

# PROJET DE BPO



# Sommaire

---

<b>Introduction.....</b>	<b>2</b>
Règles du jeu.....	2
Travail à réaliser .....	3
Prises de libertés et interprétation des règles.....	3
<b>Diagramme UML des classes .....</b>	<b>4</b>
<b>Tests unitaires des classes .....</b>	<b>5</b>
Test de l'énumération « Animal » .....	5
Test de la classe « Dompteur » .....	5
Test de la classe « Podium » .....	6
Test de la classe « Carte » .....	7
Test de la classe « Ordre » .....	8
Test de la classe « Jeu » .....	10
<b>Code du projet .....</b>	<b>10</b>
Code de l'énumération « Animal ».....	10
Code de la classe « Dompteur ».....	11
Code de la classe « Podium ».....	13
Code de la classe « Carte » .....	14
Code de la classe « Ordre ».....	17
Code de la classe « Jeu ».....	20
Code de la classe « Appli » (main).....	28
<b>Bilan du projet .....</b>	<b>31</b>
Difficultés rencontrées.....	31
Réussites.....	31
Pistes d'amélioration .....	31



# Introduction

---

Le projet consistait à adapter le jeu de société « *Crazy Circus* » en langage Java. Le programme devait permettre à des joueurs de s'affronter sur une partie dans sa totalité. Il fallait réaliser ce projet en binôme, et rendre le dossier ainsi que le code le 11 mars au plus tard.

## Règles du jeu

---

Dans *Crazy Circus*, chaque joueur incarne un dompteur ayant pour objectif de donner les bons ordres le plus rapidement aux trois animaux : le lion, l'éléphant, et l'ours blanc. Au début de chaque tour, une carte objectif est tirée au sort, elle montre la position que les animaux doivent prendre sur les deux podiums. Leur position de départ est déterminée au début du jeu, par le tirage au sort d'une carte.

Lorsqu'un objectif est atteint, les animaux conservent la position atteinte, et une nouvelle carte objectif est tirée au sort. Il existe 24 cartes objectif différentes, correspondant aux 24 positions possibles prises par les animaux sur les podiums.

Pour passer de la position de départ à celle d'arrivée, les dompteurs doivent trouver la bonne séquence d'ordres, parmi cinq possibles: « *SO* » « *MA* » « *NI* » « *KI* » et « *LO* ». Chaque ordre correspond à un déplacement, comme indiqué ci-dessous:

<b>SO</b>	L'animal au sommet du podium rouge échange sa place avec l'animal au sommet du podium bleu. Remarque : Cet ordre est interdit dans la version "difficile" du jeu.
<b>MA</b>	L'animal en bas du podium rouge va en haut de ce même podium.
<b>NI</b>	L'animal en bas du podium bleu va en haut de ce même podium.
<b>KI</b>	L'animal en haut du podium bleu va au sommet du podium rouge.
<b>LO</b>	L'animal en haut du podium rouge va au sommet du podium bleu.

Le premier joueur qui énonce une séquence d'ordre vérifie qu'elle est bonne en manipulant les animaux. S'il ne s'est pas trompé, il gagne un point. On tire alors une nouvelle carte objectif et tout le monde recommence à chercher simultanément.

Lorsque la dernière carte objectif est jouée, le joueur ayant emporté le plus de cartes est déclaré vainqueur.

## Travail à réaliser

Les identités des joueurs (leurs noms de scène) devaient être reçus sur la ligne de commande au lancement du programme. Le programme devait afficher la situation de jeu sous la forme reproduite ci-dessous.

La situation de départ est donnée en haut à gauche (ici le lion est sous l'ours sur le podium bleu alors que l' est seul sur le podium rouge) et la situation à atteindre est indiquée à droite (ici le lion est sur l'éléphant sur le podium bleu alors que l'ours est seul sur le podium rouge). Les différents ordres possibles sont rappelés en bas (les sauts d'un sommet à l'autre sont représentés par des flèches et un déplacement du bas vers 1 le haut est

ELEPHANT	OURS		LION
----	LION		OURS
BLEU	ROUGE	==>	BLEU
----	ROUGE		----
----			ROUGE
-----			
KI : BLEU	--> ROUGE	NI : BLEU	^
LO : BLEU	<-- ROUGE	MA : ROUGE	^
SO : BLEU	<-> ROUGE		

symbolisé par le caractère « ^ »). Les séquences *KIMALONI* et *SONI* sont deux solutions pour cette situation de jeu. Lorsqu'un joueur veut jouer, il saisit son identité suivie de la séquence d'ordres qu'il propose. Si « Jean » est une identité connue, le joueur peut saisir « Jean KIMALONI » par exemple. Si l'identité n'est pas connue, la séquence doit être ignorée et un message d'erreur affiche. Si le joueur n'avait pas le droit de jouer (s'il a déjà fait une erreur) ou que la séquence n'est pas la bonne, un message informatif doit être affiché par le programme. Si la séquence est correcte, le joueur remporte un point, une nouvelle situation de jeu est choisie aléatoirement par le programme parmi celles qui n'ont pas encore été jouées et un nouveau tour peut débiter. En fin de partie, le score et le rang des joueurs sont affichés (par score décroissant et par ordre alphabétique en cas d'égalité).

## Prises de libertés et interprétation des règles

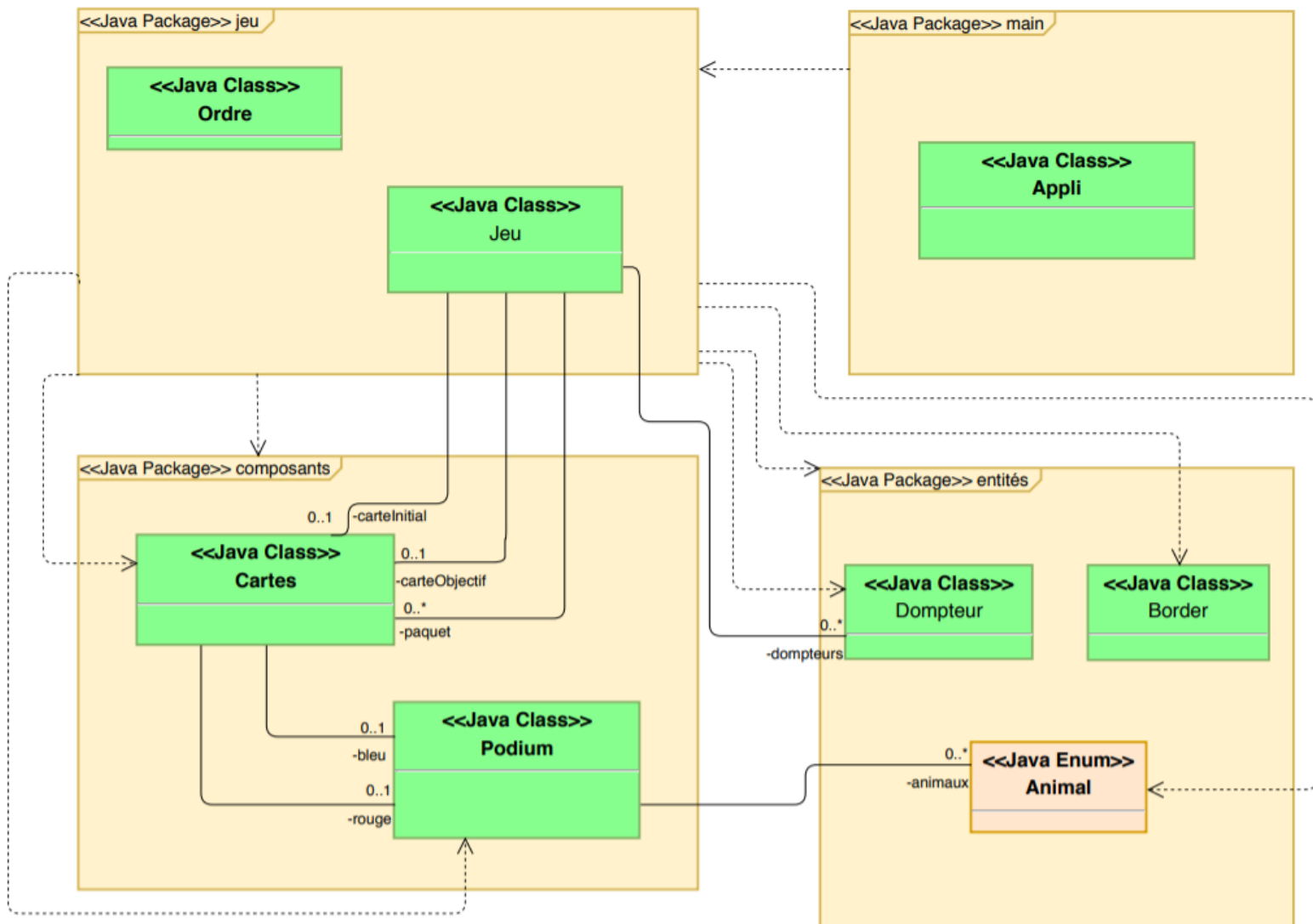
Nous avons pris quelques libertés au niveau de l'adaptation du jeu.

