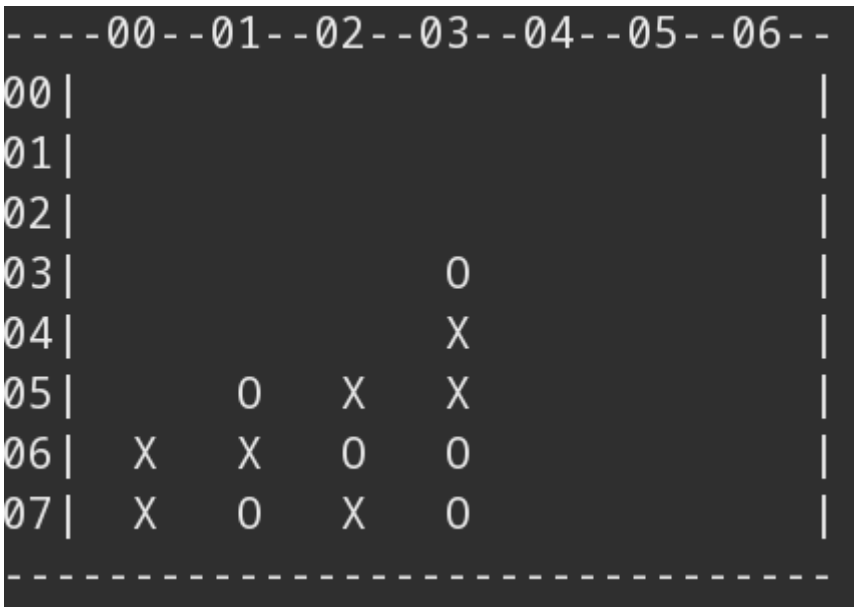
COMPTE RENDU

Puissance 4 – Projet Java



Felix Baubriaud

Corentin Prigent

Sommaire

Résumé

Le puissance 4 est un jeu assez simple à coder en langage orienté objet. Ici, nous l'avons implémenté en Java. Il faut distinguer chacun des objets intervenant dans le jeu (grille, case etc.) et créer, pour chacune des classes, des fonctions pour que les règles du jeu soit respectées (tour de jeu, pions à l'intérieur de la grille, fin du jeu si un joueur aligne un certain nombre de pions…). Il faut également utiliser les outils adéquats proposés dans le langage java pour modéliser tout ce qui intervient dans le jeu : sous-classes, table de hachage… Enfin, il faut veiller à ce que le jeu soit facile à comprendre par l'utilisateur, en expliquant dans la console ce qui est attendu et en gérant les valeurs aberrantes.

Mots clés

-Objet -Classe abstraite -Coordonnées -Programmation

-Table de hachage -Matrice -Sous-classe

-Alignement -Java -Eclipse -Type référence

-Polymorphisme -Surcharge -Vérification -Jouer

Introduction

L'objectif de ce TP est de créer en langage Java sur Eclipse un puissance 4. Le programme doit permettre à l'utilisateur de choisir la largeur et la hauteur de la grille, ainsi que le nombre de pions identiques à aligner pour gagner.

L'intérêt du TP est de se familiariser avec Java, de comparer ce langage avec le C++ et de manipuler les tables de hachage.

Coder un puissance 4 en java nécessite de créer plusieurs classes :

- Une classe Coordonnees qui contiendra les coordonnées de chaque case du jeu;
- Une classe Case avec des coordonnées en attributs;
- 3 classes héritant de la classe Case (CaseRouge, CaseJaune et CaseVide) permettant de savoir si la case est vide ou occupée par un pion rouge ou jaune;
- Une classe grille qui regroupe dans un HashMap toutes les cases;
- Une classe Partie permettant de jouer au jeu.

Dans un premier temps, nous nous concentrerons sur ce que contient la classe Coordonnees. Dans un deuxième temps, puis nous décrirons la classe Case ainsi que ses sous-classes. Ensuite, nous étudierons la classe Grille, utilisant les tables de hachage, et nous détaillerons les méthodes utilisées dans la classe Partie et le programme principal. Enfin, nous effectuerons des tests sur notre programme.

