Programmation Orientée Objet
IG2I, LE2 – 2023-2024

TP4 – Ordonnancement

Diego Cattaruzza, Maxime Ogier, Pablo Torrealba

Contexte et objectif du TP

Dans ce TP, nous poursuivons notre travail afin de proposer un outil informatique qui puisse améliorer l'ordonnancement de la production de l'entreprise Masques&Confettis.

Pour rappel, l'entreprise nordiste Masques&Confettis doit faire face à une grosse demande de production de masques colorés, car le carnaval de Dunkerque se déroule prochainement. Cette année, la demande est vraiment exceptionnelle et les gestionnaires de la production ne sont pas certains d'avoir le temps de fabriquer tous les masques. Par ailleurs, dans les contrats conclus avec ses clients, l'entreprise Masques&Confettis s'engage à respecter une date limite de livraison et à payer une forte pénalité en cas de retard (proportionnelle au retard). Il est donc essentiel de respecter au mieux les délais imposés par les clients.

Pour améliorer son planning de production, Masques&Confettis a décidé de faire appel aux meilleurs étudiants de la région Nord-Pas de Calais pour concevoir des outils informatiques qui puissent fournir un planning de production capable de gérer des demandes importantes.

Dans le TP précédent, nous avons mis en place un modèle objet pour l'atelier de production. L'objectif de ce TP est maintenant de concevoir des algorithmes qui permettent de proposer des solutions de bonne qualité pour l'ordonnancement des tâches de production.

Ce TP permet ainsi d'illustrer les notions suivantes :

- la notion d'interface ;
- la notion de méthode statique ;
- les méthodes **equals** et **hashCode** de la classe **Object** ;
- l'interface **Comparable** ;
- l'utilisation de méthodes de tri.

Bonnes pratiques à adopter

- Testez votre code au fur et à mesure de votre avancement.
- Respectez les conventions de nommage Java.
- Faites des indentations de manière correcte.
- Donnez des noms compréhensibles et pertinents à vos variables, attributs, classes, méthodes.
- Respectez autant que possible le principe d'encapsulation.
- Utilisez des structures de données adaptées.
- Faites des méthodes courtes avec peu de niveaux d'indentation.
- Ne faites pas de duplication de code, mais privilégiez le code réutilisable.
- Utilisez les outils de votre IDE pour faire du *refactoring* (clic-droit puis '*Refactor*' sur IntelliJ IDEA).
- Utilisez la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez.
- Quand c'est nécessaire, commentez votre propre code de façon pertinente au format Javadoc (commentaire commençant par `"/**"` au-dessus des définitions des attributs et méthodes).

1 Optimisation de la production : minimisation du temps total de production

C'est maintenant qu'il faut surprendre Monsieur Sorcier et lui montrer les capacités des étudiants d'IG2I !

Vous avez dû observer, à la fin du TP précédent, que Monsieur Sorcier ne tient pas compte des caractéristiques des tâches lorsqu'il détermine le planning de production. Or, il parait tout à fait logique de prendre en compte la durée des tâches et leur date limite de livraison. On pourrait ainsi espérer réduire le temps total d'exécution et les pénalités à payer.

Pour ce faire, vous allez procéder en deux temps. Dans cette section, nous nous intéressons à minimiser le temps total d'exécution (ce qui permet à Monsieur Sorcier de terminer les tâches plus vite, et ainsi de payer moins d'heures supplémentaires). Dans un second temps, en bonus, vous pourrez montrer à Monsieur Sorcier que vous pouvez aussi minimiser les coûts de pénalité !

1.1 Mise en place de classes d'ordonnancement

Nous souhaitons comparer ici les performances de plusieurs algorithmes pour ordonnancer une liste de tâches dans un atelier avec n machines en parallèle. D'un point de vue conception du code, nous proposons que chaque algorithme soit implémenté dans une classe dédiée, dans une méthode **public Atelier ordonnancer(int nombreMachines, List<Tache> taches)**. Cette méthode prend donc en paramètre un nombre de machines, et une liste de tâches et devra renvoyer l'atelier qui représente les tâches ordonnancées sur les machines selon l'algorithme utilisé.

