

# TP : Pas de jeu sans *carte* !

(enrichissons nos énumérations avec un peu d'unicode)

🔊 En fin d'énoncé, vous avez un exercice  
à rendre au début de la séance de TP de la semaine prochaine  
à faire bien sûr, en autonomie, en dehors des heures de cette séance 😊 🔊

**Cahier des charges :** Votre nouveau client souhaite vous confier le développement de différents jeux de cartes qui se jouent à partir d'un jeu de cartes à jouer de 52 cartes.

Il vous détaillera plus tard le(s) jeux qu'il souhaite vous faire développer.



Pour gagner du temps, vous décidez de vous familiariser avec ce nouveau contexte métier (celui du jeu de cartes).

Pour commencer, vous décidez de vous focaliser sur l'**élément de base** de cette application : **la carte à jouer**, puisque **sans carte, pas de jeu à développer** ! 😊

Pour mener à bien votre **phase d'analyse** autour de la carte à jouer et bien comprendre comment la modéliser, vous avez décidé de vous appuyer sur quelques pages **Wikipédia** pour commencer à vous immerger dans le contexte métier, en attendant des consignes et une définition plus précise du besoin de votre client (**voir annexe : Se familiariser avec le terminologie métier autour du jeu de cartes**)

Pour qu'un développement logiciel se passe du mieux possible, il est important que toute l'équipe ait une **compréhension commune** des **termes métiers**. Il est donc indispensable de rédiger un **glossaire** à chaque fois qu'un nouveau terme métier est employé, de mettre à jour et de se référer continuellement à ce glossaire

Pour cette étude de cas, vous pouvez écrire le glossaire en français ou directement en anglais selon votre aisance en anglais. Par contre soyez cohérent, soit vous écrivez le glossaire en français, soit en anglais, mais pas une définition dans une langue et une autre dans l'autre 😊

Si vous choisissez d'écrire le glossaire en français, les termes qui sont susceptibles de se retrouver dans le code devront être traduits en anglais comme dans l'exemple suivant...

## Glossaire

**Jeu de cartes (*card desk*)** : un jeu de cartes est composé de 52 cartes. Nous n'utiliserons pas de joker.

**Carte (*card*)** : une carte est définie par une **couleur (*suit*)** et une **hauteur (*rank*)**.

**Couleur (*suit*)** : Quatre couleurs existent :

Trèfles (♣), Carreaux (♦), Cœurs (♥), Piques (♠)  
Clubs (♣), Diamonds (♦), Hearts (♥), Spades (♠)

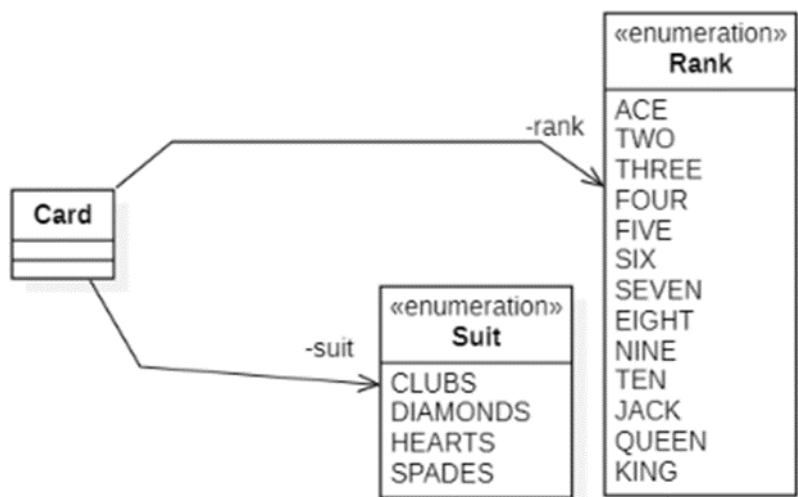
**hauteur (*rank*)** : Treize rangs sont possibles ordonnés ci-après du plus fort au plus faible :

As, Roi, Dame, Valet, 10, 9, 8, 7, 6, 5, 4, 3, 2  
Ace, King, Queen, Jack, 10, 9, 8, 7, 6, 5, 4, 3, 2

*... glossaire qui devra être complété tout au long du développement ...*

Pour jouer à la bataille, il faut absolument disposer d'un élément de base qui est la **carte (*card*)**.