1. Classe Coordonnees

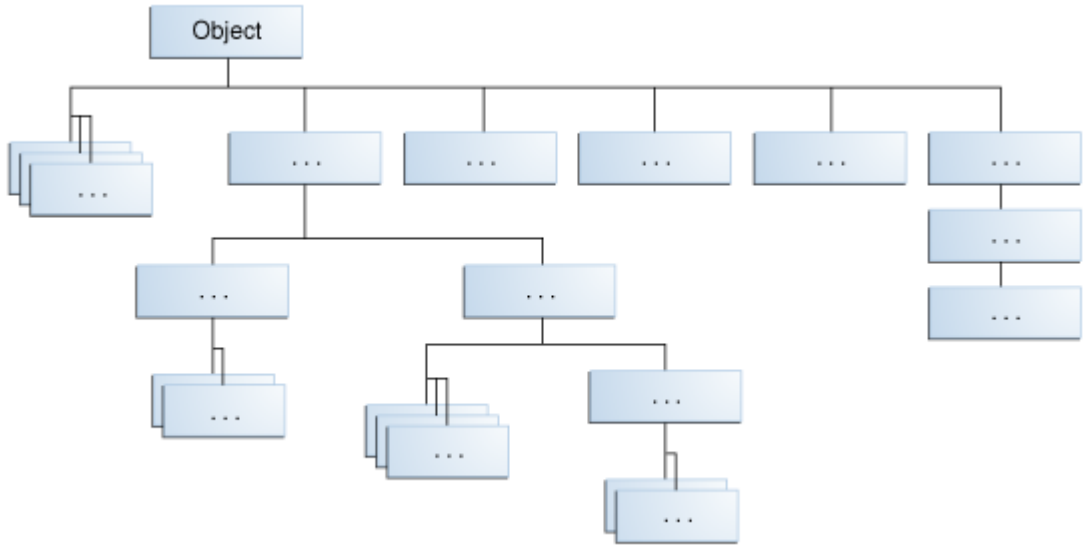
1.Coordonnées

Les coordonnées par défaut sont m_i=0 et m_j=5. Nous pouvons voir que le constructeur est vide. En java, une spécificité comparé au C++, c'est que nous pouvons initialiser les attributs sans devoir passer par le constructeur. Dans notre cas, il n'y a aucune allocation de mémoire donc nous avons décidé de laisser le constructeur vide. Pour chaque coordonnée i et j, nous avons créé des accesseurs et mutateurs, permettant d'accéder dans les autres cases aux coordonnées et de les modifier.

2.Equals(...) et HashCode()

Toutes les classes dans java sont des descendantes de la classe *Object*.

figure 1 : Hiérarchie des classes



En haut de la hiérarchie, *Object* est la classe la plus générale. Plus nous descendons dans la hiérarchie et plus les classes sont spécifiques. *Object* est aussi appelée la superclasse de toutes les classes existantes sur Java.

equals(...) et hashcode() sont des méthodes de cette classe et donc des méthodes de toutes les classes Java. Leur utilisation implique donc un polymorphisme qui est annoncé par *@Override*. Il est impossible de définir dans une classe equals(...) sans hashcode() et inversement.

equals(...) prend la référence Objet en paramètre et retourne true si les objets sont les mêmes ou false sinon.

hashcode() retourne pour chaque Objet un code de hachage. Dans notre cas le hachage est le suivant : coordonnée en x *10 + coordonnée en y. Il y a une bijection

dans notre cas car largeur < 10. En effet, 2 objets différents peuvent retourner le même code de hachage. En assurant la bijection, on assure qu'il n'y ait pas de problème par la suite.

Ces deux fonctions sont primordiales à définir pour l'utilisation du hashmap qu'on retrouvera dans la classe Grille.

Nous parlerons plus en détail de *instanceof*, présente dans la méthode equals, dans la partie suivante.

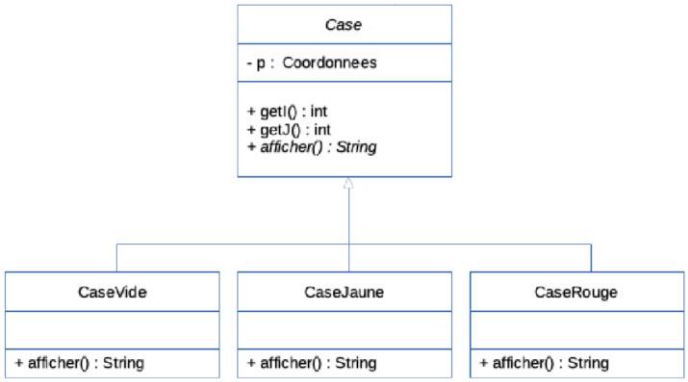
2. Classe Case et ses sous-classes

2.1.Classe abstraite

La classe Case est une classe abstraite, car elle contient la méthode abstraite afficher(), pour laquelle les classes héritières CaseRouge, CaseJaune et CasePleine ne réagissent pas de la même façon. Les classes abstraites en Java ne peuvent pas s'instancier, et s'identifient avec le mot clé *abstract*, comme pour les méthodes

