

II.1102 : Algorithmique et Programmation

TP 3 : Programmation orientée objet

Patrick Wang

16 octobre 2020

Table des matières

1	Objectifs du TP	1
2	Quelques rappels de vocabulaire	1
3	L'utilisation du debugger	2
3.1	Utilisation de points d'arrêt	2
3.2	Mode debug	2
4	Développement du jeu d'échecs	3
4.1	Objectifs du TP	3
4.2	Implémentation du jeu d'échecs	4
4.2.1	Classe <code>Player</code>	4
4.2.2	Classe <code>Position</code>	4
4.2.3	Classes représentant les pièces du jeu	5
4.2.4	Classe <code>Cell</code>	6
4.2.5	Classe <code>Chess</code>	6
4.2.6	Classe <code>Main</code>	7
4.3	Bonus	7

1 Objectifs du TP

- S'initier à la programmation orientée objet ;
- Savoir créer des classes et les définir ;
- Savoir instancier des classes et manipuler les objets créés ;
- Découvrir les outils de debug.

2 Quelques rappels de vocabulaire

Classe : Une classe est un descriptif d'un type de données que l'on souhaite créer ;

Objet : Un objet est une *instance* d'une classe. Par exemple, `Enseignant` peut être une classe et Patrick Wang serait une instance de cette classe ;

Attribut : Un attribut est une variable permettant de décrire le contenu d'une classe ;

Méthode : Une méthode est une fonction permettant de décrire le comportement d'une classe ;

Constructeur : Un constructeur est une méthode spécifique permettant d'instancier un nouvel objet ;

Visibilité : La visibilité est une caractéristique permettant de définir les conditions d'accès à une classe, une méthode, ou un attribut ;

Getters/setters : Les *getters/setters* sont des méthodes spécifiques pour manipuler les attributs de classe qui sont souvent définis comme **private**.

3 L'utilisation du debugger

Savoir déboguer un programme est une compétence importante du développeur. De nombreux outils sont à notre disposition pour réaliser ces tâches. Pour le moment, nous n'avons fait qu'introduire la possibilité d'écrire des messages en console grâce à la méthode `System.out.println()`.

Dans ce TP, nous allons présenter les outils de debug que l'on peut retrouver dans tout IDE normalement constitué. Les illustrations utilisées dans ce TP montreront l'utilisation du debugger sur IntelliJ. L'interface sera légèrement différente sur Eclipse mais les processus resteront identiques.

3.1 Utilisation de points d'arrêt

Pour utiliser le debugger, il faut d'abord placer des *points d'arrêt* (ou *breakpoints*). La Figure 1 illustre par exemple un point d'arrêt placé en ligne 55 d'un programme écrit avec IntelliJ.

Pour ajouter un point d'arrêt, la procédure à suivre est quasiment identique que l'on utilise IntelliJ ou Eclipse :

1. Repérer l'instruction où l'on souhaite mettre un point d'arrêt ;
2. Ajouter un point d'arrêt en cliquant dans la colonne se situant à gauche de la zone d'édition de texte. Attention, si on utilise IntelliJ, il faut cliquer à droite du numéro de ligne. Si on utilise Eclipse, il faut cliquer à gauche du numéro de ligne ;
3. Un marqueur visuel apparaîtra pour montrer que le point d'arrêt a été correctement ajouté ;
4. Pour supprimer un point d'arrêt, il faut simplement cliquer dessus.

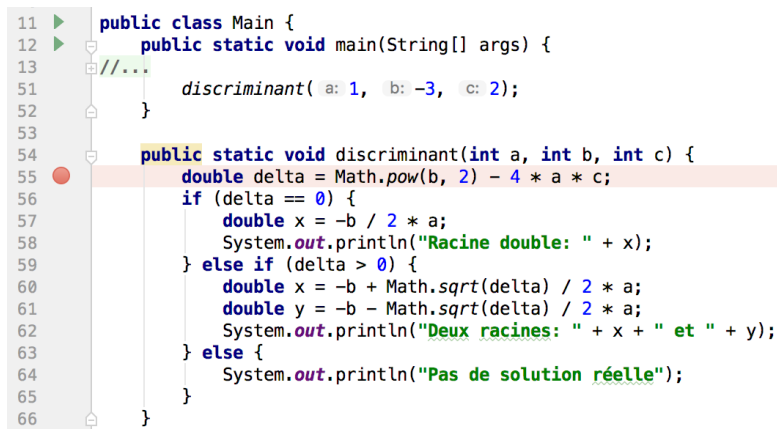
L'intérêt d'un point d'arrêt est de stopper l'exécution d'un programme avant d'exécuter une instruction particulière. Dans l'exemple de la Figure 1, le programme va donc s'arrêter avant d'exécuter la ligne 55. Un autre intérêt du mode debug est de pouvoir *inspecter* le contenu des variables et objets utilisés par le programme.

