

## SAE 1.01 - Implémentation jeu « The Game »

---

Ce document décrit le sujet de la SAE « *S1.01 - implémentation d'un besoin client* ». Le sujet se basera sur les contenus du TP10 (pour la gestion des tableaux avec des objets) et du TP11 (pour la gestion d'un paquet de cartes).



### Consigne

- **Durée** : 12h de travail étudiant.
- **Format** : travail en binôme.
- **Rendus attendus** :
  - **Après 6h de SAÉ** :
    - une première version intermédiaire à déposer sur arche ;
    - avec vos classes [Carte](#) et [PaquetCartes](#).
  - **Après 12h de SAÉ** :
    - un code commenté qui compile ;
    - des classes de test ;
    - un rapport qui présente votre réalisation.

Avant toute chose, nous vous conseillons de lire la section 7 décrivant le contenu attendu du rapport pour ne pas oublier des éléments lors de votre réalisation.



### Consigne

N'oubliez pas de sauvegarder régulièrement votre projet (sauvegarde sur votre espace personnel, envoi par mail entre vous, utilisation d'outil de partage, ...) afin de ne pas perdre votre travail entre deux séances. Aucune perte de travail ne pourra justifier une absence de rendu de la SAÉ.

L'objectif de la SAÉ va être de programmer le jeu « The Game », un jeu solo (aussi jouable à plusieurs) conçu par Steffen Benndorf et sélectionné au fameux prix international le « *Spiel des Jahres* » en 2015. Le jeu est actuellement édité en France par *Oya*<sup>1</sup>.

---

1. <http://www.oya.fr/?post/2015/06/10/76-the-game-le-jeu-n-est-pas-votre-ami>

## 1 Présentation des règles du jeu « The Game »

Ce jeu repose sur 98 cartes numérotées de 2 à 99 que le joueur doit toutes poser sur la table pour gagner la partie. Au début de la partie, quatre piles de cartes sont créées sur la table (cf figure 1), le joueur pioche une main de 8 cartes et le reste des cartes forme la pioche.



FIGURE 1 – Exemple de partie en cours avec des cartes déjà jouées sur les 4 piles. Les deux piles de gauche sont des piles de cartes croissantes (précisé par la carte à leur gauche allant de 1 à 99) et les deux piles de droite sont des piles de cartes décroissantes (précisé par la carte à leur gauche allant de 100 à 2).

**\*Règle 1.\*** À son tour, le joueur doit poser **deux cartes** de sa main sur les piles de cartes de son choix puis tire les premières cartes de la pioche pour compléter sa main à 8 cartes.

**\*Règle 2.\*** **Cependant**, le joueur ne peut pas poser ses cartes sur n'importe quelle pile. Les deux premières piles (à gauche sur la figure) doivent contenir les cartes dans l'ordre croissant (les valeurs doivent monter) et les deux dernières piles (à droite sur la figure) doivent contenir les cartes dans l'ordre décroissant (les valeurs doivent descendre).

Par exemple, sur l'image 1, les piles avec les cartes  $C_{22}$  et  $C_{34}$  sont des piles croissantes (les cartes ne peuvent que monter) et les piles avec les cartes  $C_{56}$  et  $C_{77}$  sont des piles décroissantes (les cartes ne peuvent que descendre). Ainsi, si le joueur souhaite placer la carte  $C_{62}$ ,

- il peut la placer sur la pile croissante avec la carte  $C_{22}$  (car  $62 > 22$ ) ;
- il peut la placer sur la pile croissante avec la carte  $C_{34}$  (car  $62 > 34$ ) ;
- il peut la placer sur la pile décroissante avec la carte  $C_{77}$  (car  $62 < 77$ ) ;
- **mais**, il ne peut pas la poser sur la pile avec la carte  $C_{56}$  car la pile est décroissante et que  $C_{62}$  est supérieure à  $C_{56}$ .

De plus, si le joueur décide de poser sa carte  $C_{62}$  sur la pile  $C_{22}$ , il risque de se retrouver bloqué dans la suite de la partie car il ne pourra désormais poser sur cette pile que des cartes supérieures à  $C_{62}$ .

