Partiel JD durée 3h (documents autorisés)

Gestion d'une banque

1 Solution locale

L'objet de cet exercice est de créer une application java qui implante quelques services de base d'une gestion bancaire.

1. Créer deux classes principales `BankLocal` et `BankLocalServer` qui vont agir respectivement comme client et comme serveur (en local, donc "communication" par appel direct de méthodes). La classe `BankLocal` sera déclarée publique.
2. Créer une classe `Account`, représentant un compte bancaire, munie des trois champs `String name`, le nom du propriétaire du compte, `String password`, le mot de passe associé au compte et `int balance`, le solde du compte. Cette classe ne contiendra qu'un constructeur et aucune autre méthode. Ce constructeur ne sera chargé que d'enregistrer le nom et le mot de passe ainsi que d'initialiser le solde à zéro.
3. L'ensemble des comptes bancaires est stocké dans une table de hachage (`HashMap`) réalisant l'association `nom` du titulaire d'un compte / objet de type `Account` (le compte, informatiquement parlant). Cette table sera le seul champ de `BankLocalServer`, nommé `allAccounts`.
4. Créer une sous-classe `BankingException` qui hérite de la classe `Exception` et qui ne définit qu'un seul constructeur (et aucune autre méthode) `BankingException(String)` qui appelle le constructeur de la classe mère. Cette exception servira pour des cas tels que "Solde insuffisant" ou "Mot de passe invalide".
5. Prévoir les opérations suivantes comme méthodes de `BankLocalServer` (ce sont les opérations réalisées au sein même de la banque) :
   - `public void openAccount(String name, String password) throws BankingException`.
     Le nom du titulaire `name` est donné en premier argument et son mot de passe `password` en deuxième argument. La méthode teste s'il existe déjà un compte ayant le même nom de titulaire, auquel cas elle lève une exception de type `BankingException` et dans le cas contraire crée le compte et l'enregistre dans la table `allAccounts`.
   - `public Account verify(String name, String password) throws BankingException`.
     Si le compte de nom `name` n'existe pas dans `allAccounts` ou si le mot de passe `password` est différent de celui correspondant à `name`, la méthode génère une exception de type `BankingException`. Sinon, elle renvoie une référence sur l'objet de type `Account` correspondant.
   - `public int closeAccount(String name, String password) throws BankingException`.
     Elle effectue une vérification à l'aide de `verify(...)`, retire le compte de la table `allAccounts`, met le solde à zéro et renvoie le montant disponible.
   - `public void deposit(String name, String password, int amount) throws BankingException`.
     Elle effectue une vérification à l'aide de `verify(...)` et teste si `amount` est positif (dans le cas contraire, elle génère une exception de type `BankingException`), elle incrémente le solde du montant `amount`.
   - `public int withdraw(String name, String password, int amount) throws BankingException`.
     Elle effectue une vérification à l'aide de `verify(...)`, teste si `amount` est positif, vérifie que le solde est suffisant pour le montant du retrait (dans le cas contraire, elle genère une exception de type `BankingException`), elle décrémente le solde du montant `amount` et renvoie le montant retiré.
   - `public int getBalance(String name, String password) throws BankingException`.
     Elle effectue une vérification à l'aide de `verify(...)`, puis renvoie le solde du compte.
   - La méthode `main(...)` se trouvera dans `BankLocal` et proposera à l'utilisateur un menu textuel afin qu'il puisse gérer son compte. Les entrées seront effectuées avec des appels à `readLine()` sur une référence créée comme suit :
     `BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));`
     Les sorties seront effectuées avec des `System.out.println(...)`.

2 Solution en Client/Serveur par Sockets

Il s'agit maintenant de développer une application de gestion bancaire simple en réseau, utilisant les sockets. Ce dernier mécanisme n'offrant que du transfert de flux, donc d'assez bas niveau, il est nécessaire de concevoir un protocole d'échanges de données (un *paquet*) permettant de différencier les opérations bancaires tout en acheminant les données pertinentes.

