Développement Web 2
L3 Informatique
Émilie SAMUEL

TP 3 - *Threads*

Sous Éclipse, dans votre *workspace* dédié à `DW2`, créer un nouveau projet `TP3`, avec à l'intérieur un *package* `tp3`. Dans ce *package*, créer une classe `Voiture` et une classe `Parking` (avec une méthode `+main(args: String[]): void` pour cette dernière). Ces deux classes implémentent l'interface `Runnable`.

Gestion d'un parking et de sa file d'attente

Dans ce TP, nous allons simuler l'activité d'un parking. Des voitures vont venir s'y garer pour un certain temps, puis en repartir. Quand il n'y aura plus de places libres, les nouvelles voitures arrivées feront la queue jusqu'à ce qu'une place se libère, ou que le conducteur perde patience.

Puisque les deux classes `Parking` et `Voiture` implémentent l'interface `Runnable`, elles pourront donner lieu à des *threads*. Celui du parking aura pour but de gérer sa file d'attente. Ceux des voitures auront pour but d'entrer dans la file d'attente, d'attendre que ce soit leur tour (si le conducteur est suffisamment patient), de se garer un temps, puis de repartir.

Vous trouverez à la fin de ce document un récapitulatif des classes et méthodes Java dont vous aurez besoin pour réaliser ce TP.

Le parking

Variables :

- `-capacite: int` le nombre total de places ;
- `-tarifHoraire: double` le tarif horaire de stationnement ;
- `-chiffreAffaire: double` le montant total payé par les voitures pour stationner sur toute la durée de vie du parking ;
- `-voituresGarees: HashMap<Integer, Voiture>` la table de hachage des voitures actuellement garées sur le parking (avec en clé le numéro de la place occupée par chacune) ;
- `-fileAttente: LinkedList<Voiture>` la file d'attente des voitures faisant actuellement la queue pour se garer sur le parking.

Attention : définissez les deux dernières variables avec le mot-clé `volatile`. L'idée est de contraindre les *threads* accédant à ces données à maintenir toujours une version à jour de leur contenu dès que celui-ci change.

Méthodes :

- `+Parking(c: int, th: double)` constructeur affectant la capacité du parking et son tarif horaire, initialisant le chiffre d'affaire à 0, instanciant sa table de hachage et sa file d'attente (vides toutes les deux), enfin affichant un message de confirmation de création (avec la capacité et le tarif horaire) ;
- `-estPlein(): boolean` qui renvoie `true` si le parking est actuellement plein, et `false` sinon ;
- `-existeVoituresEnAttente(): boolean` qui renvoie `true` si au moins une voiture est actuellement dans la file d'attente du parking, et `false` sinon.

Puisque plusieurs voitures vont accéder au même parking simultanément, penser à définir les deux dernières méthodes comme étant synchronisées.

Les voitures

Variables :

- `-immatriculation: String` la plaque d'immatriculation de la voiture ;
- `-dureeStationnement: int` indiquant combien de temps cette voiture va rester garée sur le parking (en millisecondes, et plus tard on considérera que 1ms équivaut à 1h) ;
- `-patience: int` indiquant combien de temps le conducteur accepte de faire la queue dans la file d'attente (idem) ;
- `-parking: Parking` le parking sur lequel cette voiture veut se garer ;
- `-statut: int` indiquant où se trouve la voiture à un instant donné. S'il vaut 0, alors elle est dans la file d'attente, s'il vaut 1, alors elle est garée sur une place, s'il vaut 2, alors elle quitte le parking car son stationnement est terminé, s'il vaut 3, alors elle quitte la file d'attente car la patience du conducteur est épuisée ;
- `-numPlace: int` le numéro de la place occupée par la voiture sur le parking.

Attention : définissez ces deux dernières variables comme `volatile`.

Méthodes :

- `+Voiture(i: String, ds: int, p: int, pk: Parking)` constructeur affectant l'immatriculation, la durée de stationnement, la patience du conducteur et le parking. Le statut reste indéfini pour le moment, et le numéro de place est initialisé à -1 ;
- `+getImmatriculation(): String` accesseur ;
- `+getDureeStationnement(): int` accesseur ;
- `+setStatut(s: int): void` mutateur ;
- `+getNumPlace(): int` accesseur et `+setNumPlace(np: int): void` mutateur.

