Environnement Client/Serveur
M1 Miage Classique
Nom :
Prénom:

lundi 8 octobre 2018

TP noté

Ce TP est ramassé et noté
Durée 2h30

Consignes : Les documents sont autorisés (ainsi que l'accès à Internet et aux corrigés des TPs précédents). Les réponses aux questions marquées d'un ⋆ doivent être écrites dans les cadres prévus à cet effet. Vous devez rendre le sujet (contenant vos réponses) en fin de séance, après avoir indiqué **lisiblement** votre nom et prénom. Le code doit être rendu via un formulaire de dépôt disponible ici :

`https://www.lri.fr/~kn/rendu_ecs.php`

Vous devez générer l'archive de rendu depuis Eclipse :

- `Export`
- `General/Archive File`
- Le nom de l'archive **doit se terminer par l'extension** « `.tar.gz` ». Vérifiez bien que les options « `Save in tar format` » et « `Create directory structure for files` » sont sélectionnées.

Vous pouvez utiliser le formulaire autant de fois que vous le souhaitez et toutes les versions soumises seront conservées. Nous corrigerons la version la plus récente. Le site de soumission se verrouille de manière automatique à l'heure indiquée sur la page (fin du TP + 5 minutes). Passé ce délai, il sera **impossible de soumettre**. Il est **vivement conseillé de ne pas soumettre à la dernière minute pour éviter tout problème**.

Notation : Les questions rédigées sont sur 5 points. La qualité du code (lisibilité et fait que le code compile) seront sur 5 points. Les 10 points restants seront attribués par des tests automatiques sur votre client et votre serveur. **Il est interdit de renommer les packages ou les classes fournis**.

Banque

L'ensemble des comptes est modélisé par la classe `Bank` (fichier `Bank.java`). Un objet de type `Bank` utilise une `Map<String, Integer>` pour associer à des noms (de compte) des montants (valeur entière, en euros). Outre le constructeur, la classe dispose de quatre méthodes :

`Integer getAccountValue(String account)` : renvoie le montant stocké sur le compte ou `null` si un tel compte n'existe pas.

`boolean createAccount(String account, String amount)` : si un compte de ce nom là existe déjà ou que `amount` est strictement négatif, renvoie `false`. Sinon ajoute une entrée correspondante dans la `map` interne.

`boolean changeAmount(String account, String amount)` : si un compte de ce nom là n'existe pas, renvoie `false`. La somme `amount` peut être positive (dépôt) ou négative (retrait). Si `amount` est négatif et que le compte ne dispose pas assez d'argent, la méthode renvoie `false`. Sinon la valeur de `amount` est ajouté à la valeur du compte et la méthode renvoie `true`.

`Map<String, Integer> getAll()` : renvoie *une copie* de la `map` interne.

Serveur et Client de compte bancaire

Pseudo code du serveur :

- le serveur initialise une socket TCP sur le port 12345 et se place en attente de connexion sur cette socket, et initialise un objet de la classe `Bank` décrit dans la section précédante.
- sur réception d'une connexion, le serveur crée un nouveau *thread de gestion de connexion*. Ce *thread* récupère les flux d'entrée et de sortie associés à la socket le reliant au client ainsi que l'objet global de type `Bank` et se met à suivre le protocole de l'application.
- une fois le *thread* crée, et s'exécutant en parallèle, le serveur se remet à l'écoute d'une connexion sur la socket.
- si aucune connexion n'a lieu dans un délais de 1000ms, le serveur vérifie si l'utilisateur a écrit sur l'entrée standard. Si c'est le cas, le serveur se termine, sinon il retourne en attente de connexion.

Pseudo code du client :

- Le client initialise une socket TCP et la connecte au port 12345 de l'adresse du serveur (`localhost` par défaut). Le client récupère les flux d'entrée et de sortie associés à la socket et le reliant au serveur.
- Une fois la connexion établie, le client attend que l'utilisateur écrive une commande sur l'entrée standard, l'envoie au serveur, et reçoit la réponse du serveur.
- Le client termine quand l'utilisateur envoie la commande de déconnexion au serveur.

Protocole de l'application : Tous les messages échangés entre le client et le serveur se terminent par un retour à la ligne (« `\n` »). Une méthode `sendMsg()` est fournie dans le client et le serveur pour envoyer les messages. Dans la suite les caractères espaces sont affichés explicitement « ␣ ».

- Une fois la connexion TCP établie, le client envoie la chaîne de caractères « `CONNECT\n` » au serveur.
- Le serveur renvoie la réponse « `CONNECT␣OK\n` » (un seul espace entre les deux mots).
- le client peut envoyer au serveur la commande « `CREATE:`xxx`:`yyy`\n` » où xxx représente un nom de compte et yyy représente une somme. Le nom ne peut contenir ni le caractère « : » ni le caractère « `\n` ». La quantité doit être un entier positif ou nul. Si l'une de ces condition n'est pas remplie, la commande est considérée comme **incorrecte**. C'est le **serveur** qui vérifie ces conditions, vous ne devez pas mettre de code de vérification dans le client.
  Le serveur rajoute à la banque un compte du montant approprié (méthode `createAccount` de la classe `Bank`) et répond au client « `OK\n` » (si `createAccount` renvoie `true`) ou « `ERROR\n` » (si `createAccount` renvoie `false`).
- le client peut envoyer au serveur la commande « `CHANGE:`xxx`:`yyy`\n` » où xxx représente un nom de compte et yyy représente une somme. Le nom ne peut contenir le caractère « : » ni le caractère « `\n` ». La quantité doit être un entier positif ou négatif. Si l'une de ces condition n'est pas remplie, la commande est considérée comme **incorrecte**. C'est le **serveur** qui vérifie ces conditions, vous ne devez pas mettre de code de vérification dans le client.
  Le serveur rajoute au compte xxx le montant yyy (méthode `changeAmount` de la classe `Bank`) et répond au client « `OK\n` » (si `changeAmount` renvoie `true`) ou « `ERROR\n` » (si `changeAmount` renvoie `false`).
- le client peut envoyer au serveur la commande « `GET:`xxx`\n` ». Si un compte xxx est présent dans la banque, le serveur répond « `VALUE:`yyy`\n` ». Si le compte xxx n'existe pas, le serveur répond « `ERROR\n` ».
- le client peut envoyer au serveur la commande « `AUDIT\n` ». Le serveur répond « lll`\n` », où lll est le nombre d'entrées de comptes en banque, suivi de chaque entrée, au format : « xxx`:`yyy`\n` » où xxx est un nom de compte et yyy la somme associée (noter l'absence d'espace autour du «:»).
- le client peut envoyer au serveur la commande « `DISCONNECT\n` » et le serveur renvoie en réponse le message « `BYE\n` » puis coupe sa connexion avec le client.
- sur réception du message « `BYE\n` », le client se termine.
- pour tout envoi **incorrect** du client (c'est à dire toute commande qui n'est pas strictement prévue par le protocole) le serveur renvoie « `INVALID:`msg`\n` » où msg est le message envoyé par le client. Le serveur ne doit pas couper la connexion.

