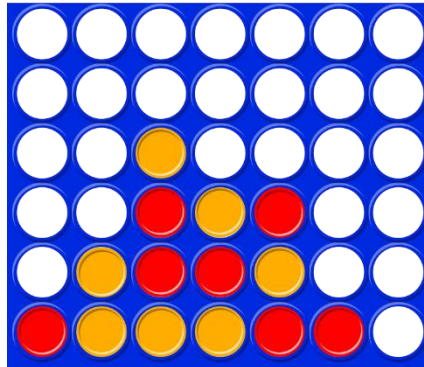


## Introduction

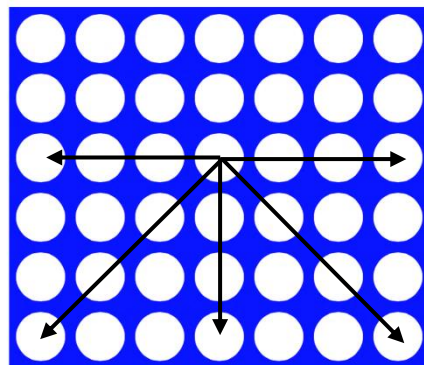
**Puissance 4** est un jeu à deux joueurs qui consiste à aligner quatre jetons d'une même couleur dans une grille.



Les joueurs disposent d'une grille vide de **sept colonnes** et **six lignes**.

Chaque joueur à son tour dépose un jeton de sa couleur dans l'une des colonnes. Le jeton tombe alors en bas de la colonne ou au-dessus du dernier jeton déposé dans cette colonne.

La partie se termine quand, en posant son jeton, un joueur forme une rangée de **quatre jetons identiques** en ligne, en colonne ou en diagonale.



## Sujet

On propose de réaliser un Puissance 4 en Java.

La grille sera affichée en console (`System.out.print`).

A chaque tour :

- On affichera la grille dans son état actuel ;
- On indiquera le joueur qui doit jouer ;
- On récupèrera le nombre saisi par l'utilisateur (`System.in`), équivalant à la colonne dans laquelle déposer le jeton ;
- Si le coup est gagnant, on affiche le vainqueur.

On proposera par ailleurs différentes options de saisie à l'utilisateur :

- « `exit` » pour quitter l'application
- « `restart` » pour réinitialiser la partie

Enfin, on sauvegardera les parties jouées dans une base de données et on permettra à l'utilisateur de la consulter.

## Prérequis

- ✓ Java 13
- ✓ Eclipse for Java EE

## Récupération des sources

Télécharger le fichier à l'adresse suivante :

<http://louiecinophile.fr/tp-java.zip>

Extraire l'archive. S'y trouvent les fichiers suivants :

- ✓ **src.zip** : les sources initiales du projet Puissance 4
- ✓ **mysql-connector-java-8.0.18.jar** : le driver MySQL qui sera utilisé plus tard
- ✓ **TP-Puissance-4.pdf** : le sujet du TP
- ✓ **Java.pdf** : le cours présenté au préalable

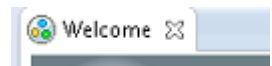
## Initialisation

### 1. Ouvrir l'environnement

- Ouvrir Eclipse.

L'assistant vous demande de choisir un emplacement pour votre **workspace**. Un workspace peut contenir plusieurs projets et permet de « ranger » ses réalisations.

- Choisir un emplacement (celui par défaut est souvent satisfaisant) et valider.
- Fermer la page de bienvenue. Vous voici dans l'IDE à proprement parler.



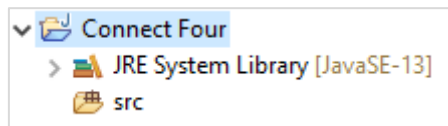
### 2. Créer le projet

- Cliquer sur « File > New > Java Project ».
- Nommer le projet « Connect Four ».
- Vérifier que la JRE est bien configurée sur **JavaSE-13**.
- Cliquer sur Finish.
- Préciser « Don't Create » à la création de module.

Les autres options vous permettent de configurer l'arborescence du projet, l'intégration des sources les bibliothèques utilisées, etc.

Vous découvrirez tout ça à l'usage. Ici, les valeurs par défaut nous conviennent.

Vous obtenez alors le résultat suivant :



- **JRE System Library** contient les bibliothèques standard Java (permettant d'utiliser `java.util.*`, etc.) ;
- **src**, vide au départ, contiendra les packages de votre application, qui eux-mêmes contiendront vos classes.

