PROJET PUISSANCE 4 SOUS ANDROID

1 – Introduction

L'objectif de ce projet est la mise en application des principes de la programmation orientée objet sous Android.

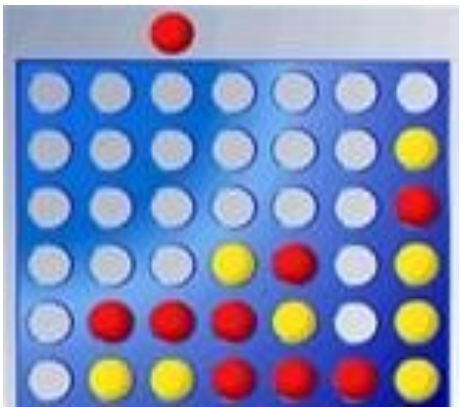
Tous ces aspects qui seront abordés pendant le cours seront mis en pratique au travers de l'implémentation du jeu « Puissance 4 ».

2 – Informations de base : Jeu - Classes - Packages

- Le Jeu et sa réalisation

Puissance4 est un jeu de stratégie. Le but du jeu est d'aligner 4 jetons de la même couleur sur une grille comptant 6 rangées et 7 colonnes.

Les éléments du jeu:
2 joueurs nommés respectivement `joueurJaune` et `joueurRouge` (en référence aux couleurs des jetons)
1 plateau de 6x7 cases (6 lignes et 7 colonnes)
21 jetons de couleur jaune
21 jetons de couleur rouge



source: http://www.stratozor.com/puissance-4/

Résumé du jeu:

Les jetons sont au nombre de 42 : 21 jetons rouges et 21 jetons jaunes.
Il y a deux joueurs et chacun d'eux reçoit 21 jetons de la même couleur. Le joueur qui reçoit les jetons de couleur jaune s'appelle « joueurJaune » et son adversaire s'appellera « joueurRouge ».
Le jeu se déroule sur un tableau de 7 colonnes et 6 lignes:

- Chaque joueur joue à tour de rôle avec des Jetons de sa couleur.
- Le joueur lâche son Jeton dans une colonne. Ce Jeton se place dans la case libre la plus basse.
- Le premier joueur à aligner 4 Jeton de sa couleur (horizontalement, verticalement, ou en diagonale) gagne la partie.
- Si le tableau est rempli sans aucun alignement (chaque joueur a joué 21 Jetons) la partie est nulle.