En vous appuyant sur la phase d'analyse menée précédemment autour de la notion de *carte*, la **phase de conception** vous a permis de proposer le diagramme de classes suivant.



Votre équipe a décidé de vous confier **l'implémentation** de ce projet.  
A vous de jouer maintenant !

1. Créez sous Eclipse un nouveau projet java (**File**→ **New**→ **Java project**) , toujours sans module, que vous appellerez **cardgame**.

*Une fois le projet créé, avant de continuer, pour plus de confort,  
assurez-vous que tous les autres projets de votre workspace sont fermés.  
(Close Unrelated Project)*

2. Vous allez commencer par **implémenter** dans **src** les énumérations **Suit** et **Rank**.  
L'énumération est une classe particulière, les IDE permettent de créer directement une **Enumeration** dans votre projet.

2.1 Par exemple sous Eclipse, vous pouvez choisir **File** → **New** → **Enum** puis dans le champ **Name**, saisir le nom de votre énumération : **Suit** pour commencer, puis terminer par **Finish**.

L'implémentation la plus simple, conforme au diagramme de classes précédent, est la suivante.

```
public enum Suit {  
    CLUBS, DIAMONDS, HEARTS, SPADES;  
}
```

Procédez à cette implémentation et n'oubliez pas de sauvegarder votre fichier **Suit.java** 😊

2.2 De la même manière procéder à l'implémentation de l'énumération **Rank** telle que conçue dans le diagramme de classes précédent.

3. Vous pouvez maintenant procéder à l'**implémentation** de la classe **Card** en respectant la notation du diagramme de classes pour le nom et la visibilité des attributs.

Générez également un constructeur à deux paramètres (**Source**→ **Generate Constructor using Fields...**) dont le premier paramètre sera la *hauteur* de la carte (**rank**).

Seul ce constructeur est demandé pour le moment.

4. **Pour visualiser l'architecture de votre code et ainsi vérifier que votre implémentation est bien cohérente avec le modèle proposé lors de la phase de conception**, il ne vous reste plus qu'à **procéder à un reverse-engineering (rétro-conception)**, c.-à-d. générer le diagramme de classes du code juste écrit à l'aide de plant UML 😊

L'implémentation sera considérée comme correcte si le diagramme de classes issu de la phase de conception (celui de la page précédente) correspond au diagramme de classes rétro-conçu à partir du code écrit dans de la phase d'implémentation (celui dans l'IDE) .

Comparer donc ces deux diagrammes (sachant que nous n'avions pas représenté de constructeur dans le diagramme issu de la phase de conception) ! 😊

5. Pour vérifier que l'implémentation que vous venez de réaliser **valide bien le comportement attendu de votre application**, vous allez maintenant procéder à **une série de jeux d'essai (test cases)**

1. Créer une classe **CardMain** (vous pouvez cocher **public static void main(String[])** avant **Finish**, cela vous évitera de l'écrire 😊)
2. Dans un premier temps, vous allez **déclarer et instancier une carte (l'as de cœur par exemple)** en **implémentant** le code suivant :

```
public static void main(String[] args) {  
  
    Card aceOfHearts = new Card(Rank.ACE, Suit.HEARTS);  
    System.out.println("Ace of Hearts well created !");  
    System.out.println("with " + aceOfHearts.rank() + " as rank and "  
                        + aceOfHearts.suit() + " as suit.");  
}
```

.... et vérifier que la création de cette carte s'est bien passée en exécutant ce code de manière à obtenir sur la console :

---

```
Ace of Hearts well created !  
with ACE as rank and HEARTS as suit.
```

💡 Pour faire compiler le code, vous devez donc enrichir la classe **Card** avec les méthodes **rank()** et **suit()**, qui ne sont autre que les *getteurs* des deux attributs.

