com.twosigma.beaker.javash.bkr4f32305e.Card

0.1.1 La classe Player

La classe joueur (« Player ») qui représente un joueur de la partie. Elle possède plusieurs attributs :

```
hand (Card[]) : un tableau de carte de taille maximal 2
balance (int) : la balance total du joureur
stake (int) : la mise du joueur :

isMyTurn (boolean) : valant true si c'est au tour du joueur
```

- Implémenter le constructeur de la classe «Player» qui prend en argument deux cartes (puis les
- Implémenter une fonction « bet » qui propose une mise sur la table mais ne retourne rien (uti
- Implémenter la méthode « showHand » qui montre les cartes sur la table
- Implémenter une methode « win » qui prend en argument la somme des gains sur la table que le

Les deux dernières méthodes s'appliquent seulement si c'est au tour du joueur.

[4]:
```java
import java.util.ArrayList;

public class Player {

    private Card[] hand;
    private int balance;
    private boolean isMyTurn;
    private String name;


    public Player(Card c1, Card c2, int initialeBalance, String name){
        hand = new Card[2];
        hand[0] = c1;
        hand[1] = c2;
        isMyTurn = false;
        balance = initialeBalance;
        this.name = name;
    }


    public void showHand(){
        for (int i = 0; i < hand.length; i++){
            hand[i].getInfo();
        }
    }

    public void miser(int value ){
        if (isMyTurn && balance-value > 0 ){
            balance -= value;
            System.out.println("Le joueur " + name + " vient de miser: " +␣
↪Integer.toString(value));
```

```
            System.out.println("La nouvelle Balance: "+  Integer.
↪toString(balance));

        } else {
            System.out.println("Pas assez d'argent!");
        }
    }

    public void win(int value){
        if (isMyTurn) {
            this.balance += value;

        }
    }

}
```

[4]: com.twosigma.beaker.javash.bkr4f32305e.Player

0.1.2 La classe Deck

La classe Deck représente un paquet de carte, cette classe possède plusieurs attributs:

- deck (ArrayList<Card>): le jeu de carte complet
- NBR_CARDS (int): un attribut final et static (constante) égale à 52
- SHUFFLE_ITER (int): une constant qui correspond au nombre de mélange effectué de valeur 100

Dans le constructeur ne prenant aucun argument, générer le deck en créant au fur et à mesure des cartes. Implémenter une fonction public shuffle() qui mélange le jeu de carte et appeler là dans le constructeur après avoir construit le jeu de carte.
Indication: utiliser Random r = new Random(); sa fonction nextInt() et utiliser des index et une variable Card temporaire pour pouvoir échanger les cartes de position.

Implémenter une fonction getCardFromDeck() qui retourn la carte en haut de la pil et la retire du jeu.

[ ]:
```
import java.util.ArrayList;
import java.util.Random;

public class Deck {

    private ArrayList<Card>  deck;
    public static final int NBR_CARDS = 52;
    public static final int SHUFFLE_ITER = 100;

    public Deck(){
        this.deck = new ArrayList<>();

        for ( int suit = Card.HEART; suit <= Card.CLUB; suit++ ) {
```

```
            for (int rank = 1; rank <= 13; rank++) {
                deck.add(new Card(suit, rank));
            }
        }
        shuffle();
    }


    public void shuffle(){
        int index_1, index_2;

        Random generator = new Random();
        Card temp;

        for (int i=0; i<SHUFFLE_ITER; i++) {
            index_1 = generator.nextInt( deck.size() - 1 );
            index_2 = generator.nextInt( deck.size() - 1 );
            temp = deck.get( index_2 );
            deck.set( index_2 , deck.get( index_1 ) );
            deck.set( index_1, temp );

        }

    }


    public Card getCardFromDeck(){
        return deck.remove(0);
    }


}
```