- Programmation Orientée Objet -

Collections

BlackJack

Avant propos

Dans ce TD nous nous chercherons à stocker des informations dans des collections dynamiques d'objets. Mélange, tri et tirage de cartes sont au programme ! Nous commencerons par une application en mode Console, puis nous réaliserons notre première interface graphique. Nous allons mettre en œuvre différentes notions :

- *Énumération*
- *Collections*
- *Mécanique de jeu*
- *Notre première interface graphique*
- *Affichage d'images dans une interface graphique*

Contexte

Nous allons nous placer dans le contexte d'un logiciel de simulation de Blackjack, jeu de cartes populaire dans les casinos, qui consiste à essayer d'approcher le plus possible un seuil (21 points) sans jamais le dépasser. Tout commence par le tirage d'une carte visible par le croupier. Ensuite, le ou les joueurs misent contre le croupier (la Banque) puis chacun des joueurs tire successivement des cartes dans le sabot (ensemble composé de 3 ou 4 jeux de 52 cartes mélangés) et le place face visible devant eux. Une fois chaque joueur satisfait ou ayant dépassé le seuil fatidique, le croupier tire à son tour des cartes pour essayer de vaincre les joueurs. Chaque joueur gagnant empoche le double de sa mise, le croupier récupérant les mises des perdants. Tout le jeu se fait face visible et est donc un jeu d'estimation et non de bluff.

Nous allons fonctionner par étape et rajouter des fonctionnalitées au fur et à mesure.

Etape 0 : Classe principale : BlackJackConsole

EXERCICE 1

Ouvrez un éditeur de texte (`gedit` ou `vim`).
Déclarez une nouvelle classe publique `BlackJackConsole` dans un fichier `BlackJackConsole.java` qui contient un constructeur qui affichera la phrase `"Welcome to the BlackJack table. Let's play "` ! et la méthode **`main`** qui se contentera de créer un nouvel objet `BlackJackConsole`

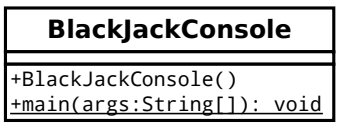
```
public class BlackJackConsole {

   public BlackJackConsole() {
      System.out.println("Welcome to the BlackJack table. Let's play !");
   }

   public static void main(String[] args) {
      new BlackJackConsole();
   }
}
```

Compilation Pour compiler, on utilise la commande (dans le Terminal) :
`javac BlackJackConsole.java`
Pour exécuter, on utilise la commande :
`java BlackJackConsole`.
Vous devriez voir :

```
Welcome to the BlackJack table. Let's play !
```

Classe principale Cette classe, en interface console, vous servira à tester votre travail au fur et à mesure du TP.

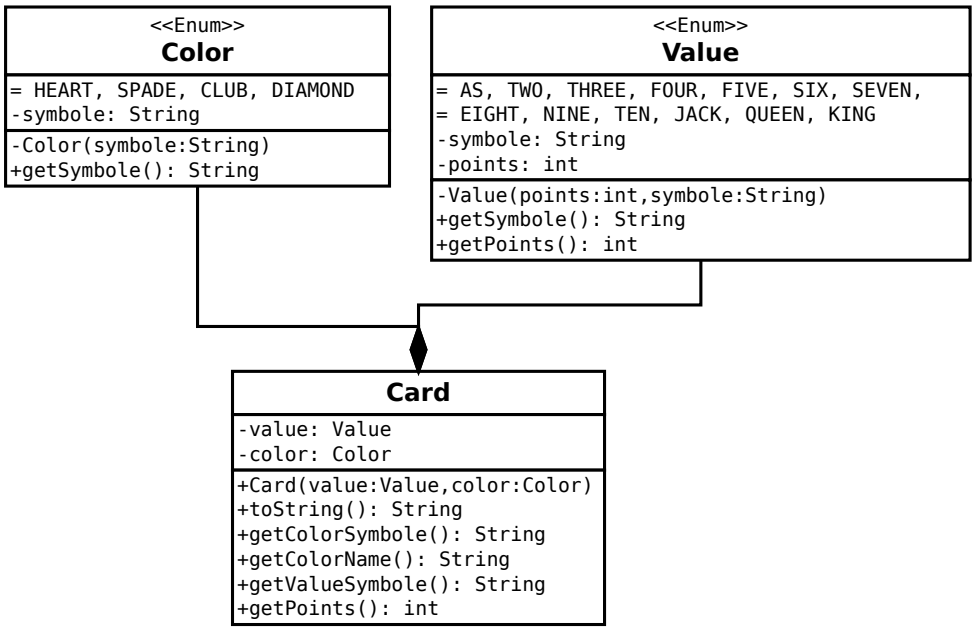
Voici son diagramme UML :



Etape 1 : Cartes, valeurs et couleurs !

Dans un paquet "classique" de 52 cartes, une carte à jouer est **composée** de deux informations, une **valeur** (As, 2, 3, ... , Valet, Dame, Roi) et une **couleur** (Cœur, Carreau, Trèfle, Pique). Nous allons utiliser des énumérations (comme dans le TD0) pour gérer ces deux attributs.

