# Contrôle Final 2 - CSC3101 - 25 Mars 2025 - 1h30

*Tout document écrit est autorisé. Pour répondre à une question, vous donnez le code correspondant (évitez les répétitions). Les classes sont à placer dans le paquetage par défaut. Sauf indication contraire, la visibilité des méthodes et attributs est celle par défaut. Il n'y a pas à écrire les directives* `import`.

Dans cet examen, nous nous aventurons dans le monde du trading algorithmique. Le terme trading algorithmique fait référence à l'utilisation d'ordinateurs sur les marchés financiers, ces lieux où s'échangent des actifs tels que des actions et obligations d'états. Nous commençons par modéliser les ordres d'achat et de vente présents dans un marché puis la notion centrale de carnet d'ordres. Plus loin, nous regarderons comment exécuter un ordre au marché.

### 1. Les ordres *(~20 minutes, 5pt)*

Un ordre contient un prix et un nombre d'actifs. Dans cet examen, le prix sera un flottant. Nous stockons un ordre dans une instance de la classe `Order`. Cette classe possède deux attributs: un prix (`float price`) et une quantité d'actifs (`int shares`). Le constructeur de cette classe reçoit deux arguments pour initialiser chacun de ses attributs. La classe `Order` contient aussi les deux getters suivantes :
- `float getPrice()` retourne le prix,
- `int getShares()` retourne la quantité d'actifs.

**[Q1a]** *(2pt)* Donnez le code de la classe `Order` (à savoir, ses attributs, son constructeur et ses deux getters).

On souhaite pouvoir comparer deux ordres. Pour ce faire, la classe `Order` doit mettre en œuvre l'interface `Comparable<Order>`.

**[Q1b]** *(1pt)* Quelle est la signature de la classe `Order` pour mettre en œuvre `Comparable<Order>` ?

L'interface `Comparable<Order>` définit la méthode `public int compareTo(Order other)`. Soient deux ordres `x` et `y`, `x.compareTo(y)` doit retourner 1 si le prix de `x` est plus grand que le prix de `y`. Dans le cas inverse, l'appel retourne -1. En cas d'égalité, la méthode retourne 0.

**[Q1c]** *(2pt)* Donnez le code de la méthode `compareTo(Order other)`.

### 2. Le carnet d'ordres *(~30 minutes, 7pt)*

Un ordre peut correspondre à un achat ("bid", en anglais) ou à une vente ("ask"). Le carnet d'ordres ("order book") stocke les ordres passés par les traders sur un actif et les exécutent. Pour ce faire, il utilise deux champs:
- `descendingBids` liste les ordres d'achat par ordre décroissant de leurs prix
- `ascendingAsks` liste les ordres de vente par ordre croissant de leurs prix

Pour rappel, le JDK fournit un support pour stocker des listes en Java. En effet, soit `T` une classe quelconque, alors la classe `List<T>` permet de stocker une liste d'instances de `T`.

**[Q2a]** *(1pt)* Rappelez le lien qui existe entre les classes `List` et `ArrayList` du JDK. Donnez le code pour créer une liste d'ordres. Comment est stockée en mémoire cette liste ?

**[Q2b]** *(1pt)* Donner le code du constructeur de la classe `OrderBook`. Ce constructeur prend en argument deux listes afin d'initialiser les champs `descendingBids` et `ascendingAsks`. On suppose que ces listes sont déjà triées.

En finance, on représente un ordre (d'achat ou de vente) par une paire $(P, N)$, où $P$ est le prix et $N$

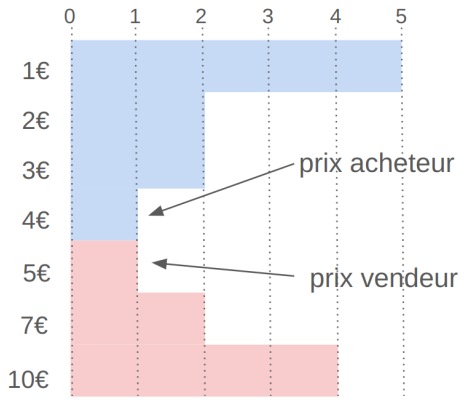


Figure 1: Illustration d'un carnet d'ordres

le nombre d'actifs. Les listes du carnet d'ordre ont donc la forme suivante:

$$\text{Ordres d'achat: } \left[\left(P_i^{(b)}, N_i^{(b)}\right) \mid 0 \leq i < m\right], P_i^{(b)} > P_j^{(b)} \text{ pour } i < j$$
$$\text{Ordres de vente: } \left[\left(P_i^{(a)}, N_i^{(a)}\right) \mid 0 \leq i < n\right], P_i^{(a)} < P_j^{(a)} \text{ pour } i < j$$

