Université de Paris **Programmation Orientée Objet**
L2 **Année 2020-2021**

TP n° 3

Introduction à l'héritage : jeu d'échecs

Le but de ce TP est de réaliser une version simplifiée d'un **jeu d'échecs**. Comme dans le TP n° 2, nous vous donnons un déroulé pas à pas au fil des différents exercices. Si vous vous sentez à l'aise, vous pouvez proposer votre propre modélisation/implémentation du jeu. Dans ce cas, prenez le temps de bien planifier vos classes avant de vous mettre à coder.

Exercice 1 Modélisation des Pièces – classe mère

On commence par la définition des pièces d'échecs.

1. Écrire une classe `Piece` qui a comme attributs :
   - Un booléen correspondant à la couleur de la pièce (*true* indique blanc, *false* indique noir).
   - Une chaîne de caractères correspondant au nom de la pièce.
2. Créer un constructeur qui initialise les deux attributs avec les valeurs données en paramètres.
3. Définir la méthode `public String toString()` pour qu'elle renvoie le nom de la pièce avec la première lettre en minuscule (si la pièce est blanche), ou en majuscule (si la pièce est noire).

Remarque : Quand on exécute `System.out.println(p);` avec une référence p non *null*, la méthode `p.toString()` est invoquée et la chaîne de caractère obtenue est affichée. Si p est *null*, `System.out.println(p);` affichera *null*.

Exercice 2 Modélisation des Pièces – classes héritées

Définir les classes `Pion, Tour, Cavalier, Fou, Dame` et `Roi` en étendant la classe `Piece`. Les attributs de `Piece` peuvent tous être hérités.

- Les constructeurs ne prendront qu'un seul paramètre en entrée, puisque le nom de la pièce est déjà déterminé par la classe à laquelle elle appartient.
  Remarque : si ni `super(...)` ni `this(...)` n'est invoqué explicitement au début du constructeur, un appel `super()` (sans arguments) est fait implicitement. Si la classe mère ne définit pas de constructeur sans arguments, telle la classe `Piece`, cela mène donc à une erreur de compilation !
- Est-ce qu'il faut redéfinir la méthode `toString()` pour qu'elle fonctionne avec les sous-classes de `Piece` ? Tester la méthode `toString()` pour la classe `Roi` et vérifier l'affichage.

(Facultatif : une autre modélisation) Vous pouvez introduire une 'grande' classe `HorVerDiag` qui remplace les classes de pièces qui se déplacent en horizontale, verticale et/ou diagonale vers les quatre directions possibles, c-à-d les classes `Fou, Tour, Dame` et `Roi`. Utilisez des attributs booléens pour préciser le type

de pièce : un attribut pour déterminer si la pièce peut se déplacer en horizontale/verticale, un autre attribut pour le pouvoir de se déplacer en diagonale et un troisième pour le pouvoir d'aller plus loin qu'une case adjacente sur le plateau. Cela laisse uniquement les classes `Pion` et `Cavalier` à définir et traiter séparément.

Exercice 3 Modélisation du Plateau

1. Écrire une classe `Case` qui a comme attributs : un booléen indiquant si elle est blanche ou noire, et un attribut de type `Piece` donnant la pièce qui est sur la case (`null` si la case est vide). Écrire le getter `Piece getPiece()` ainsi que des méthodes `boolean estVide()`, `void remplirPiece(Piece p)` et `void enleverPiece()`.
2. Écrire une méthode `toString()` dans la classe `Case`. Pour que le plateau soit lisible, l'affichage de la case se fera en un caractère : soit la lettre initiale de la pièce dessus (si pas vide), soit `"-"` pour la couleur blanche ou `"*"` pour la couleur noire.
3. Écrire une classe `Plateau` qui a comme attributs :
   — des entiers correspondant à la longueur et la largeur du plateau
   — un tableau de tableaux de `Case`s
4. Ajouter à la classe `Plateau` :
   — Un constructeur `public Plateau(int longueur, int largeur)` qui génère un plateau de taille `longueur` × `largeur` et de couleur alternées (la `Case` à (0,0) est noire).
   — Une méthode `public boolean horsLimite(int x, int y)` qui teste si la position (x,y) correspond bien à une case du plateau.
   — Un getter `public Case getCase(int x,int y)` qui renvoie la Case de coordonnées x et y, ainsi que des méthodes `void videCase(int x, int y)` et `void remplirCase(int x, int y, Piece p)` pour interagir avec la Case.
   — Une méthode `public void afficher()` qui affiche le plateau (utiliser la méthode `toString()` de `Case`).