Par exemple, nous offrons la possibilité à l'utilisateur d'inscrire des joueurs en plus de ceux passés en paramètres du programme. Par ailleurs, s'il n'y a aucun joueur passé en paramètre, le programme permet à l'utilisateur d'en ajouter.

Nous avons également décidé d'inclure un mode « *un joueur* », dans lequel le joueur peut jouer au jeu tout seul, pour s'entraîner. Ainsi, s'il commet une erreur, au lieu de passer directement au tour suivant, il peut retenter sa chance jusqu'à réussir le tour. Pour jouer dans ce mode, il suffit de n'inscrire qu'un seul joueur.

De plus, lorsqu'un joueur donne un ordre impossible à réaliser (ex : un *KI* alors que le podium bleu est vide), nous avons décidé de considérer cela comme étant une erreur (les podiums sont vides), ainsi le joueur perd son droit de donner un ordre pendant ce tour (sauf dans le mode « *un joueur* »).

# Diagramme UML des classes



Il existe également un package « test » qui contient tous les tests unitaires des classes/énumérations présentes au-dessus. Nous avons décidé de ne pas l'inclure dans ce diagramme car les tests ne font pas réellement partie de l'application.

# Tests unitaires des classes

---

Les premiers tests sont basiques, mais les suivants nécessitent de créer une fausse situation de jeu et deviennent plus complexes. Par ailleurs, certaines classes possédant des méthodes privées, il est impossible de vérifier leur fonctionnement.

## Test de l'énumération « Animal »

---

Ce test très basique s'est exécuté avec succès.

```
package test;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import entités.Animal;

class TestAnimal {


    @Test
    void test() {
        Animal a1 = Animal.ELEPHANT;
        assertTrue(a1.getNom().equals("ELEPHANT"));

        Animal a2 = Animal.LION;
        assertTrue(a2.equals(Animal.LION));
        Animal a3 = Animal.LION;
        assertTrue(a2.equals(a3));

        Animal a4 = Animal.NULL;
        assertFalse(a4.getNom().equals("NULL"));
    }
}
```

Finished after 0,054 seconds

Runs: 1/1   \* Errors: 0   \* Failures: 0



## Test de la classe « Dompteur »

---

Ce test basique s'est exécuté avec succès.

```
package test;


import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import entités.Dompteur;
```

Finished after 0,052 seconds

Runs: 1/1   \* Errors: 0   \* Failures: 0



```

class TestDompteur {

    @Test
    void test() {
        Dompteur d1 = new Dompteur("test");
        assertTrue(d1.getNomDeScene().equals("test"));
        assertTrue(d1.getScore() == 0);
        assertTrue(d1.hasDroit());

        d1.setScore(42);
        assertTrue(d1.getScore() == 42);
        assertFalse(d1.getScore() == 0);

        d1.setDroit(false);
        assertTrue(!d1.hasDroit());
        assertFalse(d1.hasDroit());

        Dompteur d2 = new Dompteur("test");
        assertTrue(d1.getNomDeScene().equals(d2.getNomDeScene()));
        assertFalse(d1.equals(d2));
    }
}

```

### Test de la classe « Podium »

Ce test s'est exécuté avec succès. Il nécessitait déjà plus de ressources que les précédents (import de la classe Animal).

```

package test;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import composants.Podium;
import entités.Animal;

class TestPodium {

    @Test
    void test() {
        Podium p1 = new Podium();
        for(Animal a : p1.getAnimaux()) {
            assertTrue(a.equals(Animal.NULL));
        }

        Podium p2 = new Podium();
        assertFalse(p1.equals(p2));
        assertTrue(p1.getAnimaux().equals(p2.getAnimaux()));
    }
}

```

Finished after 0,085 seconds

Runs: 1/1    ✖ Errors: 0    • Failures: 0

```

        assertTrue(p1.hasSameAnimals(p2));

        p1.getAnimaux().set(0, Animal.LION);
        assertTrue(p1.getAnimaux().get(0).equals(Animal.LION));
        assertTrue(p1.getTop() == 0);
        assertFalse(p1.getAnimaux().equals(p2.getAnimaux()));
        assertFalse(p1.hasSameAnimals(p2));

        p1.getAnimaux().set(1, Animal.ELEPHANT);
        assertTrue(p1.getTop() == 1);

        p1.setNull();
        assertTrue(p1.getAnimaux().equals(p2.getAnimaux()));
    }
}

```

### Test de la classe « Carte »

Le test s'est exécuté avec succès. Ce test commence à nécessiter des mises en places plus longues.

```

package test;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import composants.Carte;
import composants.Podium;
import entités.Animal;

class TestCarte {

    @Test
    void test() {
        Carte c1 = new Carte();
        assertTrue(c1.isSameAs(c1));

        Carte c2 = new Carte();
        assertFalse(c1.equals(c2));
        assertTrue(c1.isSameAs(c2));

        Podium tmp = new Podium();
        assertTrue(c1.getBleu().hasSameAnimals(tmp));
        assertTrue(c1.getRouge().hasSameAnimals(tmp));

        assertTrue(Carte.generateCartes().size() == 24);

        c2 = Carte.generateCartes().get(0);
        assertFalse(c2.isSameAs(c1));
    }
}