Par ailleurs, on utilise aussi les termes standards suivants: Nous faisons référence à $P_0^{(b)}$ en tant que *prix acheteur* ("bid price"). C'est le prix maximum qu'un acheteur est prêt à payer pour un actif. A l'inverse, $P_0^{(a)}$ est le prix minimum auquel un vendeur est prêt à céder un actif. On l'appelle le prix vendeur ("ask price"). La valeur $\frac{P_0^{(a)}+P_0^{(b)}}{2}$ est appelé prix médian ("mid price"). L'écart entre le prix acheteur et le prix vendeur ("bid-ask spread" en anglais) est défini par $P_0^{(a)} - P_0^{(b)}$. Enfin, $P_{n-1}^{(a)} - P_{m-1}^{(b)}$ est la profondeur du marché.

Un carnet d'ordres est illustré dans la Figure 1. Dans cette figure, les ordres d'achat sont en bleu et les ordres de vente en rouge. Le prix acheteur est de 4€ et le prix vendeur de 5€.

**[Q2c]** *(1pt)* Quel est le spread dans la Figure 1 ? Dans cette même figure, quelle est la profondeur du marché ?

**[Q2d]** *(4pt)* Ajoutez les méthodes suivantes à la classe `OrderBook`:
- `float bidPrice()` retourne le prix acheteur
- `float askPrice()` retourne le prix vendeur
- `float midPrice()` retourne le prix médian
- `float bidAskSpread()` retourne le spread
- `float marketDepth()` retourne la profondeur du marché

Pour ce faire, vous pourrez utiliser la méthode `get(i)` de `List` qui retourne l'élément en position i dans la liste. Par ailleurs, la méthode `size()` renvoie la taille de la liste.

### 3. Exécution des ordres *(~20 minutes, 4pt)*

Dans une activité de marché normale, le prix d'un nouvel ordre de vente est généralement supérieur au prix acheteur (c.à.d. de la meilleure offre d'achat). Cependant, si cet ordre a un prix inférieur ou égal au prix acheteur, nous disons que le marché se croise, à savoir que les plages de prix de l'offre et de la demande s'intersectent. Ceci entraîne une transaction immédiate qui épuise les offres d'achat disponibles. Le vendeur vend alors (tout ou partie) de ses actifs au prix proposé par le (ou les) acheteurs.

En détail, quand un nouvel ordre de vente $(P, N)$ arrive et que le marché se croise, deux modifications ont lieu. D'abord, on supprime les meilleures offres d'achat du carnet d'ordres. À savoir, les offres suivantes:

$$\left[ \left( P_i^{(b)}, \min \left( N_i^{(b)}, \max \left( 0, N - \sum_{j=0}^{i-1} N_j^{(b)} \right) \right) \right) \mid \left( i : P_i^{(b)} \geq P \right) \right]$$

Par exemple, les offres d'achat de la Figure 1 forme la liste suivante: $[(4€, 1), (3€, 2), (2€, 2), (1€, 10)]$. Si une offre de vente $(2€, 4)$ arrive, on supprime les ordres suivants: $[(4€, 1), (3€, 2), (2€, 1)]$. Ainsi, la liste est changée en: $[(2€, 1), (1€, 10)]$.

De plus, il se peut que la vente soit incomplète, c'est à dire que tout n'a pas été vendu. Dans l'exemple précédent, ceci se produit par exemple si l'ordre de vente est $(4€, 2)$. Dans un tel cas, on ajoute l'ordre de vente suivant au carnet d'ordres:

$$\left( P, \max \left( 0, N - \sum_{i : P_i^{(b)} \geq P} N_i^{(b)} \right) \right)$$

La méthode `void executeSellOrder(Order sell)` de la classe `OrderBook` permet l'exécution d'un ordre de vente en suivant la logique ci-dessus. D'abord, elle supprime les ordres d'achat qui sont exécutés (c'est à dire dont le nombre de `shares` tombe à 0). Ensuite, si la vente est incomplète, elle ajoute un ordre de vente approprié dans le carnet d'ordres.

