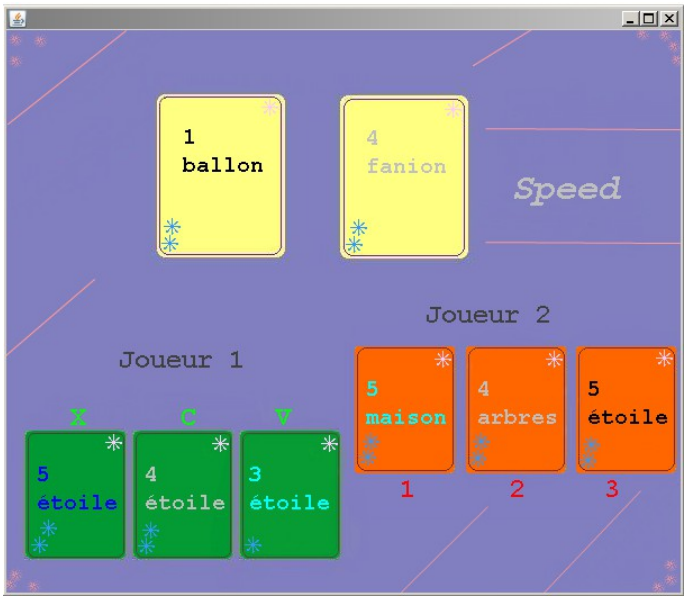
Programmation Orientée Objet en Java

TP 3

Les collections d'objets : le jeu Speed

Objectifs du TP : Définir des classes plus complexes, manipuler des ensembles d'objets, créer une classe enum tout en étant moins guidé. Certaines questions ne servent pas directement pour faire le jeu mais permettent de mieux comprendre les constructeurs. Le contexte de ce TP sera repris au TP 7.



Contexte : nous allons implémenter une partie du jeu de société « speed ». Ce jeu permet de confronter deux joueurs sur des critères d'observation et de rapidité. Chaque joueur se voit attribuer un paquet de cartes pour lequel il ne connaît que les trois premières. Deux tas sont placés au centre de la table contenant chacun une unique carte en début de partie. Chaque carte possède une couleur, un motif et une valeur. Nous aurons par exemple une carte bleue contenant quatre maisons. Chaque joueur peut poser une de ses trois cartes sur un des deux tas à condition que sa carte ait la même couleur, le même motif ou la même valeur que la carte située au sommet du tas central choisi.

Importer le projet java disponible sur Moodle.

Première étape : la classe Carte et l'enum Symbole. Créer une classe Carte dans le paquetage *jeu*. Une carte possède trois attributs privés : une couleur, un motif et une valeur. Une Carte peut également être vide. Un booléen indique pour chaque carte si elle est vide. (**NB :** la possibilité de générer une carte vide est une simplification du problème afin de nous concentrer sur d'autres aspects du TP. Cela permet par ailleurs d'aborder certains aspects des constructeurs).
Le motif répété *valeur fois* sur une carte sera issu d'une classe Symbole de type **enum,** regroupant une suite de symboles. Cet **enum** contiendra une constante **NBR_SYMBOLES** retournant le nombre de motifs différents actuellement proposés par l'énumération (affecter une valeur à cette constante en utilisant la méthode *values*). Une méthode *toString* retournera un **String** de 6 lettres exactement[1] représentant sous la forme d'une chaîne de caractères le motif en question. Cette méthode *toString* permet par exemple d'introduire des accents ou des espaces pour la visualisation de vos enum.

enum : une classe enum permet de définir un ensemble restreint de constantes, limité à certaines valeurs. Ces constantes peuvent être désignées par l'intermédiaire de chaînes de caractères ce qui simplifie la lecture et l'écriture du programme. Ces constantes (nom en majuscules sans caractère spécial) sont manipulables comme des entiers. Elles peuvent donc apparaître dans un **switch**. Pour visualiser les noms des constantes en *français*, on peut redéfinir la méthode toString (choix fait pour ce TP) ou passer par un attribut privé String (fait au TP7).

[1] La valeur de 6 est imposée pour des raisons liées à l'affichage.

Générer les accesseurs. Écrire deux constructeurs. Un constructeur *simple*, sans paramètres, qui crée une carte vide, un autre avec 3 paramètres (2 entiers et un Symbole) qui construit une carte non vide. On considère qu'une carte vide est grise, `Color.darkGray;` a pour valeur -1 et un motif vide. Pour les autres cartes, la couleur est attribuée en utilisant une fonction static privée getColor(indice) qui correspond à une instruction **switch case**. Ainsi, par exemple, Carte(**1**,2,maison) pourrait construire une carte **bleue** de valeur 2 et de motif « maison ».