abstraites. Pour la méthode afficher() de la classe Case, il suffit donc d'écrire son prototype car on la déclarera dans les sous-classes. Elle renvoie un *String* plutôt que d'afficher directement dans la console pour être utilisée plus facilement dans les prochaines classes, notamment pour la concaténation.
On aurait pu déclarer un attribut type pour différencier les cases vides, rouges et jaunes, mais ce n'est pas indispensable en Java grâce à l'opérateur *instanceof*, qui permet de vérifier le nom de la classe à laquelle appartient l'instance considérée. La classe Case présente donc comme seul attribut un couple de coordonnées. Coordonnees étant une classe, c'est un type référence en Java. Il est nécessaire d'allouer l'objet de type Coordonnees dans le constructeur de Case avec le mot clé *new*. On crée également un constructeur supplémentaire afin de pouvoir renseigner directement les coordonnées de la case à l'appel du constructeur. Contrairement au C++, il n'est pas nécessaire de créer de constructeur de recopie ni de destructeurs même en cas d'allocation de mémoire.

figure 2 : Héritage de la classe abstraite



2.2. Les sous-classes

Pour préciser que les classes CaseRouge, CaseJaune et CasePleine héritent de Case, il suffit d'ajouter à la création de la classe *extends Case*. On ajoute aussi à l'intérieur de la classe le mot clé *@Override* avant de déclarer la fonction afficher() car il s'agit d'un polymorphisme :

- Pour CaseRouge, on retourne une croix " X ".

- Pour CasePleine, on retourne un rond " O ".
- Pour CaseVide, on retourne un espace "   ".

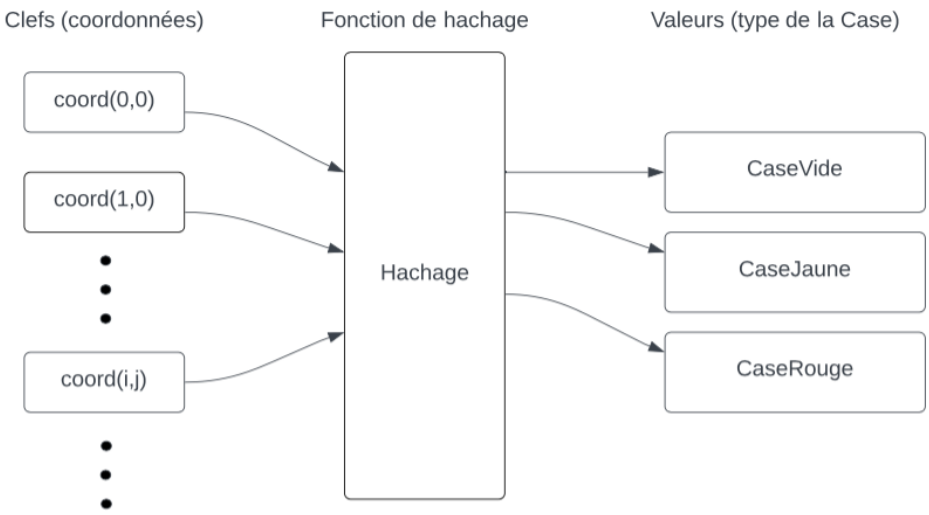
Dans les constructeurs de base et supplémentaires, le mot-clé *super* permet d'appeler le constructeur (de base ou supplémentaire si on ajoute des paramètres) de la classe mère, c'est-à-dire celui de la classe Case.

3. La classe Grille

3.1. Le HashMap

HashMap est un type référence utilisable avec la librairie *java.util.HashMap*. Il est composé d'une clé et de sa valeur. Dans notre cas, les clés sont les coordonnées de notre puissance 4 et les valeurs sont les types des cases.

figure 3 : Fonctionnement de notre table de hachage



Le constructeur de la classe Grille en est composé. Comme c'est un type référence le *HashMap* est donc alloué. *Hashmap* est une sous-classe de *Object*. Il est donc composé de ses propres méthodes dont get(...) et put(... , …) que nous avons utilisées dans notre code.

-La méthode *get* prend en entrée une clé et renvoie la valeur.

-La méthode *put* prend en entrée une clé et une valeur. La valeur est ainsi stockée dans la clé.

Ces méthodes sont équivalentes aux accesseurs *get* et *set* que nous pouvons retrouver dans d'autres classes.

3.2. Lecture de la grille

La méthode afficher() renvoie une chaîne de caractères que nous avons allouée car *String* dans Java est un type référence. À noter que Coordonnees, dans les autres méthodes, est aussi alloué car c'est une classe, donc un type référence. L'affichage des " X ", " O " et " " se fait via la méthode afficher() de la classe Case et de la méthode getCase() de Grille. On concatène chaque chaîne de caractère, en séparant les lignes du milieu des premières et dernières lignes pour un meilleur affichage, avant de renvoyer le résultat final.