Comme nous souhaitons utiliser des méthodes avec la même signature dans des classes différentes, il parait pertinent de faire une interface qui définit cette méthode, et chaque classe qui correspond à un algorithme implémentera donc cette interface.

Question 1. Créez un nouveau paquetage **ordonnancement** dans lequel on aura les objets qui permettent de fabriquer les ordonnancements dans l'atelier. Dans ce nouveau paquetage, créez une interface **Ordonnancement**. Cette interface ne définit qu'une seule méthode comme expliqué ci-dessus.

Nous allons par la suite définir plusieurs classes qui correspondent chacune à un algorithme pour ordonnancer les tâches. Ainsi, afin de pouvoir réaliser plusieurs ordonnancements d'une même liste de tâches, il faudra pouvoir faire une copie (en profondeur) de cette liste car chaque tâche aura une date de début potentiellement différente selon l'algorithme d'ordonnancement utilisé. Ce sont donc les copies des tâches qui seront ensuite ordonnancées.

Question 2. Ajoutez dans l'interface **Ordonnancement** une méthode statique **public static List<Tache> copierTaches(List<Tache> taches)** qui réalise une copie de la liste des tâches en paramètre. Pour que votre code fonctionne correctement, il faudra que la copie réalisée soit une **copie en profondeur**, i.e. que chaque tâche est également copiée, ce qui signifie qu'il faut un constructeur par copie dans la classe **Tache**. Vous pouvez tester votre méthode avec le code suivant :

```
1  List<Tache> taches = new ArrayList<>();
2  taches.add(new Tache(100));
3  taches.add(new Tache(150));
4  List<Tache> copieTaches = Ordonnancement.copierTaches(taches);
5  copieTaches.add(new Tache(400));
6  System.out.println("Nombre de taches initiales : " + taches.size());
7  // Doit etre 2
8  System.out.println("Nombre de taches finales : " +
       copieTaches.size());
9  // Doit etre 3
```

Question 3. Ajoutez une classe **OrdonnancementMonsieurSorcier** qui **implémente** l'interface **Ordonnancement**. Cette classe crée l'atelier, puis y ordonnance les tâches avec le même modus operandi que celui proposé par Monsieur Sorcier.

Question 4. Ajoutez dans la classe **Atelier** une méthode **public void afficher(boolean verbose)** qui affiche les critères (temps total d'exécution, coût total de pénalité) de l'atelier. Si le paramètre **verbose** vaut **true**, alors on affiche également l'atelier (ceci peut être utile pour du débogage). Faites un test pour vérifier que tout fonctionne bien si vous essayez d'ordonnancer quelques tâches.

En général, si on veut tester les performances d'un algorithme, il faut se construire un jeu de test avec des instances (ici nombre de machines et liste de tâches) différentes. Sur Moodle, le fichier *OrdonnancementTester.java* contient une classe **OrdonnancementTester** avec 6 jeux de test. Pour chaque jeu de test, le nombre de machines et le nombre de tâches sont donnés dans la Table 1.

jeu de test	1	2	3	4	5	6
nb. machines	2	3	4	4	4	3
nb. tâches	6	40	40	60	20	1000

Table 1: Nombre de machines à considérer pour les jeux de test.

Question 5. Le fichier *OrdonnancementTester.java* contient une classe **OrdonnancementTester** à ajouter dans le paquetage **ordonnancement**. Dans cette classe, il y a une méthode statique **comparerMethodesOrdonnancement** qui n'est pas encore implémentée. Implémentez cette méthode en vous aidant du commentaire Javadoc qui décrit le comportement attendu pour cette méthode.

Regardez la méthode **main** de la classe **OrdonnancementTester**, qui pour le moment ne met que la méthode d'ordonnancement de Monsieur Sorcier dans la liste des ordonnancements à comparer. Faites un test avec cette méthode **main** pour vérifier que tout fonctionne correctement.