- ➔ Respectez bien le nommage **rank** (et non `getRank`) et faites en sorte que `rank()` renvoie bien un **Rank** et `suit()` renvoie bien un **Suit**. Le nommage adopté ici permet une meilleure lisibilité du code, on dit qu'il est *fluent* (dans le sens où il rend *la lecture du code plus fluide*).
- ➔ Pour l'instant, seuls ces *getteurs* sont nécessaires pour compiler correctement et exécuter notre jeu d'essai, inutile d'implémenter les *setteurs* tant qu'ils ne nous servent pas 😊  
Pour l'instant, ces attributs sont donc .....  
Si ce n'est déjà fait, quel petit **mot-clé** devez-vous ajouter devant les attributs de la classe **Card** ?

## 6. Enrichir les énumérations avec des constructeurs privés...

Les types énumérés Java peuvent posséder des attributs, des méthodes et des constructeurs (privés uniquement).

L'exemple ci-dessous vous montre la syntaxe :

```
public enum Season {
    SPRING("printemps"),
    SUMMER("été"),
    AUTUMN("automne"),
    WINTER("hiver");

    private String frenchName;

    private Season(String frenchName) {
        this.frenchName = frenchName;
    }

    public String frenchName() {
        return this.frenchName;
    }
}
```

Si vous exécutez ensuite dans un **main** :

```
Season ete = Season.SUMMER;
System.out.println("La saison choisie est : " + ete.frenchName());
```

Vous obtiendrez comme affichage : **La saison choisie est : été**

Un autre exemple peut être consulté sur :

<http://blog.paumard.org/cours/java/chap04-structure-classe-enumeration.html>

Dans la suite, vous allez enrichir vos énumérations Suit et Rank en associant un **code** à chaque couleur et hauteur pour pouvoir afficher la carte sous forme de texte.

### 1. Enrichir l'énumération Suit avec un code pour chaque couleur

→ **Afficher un caractère spécial en java et vérifier que l'UTF-8 est bien pris en compte :**

A la fin de la méthode main de la classe CardMain écrivez l'instruction suivante :

```
System.out.println("\u2764");
```

Compilez et exécutez le **main**. Quel est le dernier caractère qui s'affiche sur la console : \_\_\_\_ ?

- Si le dernier caractère affiché est un cœur, vous pouvez continuer 😊
- Si vous voyez un **?**, vous devez modifier le paramétrage d'Eclipse pour que l'IDE puisse prendre en charge l'UTF-8 et reconnaisse le code de l'unicode comme tel. Pour cela, procédez de la manière suivante :

Allez dans **Windows** → **Preferences**

puis déroulez **General** pour pouvoir ensuite sélectionner **Workspace**

**En bas à gauche**, dans la dernière rubrique intitulée **Text File encoding**, cochez **Other** et sélectionnez **UTF-8**. Cliquez sur **Apply** puis sur **Apply and Close**.

Exécutez à nouveau votre jeu d'essai, vous devriez cette fois-ci bien visualiser le ♥ !

### → A propos de l'unicode :

Chaque caractère ASCII est codé sur 7 bits, permettant d'avoir 128 caractères basés sur l'alphabet anglais (ASCII signifiant American Standard Code for Information Interchange).

Or pour tous les systèmes d'écriture utilisant d'autres caractères, il a fallu créer d'autres jeux de caractères comme les différents ISO-xxxx, KOI8, JIS-yyyy, Windows-zzzz, etc., dont la plupart codent chaque caractère sur un octet. Mais ce n'était pas une solution idéale. Chaque application internationalisée devait gérer plusieurs jeux de caractères.

Finalement, à la fin des années 80 un nouveau standard, l'**Unicode**, a été inventé pour pouvoir coder tous les systèmes d'écriture modernes.

L'Unicode permet de définir 1 114 112 points de code de **0x0000 à 0x10FFFF**.

Généralement **un point de code est noté U+xxxx où xxxx est en hexadécimal, et comporte 4 à 6 chiffres**. Par exemple, U+0041 correspond à la lettre **A**.