La méthode pleine() renvoie *true* si la grille est pleine, *false* sinon. Pour cela on parcourt entièrement la grille et on regarde individuellement chaque case pour savoir si elle est vide. On utilise des boucles *while* plutôt que des *for* pour sortir des boucles dès qu'on a trouvé une case vide.

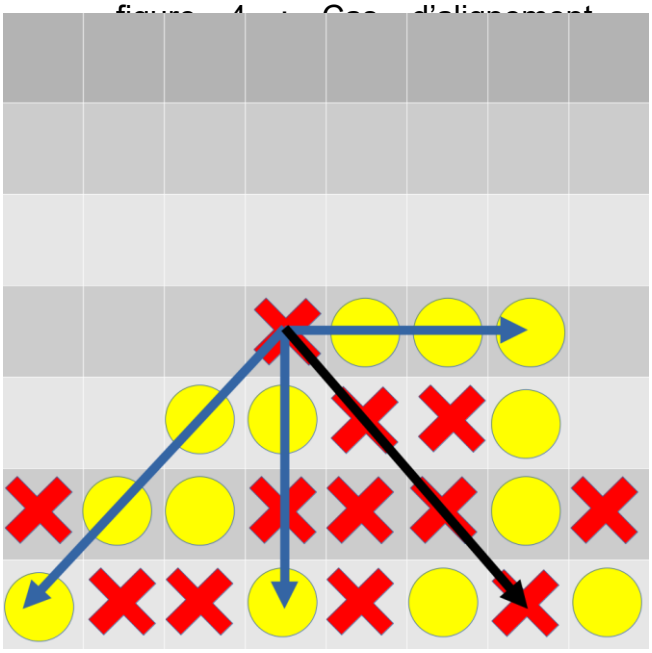
4. La classe Partie

4.1. Conditions pour mettre fin à la partie

La classe partie contient une grille m_g. On remet également en attribut m_largeur et m_hauteur (qui sont les mêmes que dans la classe Grille) pour accéder plus facilement. Pour gagner au puissance 4, il faut qu'un joueur réussisse à aligner 4 de ses pions verticalement, horizontalement ou en diagonale. Ici on doit pouvoir choisir le nombre de pions à aligner pour gagner, qu'on met dans une variable m_nbPointsAAligner en tant qu'attribut de la classe Partie. On va donc vérifier, à chaque fois qu'un joueur place un pion, si la grille présente un alignement de m_nbPointsAAligner pions de même type. Pour cela, on parcourt la grille entièrement, de gauche à droite et de haut en bas, et pour chaque case on va regarder si c'est le premier pion d'un alignement. Nous avons 4 vérifications à effectuer :

- La case peut être le pion le plus à gauche d'un alignement horizontal. On regarde donc les m_nbPointsAAligner-1 cases suivantes sur la ligne.
- La case peut être le pion le plus haut d'un alignement vertical. On regarde donc les m_nbPointsAAligner-1 cases suivantes sur la colonne.

- La case peut être le pion le plus haut et le plus à gauche d'un alignement en diagonale. On regarde donc les m_nbPointsAAligner-1 cases d'indice (i+1,j+1), …, (i+m_nbPointsAAligner, j+m_nbPointsAAligner) si la case qu'on vérifie est d'indice (i, j).
- La case peut être le pion le plus haut et le plus à droite d'un alignement en diagonale. On regarde donc les m_nbPointsAAligner-1 cases d'indice (i+1,j-1), …, (i+m_nbPointsAAligner, j-m_nbPointsAAligner) si la case qu'on vérifie est d'indice (i, j).



En faisant ces vérifications pour chaque case de la grille, on aura bien étudié tous les cas possibles car on parcourt la grille de gauche à droite et de haut en bas. Dans la classe Partie, on répartit ces vérifications dans 4 fonctions différentes. On peut alors créé la fonction gagner() qui renverra *true* si une des 4 fonctions renvoie *true* pour une ou plusieurs cases de la grille, *false* sinon.

Nous allons détailler la fonction LigneDroite, correspondant à l'alignement horizontal. Les 3 autres fonctions sont similaires.