```

Finished after 0,058 seconds

Runs: 1/1    ✖ Errors: 0    ✖ Failures: 0



```

        c2.getBleu().setNull();
        c2.getRouge().setNull();
        assertTrue(c1.isSameAs(c2));

        tmp.getAnimaux().set(0, Animal.LION);
        assertFalse(c1.getBleu().hasSameAnimals(tmp));
        assertFalse(c1.getRouge().hasSameAnimals(tmp));

        c1.setBleu(tmp);
        c1.setRouge(tmp);
        assertTrue(c1.getBleu().hasSameAnimals(tmp));
        assertTrue(c1.getRouge().hasSameAnimals(tmp));

        Carte c3 = new Carte(c2);
        assertTrue(c3.isSameAs(c2));
    }
}

```

### Test de la classe « Ordre »

Le test s'est exécuté avec succès. Pour ce test, il était nécessaire de créer une fausse situation de jeu, et pour tester les ordres, il faut utiliser des méthodes d'autres classes (Podium, Carte...). ce test est donc assez lourd (par rapport aux autres).

```

package test;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import composants.Carte;
import composants.Podium;
import entités.Animal;
import jeu.Ordre;

class TestOrdre {

    @Test
    void test() {
        assertTrue(Ordre.ordreValide("KIMALONI"));
        assertTrue(Ordre.ordreValide("kimaloni"));
        assertTrue(Ordre.ordreValide("kIMAloni"));
        assertFalse(Ordre.ordreValide("KI MALONI"));
        assertFalse(Ordre.ordreValide("azertyuiop"));

        Carte c = new Carte();
        Podium bleu = new Podium();
    }
}

```

Finished after 0,055 seconds

Runs: 1/1   ✖ Errors: 0   ✖ Failures: 0

```

Podium rouge = new Podium();
bleu.getAnimaux().set(0, Animal.LION);
bleu.getAnimaux().set(1, Animal.OURS);
bleu.getAnimaux().set(2, Animal.ELEPHANT);
c.setBleu(bleu);
c.setRouge(rouge);

Ordre.simulerOrdre(c, "KI");
assertTrue(c.getBleu().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getBleu().getAnimaux().get(1).equals(Animal.OURS));
assertTrue(c.getBleu().getAnimaux().get(0).equals(Animal.LION));
assertTrue(c.getRouge().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(1).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(0).equals(Animal.ELEPHANT));

Ordre.simulerOrdre(c, "SO");
assertTrue(c.getBleu().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getBleu().getAnimaux().get(1).equals(Animal.ELEPHANT));
assertTrue(c.getBleu().getAnimaux().get(0).equals(Animal.LION));
assertTrue(c.getRouge().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(1).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(0).equals(Animal.OURS));

Ordre.simulerOrdre(c, "NI");
assertTrue(c.getBleu().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getBleu().getAnimaux().get(1).equals(Animal.LION));
assertTrue(c.getBleu().getAnimaux().get(0).equals(Animal.ELEPHANT));
assertTrue(c.getRouge().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(1).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(0).equals(Animal.OURS));

Ordre.simulerOrdre(c, "KI");
Ordre.simulerOrdre(c, "MA");
assertTrue(c.getBleu().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getBleu().getAnimaux().get(1).equals(Animal.NULL));
assertTrue(c.getBleu().getAnimaux().get(0).equals(Animal.ELEPHANT));
assertTrue(c.getRouge().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(1).equals(Animal.OURS));
assertTrue(c.getRouge().getAnimaux().get(0).equals(Animal.LION));

Ordre.simulerOrdre(c, "LO");
assertTrue(c.getBleu().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getBleu().getAnimaux().get(1).equals(Animal.OURS));
assertTrue(c.getBleu().getAnimaux().get(0).equals(Animal.ELEPHANT));
assertTrue(c.getRouge().getAnimaux().get(2).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(1).equals(Animal.NULL));
assertTrue(c.getRouge().getAnimaux().get(0).equals(Animal.LION));

```

```

}

```

```

}

```

## Test de la classe « Jeu »

---

Cette classe étant la classe principale, elle possède beaucoup de méthodes privées donc non-testables. De plus, parmi le peu de méthodes publiques, la plupart sont des méthodes proche de toString, dépendant du jeu. Le jeu possédant beaucoup d'aléatoire, ces méthodes sont donc non-testables également. Ainsi, très peu de méthodes peuvent être testées, donc le test est très court.

Il s'est donc exécuté avec succès.

```
package test;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import jeu.Jeu;

class TestJeu {

    @Test
    void test() {
        Jeu jeu = new Jeu();

        jeu.addDompteur("test");
        assertTrue(jeu.dompteurExists("test"));

        jeu.addDompteur("test2");
        assertTrue(jeu.nbDompteursAllowed() == 2);

        assertTrue(jeu.continueJeu());
    }
}
```

Finished after 0,049 seconds

Runs: 1/1    ✖ Errors: 0    • Failures: 0

## Code du projet

---

Nous allons montrer les classes des plus simples aux plus complexes, en finissant par le main.

Vous pouvez également vous référer à la Javadoc qui se trouve dans le puits.

### Code de l'énumération « Animal »

---

Énumération élémentaire, ne dépendant d'aucune classe. Il existe 3 animaux, chacun caractérisé par son nom, plus un animal nul.

```
package entités;

/**
```

```

* Enumère les différents animaux existant
* @author BOUCHET Ulysse & VIGNARAJAH Daran
* @version 2.2 18/02/2019
*/
public enum Animal {
    OURS("OURS"), LION("LION"), ELEPHANT("ELEPHANT"), NULL(" ");

    /**
     * Nom de l'animal
     */
    private String nom;

    /**
     * Constructeur d'Animal
     * @param nom
     */
    Animal(String nom) {
        this.nom = nom;
    }

    /**
     * Getter du nom de l'Animal
     * @return le nom de l'animal
     * @see Animal#nom
     */
    public String getNom() {
        return nom;
    }
}

```

### *Code de la classe « Dompteur »*

---

Classe élémentaire ne dépendant d'aucune autre classe. Un dompteur est caractérisé par un nom, un score, et un droit de donner un ordre.

```

package entités;

/**
 * Définit un dompteur.
 *
 * @author BOUCHET Ulysse & VIGNARAJAH Daran
 * @version 1.3 27/02/2019
 */
public class Dompteur {

    /**
     * Le nom de scène du dompteur.
     */
    private String nomDeScene;

```

```

/**
 * Le score du dompteur.
 */
private int score;

/**
 * Définit si le dompteur a le droit ou non de jouer pendant ce tour.
 */
private boolean droit;

/**
 * Construit un dompteur
 * @param nomDeScene que l'on veut donner au dompteur
 */
public Dompteur(String nomDeScene) {
    this.nomDeScene = nomDeScene;
    setScore(0);
    setDroit(true);
}

// Getters & Setters

/**
 * Getter du nom de scène du dompteur
 * @return le nom de scène du dompteur
 * @see Dompteur#nomDeScene
 */
public String getNomDeScene() {
    return nomDeScene;
}

/**
 * Getter du score du dompteur
 * @return le score du dompteur
 * @see Dompteur#score
 */
public int getScore() {
    return score;
}

/**
 * Setter du score du dompteur
 * @param score que l'on veut donner au dompteur
 * @see Dompteur#score
 */
public void setScore(int score) {
    this.score = score;
}

/**
 * Getter du droit du dompteur