Comme propriété intrinsèque des ces énumérations nous allons choisir un symbole (pour l'affichage) ainsi qu'un nombre de points pour les cartes (lié à leur usage dans le jeu).



EXERCICE 2

Dans des fichiers séparés, créez les énumérations `public enum Color` (`Color.java`) et `public enum Value` (`Value.java`) puis la classe `public class Card` (`Card.java`) correspondant au diagramme UML ci-dessus.

Les anglophones utilisent une dénomination particulière des couleurs : Cœur, Épée (Pique), Bâton (Trèfle), Diamant (Carreau) qui correspond aux anciens noms français. Notez aussi qu'en anglais le valet se nomme Jack.

Symbole UTF-8 Java utilise l'encodage Unicode UTF-8 pour ses programmes, vous pouvez donc directement copier/coller les caractères ♡, ♠, ♣, ♢ dans votre programme !
Autre solution, les copier/coller depuis la page web `http://www.w3schools.com/charsets/ref_utf_symbols.asp` ou encore utilisez leur équivalents :

`"♡"`	`"\u2665"`	`"♠"`	`"\u2660"`
`"♣"`	`"\u2663"`	`"♢"`	`"\u2666"`

Valeurs des cartes Au BlackJack, les cartes numériques (2, 3...) ont une valeur en points égale à leur chiffre, alors que les figures (valet, dame, roi) valent 10 points. L'As peut valoir soit 1 soit 11, mais nous considérerons pour le moment que l'As vaut 1 point. Les couleurs ne modifient pas le nombre de points des cartes. La main (l'ensemble de carte) : [2♡, *J*♠, 7♢] vaut donc 19 points.

Qu'est ce que le symbole des cartes ?

Le symbole associé à chaque couleur (respectivement à chaque valeur) est la chaîne de caractère permettant de la représenter.

Par exemple, la couleur "pique" sera représentée par le symbole : "♠" De la même manière la valeur "valet" sera représentée par le symbole "J"

Ce qui nous permettra de dire que le valet de pique sera représenté par la chaîne "J♠"

Rappel : Récupérer le "nom" d'une instance d'une énumération Pour récupérer le "nom" d'une instance (HEART par exemple) on utilise la méthode `name()`, pour récupérer sa position dans l'énumération (0 pour HEART par exemple) on utilise la méthode `ordinal()`.
Remarque : la méthode `public String getColorName()` renvoie justement le "nom" de la couleur de la carte.

EXERCICE 3


Testez votre programme avec le code ci dessous.

```
public class BlackJackConsole {

   public BlackJackConsole() {
      System.out.println("Welcome to the BlackJack table. Let's play !");

      // This is an array of two cards
      Card[] tab = { new Card(Value.TWO, Color.HEART), new Card(Value.JACK, Color.SPADE) };
      for(Card c : tab) { // For each card
      System.out.println("This card is a "+c+ " worth "+c.getPoints()+ " points");
      System.out.print("It's a ");
      switch(c.getColorSymbole()) { // Ok from Java 1.7
      case "♡": System.out.print("heart"); break;
      case "♠": System.out.print("spade"); break;
      case "♣": System.out.print("club"); break;
      case "♢": System.out.print("diamond"); break;
      }
      if(c.getValueSymbole().matches("[JQK]")) { // Is the value symbole a J or a Q or a K ?
      System.out.println(" and a face !");
      } else {
      System.out.println(" and it's not a face.");
      }
      }
   }

    public static void main(String[] args) {
        new BlackJackConsole();
    }
}
```

Votre programme devrait afficher :

```
Welcome to the BlackJack table. Let's play !
This card is a 2♡ worth 2 points
It's a heart and it's not a face.
This card is a J♠ worth 10 points
It's a spade and a face !
```

Etape 2 : Constituer le sabot

Maintenant que nous avons nos cartes, il nous faut un sabot composé de plusieurs jeux de cartes mélangés et duquel nous tirerons nos cartes au fur et à mesure.

Nous pourrions stocker nos cartes dans un tableau, mais au fur et à mesure que nous retirons des cartes le tableau va se vider et utiliser de l'espace mémoire inutilement.