Thread d'une voiture

Quand une voiture arrive pour se garer (au lancement de son *thread*, donc dans sa méthode `+run(): void`) :

1. on appelle `-arriveeParking(): void` (de la classe `Voiture`) qui gère :
   - l'appel à la méthode synchronisée `+arriveeVoiture(v: Voiture): void` (classe `Parking`). À vous de l'implémenter : ajout de la voiture dans la file d'attente (actuellement vide ou non) du parking. Évidemment la voiture se place en dernière position ;
   - la mise à jour de son statut ;
   - l'affichage d'un message indiquant que la voiture (avec son numéro d'immatriculation) est désormais dans la file d'attente du parking.
2. une fois la voiture arrivée dans la file, son *thread* :
   - va rester actif tant qu'il ne reçoit pas de demande d'interruption (il y aura deux raisons possibles à cette demande, on y viendra) ;
   - à l'intérieur de cette boucle, il y aura une première étape d'attente dans la file, qui durera tant que le statut de la voiture l'indique. Ceci dépend :
     - de la durée d'attente de la voiture, qu'il faudra recalculer à chaque tour de la boucle d'attente (cf. annexe en fin d'énoncé). Dès qu'elle dépasse la patience, la méthode `-departParkingImpatience(): void` (classe `Voiture`) est appelée, qui gère :
       - l'appel à la méthode synchronisée `+departVoitureImpatience (v: Voiture): void` (classe `Parking`). À vous de l'implémenter : la voiture quitte la file d'attente ;
       - la mise à jour de son statut, et l'envoi dune demande d'interruption de son *thread*.
     - ou de la libération d'une place : on y vient.

Thread du parking

Le *thread* du parking consiste à gérer sa file d'attente. Tant qu'il n'est pas interrompu, il vérifie s'il existe des voitures en attente. Si c'est le cas, il tente de faire se garer la première : appel à la méthode `-garerVoiture(): void` (synchronisée, classe `Parking`) qui :

- si le parking est plein, affiche un message l'indiquant, et met le *thread* du parking en attente d'une notification de reprise (s'il reçoit une demande d'interruption pendant cette attente, il devra s'arrêter) ;
- sinon, sort la première voiture de la file d'attente (celle qui attend depuis le plus de temps), et la fait se garer sur l'une des places libres. On choisira un numéro de place pseudo-aléatoirement parmi celles disponibles. On suppose toujours que les places sont numérotées de 1 à la capacité du parking. Une fois la voiture garée, on affiche un message de confirmation (avec l'immatriculation et le numéro de la place). Enfin, on met à jour, pour la voiture, son statut et son numéro de place.

Stationnement de la voiture et départ

Le statut de la voiture ayant changé, son *thread* passe à la deuxième étape de son déroulé (étape suivant l'attente) : la voiture est garée. Cette étape n'a lieu que si son statut l'indique (car il se peut qu'elle soit partie par manque de patience). Il faut alors :

1. laisser la voiture stationner le temps nécessaire en mettant son *thread* en sommeil (dans le cadre du TP, il n'y a pas de raison que la voiture reçoive une demande d'interruption pendant son sommeil, vous pouvez donc laisser l'exception afficher la trace de l'erreur) ;
2. à son réveil, la voiture va quitter le parking : appel de la méthode `-departParking(): void` (classe `Voiture`) qui gère :
   - l'appel à la méthode `+departVoiture(v: Voiture): void` (synchronisée) de la classe `Parking`. À vous de l'implémenter : la voiture quitte le parking et, puisqu'une place se libère, on affiche un message l'indiquant (avec l'immatriculation et le numéro de la place libéré). On affiche la durée de stationnement et le gain apporté (on considère que 1000 millisecondes de stationnement correspondent à 1h). Mettre à jour le chiffre d'affaires. On indique au *thread* du parking qu'il peut reprendre ;
   - la remise à -1 de son numéro de place et la mise à jour son statut ;
   - l'envoi d'une demande d'interruption de son *thread*.
3. une fois la voiture partie (car fin de stationnement ou impatience), son *thread* va devoir s'interrompre. Faire afficher un message indiquant le départ, avec son numéro d'immatriculation et la raison (son statut vous la donne).

Fin de vie du parking

Enfin, ajouter, suite à la réception par le *thread* du parking d'une demande d'interruption, l'affichage d'un message indiquant le chiffre d'affaire total réalisé.