### 3. Importer les sources

- Cliquer avec le bouton droit de la souris sur le projet.
- Sélectionner « Import... ».
- Dans la boîte de dialogue, choisir « Général > Archive File ».
- Renseigner le fichier **src.zip**, puis cliquer sur Finish.

C'est fait ! Tout est installé, on va pouvoir démarrer !

# Présentation des sources

Le projet contient un package, contenant lui-même trois packages que sont :

- **connectfour**
  - **app** Le système de l'application toute entière
  - **model** La modélisation du jeu
  - **view** L'interface permettant à l'utilisateur d'interagir avec le modèle

## 1. connectfour.app


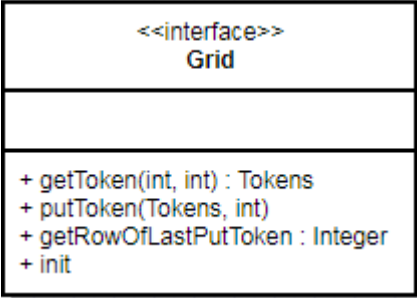

Ce package contient la classe `ConnectFour`.

Dotée d'une méthode statique `main`, c'est elle qui sera appelée à l'exécution de l'application.

Elle crée une interface graphique (`GameView`), lance le jeu, puis se termine.

## 2. connectfour.model

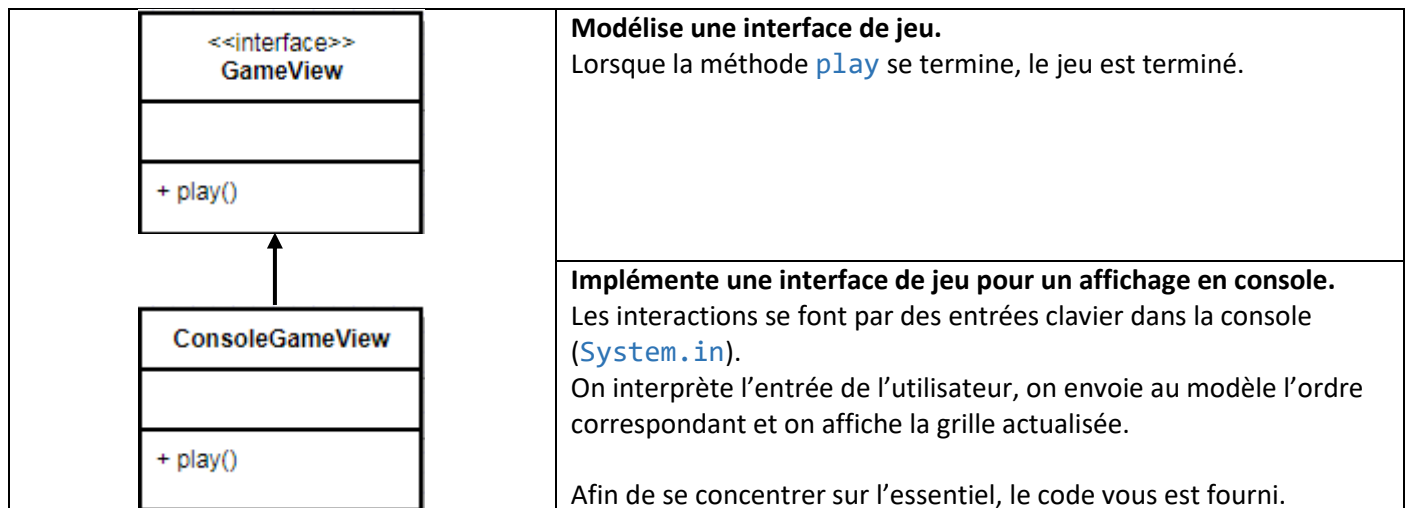
Ce package contient deux interfaces et une énumération :