1.2 Tri des éléments d'une collection

À présent, nous cherchons à développer de nouvelles méthodes d'ordonnancement afin de minimiser le temps total d'exécution. Pour cela, une première idée simple à mettre en œuvre est de trier les tâches par durée d'exécution croissante avant d'appliquer à nouveau le modus operandi expliqué par Monsieur Sorcier.

Nous allons voir que la langage Java possède des méthodes efficaces de tri qui sont déjà implémentées. Mais pour pouvoir utiliser ces méthodes de tri, il

faut au préalable définir une relation d'ordre sur les objets, i.e. étant donnés deux objets **o1** et **o2**, dire si **o1** est plus petit, égal ou plus grand que **o2**. Et donc avant de définir une relation d'ordre sur des objets, il faut au préalable avoir identifié ce que signifie que deux objets **o1** et **o2** sont égaux.

Nous proposons donc par la suite de commencer par discuter la notion d'égalité de deux objets dans la Section 1.2.1, puis de discuter de la relation d'ordre entre des objets dans la Section 1.2.2, et enfin de voir comment on peut trier les éléments d'une collection d'objets en Java dans la Section 1.2.3.

1.2.1 Égalité entre deux objets et code de hachage

Quand deux objets sont de types primitifs (par exemple **int**, **double** ou **boolean**), on peut les comparer avec l'opérateur "==". Mais que se passe-t-il quand on compare deux objets (non primitif, donc qui héritent de la classe **Object**) avec l'opérateur "==" ? Dans le langage Java, l'opérateur "==" compare les références (donc les adresses mémoires) des deux objets. Cela peut faire sens dans certains cas. Mais cette comparaison est très restrictive. En effet, la question de savoir si deux objets sont égaux (au sens identiques, les mêmes) n'est pas liée à l'adresse mémoire, qui est une notion purement informatique. Mais pour répondre à cette question, on devrait s'intéresser à comparer si les états (donc les attributs) des deux objets sont similaires.

Prenons comme exemple le code suivant où l'on suppose que le constructeur par copie de la classe **Tache** crée une copie avec les mêmes valeurs d'attributs.

```
1  Tache t1 = new Tache(120);
2  Tache t2 = t1;
3  Tache t3 = new Tache(t1);
4  System.out.println(t2 == t1); // true
5  System.out.println(t3 == t1); // false
```

Dans l'exemple ci-dessus, alors que les trois objets **t1**, **t2** et **t3** ont les même valeurs d'attributs, le test d'égalité **t1==t2** renvoie **true** car les deux objets ont la même adresse mémoire, alors que le test d'égalité **t1==t3** renvoie **false** car les objets sont stockés à des adresses différentes.

Comme l'opérateur "==" peut ne pas être satisfaisant pour comparer des objets, il existe une autre manière, adaptée au paradigme de programmation orientée objet, de comparer des objets : la méthode **equals**. La classe **Object**, dont héritent toutes les autres classes, définit la méthode suivante :

```
1  public boolean equals(Object obj) {
2     return (this == obj);
3  }
```

Cette méthode prend en paramètre un objet **obj** de type **Object** (donc n'importe quel objet), et renvoie un booléen qui vaut **true** si l'objet courant **this** est égal à **obj**, et **false** sinon. Notez bien ici que comme **equals** est une méthode, cela permet de définir l'égalité entre deux objets comme on le souhaite. Par ailleurs, il est important de noter que l'implémentation de cette méthode **equals** dans la classe **Object** fait juste une comparaison des références (adresses mémoire) de **this** et **obj**, donc est similaire à l'opérateur "==".

Cependant, tout l'intérêt de la méthode **equals** réside dans le fait que lorsque l'on fait de l'héritage, on peut également faire de la redéfinition de méthode (déjà définie dans une classe parente). Ainsi, comme chaque classe hérite de la classe **Object**, chaque classe peut redéfinir la méthode **equals** en se basant par exemple sur les valeurs de certains attributs. En pratique, lorsque l'on ajoute une nouvelle classe dans un code, de la même manière que l'on redéfinit la méthode **toString**, il est d'usage de redéfinir également la méthode **equals**. Pour se faire, les environnements de développement proposent en général des assistants pour faire de telles redéfinitions.