**\*Règle 3.\*** Il existe cependant **une exception** qui permet de ne pas respecter le sens des piles. Si la carte posée a une valeur **exactement** égale à la valeur de la pile plus ou moins 10, elle peut être posée sur la pile correspondante sans respecter le sens de la pile. C'est le seul moyen de remonter une pile décroissante ou de baisser une pile croissante.

Par exemple, sur l'image 1, le joueur peut poser la carte  $C_{87}$  sur la pile avec la carte  $C_{77}$ . La pile est censée être décroissante, mais la carte  $C_{87}$  a exactement 10 de plus que la carte  $C_{77}$ . De la même manière, le joueur peut poser la carte  $C_{12}$  sur la carte  $C_{22}$  (même si la pile est censée être croissante), la carte  $C_{24}$  sur la carte  $C_{34}$  ou la carte  $C_{66}$  sur la carte  $C_{56}$ .

**\*Règle 4.\*** Quand il ne reste plus de cartes dans la pioche, le joueur ne complète pas sa main et continue à jouer avec moins de cartes.

**\*Règle 5.\*** La partie s'arrête si le joueur a vidé toute la pioche et toute sa main, il a alors gagné. La partie s'arrête aussi si le joueur ne peut plus jouer de cartes sur aucune pile (car aucune carte ne peut être posée en respectant les règles), le joueur a alors perdu.

Par exemple, si les piles croissantes ont pour dernière carte  $C_{87}$  et  $C_{56}$  et que les piles décroissantes ont pour dernière carte  $C_{12}$  et  $C_{24}$ , le joueur ne peut pas poser la carte  $C_{48}$  (elle est plus grande que  $C_{12}$  et  $C_{24}$  et elle est plus petite que  $C_{56}$  et  $C_{87}$ ).

L'ensemble des règles officielles du jeu est disponible en français à cette adresse <https://cdn.1j1ju.com/medias/61/00/8b-the-game-regle.pdf>. Le sujet de SAE propose juste une variante demandant au joueur de poser **exactement** deux cartes avant de repiocher (à la différence des règles officielles)

## 2 Organisation de la SAÉ

La SAÉ est organisée par étapes. Chaque étape consiste à écrire certaines fonctionnalités et les tests associés.



### Consigne

Les étapes seront progressivement validées. Ne passez à l'étape suivante que quand une étape est validée (code complet et commenté et classes de test écrites et validées).



### Important

Un premier dépôt est à faire à la fin de l'étape 2 après 6h de SAÉ.

## 3 Étape 1 : Classe `Carte` (1h)

La classe `Carte` a pour objectif de représenter une carte du jeu « The Game ». Une carte doit posséder :

- un attribut `valeur` de type `int` représentant la valeur de la carte ;
- un getter `getValeur` retournant la valeur de la carte ;
- un constructeur avec un paramètre de type `int` qui construit une carte dont l'attribut `valeur` correspond au paramètre passé ;
- une méthode `toString()` qui retourne une chaîne contenant la valeur de la carte précédée de la lettre '`c`' comme `c27` ;
- une méthode `etrePlusGrand` qui prend en paramètre une autre carte et retourne `true` si et seulement si la carte `this` est plus grande que la carte passée en paramètre ;
- une méthode `avoirDiffDe10` qui prend une autre carte en paramètre et qui retourne `true` si et seulement si la carte `this` a une différence d'exactly 10 (dans un sens ou dans l'autre) avec la carte passée en paramètre.



### Question 3.1

Écrire la classe `Carte`.



### Question 3.2

Compléter la classe de test `TestCarte.java` chargée de vérifier que les méthodes de la classe `Carte` fonctionnent correctement.

## 4 Étape 2 : Classe `PaquetCartes` (4h)

L'objectif de cette partie est de pouvoir disposer de différents paquets de cartes pour représenter la pioche, la main de cartes du joueur et une partie des piles manipulables.

### 4.1 Constructeur, ajout et suppression de carte

Le TP11 vous a proposé d'écrire une classe destinée à gérer un paquet de cartes. Cette classe doit posséder

- un constructeur à partir d'un tableau de cartes ;
- une méthode `ajouterCarteFin` qui ajoute une carte à la fin d'un paquet ;
- une méthode `retirerCarte(int indice)` qui retire la carte à la position indice du paquet <sup>2</sup>.



#### Question 4.1

Adapter et finir le TP11 pour disposer d'une classe `PaquetCartes` permettant de manipuler un paquet de cartes. Vous penserez à changer la classe `Carte` utilisée.