Test du programme et trace d'exécution

Pour tester votre programme, dans le `main()` de `Parking`, instancier un parking (avec peu de places), son *thread*, et lancez-le. Ensuite, instancier quelques voitures, en choisissant les temps de stationnement et de patience judicieusement pour vérifier que tout fonctionne, puis instancier et lancer leurs *threads*. Il peut être utile de placer quelques courts temps de sommeil pour le *thread* du `main()`, afin de vous assurer que les *threads* des voitures vont a priori se lancer l'un après l'autre, pour coller avec vos tests. Forcer l'attente de la fin de tous les *threads* de voitures avant d'interrompre à son tour celui du parking.

Une trace d'exécution est proposée sur la page suivante.

```
Création du parking, capacité 2, tarif horaire 2.5
Voiture V1 dans la file d'attente du parking
Voiture V1 garée sur la place 2
Voiture V2 dans la file d'attente du parking
Voiture V2 garée sur la place 1
Le parking est plein, mise en attente de son thread
Voiture V3 dans la file d'attente du parking
Voiture V4 dans la file d'attente du parking
Voiture V5 dans la file d'attente du parking
Voiture V6 dans la file d'attente du parking
Départ de la voiture V3 cause d'impatience
Fin de stationnement de V2, place 1 libérée
Gain de 10.0 pour une durée de 4000
Départ de la voiture V2 car fin de stationnement
Voiture V4 garée sur la place 1
Le parking est plein, mise en attente de son thread
Fin de stationnement de V1, place 2 libérée
Gain de 17.5 pour une durée de 7000
Départ de la voiture V1 car fin de stationnement
Voiture V5 garée sur la place 2
Le parking est plein, mise en attente de son thread
Départ de la voiture V6 cause d'impatience
Fin de stationnement de V4, place 1 libérée
Gain de 15.0 pour une durée de 6000
Départ de la voiture V4 car fin de stationnement
Fin de stationnement de V5, place 2 libérée
Gain de 15.0 pour une durée de 6000
Départ de la voiture V5 car fin de stationnement
Fin du thread du parking, chiffre d'affaire 57.5
```

Annexe - Élements Java

▶ `java.lang.Thread`
`Thread(o: Object)`
instanciation d'un *thread* à partir d'un objet implémentant l'interface `Runnable`
`+start(): void`
démarrage du *thread*, lance automatiquement sa méthode `run()`
`+sleep(int millis): void`
met le *thread* lisant cette ligne en sommeil (appeler sous la forme `Thread.sleep(...)`)
`+join(): void`
attend la fin d'exécution du *thread*
`+interrupt(): void`
envoie une demande d'interruption :
écrire `t.interrupt()` envoie au *thread* `t`
écrire `Thread.currentThread().interrupt()` envoie au *thread* lisant cette ligne
`+isInterrupted(): boolean`
indique si le *thread* a reçu ou non une demande d'interruption
écrire `Thread.currentThread().isInterrupted()` teste par rapport au *thread* lisant cette ligne

▶ `java.lang.Object` (toute classe en hérite)
`+wait(): void`
met l'objet courant en attente d'une notification de reprise
`+notifyAll(): void`
envoie une notification de reprise à tous les *threads* en attente sur l'objet courant

▶ interface `java.lang.Runnable`
`+run(): void`
méthode lancée automatiquement au démarrage du *thread*

▶ `java.util.HashMap<T1, T2>`
`HashMap<T1, T2>()`
instanciation d'une table de hachage dont les clés sont de type `T1` et les valeurs de type `T2`
`+size(): int`
renvoie le nombre d'éléments de la table de hachage
`+containsKey(cle: T1): boolean`
renvoie `true` si la table contient une entrée pour cette `cle`, et `false` sinon
`+put(cle: T1, valeur: T2): T2`
ajoute la `valeur` dans la table, associée à sa `cle`
`+remove(cle: T1): T2`
supprime l'entrée correspondant à cette `cle` de la table de hachage

▶ `java.util.LinkedList<T>`
`LinkedList<T>()`
instanciation d'une file d'éléments de type `T`
`+size(): int`
renvoie le nombre d'éléments de la file
`+addLast(T t): void`
ajoute `t` en fin de file
`+remove(T t): boolean`
supprime la première occurrence de `t` dans la file
`+poll(): T`
renvoie et supprime l'élément en tête de la file

▶ `System`
`+currentTimeMillis(): long`
renvoie le temps actuel (en millisecondes depuis le 01/01/1970)

▶ `Random`
`Random()`
instanciation d'un générateur de nombres pseudo-aléatoires
`+nextInt(max: int): int`
renvoie un entier pseudo-aléatoire compris entre 0 inclus et `max` exclus