C'est pourquoi il est plus pertinent ici d'utiliser une `Collection`, c'est-à-dire un ensemble dynamique d'objets. Comme nous n'avons pas besoin d'accéder au cartes au milieu du paquet, mais seulement à celle qui est au dessus, une liste chaînée semble appropriée. En Java, c'est la classe générique `LinkedList<T>` qui implémente cette fonctionnalité.

http://docs.oracle.com/javase/7/docs/api/index.html?java/util/LinkedList.html

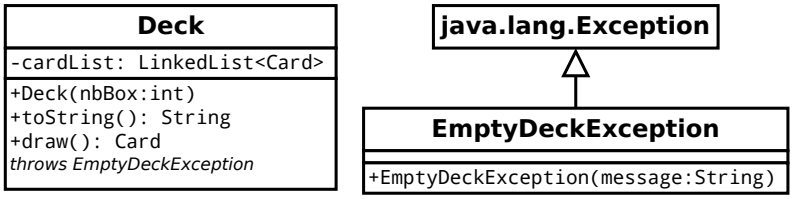
Exemple de liste chainée :

```
import java.util.LinkedList; // Need the import at the begining of the file
...
LinkedList<Car> carList = new LinkedList<Car>(); // Allocating an empty list.
carList.add(new Car("Toyota"));
carList.add(new Car("Peugeot"));
```

```
System.out.println("There is "+carList.size()+" car(s) in the list.");
System.out.println("The first car is "+carList.peekFirst()); // The car is not removed
System.out.println("There is still "+carList.size()+" car(s) in the list.");
Car myCar = carList.pollFirst(); // Retrieves and remove from the list.
System.out.println("I just get "+myCar+" out of the list.");
System.out.println("There is now "+carList.size()+" car(s) in the list.");
```

```
There is 2 car(s) in the list.
The first car is Toyota
There is still 2 car(s) in the list.
I just get Toyota out of the list.
There is now 1 car(s) in the list.
```

Peek et Poll Faites attention à la différence entre les deux méthodes `peekFirst` et `pollFirst` les deux renvoyant le premier élément de la liste, mais la seconde le retirant au passage de la liste.



EXERCICE 4

Dans un nouveau fichier `Deck.java`, créez une classe `Deck` contenant une liste chaînée de cartes. Le constructeur prend en paramètre le nombre de jeu de 52 cartes qui constitue le sabot. Pour chaque jeu, chacune des cartes (valeur/couleur) apparait une fois, donc un sabot composé de 4 jeux, aura 4 as de cœur, 4 roi de pique, etc..

La méthode `draw` retire la première carte du paquet et la renvoie. Si il n'y a plus de cartes dans le paquet une exception `EmptyDeckException` est levée.
Remarque : Vous devrez écrire votre classe `EmptyDeckException` dans un nouveau fichier `EmptyDeckException.java`.

EXERCICE 5

Modifiez votre classe `BlackJackConsole` pour afficher votre sabot puis mélangez le.

```
Deck deck = new Deck(2);
System.out.println("Here is the deck "+ deck+"\n");
for(int i = 0 ; i < 3 ; i ++) { // For each card
   try{
      Card c = deck.draw();
      System.out.println("This card is a "+c+ " worth "+c.getPoints()+ " points");
      /* ... */
   } catch (EmptyDeckException ex) {
      System.err.println(ex.getMessage());
      System.exit(-1);
   }
}
```

```
Here is the deck [A♡, 2♡, 3♡, 4♡, 5♡, 6♡, 7♡, 8♡, 9♡, 10♡, J♡, Q♡, K♡, A♠, 2♠, 3♠, 4♠, 5♠, 6
    ♠, 7♠, 8♠, 9♠, 10♠, J♠, Q♠, K♠, A♣, 2♣, 3♣, 4♣, 5♣, 6♣, 7♣, 8♣, 9♣, 10♣, J♣, Q♣, K♣, A♢,
     2♢, 3♢, 4♢, 5♢, 6♢, 7♢, 8♢, 9♢, 10♢, J♢, Q♢, K♢, A♡, 2♡, 3♡, 4♡, 5♡, 6♡, 7♡, 8♡, 9♡, 10
    ♡, J♡, Q♡, K♡, A♠, 2♠, 3♠, 4♠, 5♠, 6♠, 7♠, 8♠, 9♠, 10♠, J♠, Q♠, K♠, A♣, 2♣, 3♣, 4♣, 5♣,
    6♣, 7♣, 8♣, 9♣, 10♣, J♣, Q♣, K♣, A♢, 2♢, 3♢, 4♢, 5♢, 6♢, 7♢, 8♢, 9♢, 10♢, J♢, Q♢, K♢]

This card is a A♡ worth 1 points
```

```
It's a heart and it's not a face.
This card is a 2♡ worth 2 points
It's a heart and it's not a face.
This card is a 3♡ worth 3 points
It's a heart and it's not a face.
```

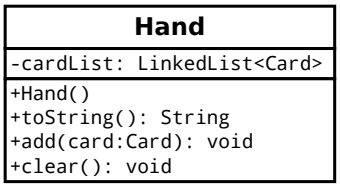
Mélanger le sabot Pour que le jeu soit intéressant, il faut que les cartes soit mélangées. Pour cela on utilise la classe `Collections` (avec un s)
http://docs.oracle.com/javase/7/docs/api/index.html?java/util/Collections.html
qui contient de nombreuses méthodes de classes utiles pour manipuler les collections comme les listes chaînées.