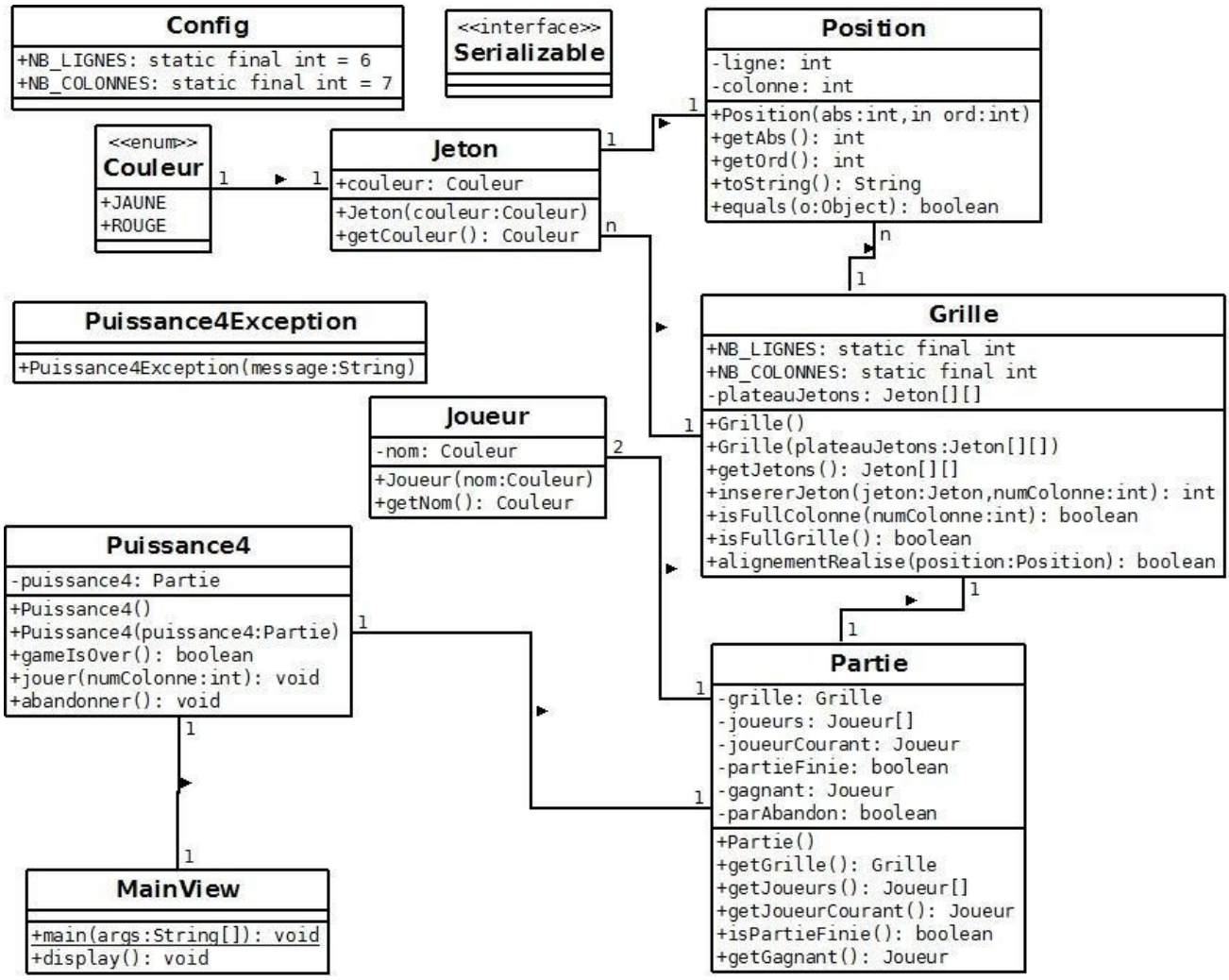
Réalisation
Nous vous demandons de réaliser une version console du jeu avec les adaptations suivantes:

- il n'y a pas d'intelligence artificielle, le jeu se déroulera entre 2 joueurs humains;
- le choix du joueur qui commence la partie se fait de manière aléatoire;
- il est possible d'abandonner la partie, le joueur adverse de celui qui a abandonné sera alors déclaré gagnant;
- chaque partie non terminée peut être sauvegardée à tout moment dans un historique, ce qui permettra de la recharger dans le but de la poursuivre.

- Les classes

Nous vous donnons ici une présentation des classes de base nécessaires à la programmation de ce jeu. Par la suite, nous donnerons une description de chacune de ces classes.

Diagramme de classes



Présentation des classes

Dans votre application, il faut faire la différence entre les classes de la partie métier (business) et celles de la partie vue (view).
La **partie métier** (***business***) rassemble les classes représentant les objets que l'on manipule ainsi que les classes qui sont responsables du travail à faire.
Dans notre cas, il s'agit des classes:

- `Couleur`, pour désigner la couleur du jeton et du joueur;
- `Jeton`, pour représenter un jeton du jeu;
- `Position`, pour désigner les positions sur une grille de jeu;
- `Grille`, pour représenter le tableau sur lequel on va jouer;
- `Joueur`, pour représenter un joueur;
- `Partie`, pour représenter une partie du jeu;
- `Puissance4`, pour gérer complètement une partie du jeu Puissance4;
- `Puissance4Exception`, pour caractériser l'exception qui sera lancée lorsqu'il y a un problème dans le déroulement du jeu. Comme par exemple essayer de placer un Jeton dans une colonne déjà pleine.
- `Config`, qui ne contiendra que des constantes.