 <pre> classDiagram     class Tokens {         &lt;&lt;enumeration&gt;&gt;         RED         BLUE         toString() String     } </pre>	<p><b>Modélise les joueurs.</b></p> <p>Tokens.RED = Joueur 1 Tokens.BLUE = Joueur 2 La méthode <code>toString()</code> est surchargée pour retourner un symbole représentant le jeton du joueur (RED = « O » ; BLUE = « X »).</p>
 <pre> classDiagram     class Grid {         &lt;&lt;interface&gt;&gt;         +getToken(int, int) Tokens         +putToken(Tokens, int)         +getRowOfLastPutToken Integer         +init     } </pre>	<p><b>Modélise une grille de jeu.</b></p> <p>Ne gère que la pose de jeton : les règles sont implémentées un niveau au-dessus.</p>
 <pre> classDiagram     class Game {         &lt;&lt;interface&gt;&gt;         +int COLUMNS         +int ROWS         +int REQUIRED_TOKENS         +getToken(int, int) Tokens         +getCurrentPlayer() Tokens         +isOver() boolean         +getWinner() Tokens         +putToken(int)         +init()     } </pre>	<p><b>Modélise le jeu.</b></p> <p>Possède une grille et permet de récupérer différentes informations sur la partie en cours (le joueur qui doit jouer, si la partie est terminée, etc.). L'interface possède également les attributs COLUMNS et ROWS qui seront passés en arguments au constructeur de la grille, ainsi que REQUIRED_TOKENS qui indique le nombre de jetons à aligner pour gagner.</p>

`Grid` et `Game` possèdent toute la `javadoc` nécessaire sur chacune de leurs méthodes.

### 3. Connectfour.view

Ce package contient une interface et son implémentation.



Vous l'aurez compris, vous allez principalement modifier le package `model` pour réaliser le jeu.

Si tout est clair, on va pouvoir entamer la réalisation.

## Développement

### 1. Une grille à remplir

En l'état, l'application ne compile pas car `ConnectFour` fait appel à une classe qui n'existe pas.

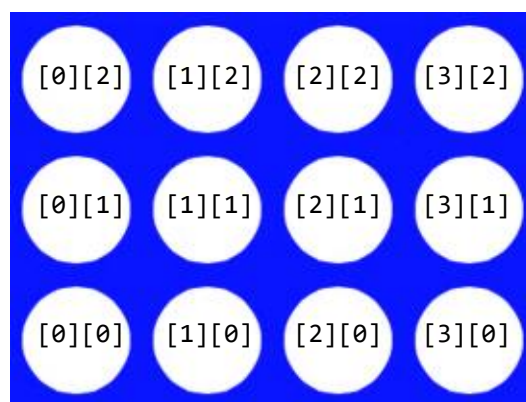
Nous allons donc commencer par implémenter les classes manquantes.

#### a) Grid

- Cliquer avec le bouton droit de la souris sur le package `model`, puis sélectionner « `New > Class` ».
- Nommer cette classe « `ImplGrid` » et lui ajouter l'interface `Grid`.
- Cliquer sur « `Finish` ».

Eclipse crée la classe et génère les méthodes définies par l'interface qui doivent être implémentées.

On propose de modéliser la grille à l'aide d'un tableau de `Tokens` à deux dimensions. La première dimension correspondra aux colonnes, la seconde aux lignes.



- Créer l'attribut privé constant « `grid` » de type « tableau de Tokens à deux dimensions ».
- Créer l'attribut privé « `rowOfLastPutToken` » de type « entier objet ». Il vaudra `null` tant qu'aucun coup n'a été joué.

- Créer le constructeur public qui initialisera le tableau à l'aide de deux arguments entiers.
- Implémenter la méthode `getToken(x, y)`.
- Implémenter la méthode `init()`. Penser à initialiser `rowOfLastPutToken` pour les parties suivantes.
- Implémenter la méthode `putToken(token, x)`. On propose l'algorithme suivant :

```

1.  $y \leftarrow 0$ 
2. Tant que  $y < \text{nombre de lignes}$  et qu'il y a un jeton en  $(x,y)$ 
    $y \leftarrow y + 1$ 
3. Placer le jeton en  $(x,y)$ 
4.  $\text{rowOfLastPutToken} \leftarrow y$ 

```

- Implémenter la méthode `getRowOfLastPutToken()`.

`Grid` possède désormais une première implémentation. Vous n'avez pas encore les outils pour tester ses méthodes, mais l'implémentation de `Game` devrait nous y aider.

## b) Game

Il nous faut désormais implémenter le système de jeu pour pouvoir (enfin) exécuter l'application.

- Créer dans le package `model` la classe `ImplGame` qui implémente l'interface `Game`.

Miracle, le projet compile !

Avant même d'implémenter les méthodes de `Game`, vous pouvez exécuter l'application. La grille s'affiche avec le nombre de lignes et de colonnes défini dans l'interface `Game`.