**La notation \uXXXX permet d'écrire en Java, un caractère ou une chaîne de caractères sous forme d'une séquence de points de code.**

Remarque : Ces phrases sont extraites de :

<https://laethy.developpez.com/tutoriels/java/jvm/unicode-et-java/>

Si vous voulez en savoir plus, n'hésitez pas à consulter cet article en entier, ainsi que

<https://www.javatpoint.com/unicode-system-in-java>

<https://www.imdoudoux.fr/java/dej/chap-encodage.htm>

### → L'unicode à la rescousse de nos couleurs !

Pour commencer, vous allez enrichir l'énumération **Suit** en **associant un symbole à chaque couleur** :

- à **CLUBS** devra être associé ♣ (ou plutôt le code *unicode* correspondant à ce symbole 😊)
- à **DIAMONDS** devra être associé ♦ (ou plutôt le code *unicode* correspondant à ce symbole 😊)
- à **HEARTS** devra être associé ♥ (ou plutôt le code *unicode* correspondant à ce symbole 😊)
- à **SPADES** devra être associé ♠ (ou plutôt le code *unicode* correspondant à ce symbole 😊)

Les codes suivants pourront par exemple être utilisés pour les couleurs de nos cartes :

\u2663 pour ♣

\u2666 pour ♦ ou \u2662 pour ◇

\u2665 pour ♥ ou \u2661 pour ♡

\u2660 pour ♠

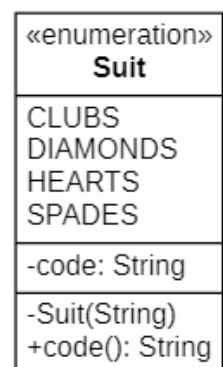
Remarque : N'hésitez pas à faire une recherche « table unicode » à l'aide de votre moteur de recherche préféré pour retrouver tous ces codes et bien d'autres 😊

### → Implémenter le code couleur dans Suit :

Comme le montre le diagramme de classes ci-contre, le **nouveau besoin** de pouvoir afficher la couleur d'une carte par son symbole, nous amène à **faire évoluer la conception de l'énumération Suit** par :

- l'introduction d'un **attribut privé** de type **String** nommé **code** et qui a la responsabilité de connaître l'*unicode* correspondant à la couleur souhaitée
- l'ajout d'une méthode **code()** qui n'est autre que le *getteur* de l'attribut précédent.

En suivant ce diagramme de classes et en tenant compte de tout ce qui précède, **implémenter dans l'énumération Suit l'ajout du code**.



## → Vérifier l'implémentation du code couleur à l'aide d'un nouveau jeu d'essai

Pour vérifier que le nouveau besoin autour de l'**unicode** a correctement été implémenté, vous pouvez compléter la classe **CardMain** avec le jeu d'essai suivant :

```
System.out.println("-----");
System.out.println("Transform each suit element into right code");
System.out.println("-----");
System.out.println("The display of " + Suit.CLUBS + " is " + Suit.CLUBS.code());
System.out.println("The display of " + Suit.DIAMONDS + " is " + Suit.DIAMONDS.code());
System.out.println("The display of " + Suit.HEARTS + " is " + Suit.HEARTS.code());
System.out.println("The display of " + Suit.SPADES + " is " + Suit.SPADES.code());
```

Ecrivez ces instructions à la fin de la méthode **main** à la place : `System.out.println("\u2764");`  
Compilez et exécutez de manière à obtenir le jeu d'essai suivant en fin de console :

```
-----
Transform each suit element into right code
-----
The display of CLUBS is ♣
The display of DIAMONDS is ♦
The display of HEARTS is ♥
The display of SPADES is ♠
```

## → Qualité de code : Améliorer la lisibilité du main de CardMain

Sélectionnez le bloc de code que vous venez d'écrire à la question précédente, faites un clic-droit, choisissez **Refactor→Extract Method**.

Dans le champ **Method name**, entrez : **transformEachSuitElementIntoRightCode**, puis cliquez sur **OK**.

Votre méthode **main** devient alors :

```
public static void main(String[] args) {

    Card aceOfHearts = new Card(Rank.ACE, Suit.HEARTS);
    System.out.println("Ace of Hearts well created !");
    System.out.println("with " + aceOfHearts.rank() + " as rank and "
                        + aceOfHearts.suit() + " as suit.");

    transformEachSuitElementIntoRightCode();
}
```

Compilez, exécutez pour vérifier que votre code fonctionne toujours 😊

Utilisez une nouvelle fois le **Refactor→Extract Method** afin que la méthode **main** se résume à :

```
public static void main(String[] args) {

    createCard();
    transformEachSuitElementIntoRightCode();
}
```

