

II.1102 : Algorithmique et Programmation

TP 5 : Programmation orientée objet

Patrick Wang

18 – 22 novembre 2019

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 | Exercice | 3 |
| 4.1 | Présentation de l'exercice | 3 |
| 4.2 | Découpage du projet | 3 |
| 4.2.1 | Classe <code>Main</code> | 4 |
| 4.2.2 | Classe <code>Artist</code> | 4 |
| 4.2.3 | Classe <code>Album</code> | 5 |
| 4.2.4 | Classe <code>Song</code> | 5 |
| 4.2.5 | Classe <code>Library</code> | 5 |

1 Objectifs du TP

- Pratiquer encore un peu avec les structures de données ;
- 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.

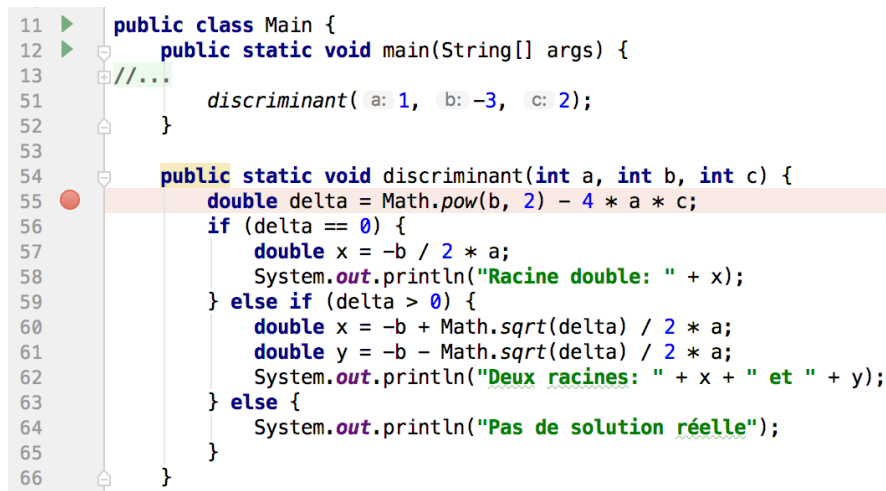
L’intérêt d’un point d’arrêt est de stopper l’exécution d’un programme avant d’exécuter une instruction particulière. Dans l’exemple de la Figure 1, le programme va donc s’arrêter avant d’exécuter la ligne 55. Un autre intérêt du mode debug est de pouvoir *inspecter* le contenu des variables et objets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



```

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.

- 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 Exercice

4.1 Présentation de l’exercice

Dans cet exercice, nous souhaitons développer un logiciel permettant de gérer une bibliothèque musicale. Dans cette bibliothèque, nous avons donc plusieurs albums, composés par des artistes, et contenant plusieurs chansons.

Avec cette bibliothèque musicale, nous serons en mesure d’ajouter ou de retirer de nouveaux albums ainsi que d’obtenir des informations sur les albums, artistes, et chansons.

4.2 Découpage du projet

Pour ce TP, nous aurons besoin de définir les cinq classes suivantes :

- La classe `Main.java`;
- La classe `Library.java`;
- La classe `Album.java`;
- La classe `Artist.java`;

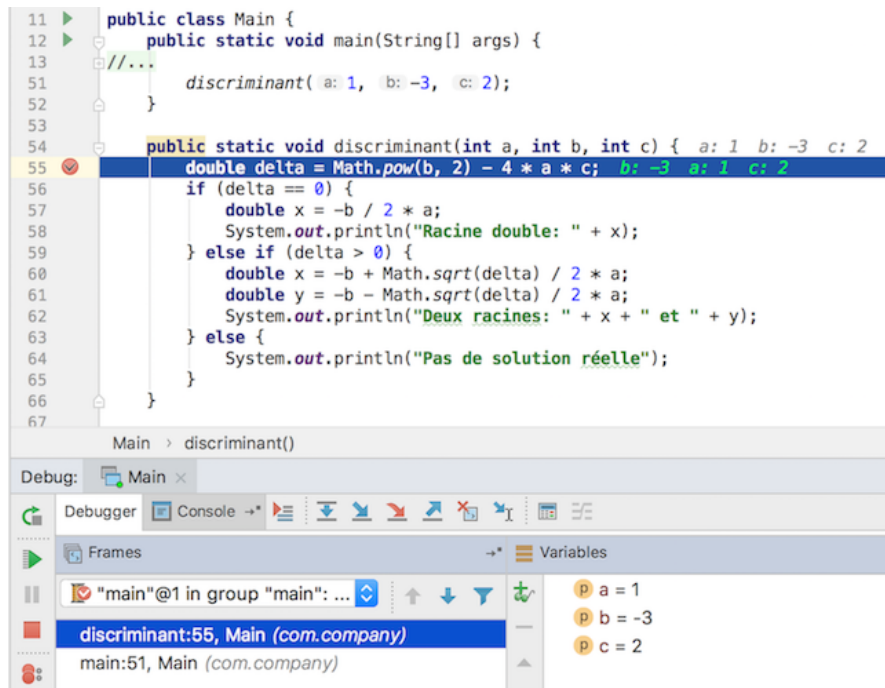


FIGURE 2 – Apparence debug sur IntelliJ.

— La classe `Song.java`.

4.2.1 Classe Main

La classe `Main` va lancer le programme en créant simplement une instance de la classe `Library`, puis en appelant la méthode `run()` de cette classe `Library`. La classe `Library` est décrite en Section 4.2.5.

4.2.2 Classe Artist

La classe `Artist` sera définie de la façon suivante :

- Un artiste possède trois attributs :
 - Un nom (sous forme de chaîne de caractères);
 - Une valeur booléenne indiquant si l'artiste est toujours actif ou non;
 - Une liste d'albums sous la forme de `List<Album>`;
- On va définir le constructeur par défaut et un autre constructeur permettant d'instancier un objet avec des valeurs précisées en paramètre;
- On va définir plusieurs méthodes :
 - Une méthode `getAlbums()` qui renvoie la liste des albums d'un artiste;
 - Une méthode `getName()` qui retourne le nom de l'artiste;
 - Une méthode `addAlbum(Album album)` qui ajoute un album à la liste des albums d'un artiste;
 - Une méthode `removeAlbum(String albumName)` qui retire un album à la liste des albums d'un artiste. Cette méthode pourrait avoir besoin d'une autre méthode `getAlbumByName(String albumName)` qui retourne un `Album` ou `null`;

- Une méthode `toString()` qui retourne simplement le nom de l'artiste ;

Comme indiqué dans la description de la classe `Library`, on va utiliser un `TreeSet` pour sauvegarder les artistes. Un `TreeSet` implémente nativement un ordre, et il faut donc être capable de *comparer* deux artistes. Pour cela, il faudra créer la classe `Artist` pour qu'elle ressemble au bout de code suivant :