```

```

    * @return le droit du dompteur
    * @see Dompteur#droit
    */
    public boolean hasDroit() {
        return droit;
    }

    /**
     * Setter du droit du dompteur
     * @param droit que l'on veut accorder au dompteur
     * @see Dompteur#droit
     */
    public void setDroit(boolean droit) {
        this.droit = droit;
    }
}

```

### Code de la classe « Podium »

---

Cette classe nécessite l'import de la classe « Animal » (package « entités ») et de la classe « ArrayList » (package « java.util »). Un podium est une « pile » de 3 animaux maximum.

```

package composants;

import java.util.ArrayList;

import entités.Animal;

/**
 * Classe définissant un podium
 *
 * @author BOUCHET Ulysse & VIGNARAJAH Daran
 * @version 1.7 21/02/2019
 */
public class Podium {

    /**
     * Liste des animaux qui composent un podium
     * @see Animal
     */
    private ArrayList<Animal> animaux = new ArrayList<>();

    /**
     * Constructeur du Podium.
     * @see Podium#setNull()
     */
    public Podium() {
        this.setNull();
    }

    /**
     * Permet de déterminer si ce podium est égal au podium p
     * @param p l'autre podium
     * @return true s'ils sont égaux, false sinon
     */
}

```

```

        public boolean hasSameAnimals(Podium p) {
            for (int i = 0; i < 3; ++i)
                if (!(this.animaux.get(i).equals(p.animaux.get(i))))
                    return false;

            return true;
        }

        //Getters & Setters

        /**
         * Remplit d'animaux NULL le podium
         * @see Podium#animaux
         */
        public void setNull() {
            animaux.clear();
            for (int i = 0; i < 3; ++i)
                this.animaux.add(Animal.NULL);
        }

        /**
         * Permet d'obtenir l'indice de l'Animal situé en haut du Podium
         * @return l'indice de l'Animal situé en haut du Podium
         */
        public int getTop() {
            for (int i = 0; i < this.animaux.size(); ++i)
                if (this.animaux.get(i).equals(Animal.NULL))
                    return (i - 1);

            return 2;
        }

        /**
         * Permet d'obtenir la liste des animaux du Podium
         * @return la liste des animaux du Podium
         */
        public ArrayList<Animal> getAnimaux() {
            return animaux;
        }
    }

```

### Code de la classe « Carte »

Cette classe nécessite l'import de la classe « Animal » (package « entités »). Elle nécessite également la classe « Podium », qui se trouve dans le même package (« composants »). Enfin, elle nécessite l'import de la classe « ArrayList » (package « java.util »). Une carte est une association de deux podiums.

```

package composants;

import java.util.ArrayList;

import entités.Animal;

/**
 * Classe déterminant une Carte. Une carte contient deux Podiums
 *
 * @author BOUCHET Ulysse & VIGNARAJAH Daran
 * @version 1.5 02/03/2019

```

```

*/
public class Carte {
    /**
     * Premier podium, le podium bleu
     *
     * @see Podium
     */
    private Podium bleu;

    /**
     * Second podium, le podium rouge
     *
     * @see Podium
     */
    private Podium rouge;

    /**
     * Constructeur de carte vide
     */
    public Carte() {
        bleu = new Podium();
        rouge = new Podium();
    }

    /**
     * Constructeur de carte par copie
     *
     * @see Jeu#carteInitiale
     */
    public Carte(Carte c) {
        Podium copyBleu = new Podium();
        Podium copyRouge = new Podium();
        for (int i = 0; i < 3; ++i) {
            copyBleu.getAnimaux().set(i, c.getBleu().getAnimaux().get(i));
            copyRouge.getAnimaux().set(i, c.getRouge().getAnimaux().get(i));
        }

        this.setBleu(copyBleu);
        this.setRouge(copyRouge);
    }

    /**
     * Vérifie si deux cartes sont égales
     *
     * @param c la 2ème carte
     * @return un boolean true si les cartes sont les mêmes, false sinon
     * @see Podium#hasSameAnimals(Podium)
     */
    public boolean isSameAs(Carte c) {
        return this.getBleu().hasSameAnimals(c.getBleu()) && this.getRouge().hasSameAnimals(c.getRouge());
    }

    /**
     * Génère les 24 cartes possibles
     *
     * @return la liste des 24 cartes possibles
     *
     * @see Carte
     * @see Podium
     * @see Animal
     */
    public static ArrayList<Carte> generateCartes() {
        ArrayList<Carte> cartes = new ArrayList<>();
        Carte tmp1;

```





```

    Carte tmp2;
    for (int o = 1; o <= 3; ++o)
        for (int l = 1; l <= 3; ++l)
            for (int e = 1; e <= 3; ++e) {
                if (o + l + e == 6 && !(o == 2 && e == 2 && l == 2)) {
                    tmp1 = new Carte();
                    tmp2 = new Carte();

                    tmp1.getBleu().getAnimaux().set(o - 1, Animal.OURS);
                    tmp1.getBleu().getAnimaux().set(l - 1, Animal.LION);
                    tmp1.getBleu().getAnimaux().set(e - 1, Animal.ELEPHANT);
                    cartes.add(tmp1);

                    tmp2.getRouge().getAnimaux().set(o - 1, Animal.OURS);
                    tmp2.getRouge().getAnimaux().set(l - 1, Animal.LION);
                    tmp2.getRouge().getAnimaux().set(e - 1, Animal.ELEPHANT);
                    cartes.add(tmp2);
                }
                if (o == 3 && l + e == 3) {
                    tmp1 = new Carte();
                    tmp2 = new Carte();

                    tmp1.getBleu().getAnimaux().set(o, Animal.OURS);
                    tmp1.getRouge().getAnimaux().set(l - 1, Animal.LION);
                    tmp1.getRouge().getAnimaux().set(e - 1, Animal.ELEPHANT);
                    cartes.add(tmp1);

                    tmp2.getRouge().getAnimaux().set(o, Animal.OURS);
                    tmp2.getBleu().getAnimaux().set(l - 1, Animal.LION);
                    tmp2.getBleu().getAnimaux().set(e - 1, Animal.ELEPHANT);
                    cartes.add(tmp2);
                }
                if (l == 3 && o + e == 3) {
                    tmp1 = new Carte();
                    tmp2 = new Carte();

                    tmp1.getBleu().getAnimaux().set(o, Animal.LION);
                    tmp1.getRouge().getAnimaux().set(o - 1, Animal.OURS);
                    tmp1.getRouge().getAnimaux().set(e - 1, Animal.ELEPHANT);
                    cartes.add(tmp1);

                    tmp2.getRouge().getAnimaux().set(o, Animal.LION);
                    tmp2.getBleu().getAnimaux().set(o - 1, Animal.OURS);
                    tmp2.getBleu().getAnimaux().set(e - 1, Animal.ELEPHANT);
                    cartes.add(tmp2);
                }
                if (e == 3 && l + o == 3) {
                    tmp1 = new Carte();
                    tmp2 = new Carte();

                    tmp1.getBleu().getAnimaux().set(o, Animal.ELEPHANT);
                    tmp1.getRouge().getAnimaux().set(l - 1, Animal.LION);
                    tmp1.getRouge().getAnimaux().set(o - 1, Animal.OURS);
                    cartes.add(tmp1);

                    tmp2.getRouge().getAnimaux().set(o, Animal.ELEPHANT);
                    tmp2.getBleu().getAnimaux().set(l - 1, Animal.LION);
                    tmp2.getBleu().getAnimaux().set(o - 1, Animal.OURS);
                    cartes.add(tmp2);
                }
            }
        }
    }

    return cartes;
}