Il convient de faire attention aux messages du protocole. Toutes les commandes sont en majuscules, et utilise au plus un caractère « : » pour séparer leurs arguments. La banque est sensible à la casse, donc "`Toto`" et "`TOTO`" ne représentent pas le même nom de compte.

Les tests automatiques vérifient la **sortie standard (System.out) du client.** Vous devrez afficher sur cette sortie les messages envoyés au serveur et les messages reçus. La console du client doit ressembler à la sortie suivante (dans laquelle est les chaînes soulignées sont celles entrées par l'utilisateur et les autre les réponses affichées par le programme client) :

```
SENT:CONNECT
RECEIVED:CONNECT␣OK
CREATE:TOTO:1000
SENT:CREATE:TOTO:1000
RECEIVED:OK
CREATE:TITI:1000
SENT:CREATE:TITI:1000
RECEIVED:OK
CHANGE:TOTO:200
SENT:CHANGE:TOTO:200
RECEIVED:OK
CHANGE:TITI:-500
SENT:CHANGE:TITI:-500
RECEIVED:OK
CHANGE:TITI:-1000
SENT:CHANGE:TITI:-1000
RECEIVED:ERROR
```

```
GET:TITI
SENT:GET:TITI
RECEIVED:VALUE:500
AUDIT
SENT:AUDIT
RECEIVED:2
TITI:500
TOTO:1200
STUPID
SENT:STUPID
RECEIVED:INVALID␣STUPID
DISCONNECT
SENT:DISCONNECT
RECEIVED:BYE
```

Notez que vous pouvez utiliser la sortie d'erreur (`System.err`) comme bon vous semble (par exemple pour vous aider à débugger votre code).

Organisation du code : le code du serveur est réparti en trois classes. Une classe principale, `TCPBankServer` qui crée le socket serveur et un stock vide et attend les connexions sur le port TCP 12345, une classe `TCPBankConnection` qui étend la classe `Thread` (ce qui permet à son code de s'exécuter en parallèle du reste de l'application) et une classe `Bank` qui modélise la banque. Le code du client ne possède qu'une seule classe, `TCPBankClient`.

Questions :

1. Compléter le code du client et du serveur aux endroits indiqués.
   — classe `Bank`, méthode `getAccountValue`, `createAccount`, `changeAmount`
   — classe `TCPBankConnection`, constructeur et méthode `run()`
   — classe `TCPBankServer`, méthode `start()` uniquement.
   — classe `TCPBankClient`, méthode `start()` et `mainLoop()`
   La méthode `split()` de la classe `String` peut être utiles lors de la manipulation des chaînes de caractères. On rappelle aussi qu'en Java, les comparaisons de chaînes se font toujours :
   — soit à l'aide de la construction `switch () { case }`
   — soit à l'aide de la méthode `.equals()`
   mais jamais avec l'opérateur « == ». Il est possible d'itérer sur toutes les entrées d'une `Map<String,Integer>` de la manière suivante :

```
//h est une Map<String,Integer>
for (Map.Entry<String, Integer> e : h.entrySet())
{
   String k = e.getKey();
   Integer v = e.getValue();
   // faire quelque chose avec k et v
}
```

   Enfin, on peut convertir une chaîne de caractères `s` en `Integer` Java avec la méthode `Integer.parseInt(s)` (cette dernière peut lever une exception `NumberFormatException` si la chaîne ne représente pas un entier). Il est conseillé de faire les choses progressivement. Ajouter d'abord la gestion de connexion dans le serveur, puis dans le client et vérifier qu'ils compilent. Puis ajouter la gestion du `DISCONNECT` (client puis serveur), puis la gestion du `CREATE/CHANGE` (client et serveur) puis la gestion du `GET` et enfin la

gestion de `AUDIT`. Faire les choses ainsi et s'assurer que le code compile entre chaque étape vous permet de vous assurer des points sur les tests automatiques correspondants à ces fonctionnalités.

2. ⋆ (2.5 points) Pourquoi la méthode `changeAmount` de la classe `Bank` doit-elle être `synchronized`? Donner un exemple précis de situation ou l'absence de `synchronized` peut causer une erreur.
3. ⋆ (2.5 points) Pourquoi en plus d'être `synchronized` la méthode `getAll()` de la classe `Bank` doit-elle effectuer une copie de la `Map`? Donner un exemple de situation où l'absence de copie pourrait provoquer une erreur.

Réponses :