Paquets à échanger

Créer une classe `BankSocketPacket` représentant un paquet du protocole. Spécification de cette classe :

```
public class BankSocketPacket {
  String    name;       // nom du compte
  int       operation;  // operation bancaire
  String    password;   // mot de passe
  int       amount;     // solde
  /* Operations */
  public static final int OPEN
  public static final int CLOSE
  public static final int DEPOSIT
  public static final int WITHDRAW
  public static final int BALANCE
  public static final int QUIT
  /* Erreurs */
  public static final int TRANSOK // Operation correcte
  public static final int EXISTANT // Compte existant
  public static final int NONEXISTANT = -2; // Compte inexistant
  public static final int INVPASSWORD = -3; // Mot de passe invalide
  public static final int NEGBALANCE  = -4; // Solde insuffisant pour retrait
  /** Constructeur : initialisation des champs **/
  BankSocketPacket(String theName, int theOperation,
                       String thePassword, int theAmount) { ... }
  /** Constructeur par defaut : paquet "vide" **/
  BankSocketPacket() { ... }
  /** Empaquetage ou formation du paquet : BankSocketPacket -> String **/
  public String foldPacket() { ... }
  /** Depaquetage ou deballage du paquet : String -> BankSocketPacket  **/
  public void unfoldPacket(String foldedPacket) { ... }
```

Outre les champs `name`, `password` et `amount` déjà dans le premier exercice, le champ `operation` représente le type d'opération bancaire à réaliser. Les différents types d'opérations possibles sont listés en constantes (`public static final int`) strictement positives. Les erreurs de type bancaire pouvant survenir sont représentées par des constantes strictement négatives et une constante nulle (`TRANSOK`) signifie une transaction qui s'est bien passée.

Considérons un paquet ayant les champs suivants : `name` valant `"Diogene"`, `password` valant `"tonneau"` et `amount` valant `1`. Supposons que le compte a été ouvert et que l'on veuille déposer (`DEPOSIT`) un montant de `100` sur le compte. Pour transmettre le paquet, on le transforme en la chaîne de caractères formée en mettant bout à bout les champs, séparés par le caractère `:` (deux-points). Ceci donne pour l'exemple ci-dessus `"Diogene:3:tonneau:100#"`. Le deuxième champ est la valeur de la constante associée à l'opération `DEPOSIT` désirée, `3`. Le dernier caractère `#` est un séparateur de paquets. On représentera donc un paquet générique transmis comme suit :
`"name:OPERATION:password:amount#"`. Le champ `amount` aura une signification différente selon les opérations à réaliser (voir tableau ci-dessous). Il n'a pas de signification en cas de retour en erreur.

Opération	Signification du champ `amount`
OPEN	Aucune
CLOSE	Aucune
DEPOSIT	Montant à déposer
WITHDRAW	Montant à retirer
QUIT	Aucune

A part le constructeur, la classe comporte deux méthodes utilitaires : `foldPacket()` qui convertit l'objet appelant (de type `BankSocketPacket`) en une chaîne (de type `String`) qui sera transmise dans le réseau, et `unfoldPacket()` qui réalise l'opération inverse.

Ecrire le code des méthodes utilitaires `foldPacket()` et `unfoldPacket()`. On pourra se servir, pour `foldPacket()`, des méthodes `append()` de `StringBuffer`, `toString()` de `Integer`. On pourra se servir, pour `unfoldPacket()`, des méthodes `indexOf()` et `substring()` de la classe `String`, ou bien des méthodes `replaceAll()` de `String` et `useDelimiter()`, `next()` et `nextInt()` de `Scanner`.

Elaboration du serveur

Le schéma de fonctionnement est le suivant : le client envoie une requête au serveur, qui la traite et qui renvoie une réponse. Cette réponse peut correspondre à une erreur ou à une transaction qui s'est bien réalisée. Par exemple, supposons qu'un compte de nom `Diogene` et de mot de passe `tonneau` ait été créé. Supposons vouloir déposer un montant de `100` euros sur ce compte sur lequel il y a déja 150 euros.
Si le client envoie la chaîne `"Diogene:3:tonneau:100#"`, le serveur répondra `"Diogene:0:tonneau:250#"`.
Si on envoie `"Diogene:3:maison:100#"` (erreur de mot de passe), le serveur répondra `"Diogene:-3:maison:100#"`. Le deuxième champ `-3` est la constante `BankSocketServer.INVPASSWORD` représentant une erreur de mot de passe invalide. Le client affichera alors un message correspondant à l'erreur survenue.