```
public class Artist implements Comparable {
    // Implements attributs, constructors, and methods

    @Override
    public int compareTo(Object o) {
        Artist other = (Artist) o;
        return this.getName().compareTo(other.getName());
    }
}
```

4.2.3 Classe Album

La classe `Album` sera définie de la façon suivante :

- Un album possède trois attributs :
 - Un titre (sous forme de chaîne de caractères) ;
 - Une année de sortie (un entier) ;
 - Des chansons (sous forme de `List` de chansons) ;
- On va définir le constructeur par défaut ainsi qu'un autre constructeur permettant d'instancier un objet avec des valeurs précisées en paramètre ;
- Une méthode `toString()` qui va retourner une chaîne de caractères au format suivant :

```
Titre Album
01 - Titre chanson 1 (mm:ss)
02 - Titre chanson 2 (mm:ss)
03 - Titre chanson 3 (mm:ss)
...
```

4.2.4 Classe Song

La classe `Song` sera définie de la façon suivante :

- Une chanson possède deux attributs :
 - Un titre (sous forme de chaîne de caractères) ;
 - Une durée (sous forme d'un nombre entier représentant le nombre de secondes) ;
- On va définir plusieurs méthodes :
 - Le constructeur par défaut ainsi qu'un autre constructeur permettant d'instancier un objet avec des valeurs précisées en paramètre ;
 - Une méthode `toString()` qui va retourner une chaîne de caractères sous la forme :
titre (mm:ss) ;
 - Une méthode `convertDuration()` qui renvoie une chaîne de caractères au format
mm:ss.

4.2.5 Classe Library

La classe `Library` sera définie de la façon suivante :

- Une bibliothèque qui sera de type `TreeSet<Artist>` ;

- On va définir plusieurs méthodes :
 - Une méthode `run()` qui va initialiser le `TreeSet` et appeler la méthode `displayMenu()` dans une boucle infinie ;
 - Une méthode `displayMenu()` qui va afficher un menu de contrôle de cette bibliothèque. La saisie de l'utilisateur sera récupérée et, selon la valeur saisie, on va appeler une des méthodes suivantes. Un exemple de menu est fourni ci-après :
 Bienvenue sur votre bibliothèque musicale !
 Tapez 1 pour ajouter un artiste à votre collection.
 Tapez 2 pour supprimer un artiste de votre collection.
 Tapez 3 pour lister tous les artistes.
 Tapez 4 pour ajouter un album à un artiste.
 Tapez 5 pour retirer un album à un artiste.
 Tapez 6 pour lister tous les albums pour un artiste donné.
 - Une méthode `addArtist()` qui demande à l'utilisateur de saisir les informations concernant un artiste puis ajoute cet artiste à la bibliothèque **si possible** ;
 - Une méthode `removeArtist()` qui demande à l'utilisateur de saisir le nom d'un artiste et supprime cet artiste de la bibliothèque **si possible**. Cette méthode pourra avoir besoin d'une méthode `getArtistByName(String artistName)` qui renvoie un `Artist` ou `null` si aucun artiste ne correspond ;
 - Une méthode `listArtists()` qui va afficher la liste des artistes. Puisqu'on utilise un `TreeSet`, cette liste doit afficher la liste des artistes triés alphabétiquement par nom ;
 - Une méthode `addAlbum()` qui va demander à l'utilisateur de saisir le nom d'un artiste. Si cet artiste est présent dans la bibliothèque, on va demander à l'utilisateur de saisir les informations de l'album. Cet album sera enfin ajouté à la liste d'albums pour cet artiste. Cette méthode pourra avoir besoin d'une méthode `createSong()` qui crée une chanson à partir des informations saisies par l'utilisateur ;
 - Une méthode `removeAlbum()` qui va demander à l'utilisateur le nom d'un artiste. Si cet artiste est présent dans la bibliothèque, on va demander à l'utilisateur de saisir le nom d'un album pour le retirer de la liste des albums pour cet artiste. Cette méthode pourra avoir besoin d'une méthode `removeAlbum()` implémentée par la classe `Artist` ;
 - Une méthode `listAlbumsforArtist()` qui demande à l'utilisateur le nom d'un artiste. Si cet artiste est présent dans la bibliothèque, on va afficher tous les albums de cet artiste en utilisant le format décrit en Section 4.2.3.