La **partie vue** (***view***) ne s'occupe que de présenter le jeu et de l'interaction avec le joueur (en lui demandant ce qu'il veut faire et le faire).
Dans notre cas il s'agit de la classe:

- `MainView`, pour s'occuper de tout l'affichage et de toutes les interactions avec l'utilisateur. C'est elle qui contient la méthode *main()*.

- Les packages

3 – Puissance4 version1

Dans cette première version du jeu, on pourra soit effectuer une insertion de jetons soit abandonner une partie (le reste suivra).
Nous donnons une description des différentes classes de cette première version de Puissance4.

[1] Classe Config

Cette classe regroupe toutes les constantes (de classe) du projet. À ce stade, elle contient:

- une constante représentant le nombre de lignes, `NB_LIGNES`,
- une constante représentant le nombre de colonnes, `NB_COLONNES`.

Les valeurs par défaut de ces constantes sont les suivantes:

```
NB_LIGNES = 6
NB_COLONNES = 7
```

[2] Enum Couleur

la classe Couleur est une énumération. Elle a comme valeurs `JAUNE` et `ROUGE`. Elle n'a aucun attribut.

[3] Classe Jeton

Un Jeton a une certaine couleur. La classe a donc un attribut *couleur* de type Couleur.

attribut:

```
Couleur couleur
```

constructeur :

```
Jeton(Couleur couleur)
```

méthodes :

Pas de mutateur car un Jeton ne change pas.

[4] Classe Puissance4Exception

Une exception propre au jeu. Inspirez-vous de vos notes de cours pour écrire cette classe.

[5] Classe Position

Cette classe permet de désigner des positions sur la grille de jeu. Une position est représentée par une ligne et une colonne. Pensez à valider ces valeurs (elles ne peuvent pas être inférieures à zéro, une IllegalArgumentException sera générée autrement).
Nous supposons que la première colonne à gauche de la grille est d'indice zéro (0) et que la ligne d'indice zéro (0) correspond à la ligne la plus élevée de la grille de jeu.

attributs:

```
int ligne ;          int colonne;
```

constructeur :

```
Position (int ligne, int colonne)
```

méthodes :

Dans un premier temps, nous allons nous contenter des méthodes de base. Nous pourrions l'enrichir par la suite.
Il faut tout de même noter qu'une Position est **immuable**, nous n'écrirons donc pas de mutateur.

[6] Classe Grille

Une Grille est composée de Jetons. Il s'agit d'un tableau de `NB_LIGNES x NB_COLONNES` Jeton.

constantes publiques de classe :

`int NB_LIGNES`	le nombre de lignes du tableau, a pour valeur la valeur `NB_LIGNES` de la classe `Config`.
`int NB_COLONNES`	le nombre de colonnes du tableau, a pour valeur la valeur `NB_COLONNES` de la classe `Config`.

attribut:

`plateauJetons,` `Jeton[][] jetons.`

constructeurs :

`Grille( ),` le constructeur crée le tableau `NB_LIGNES x NB_COLONNES` ne contenant aucun Jeton.

`Grille( Jeton[][] plateauJetons),`
ce constructeur ne sera utilisé que pour les tests.
Il initialise la grille avec le tableau passé en paramètre.
Lance une `Puissance4Exception` si le tableau est invalide.

Par invalide, on entend soit le tableau est null, soit que le nombre de lignes ou le nombre de colonnes ne correspond pas aux valeurs des constantes définies ci-dessus

méthodes :

`Jeton getJeton(Position position)`

retourne le Jeton à la position correspondante de la grille; elle retourne null si la position est libre. Si la position est invalide (c-à-d dont la ligne ou la colonne est en dehors des limites du plateau) une `IllegalArgumentException` sera lancée.

`boolean isFullColonne(int numColonne)`

retourne vrai si la colonne correspondante sur la grille est remplie et faux sinon. Si la colonne est invalide une `IllegalArgumentException` sera lancée.