Compilez, exécutez pour vérifier que votre code fonctionne et a toujours le même comportement :  
l'affichage des jeux d'essais sur la console doit être le même que précédemment 😊

## 2. Enrichir l'énumération Rank avec un code pour chaque hauteur

De la même manière que précédemment, vous allez enrichir l'énumération **Rank** en associant un code à chaque hauteur qui doit respecter les spécifications suivantes :

- à **ACE** devra être associé **1**
- à **TWO** devra être associé **2**
- à **THREE** devra être associé **3**
- à **FOUR** devra être associé **4**
- à **FIVE** devra être associé **5**
- à **SIX** devra être associé **6**
- à **SEVEN** devra être associé **7**
- à **EIGHT** devra être associé **8**
- à **NINE** devra être associé **9**
- à **TEN** devra être associé **10**
- à **JACK** devra être associé **J**
- à **QUEEN** devra être associé **Q**
- à **KING** devra être associé **K**

«enumeration» Rank
ACE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE TEN JACK QUEEN KING
-code: String
-Rank(String) +code(): String

→ En vous inspirant de ce qui a été fait précédemment, **modifiez l'implémentation de l'énumération Rank** de manière à ce qu'elle corresponde à la conception ci-contre qui prend désormais en compte un **code** pour la hauteur.

→ Dans la classe **CardMain**, implémentez maintenant une méthode qui sera appelée à la fin de la méthode **main** et qui permettra d'exécuter le jeu d'essai ci-contre.

Vous appellerez cette méthode :

**transformEachRankElementIntoRightCode()**

```
-----  
Transform each rank element into right code  
-----  
The display of ACE is 1  
The display of TWO is 2  
The display of THREE is 3  
The display of FOUR is 4  
The display of FIVE is 5  
The display of SIX is 6  
The display of SEVEN is 7  
The display of EIGHT is 8  
The display of NINE is 9  
The display of TEN is 10  
The display of JACK is J  
The display of QUEEN is Q  
The display of KING is K
```

## 7. Afficher une carte sous forme de texte : méthode **toText**

Une fois les méthodes **code()** correctement implémentées dans chaque énumération, il est possible d'ajouter à la classe **Card** une méthode **toText** qui fera appel à ces méthodes.

Et pour l'implémentation de **toText**, faites en sorte que l'instruction suivante (que vous ajouterez à la fin de votre méthode **createCard**) :

```
System.out.println("The display of this card is : "+ aceOfHearts.toText());
```

soit capable de fournir l'affichage suivant :

```
The display of this card is : 1♥
```

## 8. Redéfinir la méthode **toString** :

**Remarque/Rappel** : Dans un affichage, l'instruction suivante : `System.out.println(aceOfHearts)` est équivalente à l'instruction suivante : `System.out.println(aceOfHearts.toString())` c-a-d que dans un `System.out.println` ne pas préciser la méthode revient à appeler **toString**



Dans la classe Card, implémentez la méthode : **public String toString()** qui appelée dans le jeu d'essai de **createCard** de la manière suivante :

```
Card aceOfHearts = new Card(Rank.ACE, Suit.HEARTS);
System.out.println(aceOfHearts + " well created !");
System.out.println("with " + aceOfHearts.rank() + " as rank and "
                  + aceOfHearts.suit() + " as suit.");
System.out.println("The display of this card is : " + aceOfHearts.toText());
```

permet d'obtenir l'affichage suivant :

```
ACE of HEARTS well created !
with ACE as rank and HEARTS as suit.
The display of this card is : 1♥
```

## 9. Ajouter des jeux d'essais pour vérifier qu'une carte est correctement créée :

La méthode **createCard** permet actuellement de vérifier que la carte de l'**as de cœur** est correctement créée.