**[Q3a]** *(3pt)* Donnez le code de la méthode `void executeSellOrder(Order sell)`. Pour rappel, la méthode d'instance `removeAll(c)` de `List` permet de supprimer tous les éléments présents dans la collection `c`. Il n'est pas demandé de trier la liste des ordres de vente (variable `descendingAsks`) si tout n'est pas vendu.

**[Q3b]** *(1pt)* On souhaite ajouter une méthode `void executeBuyOrder(Order buy)` pour implémenter un ordre d'achat. Expliquer dans les grandes lignes (sans la coder nécessairement) comment mettre en œuvre cette méthode dans la classe `OrderBook`.

### 4. Ordres au marché *(~20 minutes, 4pt)*

Les ordres vus précédemment sont dit ordres limites. En effet, ils définissent des valeurs auxquelles l'actif est acheté/vendu. Un autre type d'ordre est dit au marché. Dans ce cas là, l'ordre est exécuté en utilisant les ordres présents dans le marché. En d'autres termes, une offre de vente au marché de $N$ actifs vendra les $N$ actifs aux meilleurs acheteurs présents sur le marché au prix que chacun d'eux demande. La vente rapportera donc le profit suivant:

$$\left( \sum_{i=0}^{K-1} P_i^{(b)} \times N_i^{(b)} \right) + \left( P_K^{(b)} \times \left( N - \sum_{i=0}^{K-1} N_i^{(b)} \right) \right)$$

où $K$ est défini par:

$$K = \min \left( \left\{ k \geq 0 \mid \left( \sum_{i=0}^{k} N_k^{(b)} \right) \geq N \right\} \right)$$

Par simplicité, on considère ici que les ordres d'achat présents sur le marché permettent toujours d'exécuter l'ordre au marché (quelque soit la valeur de $N$).

**[Q4a]** *(1pt)* La classe `MarketOrder` modélise un ordre au marché. Cette classe étend la classe `Order`. Elle contient un constructeur qui prend en paramètre un nombre d'actifs. (En effet le prix étant fonction du marché, il n'est pas défini par avance comme avec un ordre limite.) Donnez le code de la

classe `MarketOrder`. Pour ce faire vous utiliserez le mot clé `super` en passant une valeur abitraire de prix (e.g., 0) au constructeur de la classe mère.

**[Q4b]** *(3pt)* Écrivez le code de la méthode `float executeSellMarketOrder(MarketOrder sell)` qui permet de réaliser un ordre de vente au marché. Cette méthode utilisera la méthode `executeSellOrder(Order sell)` que vous avez précédemment écrite. La valeur retournée par un appel à `executeSellMarketOrder` est le profit réalisé par cette vente. Quelle est la complexité d'un appel à `executeSellMarketOrder` en fonction du nombre d'offres d'achat ($n$) disponibles sur le marché ?

---

Voici un exemple d'utilisation du code écrit dans cet examen:

```
List<Order> bids = new ArrayList<>(
               List.of(new Order(4,1), new Order(3,2), new Order(2,2), new Order(1,10)));
List<Order> asks = new ArrayList<>(
               List.of(new Order(5,1), new Order(7,2), new Order(10,4)));
OrderBook ob = new OrderBook(bids, asks);
ob.executeSellOrder(new Order(3,4));
System.out.println(ob);

bids = new ArrayList<>(
       List.of(new Order(4,1), new Order(3,2), new Order(2,2), new Order(1,10)));
asks = new ArrayList<>(
       List.of(new Order(5,1), new Order(7,2), new Order(10,4)));
ob = new OrderBook(bids, asks);
float gain = ob.executeSellMarketOrder(new MarketOrder(6));
System.out.println(gain);
System.out.println(ob);
```

Si on exécute ce code, il affiche sur la console le résultat suivant:

```
bids=[(2.0,2), (1.0,10)]
asks=[(3.0,3), (5.0,1), (7.0,2), (10.0,4)]
15.0
bids=[(1.0,9)]
asks=[(5.0,1), (7.0,2), (10.0,4)]
```