- `Collections.sort(List<T> list)` pour trier une liste,
- `Collections.swap(List<T> list, int i, int j)` pour échanger deux valeurs,
- `Collections.reverse(List<T> list)` pour inverser une liste,
- `Collections.shuffle(List<T> list)` pour mélanger une liste, etc.

```
Here is the deck [K♠, 10♡, 4♢, 4♡, 4♣, A♠, 3♢, J♢, Q♢, 7♡, 7♣, 2♢, 5♠, J♣, 8♣, 10♣, 7♢, 5♢, 2
    ♣, 8♠, 5♡, 3♢, K♠, 5♣, 7♣, 2♣, 5♡, 9♣, 8♠, 3♡, 6♡, 2♠, 10♣, 9♣, 3♡, 6♣, A♢, A♣, K♢, 10♠,
     3♣, 9♠, Q♣, J♢, A♣, 4♠, J♣, J♡, Q♠, K♣, 8♡, Q♠, A♡, 8♢, Q♣, Q♡, 4♠, 8♢, 10♢, 2♢, J♠, Q
    ♡, 8♣, 10♡, 9♢, 5♢, 6♢, 4♣, K♣, 7♡, 6♣, 7♠, 9♠, J♠, 3♣, A♢, 6♠, 2♡, 4♢, K♡, Q♢, 6♠, 6♢,
    9♢, K♢, K♡, 2♠, 5♣, 7♢, A♡, A♠, 6♡, 9♡, 7♠, 3♠, 8♡, 4♡, 3♠, 10♠, 5♠, 2♡, 9♡, 10♢, J♡]

This card is a K♠ worth 10 points
It's a spade and a face !
This card is a 10♡ worth 10 points
It's a heart and it's not a face.
This card is a 4♢ worth 4 points
It's a diamond and it's not a face.
```

Etape 3 : Se faire la main

Les cartes tirées du sabot constituent ce que l'on appelle **la main** des joueurs. Au BlackJack il n'y a pas de limite au nombre de carte qu'un joueur peut avoir en main, mais un joueur a perdu dès que le nombre de points de sa main dépasse le seuil de 21 points. Le nombre variable de cartes incite à utiliser à nouveau des listes chaînées pour stocker les cartes de la main dont la structure est similaire à celle du sabot (sauf que les cartes partent du sabot et s'ajoute à celle des mains).



EXERCICE 6

Dans un nouveau fichier `Hand.java`, créez la classe `Hand` en suivant le diagramme UML ci-dessus, puis testez votre classe.

```
System.out.println("Welcome to the BlackJack table. Let's play !");
Deck deck = new Deck(2);
Hand hand = new Hand();
System.out.println("Your hand is currently : "+ hand);
for(int i=0; i < 3 ; i++) {
   try {
      hand.add(deck.draw());
   } catch (EmptyDeckException ex) {
      System.err.println(ex.getMessage());
      System.exit(-1);
   }
}
```

```
System.out.println("Your hand is currently : "+ hand);
hand.clear();
System.out.println("Your hand is currently : "+ hand);
```

```
Welcome to the BlackJack table. Let's play !
Your hand is currently : []
Your hand is currently : [5♡, K♣, 10♣]
Your hand is currently : []
```

Le décompte des points associées à une main serait simple (mais moins intéressant) si il n'y avait la particularité de l'As de pouvoir prendre soit la valeur 1 soit la valeur 11 au choix du joueur. Cela veut dire que la main ($A\heartsuit, A\spadesuit, 9\clubsuit$) peut valoir 11 (1 + 1 + 9), 21 (11 + 1 + 9) ou 31 (11 + 11 + 9) ! ce qui peut signifier alternativement la victoire ou la défaite !

Nous allons donc devoir fournir, non pas UN nombre de points, mais DES nombres de points possibles, par exemple sous la forme d'une collection d'entiers puisque nous ne savons pas à l'avance combien de possibilités différentes nous avons.

En Java, les collections ne fonctionnent que sur des types Objet (et donc pas sur le type primitif `int`), mais depuis la version 1.5 Java a mis en place une correspondance exacte entre les types primitifs et les types Objets associés (`int` et `Integer` par ex.). On peut écrire :

```
Integer i = new Integer(7); // Before Java 1.5
Integer j = 3; // Since Java 1.5
int k = i; // Since Java 1.5
```

ou encore

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(4); // Add a new value
int val = list.get(0); // Get the value at index 0
list.set(0, val+2); // Set the value at index 0
```

D'un point de vu algorithmique, comment chercher tous les cas possibles ? Les listes chaînées nous proposent une solution élégante :

Soit une liste de résultats composée au début d'un seul résultat valant 0
pour chaque carte de la main **faire**
  pour i = 0 ; i < (taille de la liste de résultat) ; i++ **faire**
    Récupérer la valeur `val` *du résultat d'indice* `i`.
    Affectez au résultat d'indice `i` *la somme de* `val` *et de la valeur de la carte.*
    si la valeur de la carte vaut 1 **alors**
      Ajoutez *(*`val`*+11) à la liste de résultats*
    fin
  fin
fin

Ne pas mélanger la liste de résultats (`LinkedList<Integer>`) et la liste chaînée de cartes (`LinkedList<Card>`) !

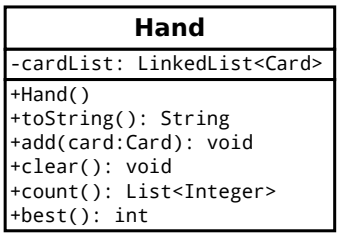
Modifier un élément de la liste Les collections, si elles ont l'avantage de permettre une allocation dynamique de la mémoire, nécessite cependant une syntaxe un peu plus lourde que les tableaux !

Exemple :