```

```

// Getters & Setters

/**
 * Getter du podium bleu
 *
 * @return le podium bleu
 * @see Carte#bleu
 * @see Podium
 */
public Podium getBleu() {
    return bleu;
}

/**
 * Setter du podium bleu
 *
 * @param bleu le nouveau podium bleu
 * @see Carte#bleu
 * @see Podium
 */
public void setBleu(Podium bleu) {
    this.bleu = bleu;
}

/**
 * Getter du podium rouge
 *
 * @return le podium rouge
 * @see Carte#rouge
 * @see Podium
 */
public Podium getRouge() {
    return rouge;
}

/**
 * Setter du podium rouge
 *
 * @param rouge le nouveau podium rouge
 * @see Carte#rouge
 * @see Podium
 */
public void setRouge(Podium rouge) {
    this.rouge = rouge;
}
}

```

### Code de la classe « Ordre »

---

Cette classe nécessite l'import de la classe « Animal » (package « entités ») et de la classe Carte (package « composants »). Un ordre permet de modifier une carte (donc la position des animaux sur les podiums).

```

package jeu;

import composants.Carte;
import entités.Animal;

/**

```



```

* Classe Ordre qui permet d'exécuter et de valider des ordres
*
* @author BOUCHET Ulysse & VIGNARAJAH Daran
* @version 1.6 17/02/2019
*
*/
public class Ordre {

    /**
     * Simule l'ordre à effectuer
     *
     * @param carte à partir de laquelle on veut simuler l'ordre
     * @param ordre que l'on veut simuler
     * @see Ordre#ki(Carte)
     * @see Ordre#lo(Carte)
     * @see Ordre#so(Carte)
     * @see Ordre#ni(Carte)
     * @see Ordre#ma(Carte)
     */
    public static void simulerOrdre(Carte carte, String ordre) {

        assert(Ordre.ordreValide(ordre));
        for (int i = 0; i < ordre.length(); i += 2)
            switch (ordre.substring(i, i + 2).toUpperCase()) {
                case "KI":
                    Ordre.ki(carte);
                    break;
                case "LO":
                    Ordre.lo(carte);
                    break;
                case "SO":
                    Ordre.so(carte);
                    break;
                case "NI":
                    Ordre.ni(carte);
                    break;
                case "MA":
                    Ordre.ma(carte);
                    break;
                default:
                    break;
            }
    }

    /**
     * Simule l'ordre ki
     *
     * @param tmp la carte pour laquelle on simule l'ordre
     */
    private static void ki(Carte tmp) {
        if (tmp.getBleu().getTop() >= 0) {
            tmp.getRouge().getAnimaux().set(tmp.getRouge().getTop() + 1,
                tmp.getBleu().getAnimaux().get(tmp.getBleu().getTop()));
            tmp.getBleu().getAnimaux().set(tmp.getBleu().getTop(), Animal.NULL);
        } else {
            tmp.getBleu().setNull();
            tmp.getRouge().setNull();
        }
    }

    /**
     * Simule l'ordre lo
     *
     * @param tmp la carte pour laquelle on simule l'ordre
     */

```

```

    */
    private static void lo(Carte tmp) {
        if (tmp.getRouge().getTop() >= 0) {
            tmp.getBleu().getAnimaux().set(tmp.getBleu().getTop() + 1,
                tmp.getRouge().getAnimaux().get(tmp.getRouge().getTop()));
            tmp.getRouge().getAnimaux().set(tmp.getRouge().getTop(), Animal.NULL);
        } else {
            tmp.getBleu().setNull();
            tmp.getRouge().setNull();
        }
    }

    /**
     * Simule l'ordre so
     *
     * @param tmp la carte pour laquelle on simule l'ordre
     */
    private static void so(Carte tmp) {
        Animal tmpA;
        if (tmp.getRouge().getTop() >= 0 && tmp.getBleu().getTop() >= 0) {
            tmpA = tmp.getBleu().getAnimaux().get(tmp.getBleu().getTop());
            tmp.getBleu().getAnimaux().set(tmp.getBleu().getTop(),
                tmp.getRouge().getAnimaux().get(tmp.getRouge().getTop()));
            tmp.getRouge().getAnimaux().set(tmp.getRouge().getTop(), tmpA);
        } else {
            tmp.getBleu().setNull();
            tmp.getRouge().setNull();
        }
    }

    /**
     * Simule l'ordre ni
     *
     * @param tmp la carte pour laquelle on simule l'ordre
     */
    private static void ni(Carte tmp) {
        if (tmp.getBleu().getTop() >= 0) {
            Animal tmpA = Animal.NULL;
            tmpA = tmp.getBleu().getAnimaux().get(0);
            tmp.getBleu().getAnimaux().remove(0);
            tmp.getBleu().getAnimaux().add(Animal.NULL);
            tmp.getBleu().getAnimaux().set(tmp.getBleu().getTop() + 1, tmpA);
        } else {
            tmp.getBleu().setNull();
            tmp.getRouge().setNull();
        }
    }

    /**
     * Simule l'ordre ma
     *
     * @param tmp la carte pour laquelle on simule l'ordre
     */
    private static void ma(Carte tmp) {
        if (tmp.getRouge().getTop() >= 0) {
            Animal tmpA = Animal.NULL;
            tmpA = tmp.getRouge().getAnimaux().get(0);
            tmp.getRouge().getAnimaux().remove(0);
            tmp.getRouge().getAnimaux().add(Animal.NULL);
            tmp.getRouge().getAnimaux().set(tmp.getRouge().getTop() + 1, tmpA);
        } else {

```

```

        tmp.getBleu().setNull();
        tmp.getRouge().setNull();
    }
}

/**
 * Vérifie si l'ordre est valide
 *
 * @param ordre à vérifier
 * @return une boolean true si l'ordre est valide, false sinon
 */
public static boolean ordreValide(String ordre) {
    if (ordre.length() % 2 != 0)
        return false;

    for (int i = 0; i < ordre.length(); i = i + 2) {
        String str = ordre.substring(i, i + 2).toUpperCase();
        if (!str.equals("KI") && !str.equals("LO")
            && !str.equals("SO") && !str.equals("NI")
            && !str.equals("MA"))
            return false;
    }
    return true;
}
}

```

### Code de la classe « Jeu »

---

Cette classe est la principale classe du projet, c'est elle qui assure le bon déroulement du jeu. Ainsi, elle nécessite l'import de presque toutes les classes précédentes. Elle nécessite également les classes « Collections » et « Comparator » du package « java.util ».

```

package jeu;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

import composants.Carte;
import entités.Animal;
import entités.Dompteur;

/**
 * Classe contenant toutes les fonctions et attributs relatifs au jeu. La
 * plupart des fonctions de cette classe return des String, pour optimiser
 * l'affichage
 *
 * @author BOUCHET Ulysse & VIGNARAJAH Daran
 * @version 2.2 02/03/2019
 */
public class Jeu {

    /**
     * Liste représentant le paquet de cartes: il contient toutes les cartes
     * possibles
     *
     * @see Carte
     * @see Carte#generateCartes()
     */
    private ArrayList<Carte> paquet;