La console attend une entrée de l'utilisateur correspondant à la colonne dans laquelle déposer le jeton. Malheureusement, tout reste à faire ! La vue n'est associée à aucun modèle, donc il ne se passe rien.

Remplissons donc un peu la classe `ImplGame`...

- Créer l'attribut privé constant `grid` de type `Grid`.
- Créer un constructeur ne prenant aucun argument et initialisant la grille.
- Implémenter les méthodes `getToken` et `init`
- Implémenter la méthode `putToken`. Avant de réaliser la mécanique des joueurs, nous allons simplement passer `Tokens.RED` à la méthode de la grille.

Exécuter l'application.

Saisissez un nombre entre 0 et 7 dans la console. Vous pouvez alors tester toutes les fonctionnalités développées jusqu'ici : la récupération de jeton (`getToken`) et la pose (`putToken`) en jouant dans différentes colonnes.

Prévoyons maintenant deux joueurs qui joueront chacun leur tour ! `ImplGame` aura cette responsabilité car sa méthode `putToken` ne prend pas de jeton en argument.

- Créer l'attribut privé `currentPlayer` de type `Tokens`.
- Implémenter la méthode `getCurrentPlayer`.

On souhaite qu'à l'initialisation, le joueur courant soit tiré au hasard.

- Compléter la méthode `init` pour donner à `currentPlayer` un `Tokens` au hasard. Cette méthode doit fonctionner quel que soit le nombre d'éléments dans `Tokens`. Pour ce faire, on utilisera :
  - La méthode statique `values()` de `Tokens`
  - `Math.random()`
  - `(int) (X)` permettant de tronquer le décimal `X` pour obtenir l'entier immédiatement inférieur
- Appeler la méthode `init` dans le constructeur de `ImplGame` afin d'initialiser le joueur courant à la création
- Créer la méthode privée `getNextPlayer()` qui retourne un `Tokens` correspondant au joueur suivant le joueur actuel

Cette méthode doit fonctionner quel que soit le nombre d'éléments dans `Tokens`.

Pour ce faire, on utilisera :

- La méthode statique `values()` de `Tokens`
- La méthode `ordinal()` de `Tokens`
- L'opérateur modulo `%`
- Compléter la méthode `putToken` afin de passer `currentPlayer` à la grille, puis le mettre à jour une fois le jeton déposé

On remarque qu'on a beaucoup utilisé la méthode `Tokens.values()`... Et qu'elle va être appelée un certain nombre de fois durant la partie. Peut-être pourrait-on stocker son résultat quelque part ?

- Créer l'attribut privé statique constant `TOKEN_VALUES` de type « tableau de `Tokens` », et l'initialiser à `Tokens.values()`
- Remplacer dans `ImplGame` toutes les occurrences de `Tokens.values()` par `TOKEN_VALUES`

Exécuter l'application.

Si tout va bien, les joueurs jouent chacun leur tour ! Tantôt « X », tantôt « O »...

### c) ConnectException

**Problème** : rien n'empêche de mettre un septième jeton dans une colonne. On peut également poser un jeton dans une colonne qui n'existe pas. Et là, c'est le drame.

```
C'est au tour de [X] ! [0-6] : 1
java.lang.ArrayIndexOutOfBoundsException: 6
    at connectfour.model.ImplGrid.putToken(ImplGrid.java:13)
    at connectfour.model.ImplGame.putToken(ImplGame.java:39)
    at connectfour.view.ConsoleGameView.play(ConsoleGameView.java:44)
    at connectfour.view.ConsoleGameView.play(ConsoleGameView.java:59)
    at connectfour.view.ConsoleGameView.play(ConsoleGameView.java:59)
    at connectfour.view.ConsoleGameView.play(ConsoleGameView.java:59)
    at connectfour.view.ConsoleGameView.play(ConsoleGameView.java:59)
    at connectfour.view.ConsoleGameView.play(ConsoleGameView.java:59)
    at connectfour.view.ConsoleGameView.play(ConsoleGameView.java:59)
    at connectfour.app.ConnectFour.main(ConnectFour.java:16)
```

Pour corriger ce dysfonctionnement, on va empêcher le système de traiter une valeur incorrecte.

- Créer dans le package `model` la classe `ConnectException` qui hérite de la classe `Exception`

Un avertissement vous indique que la classe n'a pas d'attribut `serialVersionUID`. Il sert pour la sérialisation de l'objet, c'est-à-dire sa transformation en fichier lorsqu'on souhaite le sauvegarder de cette façon. Choisir la première correction que vous propose l'IDE.