### 4.2 Constructeur par défaut

On souhaite pouvoir construire l'ensemble des cartes du jeu « The Game ».

Le constructeur par défaut de la classe `PaquetCartes` doit construire un jeu possédant 0 cartes (un tableau de taille 0).



#### Question 4.2

Ecrire le constructeur par défaut de `PaquetCartes`.

### 4.3 Remplir un paquet de cartes

Pour avoir un paquet par défaut, écrire une méthode `remplir` qui prend un entier `max` en paramètre et qui remplit le paquet en créant un tableau possédant dans l'ordre toutes les cartes de  $C_2$  à  $C_{max-1}$ .

Par exemple, pour `max` égal à 100, le paquet doit posséder 98 cartes de  $C_2$  à  $C_{99}$ .

---

2. attention, `indice` doit être un entier compris entre 0 et la taille du paquet moins un



#### Question 4.3

| Écrire la méthode `remplir`.

### 4.4 Constructeur autre

On souhaite aussi disposer d'un constructeur qui construit un jeu à partir d'un tableau d'entiers donné. Cela permet de construire des paquets de cartes dont l'ordre des cartes est connu pour pouvoir facilement faire des tests.



#### Question 4.4

| Écrire un constructeur de `PaquetCartes` qui prend un tableau d'entier en paramètre et construit un paquet de cartes contenant les cartes ayant pour valeur les entiers passés dans le tableau.

### 4.5 Accéder à des caractéristiques du paquet

Les méthodes suivantes sont proches d'accesseurs (getter) :

- la méthode `getCarte(int i)` doit retourner la carte à la position `i` d'un paquet (`null` si l'indice est incorrect). La carte reste dans le paquet ;
- la méthode `getDerniereCarte()` doit retourner la carte à la dernière position du paquet (`null` si le paquet est vide). La carte reste dans le paquet ;
- la méthode `getNbCartes()` doit retourner le nombre de cartes du paquet ;
- la méthode `etreVide()` doit retourner un `boolean` qui vaut vrai si et seulement si le paquet est vide.



#### Question 4.5

| Ajouter ces méthodes à la classe `PaquetCartes` si elles ne sont pas présentes.

### 4.6 Piocher au hasard une carte

On souhaite disposer d'une méthode permettant de piocher une carte aléatoirement dans un paquet. Cette méthode ne prend aucun paramètre, retourne une carte tirée au hasard parmi les cartes du paquet et la supprime du paquet. Si le paquet est vide, la méthode retourne simplement `null`.

Afin de sélectionner une carte au hasard, on pourra utiliser la classe JAVA `Random`. Cette classe :

- nécessite d'importer le package `java.util.Random` tout en haut de la classe `PaquetCartes` ;
- possède un constructeur vide ;
- possède une méthode `int nextInt(int bound)` qui retourne un entier choisi de manière aléatoire et compris entre 0 et `bound-1` (compris).

Le descriptif complet de la classe est disponible à l'adresse <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Random.html>.



#### Question 4.6

Ajouter une méthode `private piocherHasard` à la classe `PaquetCartes`. Cette méthode ne servira **que pour** mélanger un paquet (cf question suivante).

### 4.7 Mélanger un paquet de cartes

Il n'est pas forcément facile de mélanger un paquet de cartes pour que les cartes soient effectivement réparties de manière aléatoire dans un paquet. La méthode la plus simple pour mélanger un paquet de cartes consiste

- créer un paquet vide ;
- répéter tant qu'il reste des cartes dans le paquet initial
  - tirer une carte au hasard du paquet initial ;
  - l'ajouter au nouveau paquet ;
- échanger les tableaux de carte entre les paquets.



#### Question 4.7

En utilisant la méthode `piocherHasard`, écrire la méthode `void melangerPaquet()`.

### 4.8 Insérer une carte de manière triée

On souhaite présenter la main du joueur de manière triée pour l'aider. Pour cela, on souhaite disposer une méthode qui permette d'insérer une carte à la bonne place dans un paquet déjà trié.



#### Question 4.8

Écrire une méthode `insérerTri` qui prend une carte en paramètre et l'insère à la bonne place dans le paquet supposé déjà trié.