LigneDroite prend en argument les coordonnées d'une case de la grille pour vérifier si c'est la plus à gauche d'un alignement horizontal. k représente le nombre de cases de même type alignés horizontalement et est donc initialisé à 1. On récupère la case de coordonnées (i,j). Selon le type de la case, qu'on peut vérifier avec *instanceof*, on distingue 3 cas :

- Si c'est une case vide, on peut tout de suite renvoyer *false*.
- Si c'est une case rouge, on incrémente k jusqu'à ce qu'il égale le nombre de points à aligner, en vérifiant avant chaque incrémentation qu'on ne dépasse pas de la grille (ici on doit avoir *j+k<m_largeur*), et que la case située à une distance k à droite de la case (i, j), c'est-à-dire la case (i, j+k), est une case rouge. Si à la fin on obtient que k=nbPointsAAligner, c'est que toutes les conditions ont été vérifiées et qu'il y a bien un alignement horizontal.
- Si c'est une case jaune, on vérifie la même chose que pour la case rouge mais les cases (i, j+k) doivent être des cases jaunes.

La partie se termine lorsqu'un des joueurs a gagné ou que la grille est pleine.

4.2. Fonctionnement du jeu

Dans le programme principal, qui est également une classe en Java, il est nécessaire d'inclure *Scanner* afin de pouvoir, en plus d'écrire, lire des éléments dans la console. Au commencement de la partie, on demande aux utilisateurs de choisir la largeur et la hauteur de la grille, ainsi que le nombre de points à aligner pour gagner. Ensuite, jusqu'à la fin de la partie, on demande tour par tour aux joueurs de saisir la colonne où ils veulent placer un pion. À chaque fois qu'on demande quelque chose à l'utilisateur, on répète l'opération tant qu'une valeur saisie n'est pas correcte. En particulier lorsqu'on demande la colonne où le pion va être joué, on utilise la fonction coupPossible. Cette fonction vérifie que la colonne est présente bien dans la grille et qu'il reste de la place dans la colonne. Pour cela, il suffit de vérifier que la case la plus haute dans la colonne, donc d'indice i=0, est vide.

On peut ensuite appeler la méthode jouer, qui, dans un premier temps recherche une case vide en partant du bas de la grille (ligne *m_hauteur-1*), et dans un deuxième temps place un pion jaune ou rouge selon le joueur dans cette case et change de joueur (attritbut m_joueur de Partie).

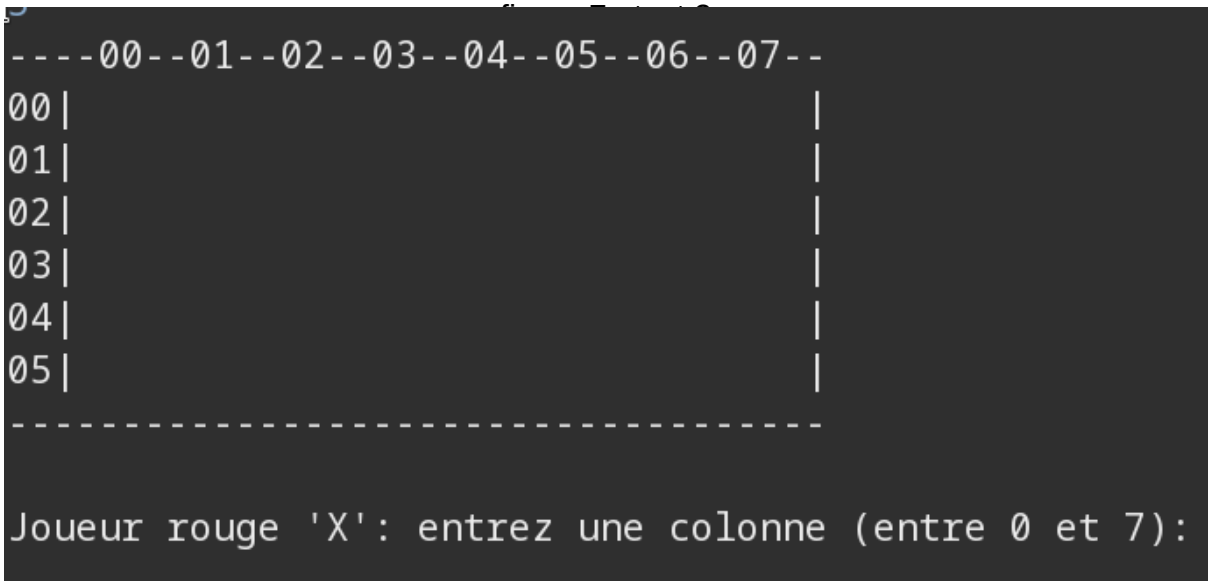
5. Tests

Lorsqu'on compile le programme, on nous demande tout d'abord d'entrer la largeur de la grille, la hauteur puis le nombre de points à aligner pour gagner. On redemande à l'utilisateur l'instruction tant qu'il n'entre pas une valeur correcte :