La classe `ConnectException` n'apportera rien de plus à `Exception` que son nom spécifique. Nous utiliserons systématiquement cette classe en cas de souci dans le système.

- Munir `ConnectException` d'un constructeur prenant un `String` en argument et appeler le constructeur similaire de sa classe parente (mot-clé `super`). Ceci nous permettra de renseigner un message d'erreur.

C'est au niveau le plus bas, celui de `Grid`, que nous allons gérer les arguments incorrects.

- Modifier l'interface de `Grid` pour indiquer que `putToken` peut lancer une `ConnectException`.
- Modifier la méthode `putToken` de `ImplGrid` pour lancer une `ConnectException` :
  - Lorsque la colonne passée en argument est inférieure à 0 ou supérieure au nombre de colonnes
  - Lorsque la colonne est déjà pleine

Tant qu'on y est, on pourrait vérifier tous les arguments qui sont transmis, afin d'offrir une classe propre, qui gère tous les cas.

- Modifier le constructeur pour lancer une `IllegalArgumentException` quand le nombre de colonnes ou le nombre de lignes est inférieur à 0.
- Modifier la méthode `getToken` pour lancer une `IllegalArgumentException` quand `(x,y)` ne correspond pas à une position dans la grille.
- Modifier `putToken` pour lancer une `IllegalArgumentException` quand le jeton argument vaut `null`.

`IllegalArgumentException` hérite de `RuntimeException` ; il est donc inutile de la déclarer dans un `throws`. On réserve les `ConnectException` aux règles du jeu (nombre de jetons max dans une grille, etc), et les `RuntimeException` aux situations qui révèlent des bugs dans le système.

Il y a une erreur de compilation dans `ImplGame` : comme `putToken` peut lancer une `ConnectException`, il faut la traiter.

Cependant, on ne souhaite pas attraper l'exception dans `Game`. On préfère la renvoyer au niveau supérieur, pour que ce soit l'interface qui la traite.

- Modifier l'interface `Game` et la classe `ImplGame` pour indiquer que `putToken` peut lancer une `ConnectException`.

Du coup, maintenant, l'erreur est dans `ConsoleGameView`.

- Ouvrir `ConsoleGameView`.

`putToken` est déjà dans un bloc `try/catch` nécessaire pour transformer la chaîne saisie par l'utilisateur en nombre entier (`Integer.parseInt()`).

- Ajouter un bloc `catch` qui attrape une `ConnectException` sur le modèle de la précédente.  
**Astuce** : utiliser `e.getMessage()` (méthode héritée de `Exception`).

Exécuter l'application.

Vérifier que tous les cas de figure sont pris en compte (entrée inférieure à 0, colonne pleine, entrée supérieure au nombre de colonnes) et que vous ne parvenez pas à faire exploser le programme.

On a désormais une grille fonctionnelle. Il ne reste « plus » qu'à implémenter la fin d'une partie !

## 2. Fin du jeu

La classe `ImplGame` va gérer les règles de fin d'une partie. Pour rappel, une partie prend fin :

- ✓ Quand la grille est remplie (match nul)
- ✓ Ou quand quatre jetons d'une même couleur sont alignés

### a) Grille pleine

Commençons par gérer ce premier cas.

- Implémenter la méthode `isOver` afin de retourner `true` lorsque la grille est complète.  
**Astuce** : on peut faire bien mieux que vérifier chaque case de la grille...
- Modifier la méthode `putToken()` afin de ne mettre à jour le joueur courant que si la partie n'est pas terminée.

On se rend compte que `isOver()` est appelée plusieurs fois à chaque tour... On pourrait peut-être optimiser un peu ça.

- Créer un attribut `over` de type `boolean`.



- Modifier la méthode `init` pour mettre à `over` la valeur `false` (par défaut l'attribut vaut `false` de toute façon, mais si on recommence une partie, on souhaite le réinitialiser).
- Créer une méthode privée `calculateOver` qui retourne un `boolean`. Cette méthode fera les calculs nécessaires, et on s'assurera qu'elle est appelée une et une seule fois, après chaque tour.
- Déplacer le corps de `isOver` dans `calculateOver`.
- Modifier le corps de `isOver` pour simplement retourner `over`.
- Modifier `putToken` pour donner à `over` la valeur de `calculateOver()` après l'appel à `grid.putToken`.