```

```

/**
 * Contient la liste des dompteurs
 *
 * @see Dompteur
 */
private ArrayList<Dompteur> dompteurs;

/**
 * Carte contenant la position initiale des animaux à chaque tour
 *
 * @see Carte
 */
private Carte carteInitiale;

/**
 * Carte contenant la position à atteindre par les animaux à chaque tour
 *
 * @see Carte
 */
private Carte carteObjectif;

/**
 * Boolean déterminant la difficulté de la partie
 */
private boolean difficile;

/**
 * Bordure permettant un affichage plus compréhensible
 */
private static final String border = "-----\n";

/**
 * Constructeur de la classe Jeu
 *
 * @see Jeu
 * @see Jeu#initialiser()
 */
public Jeu() {
    dompteurs = new ArrayList<Dompteur>();
    carteInitiale = new Carte();
    carteObjectif = new Carte();
    difficile = false;
    this.initialiser();
}

/**
 * Initialise le jeu
 *
 * @see Carte#generateCartes()
 */
private void initialiser() {
    paquet = Carte.generateCartes();
    this.carteInitiale = tirerCarteFromPaquet();

    do
        this.carteObjectif = tirerCarteFromPaquet();
    while (this.carteObjectif.equals(this.carteInitiale));
}

/**
 * Génère un nouveau tour
 */

```

```

* @return une String pour afficher à l'écran ce que la méthode a fait
* @see Jeu#tirerCarteFromPaquet()
*/
private String generateRound() {
    for (Dompteur d : this.dompteurs)
        d.setDroit(true);

    this.cartelInitiale = this.carteObjectif;
    this.carteObjectif = tirerCarteFromPaquet();

    return "Tirage d'une nouvelle carte objectif...";
}

/**
* Tire aléatoirement une carte du paquet
*
* @return la carte tirée
* @see Carte
* @see Jeu#paquet
*/
private Carte tirerCarteFromPaquet() {
    int r = (int) (paquet.size() * Math.random());

    r = (int) (paquet.size() * Math.random());

    Carte tmp = paquet.get(r);
    return tmp;
}

/**
* Permet l'affichage à la console de la situation de jeu
*
* @return une String contenant tout l'affichage du jeu
*/
public String displayJeu() {
    StringBuilder sJeu = new StringBuilder();

    String separator = " ---- ";
    String arrow = "=>";
    String sBleu = "BLEU";
    String sRouge = "ROUGE";
    String nom = new String();

    for (int i = 2; i >= 0; --i) {
        nom = (this.cartelInitiale.getBleu().getAnimaux().get(i) == Animal.ELEPHANT)
            ? this.cartelInitiale.getBleu().getAnimaux().get(i).getNom()
            : (" " + this.cartelInitiale.getBleu().getAnimaux().get(i).getNom() + " ");
        sJeu.append(nom);
        sJeu.append(" ");
        nom = (this.cartelInitiale.getRouge().getAnimaux().get(i) == Animal.ELEPHANT)
            ? this.cartelInitiale.getRouge().getAnimaux().get(i).getNom()
            : (" " + this.cartelInitiale.getRouge().getAnimaux().get(i).getNom() + " ");
        sJeu.append(nom);
        sJeu.append(" ");
        nom = (this.carteObjectif.getBleu().getAnimaux().get(i) == Animal.ELEPHANT)
            ? this.carteObjectif.getBleu().getAnimaux().get(i).getNom()
            : (" " + this.carteObjectif.getBleu().getAnimaux().get(i).getNom() + " ");
        sJeu.append(nom);
        sJeu.append(" ");
        nom = (this.carteObjectif.getRouge().getAnimaux().get(i) == Animal.ELEPHANT)
            ? this.carteObjectif.getRouge().getAnimaux().get(i).getNom()
            : (" " + this.carteObjectif.getRouge().getAnimaux().get(i).getNom() + " ");
        sJeu.append(nom);
    }
}

```

```

        sJeu.append(System.lineSeparator());
    }
    sJeu.append(separator);
    sJeu.append(" ");
    sJeu.append(separator);
    sJeu.append(" ");
    sJeu.append(arrow);
    sJeu.append(" ");
    sJeu.append(separator);
    sJeu.append(" ");
    sJeu.append(separator);

    sJeu.append(System.lineSeparator());

    sJeu.append(" ");
    sJeu.append(sBleu);
    sJeu.append(" ");
    sJeu.append(sRouge);
    sJeu.append(" ");
    sJeu.append(sBleu);
    sJeu.append(" ");
    sJeu.append(sRouge);

    sJeu.append(System.lineSeparator());
    sJeu.append(System.lineSeparator());

    for (int i = 0; i < 39; ++i)
        sJeu.append("-");
    sJeu.append(System.lineSeparator());

    sJeu.append("KI : BLEU --> ROUGE  NI : BLEU ^");
    sJeu.append(System.lineSeparator());
    sJeu.append("LO : BLEU <-- ROUGE  MA : ROUGE ^");
    sJeu.append(System.lineSeparator());
    sJeu.append("SO : BLEU <-> ROUGE");

    return sJeu.toString();
}

/**
 * Gère un tour de jeu à partir d'un ordre et d'un dompteur
 *
 * @param nomDeScene de la personne ayant donné un ordre
 * @param ordre à effectuer
 * @return une String décrivant ce qu'il s'est passé durant le tour
 * @see Jeu#dompteurExists(String)
 * @see Jeu#getDompteurFromJeu(String)
 * @see Ordre#ordreValide(String)
 * @see Ordre#simulerOrdre(Jeu, String)
 * @see Jeu#check(Carte)
 * @see Jeu#nbDompteursAllowed()
 * @see Jeu#pointForOnlyDompteur()
 * @see Jeu#continueJeu()
 * @see Jeu#generateRound()
 */
public String play(String nomDeScene, String ordre) {
    StringBuilder s = new StringBuilder();

    if (this.dompteurExists(nomDeScene)) {
        Dompteur tmp = this.getDompteurFromJeu(nomDeScene);
        if (tmp.hasDroit()) {