Regroupez ces instructions, puis en dessous dans la même méthode écrivez les des instructions identiques qui permettent d'exécuter des jeux d'essais qui seront en mesure de vérifier que au minimum que :

- La carte du **Roi de Carreau** est correctement créée
- La carte de la **Dame de Trèfle** est correctement créé
- La carte du **Dix de Pique** est correctement créé

```
ACE of HEARTS well created !
with ACE as rank and HEARTS as suit.
The display of this card is : 1♥
-----
```

```
KING of DIAMONDS well created !
with KING as rank and DIAMONDS as suit.
The display of this card is : K♦
-----
```

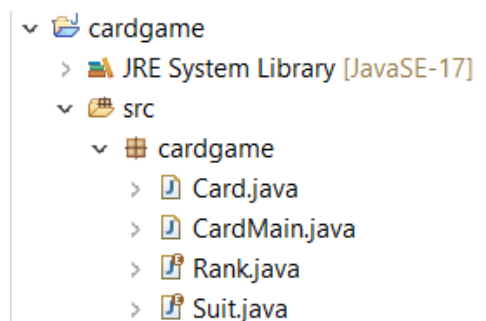
```
QUEEN of CLUBS well created !
with QUEEN as rank and CLUBS as suit.
The display of this card is : Q♣
-----
```

```
TEN of SPADES well created !
with TEN as rank and SPADES as suit.
The display of this card is : 10♠
-----
```

.... Si le cœur vous dit d'écrire d'autres jeux d'essais, ne vous privez pas 😊 ...

## 10. Qualité du code : Cohérence : Mise en place de **paquetages** pour mieux structurer le projet

Actuellement, la vue **Package Explorer** vous indique que votre projet s'organise de la manière ci-contre, c.-à-d. que toutes ces classes appartiennent au même paquetage **cardgame**. D'ailleurs, si vous consultez le code des classes **Card**, **CardMain**, **Rank** et **Suit**, vous constaterez qu'ils commencent tous par l'instruction **package cardgame;**



Pour donner **plus de cohérence au code**, il serait intéressant de regrouper les classes par *thématique*.

Actuellement, deux thématiques pourraient être identifiées :

- ➔ Une thématique autour du **cœur de métier de l'application** : **Card**, **Rank**, **Suit**
- ➔ Une thématique autour des **jeux d'essais** pour tester le **comportement** de l'application : **CardMain**

Mieux structurer son projet consiste à **regrouper au sein d'un même paquetage des classes cohérentes** nous allons donc créer deux paquetages :

- Un paquetage **model** regroupant les classes métiers : **Card, Rank, Suit**
- Un paquetage **application** regroupant les jeux d'essais : **CardMain**

1. Commençons par **créer le paquetage application depuis le fichier source directement**

Ouvrir le fichier **CardMain.java** et remplacer la première instruction **package cardgame;** par **package cardgame.application;**

Aidez-vous ensuite de l'IDE pour créer ce paquetage : une croix rouge apparaît dans la marge, cliquez dessus et sélectionnez : **Move 'CardMain.java' to package 'cardgame.application'**  
**N'oubliez pas de sauvegarder votre fichier CardMain.java après cette modification.**

Si vous consultez la vue **Package Explorer**, vous constaterez que l'IDE a bien créé le nouveau paquetage. Des erreurs de compilation apparaissent désormais dans la classe, nous y reviendrons plus tard, une fois que les classes métiers auront été déplacées dans leur nouveau paquetage.

2. Pour créer le **paquetage model**, nous allons procéder différemment et utiliser la **vue Package Explorer**. Dans la vue Package Explorer, placez-vous sur le **paquetage cardgame**.

A l'aide d'un clic droit, choisissez **New→Package** (à la place du clic droit, vous pouvez aussi passer par le menu **File**) :

- Vérifiez que le **Source folder** est bien **cardgame/src**
- Complétez le **Name** de manière à voir apparaître **cardgame.model** avant de sélectionner **Finish**

Dans la vue package Explorer le **paquetage cardgame.model** apparaît : il est vide (icône blanche).

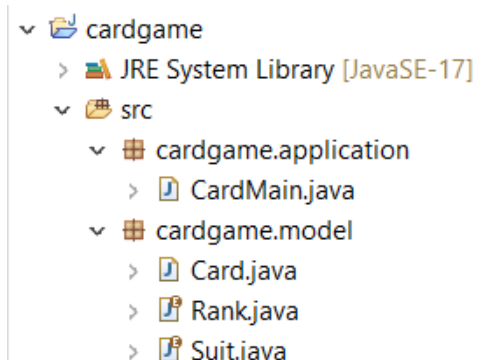
Depuis cette vue, faites glisser les classes métiers (**Card, Rank, Suit**) dans ce paquetage (cliquez sur **OK** et/ou **Continue** si des boîtes de dialogue se présentent)