figure 6 : test 1

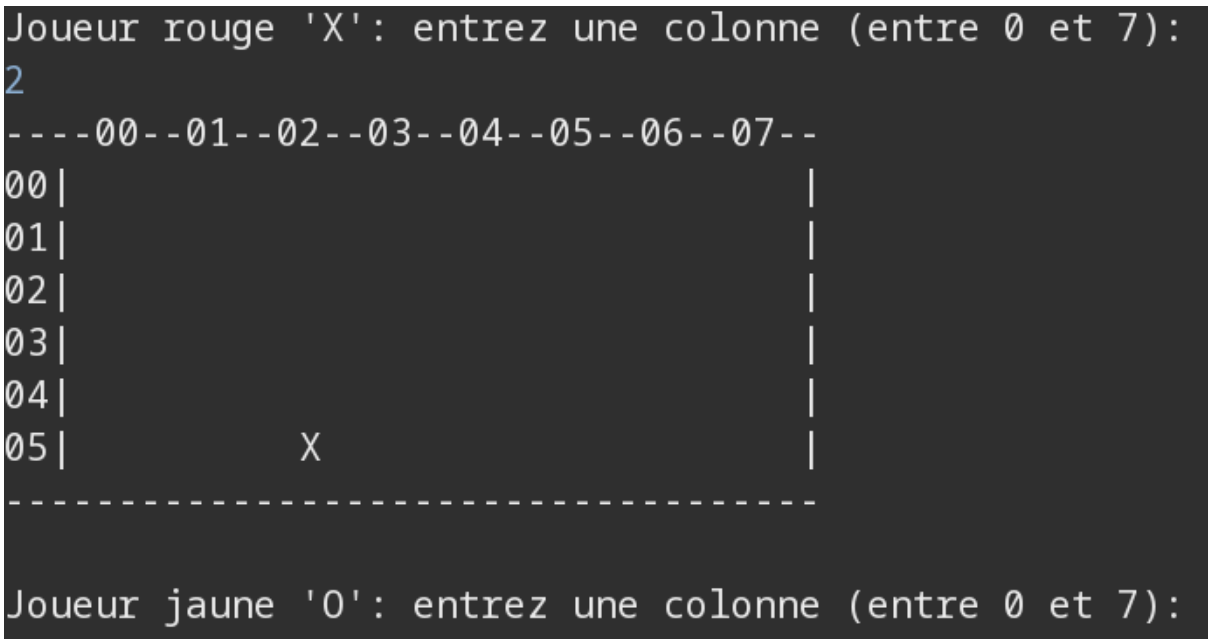
```
Entrez la largeur de la grille (entre 4 et 9):
10
Entrez la largeur de la grille (entre 4 et 9):
8
Entrez la hauteur de la grille (entre 4 et 9):
6
Choisir le nombre de pions a aligner pour gagner (entre 3 et 9):
2
Choisir le nombre de pions a aligner pour gagner (entre 3 et 9):
5
```

On affiche alors la grille vide. Le joueur rouge commence. On lui demande le numéro de la colonne dans laquelle il veut insérer son pion.