3.2 Mode debug

Une fois que les points d'arrêts ont été placés, il faut lancer le programme en mode debug. En général, cela est fait en cliquant sur une icône en forme d'insecte située à côté du bouton pour lancer le programme.

Lorsque le programme est lancé en mode debug, l'interface de notre IDE va se modifier légèrement pour adopter l'apparence du mode debug. La Figure 2 montre cette visualisation debug sur IntelliJ. On y voit plusieurs informations (une interface similaire sera adoptée par Eclipse au moment d'entrer dans le mode debug) :

- Dans le panneau "Frame", on y trouve la pile d'instructions ;



```

11  public class Main {
12  public static void main(String[] args) {
13      //...
51      discriminant( a: 1, b: -3, c: 2);
52  }
53
54  public static void discriminant(int a, int b, int c) {
55      double delta = Math.pow(b, 2) - 4 * a * c;
56      if (delta == 0) {
57          double x = -b / 2 * a;
58          System.out.println("Racine double: " + x);
59      } else if (delta > 0) {
60          double x = -b + Math.sqrt(delta) / 2 * a;
61          double y = -b - Math.sqrt(delta) / 2 * a;
62          System.out.println("Deux racines: " + x + " et " + y);
63      } else {
64          System.out.println("Pas de solution réelle");
65      }
66  }

```

FIGURE 1 – Illustration d’un point d’arrêt placé en ligne 55 sur IntelliJ.

- Dans le panneau “Variables”, on y trouve les différentes variables utilisées ainsi que leurs valeurs au moment de l’arrêt de l’exécution du programme ;
- Il y a aussi plusieurs boutons représentant des flèches allant dans différentes directions. Ces boutons vont permettre au développeur de contrôler l’exécution de la suite du programme. En particulier, on peut y trouver les boutons :
 - “Step Into” : Si la prochaine instruction contient un appel de méthode, alors le programme va *entrer* dans cette méthode pour montrer son exécution instruction par instruction ;
 - “Step Over” : Quelle que soit la nature de l’instruction suivante, le programme va l’exécuter et, une fois terminée, passer à l’instruction suivante ;
 - “Step Out” : Si le programme s’est arrêté dans l’appel d’une fonction, exécute la fin de la fonction pour en *sortir*.

Le TP suivant va donc vous permettre de vous familiariser avec le debugger, ce qui devrait grandement vous aider lors du développement de votre projet.

4 Développement du jeu d’échecs

4.1 Objectifs du TP

Dans ce TP, nous allons chercher à implémenter **une partie** du jeu d’échecs. Ce TP va ensuite servir de base pour le TP de la semaine suivante portant sur les mécanismes d’héritage et de polymorphisme.

Dans ce TP, nous allons simplement déclarer quelques classes et implémenter des méthodes bien précises. Le programme sera donc volontairement *incomplet* et le jeu ne sera pas jouable.

Le jeu d’échecs se joue à deux sur un plateau de taille 8×8 . Les échecs se jouent avec différents types de pièces : le Roi, la Reine, le Fou, le Cavalier, la Tour, et le Pion.

Les déplacements des pièces se font selon des règles bien précises. Celles-ci sont toutes décrites sur la page Wikipédia suivante : https://fr.wikipedia.org/wiki/%C3%89checs#D%C3%A9placements_des_pi%C3%A8ces. Il faudra lire attentivement cette page pour implémenter les règles de déplacement dans ce TP. On va faire exception pour les déplacements spéciaux suivants :

- Le « roque » ;
- La « prise en passant » ;