## 4.9 Prendre carte dessus

On souhaite pouvoir piocher la carte en haut du paquet de cartes. On supposera qu'il s'agit de la carte d'indice 0.

À l'aide de la méthode `retirerCarte`, la méthode `prendreCarteDessus` doit retirer la carte à l'indice 0 du paquet et la retourner.

Cette méthode retournera `null` si le paquet est vide.



### Question 4.9

Écrire la méthode `prendreCarteDessus`.

## 4.10 Méthode toString

On souhaite avoir un affichage d'un objet `PaquetCartes` par défaut. Cet affichage consiste à afficher toutes les cartes du paquet une à une sous la forme `<INDICE1>-<CARTE1>` `<INDICE2>-<CARTE2>` où les indices et les cartes sont les cartes de la main du joueur.

Par exemple, si le paquet possède les cartes `C2`, `C8`, `C24`, `C36`, la main du joueur doit s'afficher sous la forme :

```
1 0-c2 1-c8 2-c24 3-c36
```



### Question 4.10

Compléter la méthode `toString()` proposée dans le TP11 pour pouvoir afficher toutes les cartes d'un paquet de cartes sous la forme demandée.

## 4.11 Tests



### Question 4.11

Compléter la classe de test `TestPaquet.java` chargée de vérifier que les différentes méthodes écrites fonctionnent. Vous pourrez vous inspirer des tests proposés dans la classe `TestPaquet` fournie dans le TP11.



### Consigne

Valider complètement cette partie avant de passer à la suite.





### Important

Une fois cette première partie validée, **déposer sur arche cette version correspondant à la version 1** de votre projet. Ce dépôt doit être effectué à l'issue des 6 premières heures.

## 5 Étape 3 : Classe `PileCartes` (1h)

L'objectif de la classe `PileCartes` est de représenter une des piles sur laquelle poser des cartes.

### 5.1 Attributs et constructeurs

Il existe deux types de `PileCartes`, les piles croissantes et les piles décroissantes. Pour les distinguer, chaque pile possèdera un booléen précisant si la pile est dans le sens croissant (les valeurs des cartes doivent monter) ou non (les valeurs des cartes doivent descendre).

La classe `PileCartes` possède

- un attribut booléen `croissant` qui vaut `true` si les valeurs des cartes posées sur la pile doivent être dans l'ordre croissant et `false` si les valeurs des cartes posées sur la pile doivent être dans l'ordre décroissant ;
- un attribut `paquet` de type `PaquetCartes` qui correspond aux cartes de la pile – la dernière carte du paquet correspond à la carte visible en haut de la pile ;
- un constructeur qui prend en paramètre un booléen `pCroissant` et un entier `max`. L'entier `max` correspond à la taille du jeu (100 par défaut). Ce constructeur construit une pile dans le sens précisé et qui ne contient qu'une seule carte qui dépend du sens (`C1` si le sens est croissant ou `Cmax` si le sens est décroissant).



### Question 5.1

1 Déclarer la classe `PileCartes`, ses attributs et son constructeur.

### 5.2 Vérifier si une Carte est ajoutable à une pile

Pour rappel, une carte peut être posée sur une pile si et seulement si la valeur de cette carte est valide par rapport au sens de la pile et à sa dernière carte de cette pile (croissant, décroissant - cf **\*Règle 2\***). Il est aussi possible de poser une carte sur une pile lorsque la différence de valeur entre la carte et la dernière carte de la pile est exactement de 10 (cf **\*Règle 3\***).



### Question 5.2

Écrire la méthode `boolean etrePosable(Carte c)` qui permet de vérifier si la pose de la carte `c` sur la pile est valide. Cette méthode ne fait pas l'ajout mais retourne simplement un booléen qui vaut `true` si et seulement si la carte peut être posée sur la pile.

## 5.3 Ajouter une Carte à une pile

On souhaite ajouter une carte à une pile tout en vérifiant que le fait de poser cette carte est bien correct.

La méthode `poserCarte` de la classe `PileCartes` prend une carte en paramètre, essaie de la poser sur la pile et retourne un `boolean`. Cette méthode fonctionne de la manière suivante :

- si la pose de la carte passée en paramètre est valide, la carte est ajoutée à la pile et la méthode retourne `true` ;
- si la pose de la carte passée en paramètre n'est pas valide, la méthode retourne `false` et la carte n'est pas ajoutée à la pile.