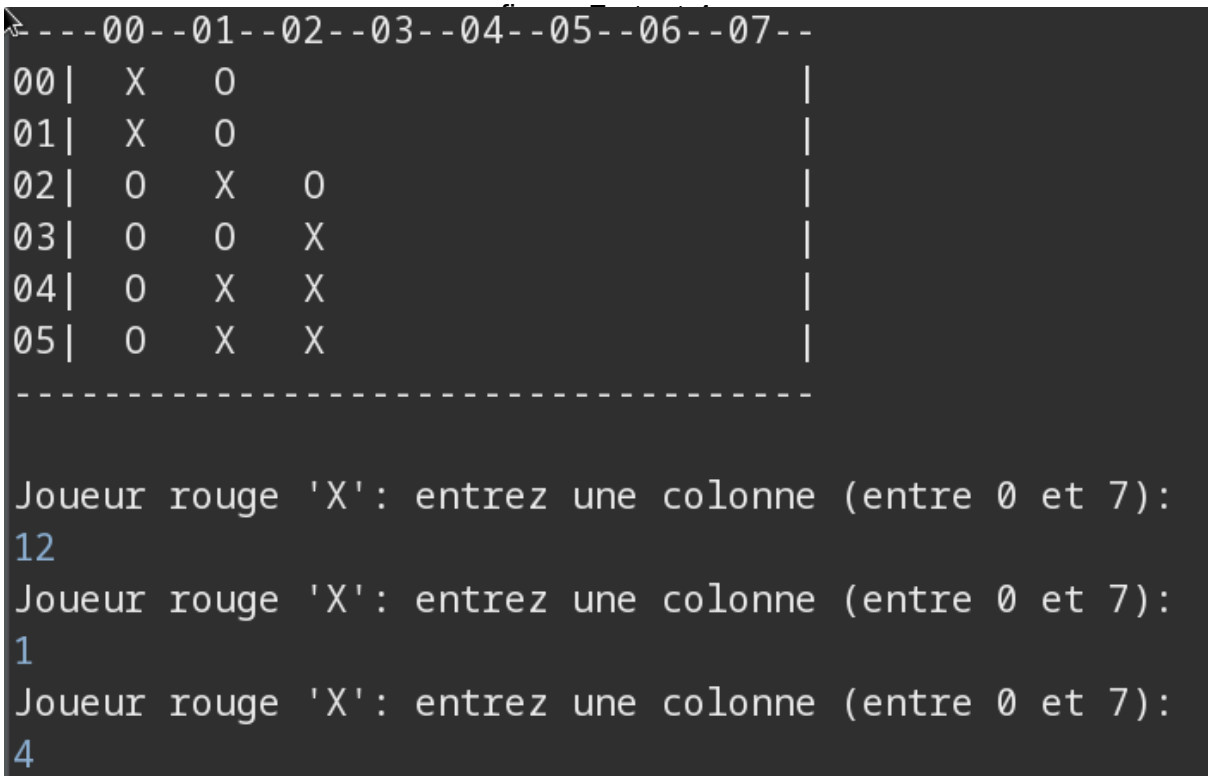


Une fois qu'il a joué, on réaffiche la grille avec le pion dans la colonne joué puis c'est le tour du joueur jaune :

figure 7 : test 3

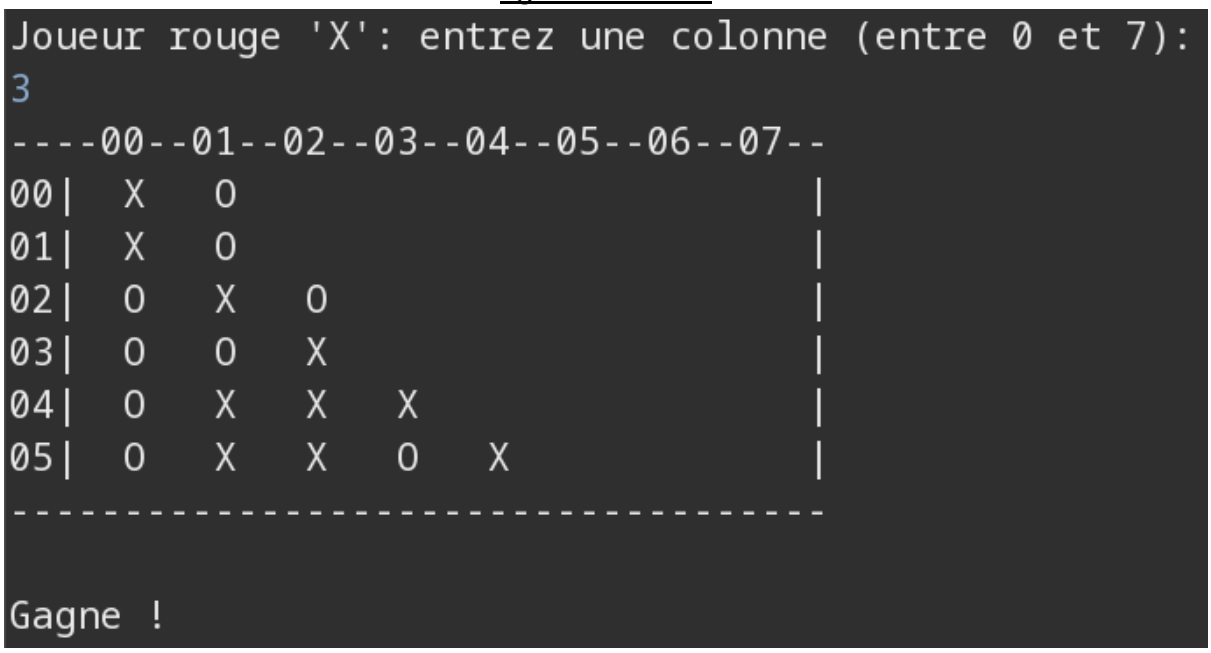


La partie continue de la même façon jusqu'à la fin. Comme pour les premières saisies de valeurs, on redemande à l'utilisateur l'instruction s'il n'est pas possible de jouer un pion dans la colonne choisie.



