Épreuve finale - volet 1 : projet « Rush au resto »

1. Contexte

Dans ce premier volet de l'épreuve finale, on doit développer une application Java (console) qui simule le service dans un petit restaurant pendant un "rush" de midi.

Des clients arrivent, passent commande, attendent leurs plats 🍕, sont servis 😋... ou partent fâchés 😡 si l'attente est trop longue.

Le programme doit :

- lire une **séquence d'actions** à partir d'un fichier texte (scénario).
- exécuter ces actions en manipulant des **structures de données** adaptées.
- utiliser des **fils d'exécution (threads)** :
  - utiliser au moins **un thread** pour simuler le travail du cuisinier.
- enregistrer **toute la sortie** (logs) dans un fichier de sortie.
- afficher des informations **compactes** à l'aide de quelques émoji.

Date de remise : date indiquée sur **Léa**

Type : **Travail individuel**

2. Objectifs d'apprentissage évalués

Cette épreuve permet d'évaluer :

1. **Manipulation efficace des structures de données**
   - Listes, ensembles, tables de hachage, etc.
2. **Utilisation judicieuse des fils d'exécution**
   - Simulation de tâches concurrentes.
3. **Choix approprié des instructions et des algorithmes**

- Parcours et mise à jour des structures utilisées et calcul de statistiques.

4. **Organisation logique des instructions et lisibilité du code**
   - Respect et complétion de l'architecture fournie (packages, classes, responsabilités).
   - Programmation orientée objet.
   - Code clair : noms significatifs, méthodes de taille raisonnable, duplication limitée, utilisation des classes utilitaires (`Constantes`, `Format`, etc.).
5. **Repérage des erreurs et fonctionnement correct**
   - Programme robuste, manipulation de fichiers, gestion de cas particuliers.

3. Description générale de l'application

On doit modéliser un **restaurant** pendant un service :

- des clients arrivent, avec un certain niveau de patience (en minutes simulées) ;
- chaque client passe une commande composée d'un ou plusieurs plats ;
- le cuisinier prépare les commandes ;
- les clients sont soit servis (😋), soit quittent le restaurant fâchés (😡) si leur patience tombe à 0.

4. Structure du projet et classes

Un projet de démarrage vous a été fourni sur Github, dans ce projet les classes sont déjà créées et placées dans leurs packages. Des classes sont implémentées partiellement ou manquent des parties à compléter (à coder). Des mentions `TODO` sont utilisées pour les repérer.

L'application principale (`App`) :

1. Lit le fichier d'actions (ex. `scenario.txt`).
2. Pour chaque ligne, crée un objet `Action` (déjà fourni).
3. Transmet l'`Action` à un objet `Restaurant` via `executerAction(Action action)`.
4. À la fin du fichier :

- le dernier `AFFICHER_STATS` doit afficher des stats **cohérentes**,
- l'action `QUITTER` termine proprement la simulation (arrêt du thread cuisinier, fermeture des ressources).

5. Structures de données exigées

L'application doit obligatoirement utiliser :

- Une structure appropriée pour le **catalogue de plats** facilitant la recherche par la valeur de l'enum `MenuPlat` correspondante.
- Une structure pour les **clients présents.**
- Une structure pour les **commandes en file d'attente** .
- Une structure pour les **statistiques de vente.**
- Autres au besoin.

6. Fils d'exécution (threads) exigés

L'application doit utiliser au minimum le thread suivant :

Thread Cuisinier

- prend des commandes dans la file d'attente ;
- déclenche le début de préparation (`commande.demarrerPreparation()`) ;
- ajoute la commande dans la liste des commandes en préparation.

La classe `Cuisinier` doit implémenter `Runnable` (ou étendre `Thread`).

Son rôle :

- surveiller la **file de commandes** ;
- lorsqu'une commande est disponible, la "prendre en charge" (début de préparation) ;
- appeler les méthodes prévues du `Restaurant` (par ex. `retirerProchaineCommande()`).

Important : la progression du temps de préparation (diminution de `tempsRestant`, détection d'une commande terminée, mise à jour des stats) est gérée par la méthode `tick()` dans `Restaurant`, appelée via l'action `AVANCER_TEMPS`.

Thread principal :

- lit le fichier d'actions ;
- exécute les actions dans l'ordre ;
- demande au `Restaurant` d'afficher l'état et les stats ;
- déclenche l'arrêt propre des threads à la commande `QUITTER`.

Attention : l'accès concurrent aux objets et collections partagées doit être protégé (ex. méthodes `synchronized` ou autre approche simple vue en cours).

7. Fichier d'actions (entrée)

Aucune saisie clavier (`System.in`) ne doit être utilisée.

Toutes les actions viennent d'un **fichier texte**.