```



```

        if (Ordre.ordreValide(ordre)) {
            if (this.difficile && ordre.contains("SO"))
                s.append("La commande <SO> est interdite en mode difficile!");
            else {
                Carte simul = new Carte(this.cartelInitiale);
                Ordre.simulerOrdre(simul, ordre);

                if (simul.isSameAs(this.carteObjectif)) {
                    tmp.setScore(tmp.getScore() + 1);

                    s.append(tmp.getNomDeScene());
                    s.append(" a trouvé une solution!");
                    s.append(System.lineSeparator());

                } else {
                    s.append("L'ordre que vous avez donné est faux! ");
                    s.append(System.lineSeparator());
                    s.append(border);

                    if (this.dompteurs.size() != 1) {
                        s.append(tmp.getNomDeScene());
                        s.append(" perd le droit de donner des ordres

durant ce tour!");

                        s.append(System.lineSeparator());
                        tmp.setDroit(false);

                        // S'il n'y a plus qu'un dompteur pouvant agir, il
gagne

                        // (sauf mode 1 joueur)
                        if (this.nbDompteursAllowed() == 1) {

                            s.append(this.pointForOnlyDompteur());

                            s.append(System.lineSeparator());
                        }
                    }
                }

                if (this.continueJeu()) {
                    s.append(border);
                    s.append(this.generateRound());
                    this.paquet.remove(this.carteObjectif);
                }
            }
        } else
            s.append("Ordre non reconnu.");

    } else
        s.append("Ce dompteur n'a plus le droit de donner d'ordres pendant ce tour.");

    } else
        s.append("Nom de scène non reconnu.");

    return s.toString();
}

/**
 * Vérifie s'il reste des cartes dans le paquet
 *
 * @return un boolean true si c'est le cas, false sinon
 * @see Jeu#paquet
 */
public boolean continueJeu() {

```

```

        if (paquet.size() == 0)
            return false;
        else
            return true;
    }

    /**
     * donne un point au seul dompteur ayant encore le droit de jouer
     *
     * @return une String décrivant ce qu'a fait la méthode
     * @see Jeu#nbDompteursAllowed()
     */
    private String pointForOnlyDompteur() {
        StringBuilder s = new StringBuilder();

        Dompteur tmp;
        for (Dompteur d : this.dompteurs)
            if (d.hasDroit()) {
                s.append("Tous les autres joueurs s'étant trompés, ");
                s.append(d.getNomDeScene());
                s.append(" gagne ce tour!");

                tmp = d;
                tmp.setScore(tmp.getScore() + 1);
            }

        return s.toString();
    }

    /**
     * Permet l'affichage du classement des Dompteurs
     *
     * @return une String contenant l'affichage
     * @see Jeu#orderDompteurs()
     */
    public String displayLeaderboard() {
        StringBuilder sLeaderboard = new StringBuilder();
        if (this.dompteurs.size() != 1) {
            int nb = 1;
            orderDompteurs();
            for (Dompteur d : this.dompteurs) {
                sLeaderboard.append(nb);
                sLeaderboard.append(": ");
                sLeaderboard.append(d.getNomDeScene());
                sLeaderboard.append(" ");
                sLeaderboard.append(d.getScore());
                sLeaderboard.append(" points.");
                sLeaderboard.append(System.lineSeparator());
                nb++;
            }

            sLeaderboard.append(border);
            sLeaderboard.append(System.lineSeparator());

            sLeaderboard.append("Félicitations à ");
            sLeaderboard.append(this.dompteurs.get(o).getNomDeScene());
            sLeaderboard.append(" qui a gagné cette partie!");
        } else
            sLeaderboard.append("Fin de l'entraînement !");

        return sLeaderboard.toString();
    }
}

```



```

/**
 * Permet le tri des dompteurs en fonction de leur score puis de leur nom de
 * scène
 *
 * @see Collections#sort(java.util.List, Comparator)
 * @see Jeu#comparePseudos(Dompteur, Dompteur)
 * @see Jeu#compareScores(Dompteur, Dompteur)
 */
private void orderDompteurs() {
    Collections.sort(this.dompteurs, new Comparator<Dompteur>() {
        @Override
        public int compare(Dompteur d1, Dompteur d2) {
            if (compareScores(d1, d2) != 0)
                return compareScores(d1, d2);
            else
                return comparePseudos(d1, d2);
        }
    });
}

/**
 * Compare le score de deux dompteurs
 *
 * @param d1 le premier dompteur
 * @param d2 le second dompteur
 * @return 1 si le second a un meilleur score, 0 s'ils ont le même score, -1 si
 *         le premier a un meilleur score
 */
private static int compareScores(Dompteur d1, Dompteur d2) {
    if (d1.getScore() < d2.getScore())
        return 1;
    else if (d1.getScore() == d2.getScore())
        return 0;
    else
        return -1;
}

/**
 * Compare les noms de deux dompteurs
 *
 * @param d1 le premier dompteur
 * @param d2 le second dompteur
 * @return 1 si le second se trouve avant alphanumériquement parlant, 0 s'ils
 *         ont le même nom, -1 si le premier se trouve avant alphanumériquement
 *         parlant
 */
private static int comparePseudos(Dompteur d1, Dompteur d2) {
    for (char c1 : d1.getNomDeScene().toCharArray())
        for (char c2 : d2.getNomDeScene().toCharArray())
            if (c1 > c2)
                return 1;
            else if (c1 < c2)
                return -1;

    return 0;
}

/**
 * Permet d'obtenir le nombre de dompteurs ayant le droit de donner un ordre
 *
 * @return 0 s'il y a un seul joueur (mode entraînement), le nombre de dompteurs
 *         ayant le droit de donner un ordre sinon

```



```

    */
    public int nbDompteursAllowed() {
        int nb = 0;
        if (this.dompteurs.size() != 1)
            for (Dompteur d : this.dompteurs)
                if (d.hasDroit())
                    nb++;

        return nb;
    }

    // Getters & Setters

    /**
     * Permet d'ajouter un dompteur au jeu
     *
     * @param nomDeScene le nom du dompteur
     * @return une String décrivant ce qu'a fait la méthode
     * @see Jeu#dompteurs
     * @see Jeu#dompteurExists(String)
     */
    public String addDompteur(String nomDeScene) {
        if (!(dompteurExists(nomDeScene))) {
            dompteurs.add(new Dompteur(nomDeScene));
            return "Dompteur <" + nomDeScene + "> ajouté.";
        } else
            return "Le dompteur <" + nomDeScene + "> existe déjà.";
    }

    /**
     * Permet de changer la difficulté du jeu
     *
     * @param difficile boolean déterminant la difficulté du jeu
     * @see Jeu#difficile
     */
    public void setDifficile(boolean difficile) {
        this.difficile = difficile;
    }

    /**
     * Permet d'obtenir un dompteur participant au jeu à partir de son nom de scène
     *
     * @param nomDeScene
     * @return le dompteur s'il participe au jeu, un dompteur "erreur" sinon
     * @see Jeu#dompteurs
     * @see Dompteur
     */
    public Dompteur getDompteurFromJeu(String nomDeScene) {
        if (dompteurExists(nomDeScene)) {
            for (Dompteur d : this.dompteurs)
                if (d.getNomDeScene().equals(nomDeScene))
                    return d;
        }

        return new Dompteur("error");
    }

    /**
     * Permet de vérifier si un dompteur participe au jeu à partir de son nom de
     * scène
     *
     * @param nomDeScene

```



```

        * @return true s'il existe, false sinon
        * @see Jeu#dompteurs
        * @see Dompteur
        */
        public boolean dompteurExists(String nomDeScene) {
            for (Dompteur d : this.dompteurs)
                if (d.getNomDeScene().equals(nomDeScene))
                    return true;

            return false;
        }
    }
}