`int  insererJeton(Jeton jeton, int numColonne)`

tente d'insérer un Jeton à la colonne correspondante de la grille. Retourne l'indice de la ligne où le pion a été inséré.
Si la colonne est invalide une `IllegalArgumentException` sera lancée.
Si la colonne est déjà remplie, une `Puissance4Exception` sera lancée.

`boolean isFullGrille()`

retourne vrai si toute la grille est remplie et faux sinon.

`boolean alignementRealise(Position position)`

Teste si il existe un alignement valide de jetons "passant" par la position fournie en argument.
Retourne vrai s'il y a un alignement de 4 jetons de même couleur qui passe par cette position et faux dans le cas contraire.

Pour vous aider voici une macro logique qui détecte un alignement horizontal :

```
MODULE alignementHorizontal (Position position) -> ENTIER
// ce module retourne le nombre de jetons de même couleur alignés horizontalement
// sur celui en position

couleur : Couleur                          // la couleur du jeton en position
ligne, colonne, nbAlignés : ENTIER
mêmeCouleur : BOOLEEN

ligne <- position.ligne
colonne <- position.colonne
couleur ← la couleur du jeton en position // attention si « pas de jeton »
nbAlignés <- 1 // le jeton même est le seul aligné pour l'instant
mêmeCouleur <- vrai

// regardons à gauche, attention à l'indice de colonne
TANT QUE mêmeCouleur ET colonne > 1
        colonne <- colonne -1     // suivant à gauche
        mêmeCouleur <- il existe un jeton en (ligne,colonne) ET il a la bonne couleur
        SI mêmeCouleur ALORS      // un de plus aligné
                        nbAlignés ← nbAlignés + 1
        FIN SI
FIN TANT

// revenons au jeton et regardons à droite
// attention à l'indice de colonne

colonne <- position.colonne
mêmeCouleur <- vrai

TANT QUE mêmeCouleur ET colonne < NB_COLONNES
        colonne <- colonne + 1    // suivant à droite
        mêmeCouleur <- il existe un jeton en (ligne,colonne) ET il a la bonne couleur
        SI mêmeCouleur ALORS      // un de plus aligné
                        nbAlignés ← nbAlignés + 1
        FIN SI
FIN TANT

RETOURNER nbAlignés
FIN MODULE
```

Vous adapterez cela pour trouver un alignement vertical ou diagonal. Dans ce dernier cas il faudra bouger de colonne et de ligne à la fois et veiller à ne pas dépasser les limites pour chacun des deux.

méthodes supplémentaires: vous êtes libres de les implémenter si besoin.

```
int alignementHorizontal(Position position)
int alignementVertical(Position position)
int alignementDiagonal1(Position position)
int alignementDiagonal2(Position position)
```

ces 4 méthodes permettent de déterminer (dans chacune des 4 directions), le nombre de jetons alignés de la même couleur "passant" par la position fournie en argument.

[7] Classe Joueur

Un Joueur représente un des participants à une partie de jeu Puissance4.

attribut :

```
Couleur  nom;
```

constructeur:

```
Joueur(Couleur nom)
```

Cette classe est immuable.

[8] Classe Partie

Une Partie nous fournit toutes les informations nécessaires au fur et à mesure de notre progression dans le jeu.

attributs:

`Grille grille`	la grille du jeu;
`Joueur[] joueurs`	le tableau de 2 joueurs (joueurJaune et joueurRouge);
`Joueur joueurCourant`	le joueur courant (joueurJaune ou joueurRouge);
`boolean partieFinie`	à true si la partie est finie et à false sinon;
`boolean parAbandon`	à true si la partie s'est terminée par abandon à false sinon.
`Joueur gagnant`	le joueur gagnant si la partie est finie et qu'il y a un gagnant; `null` sinon.

Constructeur:

```
Partie()
```

le constructeur va créer tout ce qui est nécessaire au démarrage d'une nouvelle partie entre 2 Joueur. Le joueur qui commence est choisi au hasard.

Méthodes:

les méthodes de base. Comme par exemples les getters, setters, etc.

Il ne faut pas perdre de vue la modularité de votre code.