7.1 Format général

- Une action par ligne.
- Selon l'action, il peut y avoir 0, 2 ou 3 paramètres.
- Format à 3 paramètres :

```
NOM_ACTION;param1;param2;param3
```

- Exemple : `scenario_1.txt`

```
DEMARRER_SERVICE;10;1
AJOUTER_CLIENT;1;Alice;5
PASSER_COMMANDE;1;X;PIZZA
AVANCER_TEMPS;1
AFFICHER_ETAT
AVANCER_TEMPS;3
AFFICHER_STATS
QUITTER
```

- Les lignes **vides** et celles commençant par `#` sont ignorées.

7.2 Actions minimales à gérer

Chaque ligne du fichier d'entrée suit un format fixe :

`NOM_ACTION;param1;param2;param3`

Les actions minimales à supporter :

1. `DEMARRER_SERVICE;duree;nbCuisiniers`
   - `duree` : durée maximale du service en minutes simulées (ex. 10).
   - `nbCuisiniers` : **pour cette épreuve, on exigera 1** (les autres valeurs peuvent être implémentées en bonus +10%).
   - Effet attendu :
     - création d'un objet `Restaurant` initialisé,
     - démarrage d'un **thread cuisinier**.
2. `AJOUTER_CLIENT;id;nom;patience`

   Exemple : `#1 Alice`, patience = 5, état = EN_ATTENTE.
   - `id` : identifiant numérique unique du client.
   - `nom` : nom du client (`String`).
   - `patience` : patience initiale (en minutes simulées).

   Effet :
     - ajout d'un objet `Client` dans la structure appropriée,
     - log d'un événement d'arrivée.
3. `PASSER_COMMANDE;idClient;codePlat`
   - `idClient` : identifiant du client existant.
   - `codePlat` : `PIZZA`, `BURGER` ou `FRITES`.
   - Exemple : `PASSER_COMMANDE;1;PIZZA`
   - Effet :
     - on récupère le `Client`,
     - on crée ou met à jour une `Commande` associée à ce client :

        - première commande → création d'un objet `Commande` avec une liste de plats,
        - autres plats → ajout à la même commande,
    - le `Cuisinier` pourra venir la chercher.
    - la commande est placée dans la **file d'attente des commandes**,
    - log de l'événement de commande.

4. `AVANCER_TEMPS;minutes`
    - Fait avancer la simulation de `minutes` **minutes simulées**.
    - Effet (par minute, via `tick()`):
        1. Avancer le temps de l'horloge.
        2. Permettre aux commandes en préparation de progresser (diminuer le temps restant).
        3. Marquer les commandes terminées comme prêtes, mettre à jour les stats.
        4. Diminuer la patience des clients en attente ; certains peuvent devenir 😡.
5. `AFFICHER_ETAT`
    - Affiche un **résumé compact** de l'état courant du restaurant :
        - nombre de clients,
        - nombre de servis,
        - nombre de fâchés,
        - taille de la file de commandes,
        - nombre de commandes en préparation.
    - Puis une ligne par client (emoji état + patience + plats commandés).
    - Utilise le format et les emojis fournis dans `Constantes` et `Formatter`.
6. `AFFICHER_STATS`
    - Affiche les **statistiques globales** :

    - nombre total de clients,
    - nombre de clients servis 😋,
    - nombre de clients partis fâchés 😡,
    - chiffre d'affaires total,
    - nombre de plats vendus par type (PIZZA, BURGER, FRITES).
- **Important :** plusieurs `AFFICHER_STATS` peuvent apparaître dans le scénario.

  Pour l'évaluation, **on ne tiendra compte que du dernier `AFFICHER_STATS`, placé avant `QUITTER`**.

7. `QUITTER`
    - Demande l'arrêt propre de la simulation :
        - arrêt du service,
        - arrêt du thread cuisinier (sortie propre),
        - éventuel `join` sur ce thread,
        - fin du programme.

Pour les actions qui n'ont pas de paramètres (ex. `AFFICHER_ETAT`, `AFFICHER_STATS`, `QUITTER`),

la ligne contient uniquement le nom de l'action.

8. Format d'affichage et émoji

Toute la sortie (ce qui serait normalement affiché à l'écran) doit être écrite dans un **fichier résultat**.

On veut un affichage conforme au format demandé.