Une dernière remarque importante : la méthode **equals** est très liée à une autre méthode de la classe **Object** : la méthode **hashCode**. Voici la signature de cette méthode :

```
1  public int hashCode() { ... }
```

Il s'agit donc d'une méthode qui ne prend aucun paramètre et qui renvoie un nombre entier. Ce nombre est appelé *code de hachage*. Nous verrons dans les TPs suivants à quoi il peut servir. Pour le moment, on peut voir le code de hachage comme une représentation de l'état d'un objet (la valeurs de ses attributs) simplifiée en un nombre entier. Par exemple, considérons une classe **Personne** qui serait définie par de nombreux attributs comme le nom, le prénom, l'adresse, la date de naissance, la taille, le poids, le numéro de sécurité sociale. Un exemple (parmi d'autres) de code de hachage pourrait être de renvoyer l'année de naissance de la personne : c'est bien un nombre entier, et ça représente de manière très simplifiée la personne.

Cependant, on dit que les méthodes **equals** et **hashCode** sont liées car la méthode **hashCode** doit respecter le contrat suivant (vous pouvez vérifier dans la Javadoc) :

- l'appel de la méthode **hashCode** sur un même objet (une même instance) doit toujours renvoyer la même valeur ; donc par exemple pour une classe **Personne** renvoyer le poids, la taille ou l'âge n'est pas une bonne idée ;
- si deux objets sont égaux au sens de la méthode **equals**, alors la méthode **hashCode** doit renvoyer la même valeur ; le code de hachage doit bien être une représentation simplifiée de l'objet, donc des objets égaux doivent avoir le même code qui les représente ;

- par contre, si deux objets ne sont pas égaux, rien n'est spécifié, le code de hachage de ces deux objets peut être le même ou différent : ce serait typiquement le cas si on prend comme code de hachage l'année de naissance pour une personne.

Ainsi, lorsque l'on redéfinit la méthode **equals**, il fait également redéfinir la méthode **hashCode** afin de garantir la cohérence entre les deux méthodes. En pratique, la méthode **hashCode** se base sur un sous-ensemble des attributs utilisés pour définir la méthode **equals**. Notez que la méthode **hashCode** de la classe **Object** renvoie un nombre entier qui est une représentation de l'adresse mémoire de l'objet.

Question 6. Ajoutez dans la classe **Tache** une redéfinition des méthodes **equals** et **hashCode** afin d'expliciter l'égalité entre deux objets de type **Tache**. Réfléchissez bien à quel ensemble d'attributs vous permet facilement de définir l'égalité entre deux tâches.

Vous pouvez utiliser les assistants de IntelliJ IDEA (clic-droit puis *Generate ...* puis *equals() and hashCode()*). Prenez bien le temps d'analyser le code généré et de vous assurer que vous comprenez ce code. Vous pouvez regarder l'implémentation d'une méthode avec le racourci *Ctrl+Alt+B*.

Testez dans la méthode principale de la classe **Tache** les méthodes **equals** et **hashCode**.

1.2.2 Relation d'ordre sur des objets

Nous venons de voir que dans le paradigme de programmation orientée objet, on peut définir si deux objets sont égaux en se basant sur les valeurs de certains des attributs de ces objets.

Il est possible que l'on souhaite aller plus loin, et que l'on veuille savoir si un objet est plus petit ou plus grand qu'un autre. Cela s'appelle définir une relation d'ordre : les objets peuvent alors être classés du plus petit au plus grand dans l'ordre croissant. On voit bien que cette notion de relation d'ordre est primordiale dès lors que l'on souhaite trier des objets.