```
int[] tab = {1, 1, 1, 1};
tab[1]++; // Access and modification
// 1, 2, 1, 1
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(1); // 1
list.add(1); // 1, 1
list.add(1); // 1, 1, 1
```

```
list.add(1); // 1, 1, 1, 1
list.set(1, list.get(1) + 1); // Access then Modification
// 1, 2, 1, 1
```

Taille dynamique de la liste Attention ! La méthode renvoyant la taille de la liste le fait de manière dynamique (la valeur change dès que vous ajoutez une valeur), ce qui peut provoquer des effets étranges ! **Pensez à sauver la valeur de la taille dans une variable** qui ne changera pas le temps de la boucle.



EXERCICE 7

Rajoutez la méthode `public List<Integer> count()`

Modifiez la méthode `toString` de la classe `Hand` pour afficher les différents scores possibles.

Puis testez votre nouvelle méthode.

Enfin, rajoutez la méthode `public int best()` qui renvoie le plus grand score inférieur ou égal 21, où le plus grand score à défaut.

Type de retour de `count` Pourquoi retourner une valeur sous la forme abstraite `List<Integer>` plutôt que sous une `LinkedList<Integer>` ? Au cas où vous changeriez d'avis ! Rappelez vous qu'il faut modifier les prototypes publiques le moins possible. En choisissant de renvoyer le type générique vous ne donnez pas d'information sur le type privé utilisé que vous pouvez donc modifier sans risquer de casser le code de vos utilisateurs.

Votre programme devrait afficher :

```
Welcome to the BlackJack table. Let's play !
Your hand is currently : [A♡, 8♢, A♣] : [10, 20, 20, 30]
The best score is: 20
```

```
Welcome to the BlackJack table. Let's play !
Your hand is currently : [K♠, J♣, 4♡] : [24]
The best score is: 24
```

Etape 4 : Mettons en place le jeu !

Tous les outils étant là, il nous reste à mettre en place la classe de jeu qui déroulera les étapes dans l'ordre.

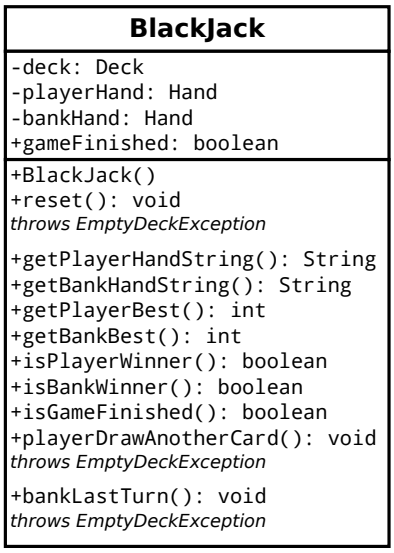
Une partie de BlackJack met en œuvre un sabot composé de 4 jeux de 52 cartes, une main pour la banque et une main pour le joueur (nous nous placerons ici dans le cas d'un joueur unique).

La partie se déroule suivant le processus suivant :
- La banque tire une carte.
- Le joueur tire deux cartes.
- Ensuite, le joueur peut demander autant de carte qu'il le désire du moment qu'il ne dépasse pas 21.
- Une fois le joueur satisfait la banque essaye de le battre (c'est-à-dire de faire un score supérieur ou égal sans dépasser 21).
- En cas d'égalité, le joueur récupère sa mise, en cas de défaite il perd sa mise, en cas de victoire il double sa mise.

EXERCICE 8

Dans un nouveau fichier `BlackJack.java`, créez la classe `BlackJack` selon le diagramme UML ci-dessous.

- Le constructeur initialisera les différents attributs puis appellera la méthode `reset`.
- La méthode `reset` videra les mains de la banque et du joueur puis attribuera 1 nouvelle carte à la banque et 2 au joueur.
- les méthodes `getPlayerHandString`, `getBankHandString`, `getPlayerBest` et `getBankBest` sont des encapsulation des méthodes de `Hand` correspondantes et permettent d'accéder aux informations des différents partis (banque et joueur).



- les méthodes `isPlayerWinner` et `isBankWinner` renvoient vrai si la partie est finie et que le joueur (respectivement la banque) a un score inférieur ou égal à 21 et [supérieur à celui de l'adversaire ou l'adversaire a un score supérieur à 21]. Notez que les deux partis peuvent perdre (en dépassant tous les deux 21).
- la méthode `isGameFinished` renvoie vrai si la partie est terminée, faux sinon.
- la méthode `playerDrawAnotherCard` tire, si la partie n'est pas terminée, une nouvelle carte pour l'ajouter à la main du joueur. Si le nouveau meilleur score du joueur dépasse 21, la méthode passe le booléen `gameFinished` à vrai.
- la méthode `bankLastTurn` tire, si la partie n'est pas terminée et si aucun des meilleurs scores n'ont dépassé 21, autant de nouvelles cartes que nécessaire pour battre le joueur. Puis il passe le booléen `gameFinished` à vrai.

Plus de cartes ! Les méthodes `reset`, `playerDrawAnotherCard` et `bankLastTurn` renvoient les éventuelles exception `EmptyDeckException` sans les traiter. Ce n'est pas à elles de décider quoi faire (mais à `BlackJackConsole`). En revanche le constructeur `BlackJack` doit traiter l'erreur (improbable vu que l'on vient de remplir le deck) et quitter le programme en cas d'exception avec le message `"Error, the deck has insuffisent cards"` !.

EXERCICE 9

Modifier la classe `BlackJackConsole` pour prendre en compte l'interface finale du jeu.

Instanciez la classe de jeu. Puis afficher la main actuelle de la banque et du joueur.
Puis tant que la partie n'est pas terminée, proposez lui de tirer une nouvelle carte. S'il répond oui, appelez la méthode `playerDrawAnotherCard` puis affichez la nouvelle main, sinon appelez la méthode `bankLastTurn` et afficher la nouvelle main (de la banque).
Enfin, affichez les scores des deux parties et annoncez le vainqueur !

Exemple d'affichage :