Créer un serveur itératif, ne traitant qu'un seul client à la fois. Pour cela créer une classe `BankSocketServer` qui représente le serveur bancaire, dont les spécifications seront les suivantes :

```
public class BankServerSocket {
  public final static int   DEFAULT_PORT = 6789;
  Hashtable                 allAccounts = new Hashtable();
  static int                transCode;                    // code d'erreur
  public synchronized void openAccount(String name, String password) { ... }
  public Account verify(String name, String password) { ... }
  public synchronized int closeAccount(String name, String password) { ... }
  public void deposit(String name, String password, int money) { ... }
  public int withdraw(String name, String password, int amount) { ... }
  public int getBalance(String name, String password) { ... }
  void sendPacket(BankSocketPacket toSend, PrintStream out) { ... }
  BankSocketPacket receivePacket(BufferedReader in) {...}
  public static void main (String[] args) throws IOException { ... }
}
```

Le champ `transCode` code l'apparition d'une erreur. On rappelle à ce propos que les constantes de la classe `BankSocketPacket` sont statiques et doivent donc être appelées en plaçant le nom de la classe devant ; par exemple la constante `BankSocketPacket.TRANSOK` désigne une transaction qui s'est bien passée. Dans les différentes méthodes que l'on trouvait déja dans la solution locale, le traitement d'erreurs effectué par un traitement d'exception de type `BankingException` est remplacé par l'affectation de `transCode` au code de l'erreur correspondante.
Les deux seules méthodes nouvelles sont `sendPacket()` et `receivePacket()` pour l'envoi et la réception des paquets. Ainsi, une fois le paquet à envoyer créé par `new BankSocketPacket(...)`, la méthode `sendPacket()` est chargée des opérations suivantes :

- création à l'aide de la méthode `foldPacket()` d'une chaîne du type `"name:OPERATION:password:amount#"` ;
- envoi de cette chaîne au client via la méthode `println()` de la classe `PrintStream`.

De son coté, la méthode `receivePacket()` est chargée des opérations suivantes :

- Réception d'une requête via la méthode `readLine()`.
- Création d'un paquet "vide" (par le constructeur par défaut de `BankSocketPacket`).
- Récupération des informations par appel de la méthode `unfoldPacket()`.

Enfin, la méthode `main()`, outre les traitements classiques, effectuera les actions suivantes :

- Appel de `receivePacket()`.
- Branchement selon l'opération demandée selon un `switch`.
- Appel de la méthode correspondant à l'opération demandée (c'est-à-dire : `openAccount()`, `closeAccount()`, `deposit()`, `withdraw()`, `getBalance()` ou, dans le cas de sortie, des `close()` de sockets)
- Création d'un paquet avec les champs adéquats.
- Envoi du paquet au client.

Elaboration du client

Créer une classe `BankSocket` dont les spécifications sont les suivantes :

```
public class BankSocket {
    public static final int    DEFAULT_PORT = 6789;
```

```
    public static final String DEFAULT_HOST = "localhost";
    public String getName(BufferedReader in) { ... }
    public String getPassword(BufferedReader in) { ... }
    // Methode utilitaire de gestion d'erreur
    static void treatError(int errorNb) { ... }
    // Methode utilitaire d'envoi/reception de paquet
    BankSocketPacket sendReceive(BankSocketPacket toSend,
        DataInputStream sin, PrintStream sout, String message) { ... }
    public static void main(String[] args) { ... }
}
```

Les méthodes `getName()` et `getPassword()` sont des méthodes de saisie qui demandent respectivement à l'utilisateur d'entrer sur l'entrée standard (au clavier) un nom et un mot de passe ; la chaîne de caractère entrée est renvoyée. La méthode `treatError()` affiche un message sur la sortie standard correspondant au code d'erreur fourni en paramètre. La méthode `sendReceive()` effectue les actions suivantes :

- Emballage du paquet `toSend` par la méthode `foldPacket()` (1er paramètre).
- Envoi du paquet emballé par `println()` sur `sout` (2e paramètre, représentera le flux de sortie vers le serveur dans le code qui appelle la méthode).
- Réception d'une ligne par `readLine()` sur `sin` (3e paramètre, représentera le flux d'entrée en provenance du serveur dans le code qui appelle la méthode).
- Déballage de la ligne reçue par `unfoldPacket()` après création d'un paquet via le constructeur par défaut de `BankSocketPacket`.
- Test sur le code (champ operation) du paquet juste déballé : s'il est égal au code `BankSocketPacket.TRANSOK`, affichage du message message, sinon appel de la méthode `treatError()` avec pour argument ce code.
- Renvoi (`return()`) du paquet déballé.

Au sein de la méthode `main()`, outre les traitements usuels pour un programme client par sockets, le traitement du service comprend :

- Affichage des différentes opérations possibles et prise d'entrée (choix) de l'utilisateur
- Choix selon l'opération souhaitée. Puis, au moins :
- Prise des nom et mot de passe via `getName()` et `getPassword()` ;
- Création d'un paquet par le constructeur de `BankSocketPacket` en y mettant les noms, mot de passe et opération sélectionnés ;
- Appel de `sendReceive()` pour envoyer le paquet précédent et recevoir la réponse du serveur.