```

### Code de la classe « Appli » (main)

Cette classe est le main du projet. Elle ne nécessite que la classe « Jeu » (package « jeu »), la classe « Dompteur » (package « entités ») et la classe « Scanner » (package « java.util »).

```

package main;

import java.util.Scanner;

import entités.Dompteur;
import jeu.Jeu;

/**
 * Classe Appli du projet crazyCircus. Elle contient le main.
 *
 * @author BOUCHET Ulysse & VIGNARAJAH Daran
 * @version 2.3 02/03/2019
 */
public class Appli {

    /**
     * Bordure permettant un affichage plus compréhensible
     */
    private static final String border = "-----\n";

    /**
     * main du projet crazyCircus
     *
     * @param noms la liste des noms de scènes des dompteurs voulant participer au
     *            jeu
     */
    public static void main(String[] noms) {
        Scanner sc = new Scanner(System.in);

        // Génération du jeu
        Jeu crazyCircus = new Jeu();
        System.out.print(border);
        System.out.println("Initialisation du jeu en cours...");

        if (noms.length != 0) {

            // Inscription des dompteurs
            System.out.print(border);
            System.out.println("Inscription de tous les dompteurs...");
            for (int i = 0; i < noms.length; ++i) {
                System.out.println(crazyCircus.addDompteur(noms[i]));
            }
        }
    }
}

```

```

    }

    // Potentielle inscription d'autres dompteurs
    System.out.print(border);
    System.out.println("En plus de ceux que vous avez passé en paramètre,");
    if (continueAdding())
        addDompteurs(crazyCircus);

} else {
    // Si aucun dompteur n'a été passé en paramètre, inscription de dompteurs
    // manuelle
    System.out.print(border);
    System.out.println("Aucun dompteur n'a été passé en paramètre.");
    System.out.println("Veuillez les entrer manuellement.");
    addDompteurs(crazyCircus);
}

chooseDifficulty(crazyCircus);
// Boucle de jeu
boolean play = true;
String nomDeScene;
String commande;
while (crazyCircus.continueJeu()) {
    // Affichage de la situation
    System.out.print(border);
    System.out.println(crazyCircus.displayJeu());

    // Scan de la commande:
    System.out.print(border);
    System.out.println("Entrez votre pseudo suivi d'une commande :");
    nomDeScene = sc.next();
    commande = sc.next().toUpperCase();

    // Simulation de la commande et affichage de ses conséquences
    System.out.print(border);
    System.out.println(crazyCircus.play(nomDeScene, commande));
}

// Affichage du classement
System.out.print(border);
System.out.println(crazyCircus.displayLeaderboard());
}

/**
 * Permet d'ajouter des dompteurs (joueurs) au jeu passé en paramètre
 *
 * @param j Jeu auquel on souhaite ajouter des dompteurs
 *
 * @see Jeu
 * @see Jeu#addDompteur(String)
 * @see Dompteur
 */
public static void addDompteurs(Jeu j) {
    Scanner sc = new Scanner(System.in);

    String nomDeScene = new String();
    boolean addDompteurs = true;
    while (addDompteurs) {
        Dompteur tmp;
        System.out.print(border);
        System.out.println("Entrez le nom de scène d'un dompteur :");

        nomDeScene = sc.next();
    }
}

```

```

        System.out.println(j.addDompteur(nomDeScene));
        addDompteurs = continueAdding();
    }
}

/**
 * Demande à l'utilisateur s'il veut continuer à ajouter de nouveaux dompteurs
 * au jeu
 *
 * @return true si l'utilisateur veut, false sinon
 */
public static boolean continueAdding() {
    Scanner sc = new Scanner(System.in);

    String choix;
    boolean valide = false;
    while (!valide) {
        System.out.println("Avez-vous d'autres dompteurs à ajouter? (OUI/NON)");
        choix = sc.next();
        if (choix.toUpperCase().equals("OUI"))
            return true;
        else if (choix.toUpperCase().equals("NON"))
            return false;
        else {
            System.out.print(border);
            System.out.println("Veuillez entrer une chaîne valide.");
        }
    }
    return false;
}

/**
 * Permet à l'utilisateur de choisir la difficulté du jeu
 *
 * @param j le jeu
 * @see Jeu
 * @see Jeu#setDifficile(boolean)
 */
public static void chooseDifficulty(Jeu j) {
    Scanner sc = new Scanner(System.in);

    String choix = " ";
    while (!choix.toUpperCase().equals("FACILE") && !choix.toUpperCase().equals("DIFFICILE")) {
        System.out.print(border);
        System.out.println("Choisissez votre difficulté (Facile/Difficile):");
        System.out.println("En mode difficile, la commande SO est interdite.");

        choix = sc.next();

        if (choix.toUpperCase().equals("FACILE"))
            j.setDifficile(false);
        else if (choix.toUpperCase().equals("DIFFICILE"))
            j.setDifficile(true);
        else {
            System.out.print(border);
            System.out.println("Veuillez entrer une chaîne valide.");
        }
    }
}
}

```

# Bilan du projet

---

## Difficultés rencontrées

---

Une difficulté rencontrée a été la recherche d'un algorithme de génération des 24 cartes possibles. En effet, nous avons trouvé un premier algorithme, qui était fonctionnel, mais très long : il faisait 48 tours de boucles, et plus de 150 lignes. Ainsi, nous avons dû beaucoup réfléchir jusqu'à trouver l'algorithme actuel (basé sur du calcul de position). Il fait dorénavant seulement 27 tours de boucles, pour un peu moins de 60 lignes.

Une autre difficulté a été de reproduire l'affichage de la situation de jeu comme il a été montré, c'est-à-dire avec les animaux centrés sur les podiums. Nous avons donc utilisé des ternaires (cf. fonction `displayJeu()` de la classe `jeu.Jeu`).

## Réussites

---

Une réussite de ce projet est donc évidemment l'algorithme léger que nous avons trouvé pour générer les cartes. Nous ne voulions pas faire la solution de « facilité », c'est-à-dire de rentrer toutes les cartes en dur. Nous ne voulions pas non plus les lire à partir d'un fichier texte, car bien que ce soit intéressant de coder la lecture du fichier, nous considérons que cela reste pareil que rentrer les cartes en dur. Ainsi, nous avons préféré réfléchir à un algorithme.

Une autre réussite est celle de l'affichage du jeu en général, qui rend le tout lisible et compréhensible sans paraître trop lourd.

Enfin, la principale réussite du projet est la structuration du code. En effet, l'organisation en différents packages du code permet une dépendance unilatérale des différents éléments. On voit dans le diagramme UML que le package « entités » peut exister seul, alors que le package « composants » nécessite l'import de la classe « Animal ». Le package « jeu » nécessite les deux packages précédents, et le main (package « Appli ») n'utilise que le package « jeu ». Il existe également un package « test », qui contient tous les tests unitaires des classes.

## Pistes d'amélioration

---

Une piste d'amélioration du projet serait de minimiser l'utilisation de l'animal nul (`Animal.NULL`). En effet, actuellement, lorsqu'un podium est « vide », il est en réalité rempli d'animaux nuls ; c'est donc plus facile à coder, mais moins optimisé.

Une autre piste d'amélioration serait d'intégrer une « customisation » du jeu par l'utilisateur, c'est-à-dire qu'il pourrait par exemple modifier les animaux présents dans le jeu, modifier la taille des podiums, etc.

## Court bilan

---

Réaliser ce projet fut une expérience enrichissante au niveau de la conception d'applications, mais également au niveau de l'algorithmique.