8.1 Les émoji à utiliser

Dans le fichier `Constantes.java` vous allez trouver les différentes constantes ainsi que celles utilisées pour les émojis tel que :

- 👥 : nombre de clients présents
- 😋 : clients servis

- 😡 : clients partis fâchés
- 📥: commandes en file
- 🍳 : commandes en préparation
- 📊 : statistiques
- `t` : temps simulé (en minutes)

8.2 Ligne de résumé

À chaque `AFFICHER_ETAT`, on affiche une ligne de la forme :

```
[t=3] 👥2 😋1 😡0 📥1 🍳1
```

Signification :

- `t=3` : temps simulé = 3 min
- `👥2` : 2 clients présents
- `😋1` : 1 client servi
- `😡0` : 0 client parti fâché
- `📥1` : 1 commande en file d'attente
- `🍳1` : 1 commande en préparation

8.3 Liste des clients

Après la ligne de résumé, on affiche une ligne par client, par exemple :

```
#1 Alice 🙂 (pat=4, 🍕)
#2 Bob 😐 (pat=2, 🍔 🍟)
```

- `#1 Alice` : id + nom
- 🙂 / 😐 / 😋 / 😡 : état du client (en attente, servi, fâché, etc.)
- `pat=4` : patience restante (minutes)
- `🍕` ou `🍔 🍟` : plats commandés (codes ou émoji)

Menu des plats

Le restaurant propose **exactement trois plats** :

Code	Nom affiché	Temps de préparation (min simulées)	Prix
PIZZA	Pizza fromage	3	14.99 $
BURGER	Burger classique	2	11.49 $
FRITES	Frites	1	3.99 $

8.4 Statistiques (`AFFICHER_STATS`)

Format exigé :

```
📈 AFFICHER_STATS
[t=5] 📊 Stats
   Clients totaux : 3
   😋 Servis       : 1
   😡 Fâchés       : 2
   💰 Chiffre d'affaires : 29.98 $
   Plats vendus :
      🍕 PIZZA : 1
      🍔 BURGER : 1
      🍟 FRITES : 1
```

Données à afficher :

- nombre total de clients passés ;
- nombre de clients servis ;
- nombre de clients partis fâchés ;
- chiffre d'affaires (somme des commandes réellement servies) ;
- au moins une statistique par plat (par ex. nombre de ventes par plat).

8.5 Logs d'événements

Lors d'événements importants, on affiche une ligne courte, par ex. :

```
[📥 t=1] Cmd #1 (Alice) → 🍕
[🍳 t=1] Cmd #1 commence (3 min)
[✅ t=3] Cmd #1 terminée → Alice 😋
[😡 t=5] Bob part fâché (pat=0)
```

Format : `[EMOJI_ORIGINE t=...] message`.

L'événement qui log l'arrivée d'un client est un plus :

```
[🚪 t=1] 🧍 Client #1 "Alice" (pat=5)
```

Exemple d'exécution pour le scenario:

```
DEMARRER_SERVICE;10;1
AJOUTER_CLIENT;1;Alice;5
PASSER_COMMANDE;1;PIZZA
AVANCER_TEMPS;1
AFFICHER_ETAT
AVANCER_TEMPS;3
AFFICHER_ETAT
AFFICHER_STATS
QUITTER
```

Exemple du résultat attendu :

```
========== RUSH AU RESTO ==========
🍽Lecture des actions : data/scenario_clients.txt

[⏱] Service = 10 min, 👨‍🍳 = 1
[📥 t=0] Cmd #1 (Alice) → 🍕
⏩ AVANCER_TEMPS;1
[🍳 t=1] Cmd #1 commence (3 min)
[t=1] 👥1 😋0 😡0 📥0 🍳1
   #1 Alice 🙂 (pat=4, 🍕 )
⏩ AVANCER_TEMPS;3
```

```
[✅ t=3] Cmd #1 terminée → Alice 😋
[t=4] 👥1 😋1 😡0 📥0 🍳0
   #1 Alice 😋 (pat=3, 🍕 )
📈 AFFICHER_STATS
[t=4] 📊 Stats
   Clients totaux : 1
   😋 Servis       : 1
   😡 Fâchés       : 0
   💰 Chiffre d'affaires : 14,99 $
   Plats vendus :
      🍟 FRITES : 0
      🍔 BURGER : 0
      🍕 PIZZA : 1
==================================
```

9. Contraintes techniques

- Projet Java : **Maven**, archétype `maven-archetype-quickstart`.
- Langage : Java (version 21 ou plus).
- Application console (pas d'interface graphique).
- Pas de bibliothèques externes non standard.
- On doit recycler et compléter les classes fournies (ne pas commencer à zéro).
- Respect des conventions de codage Java (noms de classes, méthodes, etc.).
- Code structuré en plusieurs classes (pas tout dans `App` ou dans `main`).
- Les classes doivent être organisées dans les packages :
  - `model` pour les entités,
  - `sim` pour `Restaurant`,
  - `sim.thread` pour les threads,
  - `io` pour actions
  - `utils` pour les constantes, logs et formatage.

- Tout ce qui est affiché (logs, résumés, statistiques) doit passer par la classe de journalisation (`Logger`).