[9] Classe Puissance4

C'est la classe métier (business), elle a en charge toute la gestion du jeu mais **aucune entrée/sortie**. Cette classe connait le vainqueur de la partie. Elle dit également si la partie s'est terminée par l'abandon d'un des joueurs.

attributs:

`Partie puissance4` une partie du jeu;

Constructeurs:

`Puissance4()`

le constructeur crée le jeu.

`Puissance4(Partie puissance4)`

le constructeur utilisé pour la récupération d'une partie sauvegardée afin de créer le jeu.

Méthodes:

`boolean gameIsOver()`

retourne vrai si la partie est finie et faux sinon.

`jouer(int numColonne)`

permet de jouer un jeton,
jouer un jeton = si la partie n'est pas finie, tenter d'insérer un Jeton dans la Grille et mettre à jour ce qu'il faut.

`abandonner()`

permet d'abandonner la partie.
Cette méthode met à vrai l'attribut `parAbandon`, désigne comme `gagnant` le joueur autre que le joueur courant et met fin à la partie.

[10] Classe MainView

Cette classe est la vue principale (le point d'entrée) de l'application.
Cette classe devra avoir comme variable locale une instance de `Puissance4`. Elle se charge de l'affichage et de l'interaction avec l'utilisateur.
C'est cette classe qui contient la méthode *`main()`*.

La logique de cette méthode ***`main()`*** est :

```
Créer une instance de  Puissance4
Tant que la partie du jeu n'est pas finie
      Afficher la partie du jeu puissance4
      Demander au joueur courant l'action qu'il propose
      Effectuer cette action
Fin tant que
Dire s'il y a eu abandon
Afficher le gagnant.
```

En plus de la méthode `main()`, on devrait retrouver ici une méthode permettant d'afficher une partie de jeu.

`display()`

affiche la grille avec son contenu c'est-à-dire les jetons, et si possible d'autres informations qui concernent la partie du jeu.

4 – Puissance4 version2

Nous allons ajouter d'autres fonctionnalités à notre application comme par exemple:

- la configuration et le fichier *properties*.
- la possibilité de sauvegarder une partie et de la reprendre par la suite;
- la distribution de votre application via un fichier jar;
- l'implémentation d'un script **`puissance4`** permettant de lancer votre application;
- les arguments en ligne de commande.

Dans cette troisième phase, nous ne donnons pas de diagramme de classes et nous vous dirigeons moins dans la réalisation des diverses fonctionnalités. À vous d'être plus autonomes.

7.1- Configuration et fichier properties

Fichier properties.

La lecture des valeurs des paramètres NB_LIGNES et NB_COLONNES se fera depuis un fichier de propriétés.

Si nous considérons le fichier `config.properties`

```
$cat config.properties
NB_LIGNES = 6
NB_COLONNES = 7
```

Ces valeurs de `NB_LIGNES` et `NB_COLONNES` pourront être récupérées par l'intermédiaire de la classe `java.util.Properties`

```
Properties properties = …
… properties.getProperty("NB LIGNES")
```

Pour en savoir plus, consultez la classe **`java.util.Properties`** dans l'API java.

Block static.

Pour permettre l'initialisation de nos deux constantes une seule fois au chargement de la classe, nous allons utiliser un bloc statique.

```
public class Config{
      public static final int NB_LIGNES;
      public static final int NB_COLONNES;
      static{
            // Faire tout un travail ici !
             NB_LIGNES = 6;
             NB_COLONNES = 7;
      }
}
```

Il faudra modifier la classe `Config` pour qu'elle puisse lire ces deux constantes dans un fichier de propriétés.

Modification de Config.

- créer une variable de type Properties;
- charger un fichier *properties* (s'il est dans le même répertoire)

```
InputStream file = Config.class.getResourceAsStream("config.properties");
properties.load(file);
```

- lire les différentes propriétés et les assigner aux attributs statiques constants.

- Possibilité de sauvegarder une partie

Nous devons donner aux joueurs la possibilité de faire une pause et de reprendre la partie qu'ils ont démarrée.