Un constructeur permet de réserver « sur le tas » un emplacement mémoire pour une instance de la classe où il est défini. L'instance aura ainsi accès à ses attributs et à ses méthodes propres et héritées de sa classe mère. Le constructeur a le nom de la classe où il est défini et ne retourne aucune valeur, pas même `void.`
Comme il vient d'être vu, il est possible de **surcharger** un constructeur, à savoir faire deux constructeurs qui ne diffèrent que par leurs paramètres.

Les couleurs sont celles proposées par la classe `java.awt.Color` en particulier : `Color.blue; Color.orange; Color.cyan; Color.black; Color.lightGray;`

Contrairement au TP avec les bestioles, les couleurs des cartes définies dans le métier sont directement utilisées pour l'IHM. La méthode `private static Color getCouleur(Carte carte)` du Controleur peut prévoir une mise en correspondance de ces codes couleurs pour transmission à l'IHM, si le codage de la couleur d'une carte venait à évoluer dans le métier.

Par ailleurs, la classe Carte contient trois constantes (ici de valeur 5) : le nombre de couleurs existantes (**NBR_COULEURS**), le nombre de motifs différents (**NBR_MOTIFS**) et la valeur maximale d'une carte (**NBR_VALEURS**). Utiliser ensuite ces constantes dans le code et prendre en compte qu'elles peuvent être changées ultérieurement afin de permettre une certaine évolutivité du code. Modifier le constructeur de Carte pour qu'il s'assure que les paramètres sont compatibles avec les constantes. En cas de problème, afficher un message d'erreur dans la console et forcer la carte à être vide.

Dans le cadre de ce programme, une erreur peut survenir si le programmeur programme mal, ce n'est pas l'utilisateur du programme qui générera l'erreur. Vérifier ces constantes propose un garde-fou supplémentaire aux programmeurs.

Deuxième étape : la méthode main. Dans la classe Speed, créer une instance de Carte vide et une autre instance de Carte. Les afficher après avoir implémenté ou généré la ou les méthodes *toString* adéquates.

Troisième étape : les classes Joueur et PaquetCarte du paquetage *jeu***.**
Créer une classe Joueur qui possède comme attribut constant un nom, initialisé par le constructeur. Faire un accesseur sur ce nom.
Créer une classe PaquetCarte dont chaque instance sera une agrégation de Carte. Plutôt que d'étendre la classe ArrayList de java (**extends ArrayList<Carte>**) l'usage veut qu'on ajoute un attribut privé *cartes* de type ArrayList<Carte>. Cela s'accompagne en particulier d'une redéfinition des méthodes de base d'ajout (add) ou de calcul de taille du paquet (size), qui s'appliquent directement sur l'attribut *cartes* mais qui seront accessibles sur une instance de PaquetCarte, grâce à ces ajouts.

Cet usage est lié au fait qu'on ne puisse faire qu'un seul *extends* par classe. Il est donc préférable de se garder cette possibilité pour étendre nos propres classes plutôt que celles qui sont prédéfinies.

Le paquet possède également comme attribut un Joueur, initialisée par les constructeurs, dont le nom sera accessible depuis une instance de PaquetCarte par la méthode `public String getNomJoueur()`. Générer un accesseur et un mutateur sur *joueur*. Il possède également une constante correspondant au nombre de cartes souhaitées pour le jeu (**NBR_CARTES**), en l'occurrence 72. Le Constructeur de PaquetCarte est le premier constructeur complexe à implémenter. Il permet d'instancier un paquet de **NBR_CARTES** cartes. Créer les cartes du paquet en commençant d'abord par toutes les cartes de la même couleur **C**. Pour ce faire, créer toutes les cartes de cette couleur ayant la même valeur **V** en créant ainsi une carte de couleur **C** et de valeur **V** par motif. Procédons par étapes. Créer et afficher un paquet contenant

NBR_MOTIFS cartes de la même couleur (bleue) avec une valeur de 1. Pour cela, définir une méthode `public static Symbole get(int i)` dans la classe Symbole qui retourne le ième Symbole de l'**enum.**

NB : vérifier avant chaque création de carte que le nombre de cartes maximal ne sera pas dépassé. Par ailleurs, si **NBR_CARTES** n'est pas atteint, recommencer de nouvelles séries de cartes, même si elles ont déjà été créées.