Lancer l'application, et remplir une grille pour vérifier que la fin de la partie est bien détectée.

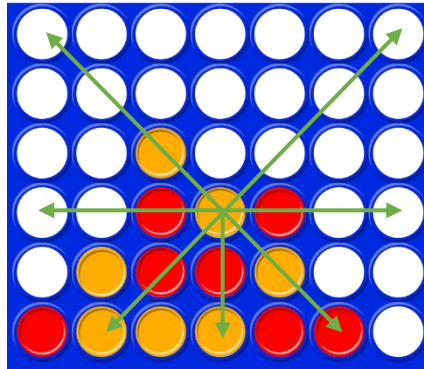
### b) Victoire

- Implémenter `getWinner()` de la même façon que `isOver()` : à l'aide d'un attribut de type `Tokens`, retourné par `getWinner()`. Cet attribut sera mis à jour par `calculateOver()`.

**Attention :** il faut penser à l'inclure dans la méthode `init()`.

Pour étudier l'état de la grille, on ne va pas systématiquement tout parcourir : on va regarder si le dernier coup joué était gagnant, c'est-à-dire s'il a contribué à une rangée de quatre jetons de même couleur.

En admettant qu'on ait posé le jeton jaune dans la colonne 3 :



On regarde les deux diagonales, ainsi que les rangées verticale et horizontale. Si on trouve au moins quatre jetons de même couleur alignés, c'est gagné !

- Modifier la méthode `calculateOver()` pour qu'elle prenne en argument la colonne choisie par le joueur qui vient de jouer. La ligne dans laquelle est tombé le jeton pourra être récupérée par `grid.getRowOfLastPutToken()`.
- Créer les méthodes privées :
  - `inspectSouth()` qui retourne le score de la colonne (nombre de jetons de même couleur consécutifs)
  - `inspectWestEast()` qui retourne le score de la ligne
  - `inspectNWSE()` qui retourne le score de la diagonale nord-ouest → sud-est
  - `inspectNESW()` qui retourne le score de la diagonale nord-est → sud-ouest
 Ces méthodes prennent toutes en argument `x` et `y` coordonnées du jeton posé. L'entier retourné par ces méthodes sera compris entre 1 (le jeton est isolé) et 7 (le jeton tombe entre trois jetons de même couleur d'un côté et trois de l'autre). Pour vous aider, on vous propose le corps de la méthode `inspectNWSE(x, y)` :



```
private int inspectNWSE(int x, int y) {
    int foundInLine = 0;
    for (int i = 1; x - i >= 0 && y + i < ROWS && getToken(x - i, y + i) == currentPlayer; i++) {
        foundInLine++;
    }
    for (int i = 1; x + i < COLUMNS && y - i >= 0 && getToken(x + i, y - i) == currentPlayer; i++) {
        foundInLine++;
    }
    return foundInLine + 1;
}
```

- Modifier la méthode `calculateOver()` pour qu'elle affecte l'attribut `winner` si l'une des quatre méthodes précédentes retourne une valeur supérieure ou égale à `REQUIRED_TOKENS`.  
Dans ce cas, la méthode doit évidemment retourner `true` avant de tester la complétude de la grille.

C'est le moment de tester !

Lancer l'application et essayer d'atteindre les différents cas de victoire (en colonne, en ligne, les deux diagonales).

### 3. Commandes supplémentaires

On souhaite permettre à l'utilisateur de rejouer ou de quitter proprement l'application.

A cette fin, nous allons lui offrir des commandes supplémentaires.

- Créer dans `ConsoleGameView` les attributs constants statiques `KEYWORD_EXIT` et `KEYWORD_RESTART` de type `String`, contenant le mot-clé que devra saisir l'utilisateur.
- Modifier la méthode `play()` pour interpréter l'entrée de l'utilisateur.  
**Astuce :** utiliser `switch/case/default` et `exitRequest`
- Ajouter dans la méthode `play()` juste avant l'appel à `displayGrid()` l'affichage d'une phrase décrivant les commandes possibles pour l'utilisateur.  
Utiliser évidemment la valeur des attributs, afin qu'on puisse les changer sans impacter le code...

Tester l'application.

Les commandes `restart` et `exit` doivent être accessibles n'importe quand dans la partie et une fois que la partie est terminée.