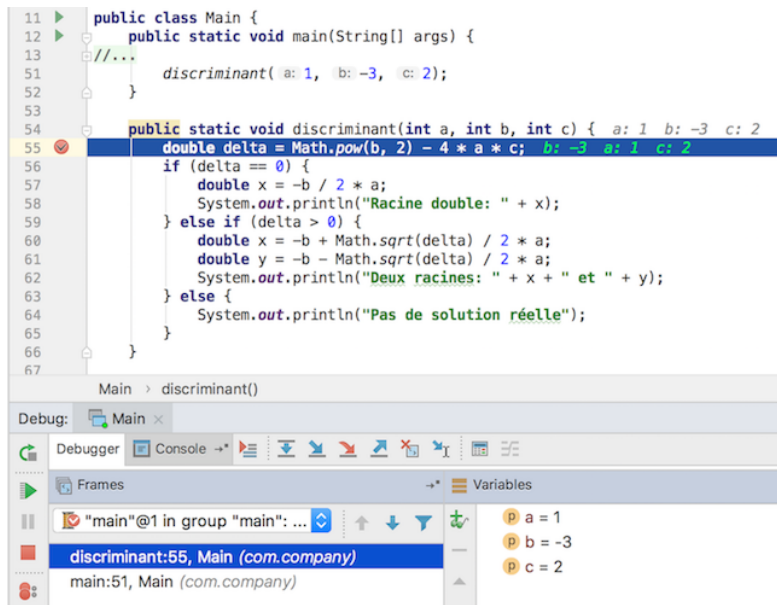


FIGURE 2 – Apparence debug sur IntelliJ.

— La promotion des pions.

Le plateau, dans son état initial, est illustré en Fig. 3.

4.2 Implémentation du jeu d'échecs

Pour chacune des classes suivantes, nous avons listé quelques attributs et méthodes. La structure des classes est parfois incomplète, et il vous reviendra de les compléter (en particulier, si vous souhaitez implémenter des constructeurs spécifiques ou des *getters/setters*).

Aussi, observez bien les visibilitées utilisées sur les différentes méthodes proposées, et cherchez à comprendre la raison de ces choix.

4.2.1 Classe Player

La classe **Player** va permettre de représenter un joueur d'échecs. Cette classe va déclarer deux attributs :

- **name** qui est une chaîne de caractères;
- **color** qui est un entier (0 pour blanc, 1 pour noir).

4.2.2 Classe Position

La classe **Position** va permettre de représenter une position sur l'échiquier en suivant le standard présenté en Fig. 3. Par exemple, le Roi blanc se trouve en **e1** tandis que le Roi noir se trouve en **e8**.

La classe **Position** va déclarer deux attributs et une méthode :

- **column** qui est un caractère;
- **row** qui est un entier;

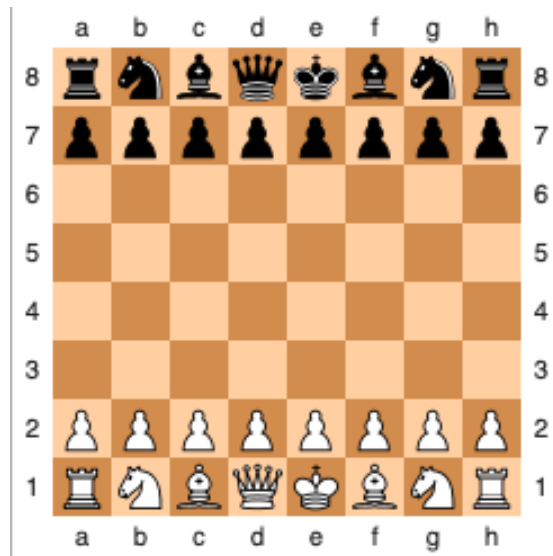


FIGURE 3 – État initial de l'échiquier.

- `public String toString()` qui retourne la position sous forme de chaîne de caractères comme mentionné plus haut.

4.2.3 Classes représentant les pièces du jeu

La suite du TP va consister en la création des six types de pièces utilisées dans le jeu d'échecs :

- `King` pour le Roi ;
- `Queen` pour la Reine ;
- `Bishop` pour le Fou ;
- `Knight` pour le Cavalier ;
- `Rook` pour la Tour ;
- `Pawn` pour le Pion.

Toutes ces classes vont déclarer les attributs suivants :

- `position` qui est de type `Position` ;
- `color` qui est un entier (0 pour blanc, 1 pour noir).

Puis, chacune de ces classes va définir sa propre implémentation des méthodes suivantes :

- `public boolean isValidMove(Position newPosition, Cell[][] board)` : l'implémentation de cette méthode va donc suivre les règles de déplacement mentionnées en Sect. 4.1 (en prenant en compte les exceptions concernant les déplacements *spéciaux*).
- `public String toString()` : cette méthode renvoie une lettre unique permettant d'identifier le type de pièce selon la Table 1 :

Attention : L'implémentation de la méthode `isValidMove()` est complexe et longue. Il faut prendre en compte plusieurs points :

- Les règles de déplacement de chaque type de pièce ;
- Le coup est-il valide ? (e.g., le coup est bien placé sur l'échiquier) ;
- Y a-t-il une pièce sur le chemin ?

Prenez donc le temps de réfléchir un peu à ces méthodes. Si vous n'y arrivez pas, vous pourrez par exemple simplement compléter cette méthode par un `return false;`.

