Rapport de TP

CSR

Alan Huitel
Gwendal Denoual

Partie A - Synchronisation

1) Structure du supermarché

La classe Supermarche initialise tous les éléments du supermarché dans la méthode static d'exécution main().
Pour garder une **référence sur tous les objets partagés**, on utilise un objet **StructureSupermarche**.

2) Énumérations

Les états d'un client sont regroupés dans l'enum **EtatClient**.
Les produits disponibles dans le magasins sont regroupés dans l'enum **Produit**. Chaque rayon aura un attribut de type Produit pour indiquer quel est le produit qu'il propose. Le marqueur de client suivant attendu à la caisse est également présent dans l'enum Produit.

3) Threads

Les classes **ChefRayon**, **Client** et **EmployeCaisse** sont les threads utilisant les ressources partagées de l'application Supermarche.
La liste de courses de chaque client est générée aléatoirement de la manière suivante:
Un HasMap<Produit, Integer> est rempli avec tous les produits du magasins et une quantité choisie aléatoirement dans l'intervalle d'entiers [0; 10].

4) Ressources partagées

TasDeChariots:

Classe représentant le tas de chariots à l'entrée du supermarché.
Pour résoudre le problème de capacité limitée, nous utilisons un sémaphore initialisé au nombre de chariots du magasin.
Chaque client viendra prendre un chariot utilisera la méthode acquire() du sémaphore et il utilisera release() pour rendre un chariot. Un client appelant la méthode acquire() alors que le compteur du sémaphore est à 0 devra attendre qu'un autre client ait appelé la méthode release().
Après avoir reposé son chariot, le client est **retiré de la liste de clients** du supermarché.

Rayon:

Classe représentant un rayon du supermarché.
Nous rencontrons un problème de type **producteur/consommateur**: le chef de rayon remplit le stock du rayon alors que les clients le consomment.
Nous utilisons des **moniteurs** pour gérer ce problème.
Les actions "*prendre un produit du stock du rayon*" et "*ajouter un produit au stock du rayon*" doivent ainsi être **atomiques**, ce sont les méthodes associées qui seront **synchronized**.

Ces actions sont deux sous-problèmes:

- "Prendre un produit du stock du rayon" doit être encadré par un **while(condition sur quantité du stock) wait() et un notifyAll()** car un client ne doit pas quitter le rayon tant qu'il n'a pas rempli son chariot de la quantité de produit indiquée sur sa liste de courses. *Note: nous utilisons des moniteurs pour d'autres problèmes de synchronisation, nous ne pouvons donc pas juste utiliser une condition if et notify()***.**

- Pour "Ajouter un produit au stock du rayon", l'état du stock n'est pas bloquant. Si le stock est plein, le chef de rayons passe au rayon suivant.

Des méthodes supplémentaires utilisent ces méthodes synchronized pour répondre au besoin des clients et pour gérer le travail du chef de rayons.

Caisse:

Classe gérant la caisse du magasin. Elle résout trois problèmes de synchronisation:

- Problème d'**exclusion mutuelle** (passage d'un seul client a la fois en caisse):

Problème géré avec un **sémaphore initialisé à 1**.

- Problème de **rendez-vous** entre un client et un caissier:

Problème géré avec **deux sémaphores initialisés à 0**.

- Problème de **producteur/consommateur** (client met des produits sur le tapis alors que le caissier les enlève):

Problème géré avec des **moniteurs**. Les actions atomiques sont "poser un produit sur le tapis" et "retirer un produit du tapis".

Partie B - API REST

1) Application

La classe SupermarcheRestApp possède un attribut StructureSupermarche (référence vers les ressources partagées de la simulation).
Elle initialise le routeur de l'API REST, en redirigeant les requêtes de la manière suivante:

- Les requêtes sur les uri "/supermarche/clients" et "/supermarche/clients/" sont redirigées vers la ressource de serveur **ResourceClients**.

- Les requêtes sur les uri "/supermarche/clients/{id]" et "/supermarche/clients/{id}/" sont redirigées vers la ressource de serveur **ResourceClient**.

- Les requêtes sur les uri "/supermarche/clients/stock" et "/supermarche/clients/stock/" sont redirigées vers la ressource de serveur **ResourceStocks**.

L'application gère la présence ou non du caractère '/' à la fin de l'uri des requêtes. Elle s'occupe de l'ajouter si besoin quand elle renvoie un objet JSON avec un uri dedans.

2) Parser JSON

La classe JsonParser permet d'effectuer le parsing des clients, des listes de clients ou des listes de rayons en objets JSON.

3) Ressources de serveur

a) ResourceClient

Exemple de requête avec méthode GET:
curl localhost:5000/supermarche/clients/2
La méthode avec le tag @Get() va parser le client présent dans la liste de clients du supermarché dont l'id vaut 2. S'il n'existe pas, le code de retour vaudra 404 et les données de retour seront un objet JSON vide. Sinon, le code de retour vaudra 200.

Exemple de requête avec méthode POST:
curl -X POST localhost:5000/supermarche/clients/2
Pas de gestion de la méthode POST, donc le code de retour vaut 405 - Method Not Allowed.

b) ResourceClients

Exemple de requête avec méthode GET:
curl localhost:5000/supermarche/clients/
La méthode avec le tag @Get() va parser la liste de clients du supermarché et retourner l'objet JSON ainsi créé. Même si la liste est vide, l'uri est valide et le code de retour vaut 200.

Exemple de requête avec méthode POST:
curl -X POST localhost:5000/supermarche/clients
La méthode avec le tag @Post() va instancier et lancer un nouveau thread client puis l'ajouter à la liste de clients du supermarché. Cette méthode retourne une représentation JSON du client ainsi créé. Le code de retour vaut 200.

Script bash pour créer plusieurs clients:

```
#!/usr/bin/env bash

for i in {1..8}
do
  curl -X POST localhost:5000/supermarche/clients
done
```

c) ResourceStocks

Exemple de requête avec méthode GET:
curl localhost:5000/supermarche/stock
La méthode avec le tag @Get() va parser la liste de rayons du supermarché avec les stocks associés et retourner l'objet JSON ainsi créé. Même si la liste est vide, l'uri est valide et le code de retour vaut 200.