Exercice 4 Gestion du jeu

Écrire une classe `Echecs` (la classe principale contenant la méthode `main`) qui a comme but le lancement et la gestion du jeu. Dans un premier temps, elle permet l'initialisation du plateau avec des pièces dessus et l'affichage de son état actuel. Si vous en voulez, vous pouvez vous restreindre à un plateau de taille 4×4 ou 4×5 (voir les règles des mini-échecs ici : https://fr.wikipedia.org/wiki/Mini-échecs), car un plateau petit facilite la vérification du bon comportement du déplacement des pièces et du jeu en général. Compiler et exécuter le programme pour vérifier le bon affichage du plateau.

Exercice 5 Gestion du déplacement

On veut maintenant ajouter la possibilité de déplacer les pièces d'une case à l'autre. Tout d'abord, on définit une classe `Deplacement`. Un objet de type

`Deplacement` a quatre attributs `int x0,y0,x1,y1`, les coordonnées du point de départ `(x0,y0)` et d'arrivée `(x1,y1)` du déplacement.

La classe `Deplacement` peut fournir aussi à ses objets des fonctionnalités diverses pour bien déterminer et préciser le déplacement. Par exemple, elle peut avoir une méthode `char typeDeplacement()`, qui détermine le type du déplacement (horizontal, vertical, diagonal, ou cavalier), ou une autre méthode `int dist()` qui retourne la distance entre les points de départ et d'arrivée. Pour des descriptions plus précises de ces méthodes regarder ci-dessous.

A. `char typeDeplacement()` : cette méthode retourne un charactère qui détermine le type de déplacement ('v' : vertical, 'h' : horizontal, 'd' : diagonal, 'c' : cavalier, 'x' : autre). Par exemple :
   - un déplacement $d1$ entre les cases (1,5) et (3,3) est de type 'd'.
   - un déplacement $d2$ entre les cases (1,5) et (1,2) est de type 'v'.
   - un déplacement $d3$ entre les cases (1,1) et (2,3) est de type 'c'.
   - un déplacement $d4$ entre les cases (5,6) et (1,5) est de type 'x'.

B. `int dist()` : cette méthode retourne la distance absolue entre le point de départ et le point d'arrivée si le déplacement est de type 'h', 'v' ou 'd'. Si le déplacement est de type 'c' ou 'x', elle retourne -1. Par exemple, pour les déplacements $d1, d2, d3$ ci-dessus :
   - pour $d1$ la méthode retourne 2.
   - pour $d2$ la méthode retourne 3.
   - pour $d3$ la méthode retourne -1.

Définir d'autres méthodes dans la classe `Deplacement` selon vos besoins.

Une fois la classe `Deplacement` écrite, penser à surcharger la méthode `boolean horsLimite(int x, int y)` de la classe `Plateau` pour qu'elle puisse prendre en argument un déplacement au lieu d'un point (x,y).

Exercice 6 Déplacement valide – héritage

La validité d'un déplacement dépend parfois des conditions qui sont communes à toutes les pièces (une pièce ne peut pas se déplacer sur un point d'arrivée occupé par une pièce de la même couleur), mais aussi des conditions qui ne s'appliquent qu'à la pièce concernée (le roi peut se déplacer d'une seule case en horizontale, verticale ou diagonale, le fou peut se déplacer en diagonale d'autant de cases vides que voulu etc).