Type de pièce	Caractère identifiant
Roi	K
Reine	Q
Fou	B
Cavalier	N
Tour	R
Pion	P

TABLE 1 – Table de correspondance entre type de pièce et notation standard.

4.2.4 Classe Cell

La classe `Cell` va représenter une cellule de l'échiquier. Cette classe va déclarer trois attributs :

- `position` qui est de type `Position` et qui peut être déclarée comme `final` ;
- `isEmpty` qui est une valeur booléenne et qui renvoie `true` si la cellule est vide, `false` sinon ;
- Un attribut serait utile pour pouvoir spécifier quelle pièce se trouve sur cette cellule. Mais nous nous heurtons au problème suivant : **Puisqu'il y a six types de pièces, devrions-nous créer six attributs (un par type de pièce) ?**

4.2.5 Classe Chess

La classe `Chess` va permettre de gérer une partie d'échecs entre deux joueurs. Pour ce TP, nous allons implémenter une partie de cette classe. Nous allons donc définir les attributs suivants :

- `board` qui sera de type `Cell[][]` ;
- `players` qui sera de type `Player[]` ;
- `currentPlayer` qui sera de type `Player` et qui permettra d'identifier le joueur en cours ;

Nous allons aussi déclarer les méthodes suivantes. **Attention**, certaines méthodes ne pourront pas encore être implémentées en raison de la limitation sur la classe `Cell`. Ce sera en particulier le cas pour la méthode `isCheckMate()`.

- `public void play()` qui va démarrer une partie ;
- `private void createPlayers()` qui va demander à chaque joueur de saisir son nom, puis va instancier deux objets de type `Player`. Cette méthode va aussi initialiser la valeur de `currentPlayer` ;
- `private void initialiseBoard()` qui va initialiser l'échiquier ;
- `private void printBoard()` qui va afficher l'état actuel de l'échiquier ;
- `private String askMove()` qui va demander au joueur en cours de saisir son prochain coup en spécifiant la pièce à déplacer puis l'emplacement suite au coup. Par exemple, si on reprend l'état initial de l'échiquier illustré en Fig. 3, alors les coups suivants sont tous valides : Nb1 Nc3, Pd1 Pd3, Pc1 Pc4 ;
- `private boolean isCheckMate()` est une méthode qui va permettre de déterminer si un joueur est échec et mat. C'est une méthode complexe à implémenter, et nous allons donc simplement la déclarer sans l'implémenter.
- `private boolean isValidMove(String move)` est une méthode qui va ensuite appeler la méthode `isValidMove()` de la pièce qui doit être déplacée ;
- `private void updateBoard(String move)` qui va mettre à jour l'échiquier suite a déplacement d'une pièce ;
- `private void switchPlayer()` qui va changer la valeur de `currentPlayer`.

La classe `Chess` aura donc la structure suivante :

Listing 1 – Classe `Chess`.

```
public class Chess {
    private Cell[][] board;
    private Player[] players;
    private Player currentPlayer;

    // On decrit brievement la logique du jeu
    public void play() {
        while (true) {
            createPlayers();
            initialiseBoard();
            while (!isCheckMate()) {
                printBoard();
                String move;
                do {
                    move = askMove();
                }
                while (!isValidMove(move));
                updateBoard(move);
                switchPlayer();
            }
        }
    }

    // On declare et implemente les autres methodes mentionnees plus haut
}
```

4.2.6 Classe `Main`

La classe `Main` va simplement instancier un objet de type `Chess` puis appeler sa méthode `play()`. Ces quelques lignes suffisent pour démarrer une partie d'échecs.

Pourquoi avoir créé une classe `Chess` et ne pas avoir implémenter cette logique directement dans la classe `Main`? Parce que nous avons vu qu'une méthode statique (la méthode `main()`) ne peut appeler que d'autres méthodes statiques et utiliser des attributs statiques, ce qui restreint (voire fausse) l'implémentation du programme.

Ce type d'implémentation est assez classique, et cela peut probablement être repris dans le cadre de votre projet.

Listing 2 – Classe `Main`.

```
public class Main {
    public static void main(String[] args) {
        Chess chess = new Chess();
        chess.play();
    }
}
```

4.3 Bonus

Comme bonus, vous pouvez réfléchir à l'implémentation des règles de déplacements spéciaux. **Attention** : il ne sera peut-être pas possible de les implémenter tout de suite, sans changement

majeur dans votre programme.

Vous pouvez également réfléchir à l'implémentation d'une méthode `isCheckMate()` de la classe **Chess**, qui permet de déterminer si un joueur est échec et mat.