```
Welcome to the BlackJack table. Let's play !
The bank draw : [7♡] : [7]
You draw : [K♠, 4♢] : [14]
```

```
Do you want another card ? [y/n]
y
Your new hand : [K♠, 4♢, A♣] : [15, 25]
Do you want another card ? [y/n]
y
Your new hand : [K♠, 4♢, A♣, 4♣] : [19, 29]
Do you want another card ? [y/n]
n
The bank draw cards. New hand : [7♡, 8♢, 4♠] : [19]
Player best : 19
Bank best : 19
Draw !
```

Félicitations, nous avons notre jeu de BlackJack en interface console !

Etape 5 : Interface graphique

Afin de rendre le jeu plus attractif nous allons lui créer une interface graphique. Comme nous avons pris soin de séparer le moteur de jeu (classe `BlackJack`) de l'interface console (classe `BlackJackConsole`), cela devrait être assez facile.

Nous aimerions réaliser une interface ressemblant à ceci :

On observe :
- une zone de boutons, en haut de la fenêtre, avec trois options : `"Another Card"`, `"No More Card"` ou `"Reset"`.
- Une zone centrale avec redécoupée en deux parties `"Bank"` et `"Player"` qui affiche les mains de la banque et du joueur ainsi que leur meilleur score actuel.

Création du squelette

EXERCICE 10

Dans un nouveau fichier `BlackJackGUI.java`, créez une classe `BlackJackGUI` et dotez de l'interface graphique minimale ci-dessous.

```
public class BlackJackGUI {
   public BlackJackGUI() {

      JFrame frame = new JFrame("BlackJack GUI");
      frame.setMinimumSize(new Dimension(640,480));
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.pack();
```

```
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        new BlackJackGUI();
    }
}
```

EXERCICE 11

Rajoutez deux panneaux (`JPanel`) `topPanel` et `centerPanel`, puis trois boutons (`JButton`) dans le `topPanel` et deux nouveaux panneaux `bankPanel` et `playerPanel` dans le `centerPanel`.
http://docs.oracle.com/javase/7/docs/api/index.html?javax/swing/JFrame.html
http://docs.oracle.com/javase/7/docs/api/index.html?javax/swing/JPanel.html
http://docs.oracle.com/javase/7/docs/api/index.html?javax/swing/JButton.html

Bordure Pour afficher une bordure autour de votre panneau, on utilise une méthode de classe appelée une "factory" : `BorderFactory.createTitledBorder(String name)`
http://docs.oracle.com/javase/7/docs/api/index.html?javax/swing/BorderFactory.html

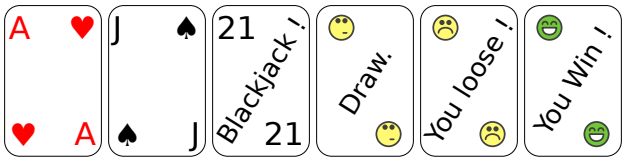
Exemple :