## Pour aller plus loin

### 1. ImplListGrid

On constate dans `ImplGrid` qu'on effectue plusieurs boucles, qui pourraient être évitées avec un modèle de données différent.

- Créer la classe `ImplListGrid` qui implémente `Grid`.
- Implémenter toutes les méthodes en reposant non plus sur un tableau à deux dimensions de `Tokens` mais sur une liste de listes de `Tokens` (`List<List<Tokens>>`).  
**Rappel :** on utilise `ArrayList` pour instancier une `List`.
- Dans `ImplGame`, instancier une `ImplListGrid` plutôt qu'une `ImplGrid`.

C'est ici qu'on constate toute la magie des interfaces (en l'occurrence `Grid`) : on peut changer son implémentation sans devoir revenir sur tous les endroits où ses instances sont utilisées.

Lancer l'application. Vérifier que tous les cas d'erreur sont pris en compte.

## 2. Base de données

On souhaite enregistrer dans une base de données le résultat de chaque partie.

Pour ce faire, on dispose d'une base MySQL distante de test, qui contient les tables `games_[0-9]` qui ont toutes le schéma suivant :

<u>games_X</u>
+ id : int + date : datetime + winner: varchar(20) NULL + winnerType : enum("Human", "CPU") NULL

`id` : identifiant unique auto-incrémenté  
`date` : date et heure de la partie  
`winner` : nom du vainqueur (on utilisera ici « RED » et « BLUE »)  
`winnerType` : indique si le gagnant est un humain ou une IA (pour plus tard...)

- Demander à l'intervenant le numéro de table à utiliser.

### a) Driver JDBC

Comme on l'a vu pendant le cours, il faut installer un driver de base de données pour pouvoir interroger un SGBD en Java. Pas de panique : il vous est fourni !

- Cliquer avec le bouton droit de la souris sur le projet, puis sélectionner « [Build Path > Add External Archives...](#) ».
- Sélectionner le fichier « [mysql-connector-java-8.0.18.jar](#) »

Le driver est installé. On peut désormais utiliser ses classes.

### b) Entité Score

La classe `Score` nous permettra de modéliser une ligne de la base. On appelle ça une entité : elle ne possède que des attributs, des getters et éventuellement des setters, mais aucun code métier. Son rôle est seulement de stocker de l'information.

- Créer le package `connectfour.model.score`.
- Créer dans ce package score une interface `Score` contenant les méthodes suivantes :
  - ✓ `getDate()` : retourne la date et l'heure de la fin de la partie de type `Date (java.util)`.
  - ✓ `getWinner()` : retourne le vainqueur de type `Tokens`.
  - ✓ `getWinnerType()` : retourne le type du vainqueur (humain ou ordinateur) de type `String`.
- Implémenter cette interface dans une classe `ImplScore`. Chaque getter ne fera que retourner l'attribut constant correspondant.
- Ajouter à `ImplScore` un constructeur qui prend en argument les trois valeurs nécessaires pour initialiser ses attributs.

### c) ScoreManager

Nous allons maintenant mettre en place la mécanique qui requête la base de données et qui retourne une liste de scores.

- Créer dans le package `score` l'interface `ScoreManager` contenant les méthodes suivantes :
  - ✓ `getLastScores(int count)` : retourne les count dernières parties enregistrées sous la forme d'une liste de `Score`.
  - ✓ `saveScore(Tokens winner, String winnerType)` : enregistre en base le score et ne retourne rien.Ces deux méthodes peuvent lancer une `ConnectException` (si les scores n'ont pu être récupérés par exemple).

On choisit d'en réaliser une implémentation par base de données, mais demain, on pourrait en faire une nouvelle implémentation par système de fichiers...

- Créer la classe `DatabaseScoreManager` qui implémente `ScoreManager`.

- Y renseigner les constantes privées suivantes :
  - ✓ `DRIVER = jdbc:mysql`
  - ✓ `DATABASE_HOST = remotemysql.com`
  - ✓ `DATABASE_NAME = UVxhCmnKnO`
  - ✓ `DATABASE_USER = UVxhCmnKnO`
  - ✓ `DATABASE_PASSWORD = NADGLcwBZR`
  - ✓ `TABLE_NAME = games_X` (avec `X` le numéro qui vous a été attribué)

Vous pouvez consulter la base à l'adresse <https://remotemysql.com/phpmyadmin/index.php?db=UVxhCmnKnO>

- Créer l'attribut `connection` de type `Connection`

On ne souhaite pas que l'attribut `connection` soit initialisé si on ne s'en sert jamais. On ne l'initialise donc pas dans le constructeur (qu'on a, du coup, même pas besoin d'écrire).