Lorsque la méthode `poserCarte` s'effectue, elle doit ajouter la carte en fin du paquet de la pile (c'est la carte qui sera visible par le joueur).



### Question 5.3

Écrire la méthode `poserCarte`.

## 5.4 Méthode `toString()`

La méthode `toString()` a pour objectif d'afficher une pile.

L'affichage d'une pile se limitera

- au sens de la pile sous la forme d'un caractère '`c`' pour croissant et '`d`' pour décroissant ;
- la carte située en fin du paquet de carte (c'est à dire la dernière carte a avoir été ajoutée au paquet - vous utiliserez la méthode `getDerniereCarte`) ;
- le nombre de cartes dans la pile (entre parenthèses).

Par exemple, si la pile est croissante avec pour dernière carte la carte `C15` et que la pile contient 4 cartes, la méthode `toString` doit retourner la chaîne "`c-c15-(4)`".



#### Question 5.4

Écrire la méthode `toString()`.

### 5.5 Tests



#### Question 5.5

Pour effectuer les tests plus facilement, ajouter la méthode `getDerniereCarte` qui retourne la dernière carte du paquet de la pile (la carte qui a été posée le plus récemment).



#### Question 5.6

Compléter la classe de test `TestPileCartes` pour vérifier que vos méthodes de la classe `PileCartes` fonctionnent correctement.

## 6 Étape 4 : Mise en place et exécution du jeu (4h)

### 6.1 Attributs

La classe `Jeu` permet de représenter un jeu complet. Un `Jeu` possède 3 attributs :

- un attribut `main` de type `PaquetCartes` correspondant à la main du joueur ;
- un attribut `pioche` de type `PaquetCartes` correspondant à la pioche des cartes restantes ;
- un attribut `piles` de type tableau de `PileCartes` correspondant aux quatre piles du jeu. Les deux premières piles devront être dans l'ordre croissant et les deux piles suivantes dans l'ordre décroissant.



#### Question 6.1

Déclarer la classe `Jeu` et écrire un `getter` pour chaque attribut dans le but de faciliter les tests.

### 6.2 Constructeur

Le constructeur de la classe `Jeu` prend en paramètre un entier `max` et a pour objectif de mettre en place une partie standard dans un jeu de `max` cartes. Il doit créer la pioche mélangée des cartes, les différentes piles et créer la main initiale du joueur (composée de 8 cartes) triée dans l'ordre des cartes.

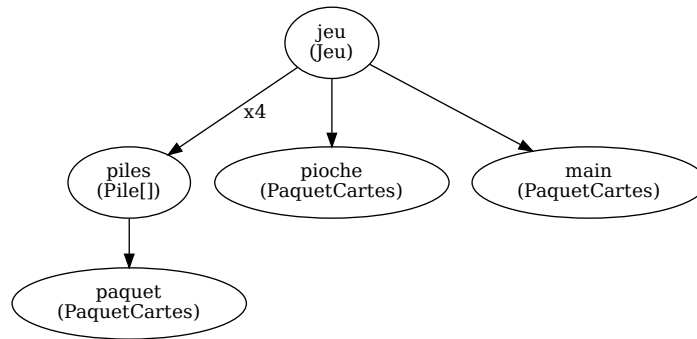


FIGURE 2 – Représentation des attributs de la classe **Jeu**

Si **max** vaut 20, la pioche initiale est constitué de 18 cartes (de **C<sub>2</sub>** à **C<sub>19</sub>**), les piles ont des cartes de valeur **C<sub>1</sub>** ou **C<sub>20</sub>** (en fonction de leur sens). L'intérêt de créer un jeu avec moins de cartes (par exemple 15) est de pouvoir le tester facilement dans un **main** sans devoir faire une partie complète.



### Question 6.2

Écrire le constructeur.

## 6.3 Méthode `toString`

La méthode `toString` doit afficher l'état du jeu à l'instant courant. Cela consiste à afficher plusieurs lignes dans l'ordre

- une ligne pour chaque pile affichant le numéro de la pile et son descriptif (sens de la pile, carte au dessus de la pile et nombre de carte de la pile comme demandé dans la classe **PileCartes**) ;
- une ligne donnant le nombre de cartes restant dans la pioche entre parenthèses ;
- les dernières lignes correspondant à l'affichage de la main du joueur (comme vu précédemment).