Pour cela, avant de faire cette pause, la classe Puissance4 doit pouvoir sauvegarder la partie en cours dans un fichier.

À la fin de la pause c'est-à-dire à la reprise, la classe Puissance4 récupère la partie sauvegardée depuis le fichier.

Vous devez maintenant vous documenter sur la façon de sauvegarder un objet dans un fichier, de lire un objet à partir d'un fichier et sur la *sérialisation* des objets.

Modification de la classe Puissance4.

- ajouter un attribut `lastParties` de type `Stack<Partie>`
- maintenir cette `Stack` à jour à chaque fois que l'on sauve ou récupère une partie;
- ajouter une méthode `sauver(String nomFichier)` qui permet de sauver la partie en cours dans un fichier (sauver la pile de Partie dans fichier);
- ajouter une méthode `charger(String nomFichier)` qui permet de récuper une partie qui a été sauvée (affecter la pile avec la pile de Partie contenue dans le fichier).

Sérialisation.

Si vous optez pour la *sérialisation* (ce que nous vous conseillons vivement), vous devez penser à rendre les classes `Partie`, `Grille`, `Joueur`, `Position` et `Jeton` sérialisables.

- Distribution via un fichier jar

Dans la pratique, pour distribuer un programme java, toutes les classes du programme sont archivées dans une archive au format `.jar`.

S'il existe un JRE sur la machine, le programme pourra être lancé soit en double-cliquant sur l'icône, soit en invoquant la machine virtuelle avec l'option `-jar`, comme suit:

```
java -jar puissance4.jar
```

Il faut lire la documentation concernant les fichiers `.jar` fournie par *Oracle*.

Création d'une archive java.

```
echo "Main-Class: g12345.puissance4.MainView" > manifest.mf
jar cfmv puissance4.jar manifest.mf  *.class
```

- Script Puissance4

Écrire un script très simple qui permet de lancer le jeu. Ce script s'appelle **`puissance4`** et permet de lancer le fichier `jar`.

Le script fera suivre au programme java les paramètres éventuels qu'il reçoit (voir **7.5** arguments en ligne de commandes ou arguments au programme ci-dessous).

Il faut se documenter sur **la notion de script sous Linux.**

- Arguments au programme (arguments en ligne de commandes)

Nous devons offrir la possibilité de donner des arguments à son programme (par le biais du tableau de String ***`args`***).

`puissance4`

lance le jeu avec les paramètres par défaut

`puissance4 help`

affiche une aide sur l'utilisation du programme

`puissance4 config`

permet de choisir les valeurs des différents attributs (nombre de lignes, nombre de colonnes) et de les sauvegarder dans un fichier propre à l'utilisateur.

`puissance4 load`

relance une partie qui a été précédemment sauvegardée.

Action config.

Pour ajouter cette fonctionnalité, il faut:

- gérer l'écriture d'une configuration dans un fichier
- gérer l'interface avec l'utilisateur

Écriture d'une configuration dans un fichier.

Modifier la classe `Config` afin qu'elle propose une méthode `save` permettant de sauvegarder un fichier *properties* dans le répertoire *`home`* de l'utilisateur.

- Obtenir le répertoire *`home`* de l'utilisateur se fait grâce à

```
System.getenv("HOME");
```

- Le fichier portera le nom `.puissance4.properties` et sera placé dans le `home` de l'utilisateur
- Il faut réfléchir à quels paramètres passer à la méthode `save`
- Il est important (re)lire la `javadoc` de la classe `java.util.Properties` avant d'écrire cette sauvegarde

Modifier le bloc statique afin que le code qu'il contient essaie d'abord de lire le fichier *properties* de l'utilisateur avant de lire le fichier par défaut.

Gérer l'interface avec l'utilisateur.

Lorsque l'utilisateurr choisit l'action `config`, il faut lui demander ses choix de valeurs et ensuite les sauver (par appel à la bonne méthode dans la classe `Config`).

Cette interaction se fait bien sûr dans la classe `MainView`.