```
JPanel bankPanel = new JPanel();
bankPanel.setBorder(BorderFactory.createTitledBorder("Bank"));
```

Gestionnaire de disposition Si votre interface ne ressemble à rien alors que vous avez bien ajouté les composants (`JButton`, `JPanel`) aux conteneurs (`JFrame`, `JPanel`), c'est très certainement par ce que vous n'avez pas (bien) spécifié les gestionnaire de disposition ("layout") des conteneurs avec la méthode (`setLayout`). Retournez dans le cours voir les différents types de Layout ou allez sur l'api Java
http://docs.oracle.com/javase/7/docs/api/index.html?java/awt/FlowLayout.html
http://docs.oracle.com/javase/7/docs/api/index.html?java/awt/GridLayout.html
http://docs.oracle.com/javase/7/docs/api/index.html?java/awt/BorderLayout.html

Vous devriez obtenir :

Afficher des cartes Vous trouverez sur http://learning.esiea.fr/ une archive contenant des images de cartes au format PNG.



Ces cartes sont nommées sur le format `card_(couleur)_(valeur)` par exemple `card_♠_J.png` il est donc facile de déterminer le nom du fichier image associé à une carte de notre jeu.

L'affichage d'images dans une interface graphique peut se faire par le biais des classes `JLabel` (qui servent à afficher des "étiquettes" de textes ou des icônes) et `ImageIcon` (qui gère l'image d'un point de vue stockage de donnée).

http://docs.oracle.com/javase/7/docs/api/index.html?javax/swing/JLabel.html

http://docs.oracle.com/javase/7/docs/api/index.html?javax/swing/ImageIcon.html

Exemple :

```
File file = new File("./img/card_SPADE_J.png");
if (!file.exists()) {
   throw new FileNotFoundException("Can't find "+file.getPath());
}
ImageIcon icon = new ImageIcon(file.getPath()); // Create the image from the filename
JLabel label = new JLabel(icon); // Associate the image to a label
playerPanel.add(label); // Add the label to a panel
```

Chemin vers l'image Attention à bien spécifier le chemin exact jusqu'à l'image (ici l'image est dans le sous-répertoire `img`). Sinon, l'image n'apparait tout simplement pas ! C'est pourquoi nous vérifions d'abord l'existence du fichier sur le disque dur.

EXERCICE 12

Rajoutez la méthode
`private void addToPanel(JPanel p, String token)throws FileNotFoundException` qui ajoute un `JLabel` représentant l'image dont le `token` est passé en argument. Le token correspond à la partie variable du nom de l'image. (i.e le nom de l'image est `"./img/card_"+token+".png"`) au `JPanel` passé en paramètre. Si le fichier image n'existe pas la méthode renvoie une `FileNotFoundException` (comme dans l'exemple ci-dessus).

Puis testez votre méthode.

Exemple de lignes ajoutées au constructeur :

```
try{
   addToPanel(playerPanel, "CLUB_J");
} catch(FileNotFoundException ex) {
   System.out.println(ex.getMessage());
}
```

Afficher une main En instanciant la classe `BlackJack`, nous aimerions avoir accès, non pas à la représentation textuelle des mains, mais à la liste des cartes contenues dans chaque main.

EXERCICE 13

Rajoutez les méthodes :
- `public List<Card> getCardList()` à la classe `Hand`
- `public List<Card> getPlayerCardList()` et `public List<Card> getBankCardList()` à la classe `BlackJack` qui retourne une copie (pour éviter des modifications de l'extérieur) des listes de cartes stockées.

Faire une copie d'une LinkedList<E> Pour créer une nouvelle instance avec des copies des éléments d'une liste on utilise l'astuce suivante : on crée un nouvelle objet avec l'ancienne liste en argument du constructeur !

```
LinkedList<Card> originalList = playerHand.getCardList();
LinkedList<Card> copyList = new LinkedList<Card>(originalList); // Copy !
```

BlackJack
-deck: Deck -playerHand: Hand -bankHand: Hand -gameFinished: boolean
+BlackJack() +reset(): void *throws EmptyDeckException* +getPlayerHandString(): String +getBankHandString(): String +getPlayerBest(): int +getBankBest(): int +isPlayerWinner(): boolean +isBankWinner(): boolean +isGameFinished(): boolean +playerDrawAnotherCard(): void *throws EmptyDeckException* +bankLastTurn(): void *throws EmptyDeckException* +getPlayerCardList(): List<Card> +getBankCardList(): List<Card>

Hand
-hand: LinkedList<Card>
+Hand() +toString(): String +add(card:Card): void +clear(): void +count(): Collection<Integer> +best(): int +getCardList(): List<Card>

Pour une main donnée, nous voulons ajouter au panel associé :
– Chacune des images des cartes de la main
– La valeur de score sous la forme d'un `JLabel` textuel.
– la carte `blackjack` si le meilleur score est 21.
– si le jeu est terminé l'une des cartes `winner`, `looser` ou `draw`

Nettoyer avant d'ajouter Pensez à supprimer tous les anciens composants avant d'en rajouter de nouveaux avec la méthode `removeAll()` de `JPanel`. Une fois terminé l'ajout des nouveaux composants, mettez à jour l'affichage avec la méthode `updateUI()` de la même classe.

EXERCICE 14

Modifiez votre classe `BlackJackGUI` pour rajouter les méthodes :
– `private void updatePlayerPanel()throws FileNotFoundException`
– `private void updateBankPanel()throws FileNotFoundException` qui mettent à jour les panneaux `playerPanel` et `bankPanel` en fonctions des mains du joueur et de la banque.

BlackJackGUI
-bj: BlackJack -playerPanel: JPanel -bankPanel: JPanel
+BlackJackGUI() +addToPanel(p:JPanel,token:String) +updatePlayerPanel(): void *throws FileNotFoundException* +updateBankPanel(): void *throws FileNotFoundException* +main(args:String[]): void

Un peu d'action ! Pour déclencher le jeu, il ne manque plus qu'à rendre nos boutons réactif ! Pour cela on utilise un gestionnaire d'événement : un `ActionListener`

http://docs.oracle.com/javase/7/docs/api/index.html?java/awt/event/ActionListener.html

Exemple d'utilisation :