- Créer la méthode privée `initConnection()` qui teste si `connection` vaut `null` et qui l'initialise si c'est le cas.

Cette méthode peut lancer une `SQLException`.

- Créer la méthode privée `resultToScore()` qui prend un `ResultSet` en argument et qui retourne une instance de `Score` correspondante.

On utilisera les méthodes `getDate()` et `getString()` de `ResultSet`.

La date pourra être récupérée dans un objet de type `java.util.Date` par la commande :

```
Date date = new Date(result.getTimestamp("date").getTime());
```

**Attention :** `winner` et `winnerType` peuvent être `null`.

- Implémenter la méthode `getLastScores(count)`.

Commencer par créer une liste de scores et un `state` initialisé à `null`.

Dans un bloc `try/catch`, initialiser `connection`, puis initialiser `state`, exécuter la requête SQL nécessaire, boucler sur les résultats en appelant `resultToScore` afin de remplir la liste, et enfin fermer le `statement`.

Enfin, en-dehors du `try/catch`, retourner la liste.

Y a rien de plus simple !

- Implémenter `saveScore` de la même façon, mais sans retour.  
Pour ce faire, on n'utilisera pas `executeQuery`, incompatible avec la manipulation de données, mais `executeUpdate` (même fonctionnement).  
La colonne `id` est remplie automatiquement, inutile de la saisir.  
La colonne `date` doit être remplie avec la date actuelle que l'on formatera de la manière suivante :  
`LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss.SSS"))`  
À nouveau, attention aux valeurs `null` !

Nous voilà fin prêts pour enregistrer et récupérer le récapitulatif de nos parties.

#### d) Mise à jour de la mécanique de jeu

Pour l’affichage des scores, on propose la méthode suivante, à ajouter à `ConsoleGameView` :

```
private void displayScores(List<Score> scores) {
    if (scores.size() == 0) {
        System.out.println("Vous n'avez encore joué aucune partie. :)");
    } else {
        System.out.println(String.format("%-20s", "Date") + "| Vainqueur | Type");
        System.out.println("-----|-----|-----");
        for (Score score : scores) {
            System.out.println(String.format("%-20s", LocalDateTime.ofInstant(score.getDate().toInstant(),
ZoneId.systemDefault())) +
                " | " + String.format("%-10s", score.getWinner() == null ? "Match nul" : score.getWinner().name()) +
                " | " + (score.getWinnerType() == null ? "" : score.getWinnerType()));
            System.out.println("-----|-----|-----");
        }
    }
}
```

- Ajouter à `ConsoleGameView` le mot-clé « `KEYWORD_SCORES` » afin d’offrir cette nouvelle fonctionnalité à l’utilisateur.
- Ajouter l’attribut constant `scoreManager` de type `ScoreManager` et l’initialiser dans le constructeur.
- Modifier la méthode `play()` pour appeler la méthode `displayScores()` lorsque l’utilisateur saisit la valeur de `KEYWORD_SCORES`.  
Avant cet appel, on affichera « `Chargement...` » pour signifier à l’utilisateur que sa demande a été prise en compte.  
Après cet appel, on demandera à l’utilisateur une saisie quelconque (type « `Appuyer sur la touche Entrée` ») afin de revenir au jeu.
- Modifier la méthode `play()` pour sauvegarder la partie si elle est terminée.  
L’idéal est de faire ça juste après l’appel à `putToken()`.  
On passera « `Human` » comme type de vainqueur s’il y en a un.

Tester tout ça.

### 3. IA

Enfin, vous pouvez tenter de réaliser une IA simpliste qui jouera un coup sur deux.

Pour ce faire, il faut évaluer la viabilité de chaque coup.

On conseille donc de créer dans `Game` une méthode `evaluate()` qui retourne un score en fonction du coup simulé. L’IA choisira de jouer la colonne pour laquelle `evaluate()` aura retourné la plus grande valeur.

Une fois qu’elle est fonctionnelle, on peut tenter de compliquer les choses en implémentant l’algorithme `minimax`.

[https://fr.wikipedia.org/wiki/Algorithme\\_minimax](https://fr.wikipedia.org/wiki/Algorithme_minimax)