Dès qu'un joueur a aligné le nombre de pions choisis au départ (ici 5) verticalement, horizontalement ou en diagonale, la partie s'arrête et le message "Gagne !" apparaît à l'écran :

figure 7 : test 4



En revanche, si personne ne parvient à aligner ses pions et que la grille est remplie, le jeu s'arrête et le message "Egalite !" apparaît :

figure 8 : test 4

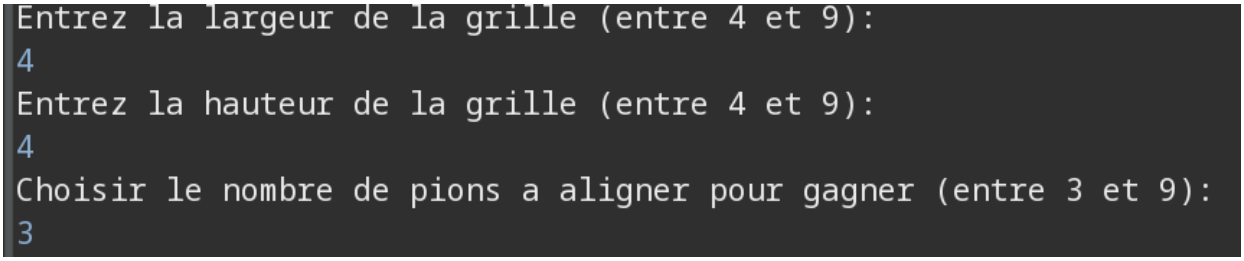
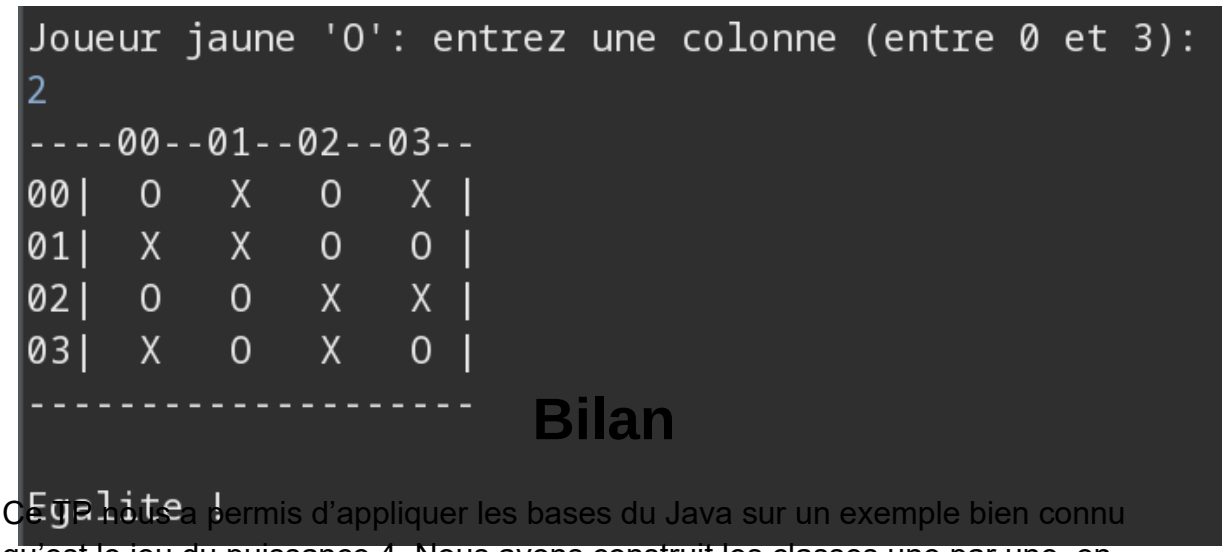


figure 8 : test 5



Ce projet nous a permis d'appliquer les bases du Java sur un exemple bien connu qu'est le jeu du puissance 4. Nous avons construit les classes une par une, en commençant par Coordonnees, puis en continuant avec les classes de plus en plus générales (ordre du plan du rapport). Une fois toutes les classes implémentées, le jeu n'a pas fonctionné immédiatement. Il a fallu revenir plusieurs fois sur le code. En particulier, nous avions au départ oublié de définir les méthodes hashCode et equals dans Coordonnees. Il s'agissait sinon d'erreurs d'étourderie (un j non initialisé à 0 avant une boucle while, else au lieu de else if, conditions pour ne pas sortir de la grille dans les vérifications d'alignement…).

Conclusion

Pour conclure, les principaux points à retenir dans ce TP est l'utilisation de la classe Object pour Java et de ses méthodes comme equals et hashcode ainsi que la classe hashmap et des types référence en général.

Ce TP nous a également permis de comparer C++ à Java. Par exemple, le main n'est plus une fonction mais une classe, et les types référence se rapprochent plus de pointeurs que de références en C++. Le *instanceof* qui permet de prendre le type d'une classe fille sans passer par un attribut type est aussi une nouveauté. Il est bien de noter que la lecture se fait via une classe et que donc c'est aussi un type référence (classe Scanner).