```
public class GUI implements ActionListener {
   public GUI() {
      JButton button1 = new JButton("Bouton1");
      button1.setActionCommand("key1");
      button1.addActionListener(this);
      JButton button2 = new JButton("Bouton2");
      button2.setActionCommand("key2");
      button2.addActionListener(this);
   }
   public void actionPerformed(ActionEvent e) {
      switch(e.getActionCommand()) { // Valid since Java 1.7
         case "key1": /* do something */ break;
         case "key2": /* do something */ break;
      }
   }
}
```

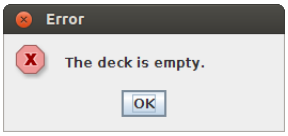
En implémentant l'interface `ActionListener` nous redéfinissons la méthode
`public void actionPerformed(ActionEvent e)` qui sera appelée lors de l'événement clic bouton. C'est dans cette méthode que nous allons décrire ce qui doit se passer lors de l'appuie sur le bouton.

On relie le bouton au gestionnaire par le biais de la méthode `addActionListener`, mais comme on va relier plusieurs bouton au même gestionnaire on associe chacun des boutons à un mot clef, qui nous permettra de distinguer les sources lors de l'appel de la méthode `actionPerformed`.

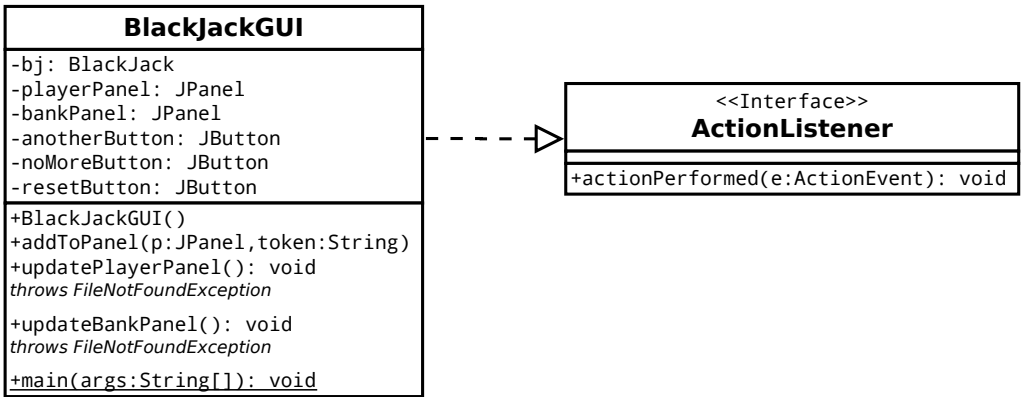
Dans notre cas, la méthode `actionPerformed` va :
- appeler les méthodes `playerDrawAnotherCard`, `bankLastTurn` ou `reset` de notre instance de `BlackJack` en fonction du bouton source de l'événement.
- si la partie est terminée, elle va désactiver (`button1.setEnabled(false)`) les boutons `anotherButton` et `noMoreButton`, en revanche si la partie n'est pas terminée, elle va les activer (`button1.setEnabled(true)`).
- elle mettre à jour les panneaux avec les méthodes `updatePlayerPanel()` et `updateBankPanel()`. Pensez à attraper l'exception potentielle et à afficher un message au cas où.

En cas d'exception vous pouvez afficher une fenêtre de dialogue d'erreur avec la commande suivante :

```
try { /* ... */ } catch (EmptyDeckException ex) {
   JOptionPane.showMessageDialog(null, ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
   System.exit(-1);
}
```



Le nouveau diagramme UML de la classe devient :



Félicitations, vous venez de finir votre premier jeu de BlackJack avec interface graphique en Java !

Pour aller plus loin

Parce qu'on peut toujours avoir d'autres envies, on peut envisager de :
– rajouter la notion de mise et de solde (argent restant au joueur).
– rajouter la possibilité d'avoir plusieurs joueurs (dans un jeu à plusieurs joueurs, la Banque s'arrête lorsqu'elle atteint 17 ou plus.)
– Ce que vous avez envie !