10. Légende des émoji et format à respecter

10.1 Légende des émoji

Vous devez utiliser les émoji et libellés suivants exactement comme indiqué :

Élément	Affichage à utiliser
Résumé temps	`t=3` (par exemple pour le temps simulé 3)
Clients présents	`👥` (ex. `👥2`)
Clients servis	`😋` (ex. `😋1`)
Clients partis fâchés	`😡` (ex. `😡0`)
Commandes en file	`📥` (ex. `📥1`)
Commandes en préparation	`🍳` (ex. `🍳1`)
Statistiques	`📊 Stats`
Affichage stats (titre)	`Plats vendus :`
Début scénario	`========== RUSH AU RESTO ==========`
Ligne lecture scénario	`🍽 Lecture des actions : ...`
Lancement service	`[⏱] Service = X min, 👨‍🍳 = Y`
Avancement temps	`⏩ AVANCER_TEMPS;N`
Stats	`📈 AFFICHER_STATS` (juste avant les stats)
Fin scénario	`==========`

Attention : Toutes les constantes devraient se trouver dans la classe `Constantes`.

10.2 Ligne de résumé (obligatoire)

À chaque `AFFICHER_ETAT`, vous devez produire **exactement** une ligne de résumé au format :

> [t=<temps>] 👥<nbClients> 😋<nbServis> 😡<nbFaches> 📥<nbEnFile> 🍳<nbEnPreparation>

Exemple :

```
[t=3] 👥2 😋1 😡0 📥1 🍳1
```

Ordre et symboles **à ne pas modifier** :

1. `t=...`
2. 👥
3. 😋
4. 😡
5. 📥
6. 🍳

10.3 Ligne client

Pour chaque client, utilisez une ligne du type :

```
    #<id> <Nom> <emojiEtat> (pat=<patience>, <plats>)
```

Exemple :

```
    #1 Alice 🙂 (pat=4, 🍕)
```

11. Entrée / sortie et exécution (Maven)

Votre projet doit être exécutable en ligne de commande avec **deux paramètres** :

```
mvn clean package
mvn exec:java -Dexec.mainClass="votre.package.App" \
              -Dexec.args="scenario_simple.txt sortie_simple.txt"
```

- 1er argument : **fichier d'actions** (ex. `scenario_simple.txt`)
- 2e argument : **fichier résultat** (ex. `sortie_simple.txt`)

Le programme doit :

- lire **uniquement** les actions à partir du fichier (pas de `Scanner` sur le clavier) ;
- écrire **toute la sortie** (logs, résumés `[t=...]`, états, stats) dans le fichier résultat.

Une classe `Logger` est utilisée pour centraliser l'écriture vers le fichier (et éventuellement vers la console).

Livrables

Vous devez remettre :

1. **Le projet Maven complet**, incluant :
   - `pom.xml`
   - tout le contenu de `src/main/java/...`
2. **Au moins un fichier de scénario** (par ex. `scenario_simple.txt`) montrant un cas de test personnel (dans le dossier data).
3. **Un fichier de sortie généré** à partir d'un scénario fourni par l'enseignant·e (dans le dossier data)..

   (par ex. `scenario_simple.txt` → `sortie_simple.txt`).

L'enseignant(e) doit pouvoir exécuter les commandes :

```
mvn clean package
mvn exec:java -Dexec.args="data/scenario_1.txt data/sortie_1.txt"
```

et obtenir un fichier `sortie_1.txt` cohérent, sans modifier le code.

4. Votre fichier exécutable `.jar` devrait commencer par votre numéro de DA.
4. Votre code devrait être **versionné sur Github** :
   a. Ajoutez un fichier README et mettez le lien vers votre repo.
   b. Une invitation devrait être envoyée à sara.boumehraz@cegepmv.ca si votre repository est privé.

Grille d’évaluation

1. Structures de données (25 %)

- Choix pertinents (`List`, `Map`, `Set`,...) : 10 %
- Utilisation correcte (ajout, suppression, parcours) : 10 %
- Structures alignées avec les besoins de l’application : 5 %

2. Fils d’exécution (25 %)

- Présence de threads (au moins le principal (main) et cuisinier) : 10 %
- Rôles clairs et correctement implémentés : 10 %
- Gestion minimale de la synchronisation (pas de bugs évidents) : 5 %

3. Algorithmes et logique (20 %)

- Logique correcte pour la gestion du temps, des patiences et des états : 10 %
- Statistiques calculées correctement : 5 %
- Utilisation adéquate de tri / filtrage / parcours : 5 %

4. Organisation et lisibilité du code (15 %)

- Architecture orientée objet cohérente : 10 %
- Qualité du code (noms, commentaires, clarté) : 5 %

5. Fonctionnement et robustesse (15 %)

- Exécution sans erreur avec les scénarios fournis : 10 %
- Gestion des cas d’erreur simples (fichier manquant, ligne d’action invalide, etc.) : 5 %

Attention : Un code qui ne compile pas ou qui ressemble au projet de démarrage fourni (sans aucune fonctionnalité implémentée) aura une note de 0.