Dans le langage Java, pour qu'une classe **E** définisse une relation d'ordre, il faut que cette classe implémente l'interface générique **Comparable<E>** où **E** est le type des objets à comparer. L'interface **Comparable<E>** est déjà définie dans le langage Java, de la manière suivante :

```
1  public interface Comparable<E> {
2     public int compareTo(E o);
3  }
```

L'interface **Comparable<E>** déclare donc une seule méthode : **compareTo**. Cette méthode doit comparer l'objet courant (**this**) à l'objet reçu en paramètre (**o**) et renvoyer un entier dont la valeur doit être :

- négative si "**this** < **o**" ;
- nulle si "**this** = **o**" ;
- positive si "**this** > **o**".

On définit alors un ordre sur les objets manipulés. Notez que dans le cas des valeurs positives et négatives, la valeur exacte n'a pas d'importance, seul le signe compte.

Par ailleurs, comme nous l'avons vu précédemment, chaque objet possède une méthode **public boolean equals(Object obj)**. Afin d'assurer la cohérence du code, il est très fortement recommandé que l'ordre naturel défini par la méthode **compareTo** soit consistant avec l'égalité définie par la méthode **equals**. Ainsi, le contrat à respecter entre les méthodes **compareTo** et **equals** est le suivant :

- si la méthode **compareTo** renvoie 0, alors la méthode **equals** renvoie **true** ;
- si la méthode **equals** renvoie **true**, alors la méthode **compareTo** renvoie 0.

Par conséquence, la méthode **compareTo** renvoie une valeur non nulle si et seulement si la méthode **equals** renvoie **false**.

Pour terminer, regardons un petit exemple. Nous considérons une classe **CompteBancaire** définie par deux attributs : un numéro de compte (supposé unique), et le solde disponible sur le compte. Deux comptes sont égaux s'ils ont le même numéro de compte, et le code de hachage se basera donc également sur le numéro de compte. Par ailleurs, on souhaite introduire une relation d'ordre entre les comptes bancaires, basée sur le solde afin que les comptes bancaires soient triés par solde croissant. On peut alors écrire le code suivant, dans lequel on assure la consistance entre les méthodes **equals**, **hashCode** et **compareTo**.

```
1  public class CompteBancaire implements Comparable<CompteBancaire> {
2     private int numeroCompte;
3     private double solde;
4
5     @Override
6     public boolean equals(Object o) {
7        if (this == o) return true;
8        if (o == null || getClass() != o.getClass()) return false;
9        CompteBancaire that = (CompteBancaire) o;
10       return numeroCompte == that.numeroCompte;
11    }
12
13    @Override
14    public int hashCode() {
15       return Objects.hash(numeroCompte);
16    }
17
```

```
18      @Override
19      public int compareTo(CompteBancaire o) {
20          if(o == null) {
21              return -1; // this < o
22          }
23          if(this.numeroCompte == o.numeroCompte) {
24              return 0; // coherence avec equals
25          }
26          if(this.solde < o.solde) {
27              return -1; // this < o
28          } else if(this.solde > o.solde) {
29              return 1; // this > o
30          } else { // coherence avec equals
31              if(this.numeroCompte < o.numeroCompte) {
32                  return -1; // this < o
33              } else { // this.numeroCompte > o.numeroCompte
34                  return 1; // this > o
35              }
36          }
37      }
38  }
```

Ici, les méthodes **equals** et **hashCode** ont été générées automatiquement avec IntelliJ IDEA en se basant sur l'attribut **numeroCompte**. Notez que l'on pourrait tout à fait réécrire la méthode **hashCode** de la manière suivante :

```
1   @Override
2   public int hashCode() {
3       return numeroCompte;
4   }
```

Pour ce qui concerne la relation d'ordre, la classe **CompteBancaire** implémente donc l'interface **Comparable<CompteBancaire>**, et implémente donc la méthode **compareTo**. Notez dans cette méthode que dans les lignes 23 à 25, on s'assure que si les objets sont égaux (même numéro de compte) alors la méthode renvoie 0. Dans les lignes 31 à 35, on s'assure de ne pas renvoyer 0 si deux comptes avec des numéros de compte différents ont le même solde. Dans ce cas, on choisit donc on second critère d'ordre basé sur le numéro de compte. Ceci permet d'assurer que si les objets ne sont pas égaux, alors la méthode renvoie bien une valeur non nulle. Notez au passage que comme la valeur exacte renvoyée par la méthode **compareTo** n'a pas d'importance, on pourrait remplacer les lignes 31 à 35 par :

```
1   return this.numeroCompte - o.numeroCompte;
```

Question 7. Modifiez la classe **Tache** de sorte qu'elle définisse une relation d'ordre pour ordonner les tâches par temps de production croissant. Faites bien attention à assurer la cohérence entre la méthode **compareTo** et la méthode **equals**. Testez dans la méthode principale de la classe **Tache** que la méthode **compareTo** fonctionne correctement.