Les classes apparaissent maintenant sous **cardgame.model** dans la vue Package explorer

Si vous ouvrez le code, la première ligne de ces classes est devenue : **package cardgame.model;**

Il ne reste plus qu'à corriger les erreurs de compilation dans **CardMain** dues au fait que les classes métiers ont changées de paquetages. Deux possibilités pour corriger ces erreurs :

- Cliquez sur chaque croix rouge dans la marge et sélectionnez **Import '...' (cardgame.model)**
- Ou rendez-vous dans le menu **Source → Organize Import** pour ajouter en une seule fois tous les imports manquants. Cette option vous sera très souvent utile, retenez-la bien et retenez surtout son raccourci clavier **Ctrl + Shift + O** qui vous fera gagner du temps 😊



Quelle que soit la solution choisie, n'oubliez pas de **sauvegarder votre fichier** et **d'exécuter CardMain** pour vérifier que les jeux d'essais sont toujours conforme au comportement attendu !

### Diagramme de classe final 😊

Pour mieux répondre aux besoins de l'application, la conception du projet a évolué pendant la phase d'implémentation (des attributs et des méthodes ont été ajoutées), ce qui signifie que le diagramme de classes actuel a été enrichi par rapport à au diagramme qui avait été proposé initialement dans la phase de conception ! Procédez à un nouveau reverse-engineering (rétro-conception) de votre code via **PlantUML** en sélectionnant **uniquement le package model c-a-d uniquement les classes métiers** !

Mettez à jour le diagramme de classes initial (celui en début d'énoncé) avec les nouveaux attributs et services offerts actuellement par les différents composants de ce projet 😊

***N'oubliez pas de faire vérifier votre projet (code et diagramme de classes) à votre enseignant de TP !***



## Travail en autonomie : A rendre pour la prochaine séance de TP

ou à la fin de cette séance s'il vous reste du temps  
et que l'exercice précédent est bien terminé 😊



### ***Et si on cancanait un peu ...***

Vous allez participer au développement d'un jeu de simulation de mare aux canards.

Le jeu doit pouvoir afficher toutes sortes de canards. Les canards peuvent nager et cancaner (c-a-d émettre des sons).

Chaque sous-type de canard (Colvert, Mandarin, Canard en plastique, Leurre, ...) est chargé de s'afficher correctement, lui seul connaît la manière dont il apparaît à l'écran (dans un premier temps l'affichage se limitera à une phrase du style "Je suis un vrai colvert", "Je suis un vrai mandarin", ...)

Tous les canards peuvent nager (et oui, tous les canards flottent, même les leurres!)

Les canards doivent aussi pouvoir voler... mais attention, pas tous : les canards en plastique ne doivent pas voler, cela ferait désordre de voir des canards en plastique voler dans tout l'écran 😊

Les canards n'émettent pas tous le même son : un *vrai* canard (comme le colvert ou le mandarin) émet un cancanement (*can-can*), alors qu'un canard en plastique émet un couinement (*coin-coin*)... ah oui, et un leurre n'émettra aucun son (ce sera un canard muet 😊) .

Et pour finir, sachiez-vous que durant l'hiver, le canard perd ses plumes de vol (rémiges) et ne peut donc pas voler pour un certain temps ?

Du coup, ce serait bien de faire en sorte qu'un canard ait un comportement de vol qui lui est propre et qui pourra être modifier dynamiquement au cours du jeu c-a-d n'importe quand durant le cycle de vie de cet objet.



**Proposer un diagramme de classes pour modéliser ce problème**

# Annexe : Se familiariser avec la terminologie métier autour du jeu de cartes (analyse)

Pour pouvoir jouer au jeu de cartes de la bataille, il faut un élément de base qui est la **carte** ou **carte à jouer**.