Le retour de la méthode n'est pas défini strictement (on ne le testera pas à la lettre dans l'évaluation du projet) mais devra respecter les contraintes ci-dessus. Un exemple de retour possible donnera l'affichage suivant :



### Affichage console

```

#####
- PILE 0 : c-c1(1)
- PILE 1 : c-c1(1)
- PILE 2 : d-c100(1)
- PILE 3 : d-c100(1)
  
```

```
#####
Reste 90 cartes dans la pioche
#####
Main du joueur :
0-c9 1-c24 2-c28 3-c42 4-c44 5-c51 6-c55 7-c60
#####
```



### Question 6.3

| Ecrire la méthode `toString()`.

## 6.4 Constructeur avec un paquet de cartes

On souhaite pouvoir créer un jeu à partir d'un paquet de carte déjà donné dont l'ordre est connu (ce qui permet de pouvoir faire facilement des tests). Cela se programmera avec un constructeur de `Jeu` qui utilise un `PaquetCartes` en paramètre. Ce constructeur doit initialiser le jeu en utilisant les cartes et l'ordre des cartes dans le paquet fourni.



### Question 6.4

| Ecrire le constructeur avec un `PaquetCartes` en paramètre.

## 6.5 Méthode jouerCarte

La méthode `boolean jouerCarte(int indice, int numPile)` permet au joueur de jouer la carte d'indice `indice` de sa main sur la pile de numéro `numPile`. Cette méthode est en charge de faire évoluer le jeu en conséquence. S'il est possible de poser la carte sur la pile souhaitée en respectant les règles, l'opération se fait, le joueur complète sa main et la méthode retourne `true`. Sinon, rien ne se passe et la méthode doit retourner `false`.



### Question 6.5

| Écrire la méthode `jouerCarte` en fonction des autres classes et méthode déjà écrites.

## 6.6 Méthode `etreFini()`

Cette méthode a pour objectif de vérifier que le jeu est fini qu'il ait été gagné ou perdu (cf **\*Règle 5.\***). Elle retourne un entier qui vaut

- soit 0 si le jeu n'est pas fini ;
- soit 1 si le jeu est fini et la partie gagnée ;
- soit -1 si le jeu est fini et la partie perdue.

Pour rappel, la partie s'arrête si le joueur ne peut plus jouer de cartes (il a alors perdu) ou si le joueur a placé toutes ses cartes et que la pile est vide (il a alors gagné).

### Question 6.6

Écrire la méthode `int etreFini()`.

## 6.7 Méthode `lancerJeu`

La méthode `lancerJeu` est en charge de gérer toute la logique du jeu.

- elle affiche l'état du jeu ;
- elle demande les actions du joueur une par une (deux cartes à poser) ;
- elle met à jour l'état du jeu en fonction des actions du joueur ;
- elle continue tant que le jeu n'est pas fini ;
- enfin, à la fin du jeu, elle précise si le joueur a gagné ou perdu.

La méthode `lancerJeu` est aussi en charge de gérer les saisies clavier de l'utilisateur et d'afficher à l'écran les attentes au clavier. Elle pourra utiliser un `Scanner` et précisera une erreur lorsque le joueur essaie de jouer un coup impossible.

Pour simplifier l'écriture du code, vous pouvez écrire une méthode `completerMain` qui complète la main du joueur à 8 cartes qui sera appelée à chaque fois que le joueur aura joué deux cartes.

### Question 6.7

Écrire des commentaires qui expliquent le déroulement du jeu (et les méthodes à utiliser) dans la méthode `lancerJeu`.


### Question 6.8

Traduire ces commentaires pour écrire la méthode `lancerJeu` qui déroule l'exécution du jeu jusque la fin de la partie.

### Question 6.9

Écrire un `main` qui lance le jeu.