1.2.3 Trier des objets

La classe **Collections** dispose d'une méthode statique **sort(List<T> list)** qui réalise un tri des éléments de **list** (qui doit implémenter l'interface **List**). Les éléments de **list** sont réorganisés de façon à respecter l'ordre naturel des éléments de la liste (les éléments doivent donc implémenter l'interface **Comparable**). L'efficacité du tri est en $\mathcal{O}(n \log n)$ avec n le nombre d'éléments dans la liste (n'hésitez pas à jeter un œil à la Javadoc pour plus de précisions).

Pour trier une liste selon l'ordre inverse de l'ordre naturel, on peut commencer par faire un tri selon l'ordre naturel, puis on utilise la méthode statique **reverse (List<T> list)** de la classe **Collections**. Cette méthode inverse l'ordre des éléments de la liste (en temps linéaire).

Par ailleurs, pour mélanger les éléments d'une liste, vous pouvez faire appel à la méthode statique **shuffle(List<T> list)** de la classe **Collections** : elle "mélange" les élément dans la liste. Cela peut être utile lors de vos tests par exemple.

Question 8. Définissez deux nouvelles classes qui implémentent l'interface **Ordonnancement** :

- **OrdonnancementTempsCroissant**, qui trie les tâches par temps d'exécution croissant avant de faire l'affectation aux machines selon le modus operandi de Monsieur Sorcier ;
- **OrdonnancementTempsDecroissant** qui trie les tâches par temps d'exécution décroissant avant de faire l'affectation aux machines selon le modus operandi de Monsieur Sorcier.

Modifiez en conséquence les tests de la méthode **main** de la classe **OrdonnancementTester**, de sorte que l'on puisse comparer les trois méthodes d'ordonnancement en terme de temps total d'exécution. Faites les tests sur les 6 jeux de test. Que dire des résultats trouvés ? En regardant plus en détail le jeu de test 1, est-il possible de faire encore mieux (si oui proposez une meilleure solution, si non donnez un argument) ?

1.3 Un peu d'aléatoire

Question 9. Maintenant, c'est à vous ! Proposez d'autres méthodes d'ordonnancement qui permettent d'améliorer encore les résultats précédents. En particulier, on pourra penser à introduire un peu d'aléatoire.

2 Bonus - Minimisation du coût total de pénalité

Trier les tâches pour minimiser le temps total d'exécution de l'ordonnancement vous a permis d'obtenir un planning suffisamment efficace pour surprendre Monsieur Sorcier. Maintenant que vous savez construire de tels ordonnancements, Monsieur Sorcier vous fait remarquer que pour un même ordonnancement il est possible, en échangeant l'ordre d'exécution des tâches sur une machine, de minimiser les coûts de pénalités sans augmenter le temps total d'exécution.

Question 10. Modifiez la méthode **comparerMethodesOrdonnancement()** de la classe **OrdonnancementTester** de telle sorte qu'après avoir ordonnancé les tâches, on affiche les valeurs du temps total d'exécution et du coût total de pénalité, puis on essaye d'améliorer l'ordonnancement de sorte que le temps total d'exécution reste le même et le coût total de pénalité diminue. Vous êtes évidemment fortement encouragés à rajouter de nouvelles méthodes dans les classes du paquetage **atelier** si nécessaire. Si vos résultats sont stupéfiants, courez chez Monsieur Sorcier, peut-être qu'il va vous embaucher !!

References

[1] Delannoy, C., *Programmer en Java, Eyrolles, 2014*