PCOO – TP 10

Exercice 1 – Création d'un serveur

Nous allons créer un serveur en utilisant des sockets sur IP. La classe `SocketServer` permet d'ouvrir un socket côté serveur en mode connecté. Par exemple, pour écouter sur le port 8088, il suffit d'instancier la classe `SocketServer`. Dans la méthode `main`, ajoutez la ligne suivante :

```
ServerSocket serverSocket = new ServerSocket(8088);
```

Ensuite, nous allons attendre la connexion d'un client au serveur. La méthode `Socket accept()` de la classe `ServerSocket` arrête l'exécution du programme jusqu'à la connexion d'un client au serveur. Cette méthode retourne une instance de la classe `Socket` qui représente la connexion (établie) entre le client et le serveur. Par conséquent, pour attendre un client, il suffit de rajouter la ligne suivante :

```
Socket socket = serverSocket.accept();
```

Il est ensuite possible d'envoyer des données au client via le flux de données retourné par la méthode `OutputStream getOutputStream()`. De la même façon, il est possible de lire des données envoyées par le client dans le flux retourné par la méthode `InputStream getInputStream()`. Afin de travailler sur du texte ligne par ligne, nous allons "convertir" ces deux flux en instances de `BufferedReader` et `PrintWriter` à l'aide des lignes suivantes (le `true` de la dernière ligne indique que nous souhaitons que les données soient envoyées au client à chaque retour chariot) :

```
BufferedReader reader = new BufferedReader(
                            new InputStreamReader(
                               socket.getInputStream()));
PrintWriter writer = new PrintWriter(
                            socket.getOutputStream(),
                            true);
```

La méthode `String readLine()` de `BufferedReader` permet de lire une ligne envoyée par le client. Elle est bloquante. La méthode `void println(String s)` de `PrintWriter` permet d'envoyer une ligne au client (notez que `System.out` est également une instance de `PrintWriter` connectée à la sortie standard du programme). Il est donc très facile de renvoyer au client toutes les lignes qui nous envoient à l'aide des lignes suivantes :

```
for(;;) {
  String line = reader.readLine();
  if (line==null) break;
  writer.println("["+line+"]");
}
```

À la fin du programme, il est important de fermer correctement toutes les ressources en utilisant les lignes suivantes :

```
socket.close();
serverSocket.close();
```

Vous devez également traiter les exceptions (déclenchées si des erreurs d'entrée/sortie se produisent). Ici, nous allons simplement stopper l'exécution du programme et afficher une trace (Bien évidemment, en pratique, un traitement plus fin des exceptions est préférable). Pour ce faire, entourez le code de la méthode `main` des lignes suivantes :

```
try {
 /* Code de la méthode main. */
} catch (IOException e) {
 e.printStackTrace();
}
```

Vous pouvez vous connecter au serveur en utilisant la ligne de commande suivante `telnet localhost 8088`. Pour interrompre la connexion `telnet`, il faut appuyer simultanément sur `Ctrl`, `AltGr` et `]`.

Exercice 2 – Les threads

En Java, il est possible d'exécuter (ou de simuler l'exécution) de plusieurs tâches en parallèle. À chacune de ces exécutions est associée une instance de la classe `Thread`. Ces tâches ne sont pas des processus car ils partagent les mêmes données.

Une instance qui implémente l'interface `Runnable` peut être exécutée dans un `thread` Java. Cette interface contient l'unique méthode `void run()` qui correspond à la méthode appelée pour débuter l'exécution du thread :

```java
class MyRunnable implements Runnable {
  private String name;

  public MyRunnable(String name) { this.name = name; }

  public void run() {
    for (int i = 1; i <= 5; i++) {
      if (Thread.interrupted()) break;
      System.out.println(name+":"+i);
      try {
        Thread.sleep(1000); // endort le thread une seconde.
      } catch (InterruptedException e) {
        // Un autre thread demande à l'exécution
        // du thread courant de s'arrêter.
        break; // on sort de la boucle.
      }
    }
  }
}
```

Pour exécuter la méthode `void run()` d'une instance de la classe `MyRunnable` dans un nouveau `thread`, il suffit d'instancier la classe `Thread` (en passant au constructeur l'instance de `MyRunnable`) et d'invoquer la méthode `void start()`. Que génère le code suivant sur la sortie standard ?

```java
new Thread(new MyRunnable("a")).start();
new Thread(new MyRunnable("b")).start();
new Thread(new MyRunnable("c")).start();
new Thread(new MyRunnable("d")).start();
```

Exercice 3 – Accepter plusieurs clients

Afin de traiter les requêtes de plusieurs clients en parallèle, nous allons créer un `thread` par client. Il est à noter que cette méthode n'est pas optimale en pratique car un grand nombre de connexions peut saturer rapidement le serveur. Cependant, la méthode "optimale" étant plus compliquée à mettre à place, nous allons utiliser autant de `threads` que de clients connectés.

1. Écrivez une classe `Client` qui implémente `Runnable`. Le constructeur prend en paramètre une instance de `Socket` et place sa référence dans une variable privée `socket`. La méthode `void run()` lit ligne par ligne les données envoyées par le client et les renvoie au client. Le code de la méthode est composé de codes donnés au premier exercice.

2. Dans la méthode `main`, faites en sorte que le serveur écoute sur le port `8088`. À chaque fois qu'un client se connecte, le serveur doit créer une instance de la classe `Client` et exécuter sa méthode `run` dans un nouveau `thread`.

Exercice 4 – Broadcasting

Nous souhaitons que les messages envoyés par un client au serveur soient renvoyés à tous les clients.

1. Ajoutez une méthode `void sendMessage(String msg)` à la classe `Client` permettant d'envoyer un message au client.
2. Définissez une classe `Server` qui propose les méthodes suivantes :
   — `void run()` (contenu du `main` actuel)
   — `void subscribeToBroadcast(Client client)`
   — `void unsubscribeToBroadcast(Client client)`
   — `void broadcastMessage(String message)`
3. Modifiez la classe `Client` de sorte que son constructeur prenne en paramètre une instance de la classe `Server`. Modifiez également sa méthode `run` de façon à s'abonner au `broadcast`, à diffuser les messages reçus du client et à se désabonner du `broadcast` à la fin de la connexion.
4. Modifiez la méthode `main` de façon à instancier un serveur et exécuter sa méthode `run`.

Exercice 5 – Programmation du client

Nous allons écrire un programme qui reproduit le comportement de `telnet`.

1. Dans un nouveau projet, écrivez une nouvelle classe `Client`. Cette classe possède un constructeur qui prend en paramètre une `URL` et un numéro de port. Le constructeur se connecte au serveur en instanciant la classe `Socket` de la façon suivante :

   ```java
   socket = new Socket(url, port);
   ```

2. Ajoutez à la classe `Client` une méthode `void sendMessage(String msg)` qui permet d'envoyer un message au serveur.
3. Ajoutez une méthode `void close()` qui permet de fermer la connexion.
4. Modifiez la classe `Client` de sorte qu'elle implémente l'interface `Runnable`. La méthode `void run()` doit recopier toutes les lignes reçues via le `socket` sur la sortie standard.
5. Ajoutez une classe `Main` avec la méthode `main` suivante :

```java
public static void main(String[] args) throws IOException {
  Client client = new Client("localhost", 8088);
  new Thread(client).start();

  Scanner scanner = new Scanner(System.in);

  for (;;) {
    String line = scanner.nextLine();
    if (line.equals("STOP")) break;
    client.sendMessage(line);
  }

  client.close();
  scanner.close();
}
```