## 6.8 Exemple de déroulement de partie avec 100 cartes

 Affichage console

```
#####
- PILE 0 : c-c1(1)
- PILE 1 : c-c1(1)
- PILE 2 : d-c100(1)
- PILE 3 : d-c100(1)
#####
Reste 90 cartes dans la pioche
#####
Main du joueur :
0-c10 1-c21 2-c26 3-c37 4-c43 5-c55 6-c56 7-c94
#####

Quelle carte poser ?
7
Quelle pile ?
3
#####
- PILE 0 : c-c1(1)
- PILE 1 : c-c1(1)
- PILE 2 : d-c100(1)
- PILE 3 : d-c94(2)
#####
Reste 90 cartes dans la pioche
#####
Main du joueur :
0-c10 1-c21 2-c26 3-c37 4-c43 5-c55 6-c56
#####

Quelle carte poser ?
0
Quelle pile ?
4
**ERREUR** pile inexistante
Erreur, l'action n'est pas possible !!
#####
- PILE 0 : c-c1(1)
- PILE 1 : c-c1(1)
- PILE 2 : d-c100(1)
- PILE 3 : d-c94(2)
#####
Reste 90 cartes dans la pioche
#####
Main du joueur :
0-c10 1-c21 2-c26 3-c37 4-c43 5-c55 6-c56
#####

Quelle carte poser ?
```

## 6.9 Tests



### Question 6.10

Compléter la classe `TestJeu` permettant de tester les méthodes de la classe `Jeu`.

## 7 Rendu attendu

### 7.1 Version 1

La version 1 de votre projet a du être déposée à la fin de la classe `Carte`. Elle ne constitue qu'une étape intermédiaire de votre rendu.

### 7.2 Version 2

La version 2 de votre application est une version complète de votre projet. Elle doit contenir

- votre code complet (classes, classes de test, classes main attendues) ;
- un code commenté (commentaires internes, javadoc) ;
- un petit compte-rendu qui présente votre travail.

Ce compte-rendu devra préciser

- les noms, prénoms et groupe des membres du binôme ;
- l'avancée de votre projet (à quelle partie en êtes vous) ;
- un déroulé de votre travail (combien de temps pour chacune des parties réalisées) ;
- les difficultés éventuelles rencontrées lors du travail (éventuellement les questions que vous n'avez pas faites) ;
- la manière dont vous avez géré les différents problèmes rencontrés ;
- quelques explications sur la manière dont vous avez programmé la classe `Jeu` ;
- ce que vous avez appris en effectuant cette SAÉ.

L'objectif du compte rendu est de prendre le temps de détailler vos difficultés et de montrer que vous avez bien compris le sujet. Il ne s'agit de faire des discours abstraits mais d'expliquer en détail ce que vous avez découvert sur CE projet et les éléments qui vous ont pris du temps sur CE sujet (et pour quelle raison).



### Consigne

IMPORTANT : Avant de passer à la suite, sauver une version 2 de votre application dans un fichier zip et déposer cette version sur arche.



### 7.3 Version 3

Une fois votre version 2 déposée sur arche, copier cette version et la renommer en une version v3 pour aborder les questions optionnelles qui arrivent ensuite. Cette nouvelle version servira de base à la suite de votre travail pour éviter les confusions.

Si vous répondez à des questions optionnelles, vous penserez à déposer votre version 3 dans le second dépôt arche.



#### Important

Les questions optionnelles pourront vous donner quelques points bonus si vous avez fini les parties précédentes. **Ne débutez pas cette partie, si votre projet n'est pas complet ni correctement finalisé.** Les parties suivantes ont pour objectif de vous permettre de continuer à travailler sur la SAE si vous avez fini votre projet avant les 12 heures.

## 8 Étape 5 : Options de jeu (partie optionnelle)

Une fois la version 2 complétée, différentes extensions sont possibles pour améliorer votre jeu. Si vous avez le temps, vous pouvez développer une (ou plusieurs) de ces extensions.

### 8.1 Gestion du score

Lorsque le joueur perd, affichez son score. Ce score correspond aux nombres de cartes restant à poser.

### 8.2 Sauvegarde du score

Proposer une classe `Score` pour stocker les 5 meilleurs scores obtenus et les joueurs qui ont fait ces scores. À chaque fois qu'un joueur fait un score devant être enregistré dans ce palmarès, le jeu demande le nom du joueur et enregistre ces informations. Le jeu se relance ensuite.

Au lancement de chaque nouvelle partie, les 5 meilleurs scores sont affichés à l'écran <sup>3</sup>

---

3. si l'application s'arrête les meilleurs scores seront perdus. On ne gèrera pas la sauvegarde de ces scores.