On veut séparer la partie de code qui s'applique à tous les pièces de celle qui dépend des spécificités de chaque pièce.

1. Ecrire une méthode `boolean estValide(Deplacement d, Plateau p)` dans la classe `Piece` qui teste si le déplacement est valide pour la pièce concernée et l'état actuel du plateau.
2. Redéfinir la méthode `boolean estValide(Deplacement d, Plateau p)` pour chaque sous-classe de `Piece`. Ici, on testera les conditions de déplacement propres à chaque pièce (par exemple : un fou ne peut se déplacer qu'en diagonale). N'oubliez pas d'utiliser l'instruction `super.estValide(d,p);` pour tester si les conditions générales de déplacement sont vérifiées.

Remarque : La méthode `estValide` est polymorphe : lorsqu'on exécute `p.estValide(deplacement,plateau);` la JVM exécute la variante de la méthode `estValide` correspondant au type de p (liaison dynamique).

Exercice 7 Gestion d'un tour

1. Un tour correspond à un joueur courant. Donc, il faut tout d'abord ajouter dans la classe `Echecs` un attribut d'instance `boolean joueur` qui indique le joueur courant, `true` pour *Blanc* et `false` pour *Noir*. Vu qu'une partie d'échecs commence toujours par le *Blanc*, dans le constructeur de la classe `Echecs`, on initialise l'attribut `joueur = true`.
2. Un déplacement d'une pièce dans le plateau est valide s'il vérifie les conditions suivantes :
   - Les coordonnées correspondent aux vraies cases dans le plateau.
   - Il y a bien une pièce p sur la case de départ.
   - La pièce p appartient au joueur en train de jouer.
   - Il n'y a pas de pièce appartenant au même joueur sur la case d'arrivée.
   - Le mouvement est valide pour la pièce p.

   Écrire dans la classe `Echecs` une méthode `void jouerTour(Deplacement d, boolean joueur, Plateau p)` qui teste si un déplacement est valide et réalise le déplacement dans ce cas (vider la case de départ, vider la case d'arrivée si elle contient une pièce de l'adversaire, remplir la case d'arrivée avec la pièce à déplacer).
3. Tester votre méthode sur plusieurs exemples. Afficher le plateau après chaque exécution de la méthode `jouerTour`.

Exercice 8 (bonus) Gestion d'une partie

On implémente enfin la gestion complète d'une partie. Elle doit pouvoir s'exécuter depuis la fonction `main` en déclarant un objet de type `Echecs` et en exécutant la méthode `jouerPartie` (voir ci-dessous).

L'interaction entre l'utilisateur et le programme est gérée de la sorte :

1. Créer une classe `Communication` avec un attribut de type `Scanner`. On utilisera un objet de cette classe pour communiquer avec l'utilisateur (demande de saisie, affichage de messages...).
2. Ajouter une méthode
   `public Deplacement demanderDeplacement(boolean joueur)`
   qui demande à l'utilisateur d'entrer au clavier les quatre coordonnées d'un déplacement, et retourne un objet de type `Deplacement` avec ces paramètres.

L'alternance des tours : le jeu commence avec le joueur *Blanc* puis alterne entre les deux joueurs jusqu'à ce qu'un roi soit éliminé du plateau. Implémenter ce déroulement :

3. Ajouter une méthode `boolean estRoi()` (dans la classe `Piece` et les sous-classes héritées) qui indique si une pièce est un roi.
4. Ajouter une méthode `jouerPartie()` dans la classe `Echecs` qui implémente une boucle se répétant jusqu'à la fin de la partie. Dans cette boucle, on demande au joueur courant de proposer un déplacement. Si le déplacement proposé n'est pas valide, il faudra lui reposer la question. Une fois le déplacement correct, on lance la méthode `jouerTour()`. Si le roi de l'adversaire est capturé, le joueur courant gagne et on sort de la boucle, sinon le joueur courant change et on répète.