Quatrième étape : terminer les constructeurs de PaquetCarte. Compléter votre constructeur initial afin de construire toutes les cartes de couleur bleue (avec toutes les valeurs et tous les motifs), puis toutes les cartes de toutes les couleurs.

NB : dès que **NBR_CARTES** est atteint, il faut sortir des boucles puis du constructeur.

Mélanger votre paquet de cartes, après l'avoir généré, en utilisant la méthode *static* `Collections.shuffle(cartes);`

Créer un second constructeur qui permet de créer un sous-paquet de **N** cartes à partir d'un paquet initial **P**. Il conserve le même joueur. Les cartes doivent être piochées au hasard dans le paquet. S'assurer en début de constructeur que le paquet d'origine aura assez de cartes. Si ce n'est pas le cas, arrêter le programme (de façon brutale) après avoir affiché un message d'erreur. Utiliser la fonction **Math.random()** qui génère des nombres aléatoires entre 0 et 1. Utiliser éventuellement **nextInt**() sur un objet de **Random**.

int i = (int) (Math.random()*20); permet d'affecter à i un entier entre 0 et 19 de façon aléatoire.

Adapter maintenant le premier constructeur pour générer deux fois plus de cartes que prévu et en supprimer la moitié, de façon aléatoire. Ainsi, il n'y aura quasiment jamais le même paquet de cartes.

Cinquième étape : compatibilité de cartes et carte vide. Créer dans Carte une méthode booléenne **estCompatible(Carte c2)** qui indique si deux cartes (`this` et *c2*) sont compatibles c'est-à-dire qui indique si *c2* peut être posée sur `this`. S'assurer par ailleurs de l'existence de l'accesseur **isVide()** qui indique simplement si une carte est vide ou non. Une carte vide n'est jamais compatible.

Vous pourrez utiliser pour cette méthode les propriétés d'optimisation des opérateurs booléens && et ||

Sixième étape : Contrôler le jeu des joueurs via PaquetCarte et Joueur. La méthode `boolean gagne()` permet de déterminer si un paquet gagne ou non. Un paquet est gagnant si la première carte du paquet est associée au symbole vide.

Pour maintenir un jeu intéressant, une pénalité va être attribuée aux joueurs qui proposent une carte alors qu'elle n'est compatible avec aucune carte sur la table. Si un joueur propose une mauvaise carte, l'éventuelle pénalité de l'autre joueur disparaît et il adopte lui-même une pénalité de 3 coups : l'autre joueur va pouvoir librement jouer trois fois sans être interrompu (à condition qu'il ne joue pas lui même une mauvaise carte). La méthode keyPressed dans Controleur permet de comprendre un peu comment sont attribuées les pénalités en lisant les commentaires du `switch`.

Controleur implémente l'interface `KeyListener` qui est utilisée pour surveiller le clavier. Trois méthodes sont liées à cette interface. `keyPressed` propose un traitement à réaliser quand une touche du clavier est enfoncée (celle-ci peut être maintenue enfoncée longtemps ou relâchée). Dans ce contrôleur, c'est la méthode utilisée pour lire le clavier. Les traitements se font pour chaque touche enfoncée et ne seront pas répétés tant que la touche n'a pas été relâchée. `keyReleased` indique le traitement à réaliser lorsque une touche est relâchée, dans notre cas, c'est la méthode où on bloque le jeu dès que la partie est terminée. `KeyTyped` indique ce qu'il faut faire lorsqu'une touche a été enfoncée et relâchée. Le contrôleur est relié à l'ihm par l'intermédiaire de son constructeur, *via* la méthode `addKeyListener.`

Déclarer deux attributs dans la classe Joueur `PENALITE = 3 et penalite = 0`

Déclarer les méthodes suivantes, dans Joueur, permettant de gérer ces pénalités.

```
public void annulerPenalite()    public void ajoutePenalite()
public boolean sansPenalite() public void oterEventuellementUnePenalite()
```

Dans PaquetCarte, la méthode `sansPenalite()` de PaquetCarte fait appel à la méthode définie sur le joueur du paquet. La méthode `public void gererErreur(PaquetCarte paquet2)` permet d'annuler l'éventuelle pénalité du joueur associé au paquet2 et d'initialiser une pénalité sur le joueur du paquet `this`.
La méthode `public Carte testerCarteSommet(int positionCarte, Carte sommet, PaquetCarte autrePaquet)` ôte et retourne la carte située en position `positionCarte` dans le PaquetCarte si elle est compatible avec le `sommet` donné, `null` sinon.
Si elle est compatible, on `oterEventuellementUnePenalite` au joueur de l'autre paquet.