La définition du terme carte va entraîner d'autres définitions comme :  
**couleur, hauteur, jeu de cartes, ...**

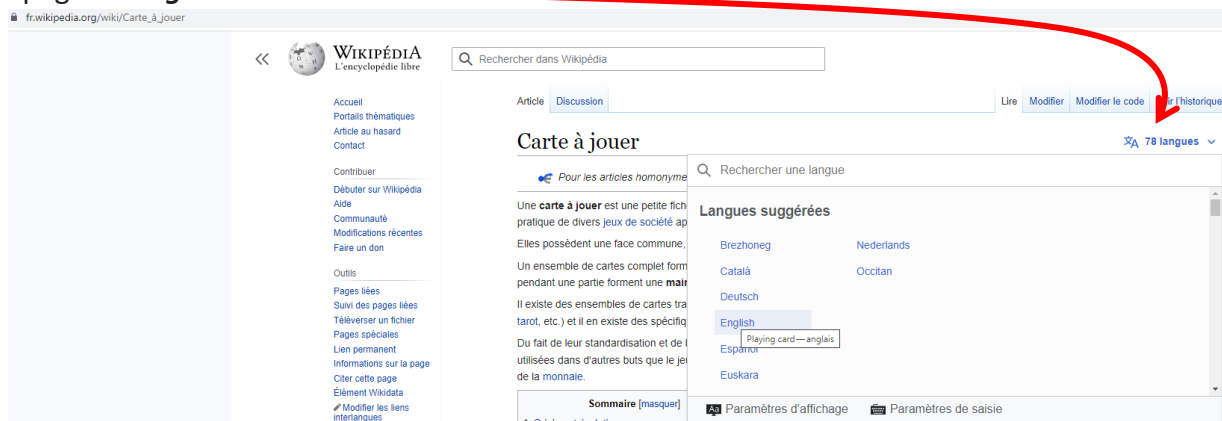
❑ Extrait : [https://fr.wikipedia.org/wiki/Carte\\_%C3%A0\\_jouer](https://fr.wikipedia.org/wiki/Carte_%C3%A0_jouer)

Une **carte à jouer** est une petite fiche illustrée de motifs variés et utilisée, au sein d'un ensemble, dans la pratique de divers **jeux de société** appelés **jeux de cartes**.

Elles possèdent une face commune, appelée **dos**, et une face particulière qui distingue chaque carte.

Un ensemble de cartes complet forme un **jeu** ou un **paquet**, tandis que les cartes qu'un joueur tient en main pendant une partie forment une **main**.

**Astuce :** Pour commencer à se familiariser avec la **terminologie anglaise**, ne pas hésiter à afficher la même page en **English**



⇒ Ci-dessous un petit extrait de la même page (version **English**) :

[https://en.wikipedia.org/wiki/Playing\\_card](https://en.wikipedia.org/wiki/Playing_card)

*Playing cards are typically palm-sized for convenient handling, and usually are sold together in a set as a **deck of cards** or **pack of cards**.*

*The most common type of playing card is that found in the **French-suited, standard 52-card pack**,*

**Remarque :** les contenus des pages **Français** et **English** ne sont pas forcément les mêmes : il ne faut pas s'attendre à voir une traduction littérale au mot près, mais au moins naviguer entre les deux pages a le mérite de se familiariser avec **les termes métiers** 😊

❑ Maintenant selon [https://en.wikipedia.org/wiki/Standard\\_52-card\\_deck](https://en.wikipedia.org/wiki/Standard_52-card_deck)








































A standard 52-card deck comprises 13 ranks in each of the four French suits: clubs (♣), diamonds (♦), hearts (♥) and spades (♠), with reversible (double-headed) court cards (face cards). Each suit includes an Ace, a King, Queen and Jack, each depicted alongside a symbol of its suit; and numerals or pip cards from the Deuce (Two) to the Ten, with each card depicting that many symbols (*pips*) of its suit. Anywhere  
*Pas de jeu sans carte (enrichissons nos énumérations avec un peu d'unicode) Isabelle BLASQUEZ*

from one to six (most often two or three since the mid-20th century) Jokers, often distinguishable with one being more colourful than the other, are added to commercial decks, as some card games require these extra cards.

**Numerals** or **pip cards** are the cards numbered from 2 to 10.

- ➔ "2" cards are also known as **deuces**.
- ➔ "3" cards are also known as **treys**.
- ➔ "4" cards are also known as **sailboats**
- ➔ "8" cards are also known as **snowmen**

Example set of 52 playing cards; 13 of each suit: clubs, diamonds, hearts, and spades

	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
Clubs													
Diamonds													
Hearts													
Spades	