Septième étape : le contrôleur et l'IHM. Étudier la classe Controleur. Les constantes correspondent aux touches que peuvent utiliser les joueurs.

Cette classe permet de faire le lien, conformément au patron de conception MVC, entre le métier et l'IHM. Il organise les passages d'un affichage à un autre et traite les entrées clavier (keyPressed) en utilisant la partie métier de l'application. L'IHM compte une fenêtre hôte et un contenu graphique. Le Controleur commande la fenêtre hôte, ce qui peut impacter son contenu graphique.

Terminer les classes afin de pouvoir jouer.

Trouver comment changer le positionnement des noms des joueurs ainsi que du mot « speed » dans la partie graphique de l'IHM.

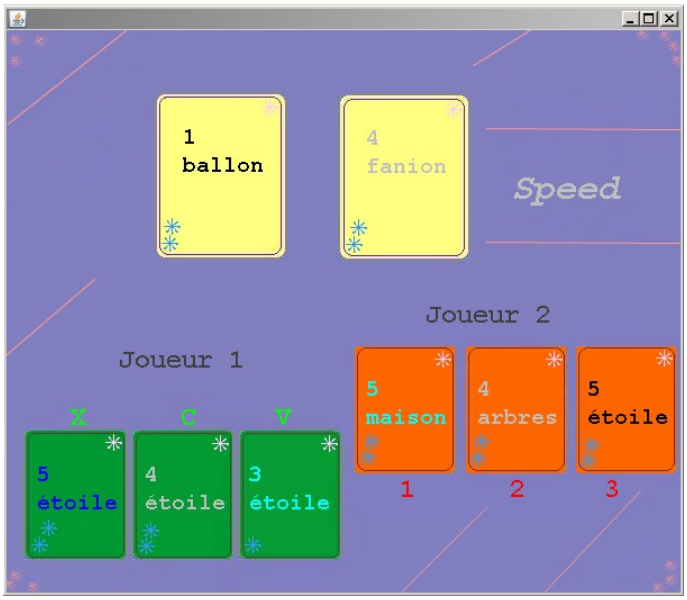
Trouver des réponses à vos questions :
en tapant « java API <nom de l'api> » dans un moteur de recherche, vous obtiendrez la javadoc complète de l'API que vous recherchez.
Ainsi, pour les ArrayList, en tapant « arraylist api » sur google, la première réponse obtenue sera le site de documentation d'Oracle : http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html

Programmation Orientée Objet en Java

TP 7

le jeu Speed avec Interface, classe abstraite et types génériques

Objectifs du TP : Reprendre tous les concepts liés au polymorphisme sur le TP Speed.



Contexte : Celui du TP3. Importer la nouvelle archive disponible sur Moodle. Vous pourriez réutiliser peut être quelques classes codées lors du TP2, mais il est plus sage de reprendre ce travail en entier (sachant qu'il ne représente pas beaucoup de temps de travail de recopie et que cela constitue un bon exercice de révision).

Vous pouvez retravailler dans le workspace du TP2.

Toutes les étapes :

Définir une nouvelle classe `public enum Symbole` qui aura un attribut `private final String nomAffichage` utilisé lors de l'affichage (la méthode toString retourne cet attribut). L'attribut est initialisé par le constructeur qui reçoit comme paramètre une chaîne de caractères. Ainsi, une valeur est définie en tête de classe de la façon suivante : `ETOILE("étoile")`.

Définir une classe Carte et une classe CarteVide qui hérite de la classe Carte. CarteVide ne contient qu'un constructeur simple (qui affecte les valeurs *neutres* aux trois attributs de Carte) et qui redéfinit la méthode boolénne isVide(). Retrouver le contenu de Carte grâce au sujet du TP2.

Définir une interface générique `public interface IPaquet<T>` qui permet de manipuler un paquet d'objets en redéfinissant les méthodes add() get() remove() remove() size()

Définir une classe abstraite générique `public abstract class Paquet<T>` qui implémente `IPaquet<T>` et qui a pour attribut `private List<T> lePaquet`.

Définir enfin une classe concrète PaquetCarte qui étend `Paquet<Carte>`.

Faire les derniers ajouts de code pour faire tourner le jeu.

Bonus :

Modifier le code, y compris dans le contrôleur, pour ne plus avoir besoin de la classe CarteVide.