Programmation Orientée Objet(POO)
IG2I - LA1 – 2019-2020
TP – Chat

Contexte et objectif du TP

Dans ce TP, nous nous attachons à réaliser un tchat multi-utilisateurs. On se base pour cela sur la relation client/serveur, avec des clients communiquant par l'intermédiaire du serveur. Du côté du serveur, il s'agit donc de gérer un ensemble de clients connectés. Du côté du client, l'envoi et la réception des messages sont réalisés au travers d'une interface graphique.

Ce TP permet ainsi d'illustrer les notions suivantes :
• la programmation concurrente en Java ;
• les processus légers en Java : Thread ;
• la programmation réseau en Java : Socket ;
• les flux d'entrées/sorties.

Rappels

Pour réaliser ce TP, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez, en particulier en ce qui concerne celles de l'API Swing, ainsi que des classes Thread, Socket et ServerSocket. Par ailleurs, il est aussi important que votre propre code soit commenté au format Javadoc (commentaire commençant par "/**" au-dessus des définitions des attributs et méthodes), et correctement indenté.

---

1 Programmation client/serveur de base

Dans cette section, nous étudions comment créer une connexion client/serveur de base.

1.1 Les processus légers

Un processus est une unité d'exécution gérée par le système. On associe donc à un processus des lignes de code qui décrivent son déroulement ainsi qu'un espace mémoire dédié. Or, dans de nombreux contextes il est intéressant d'effectuer plusieurs traitements distincts en même temps. "En même temps" ne signifie pas une simultanéité parfaite, mais il est possible de simuler l'exécution simultanée de deux traitements en leur permettant d'avoir accès, chacun à leur tour, au processeur. On parle alors de traitement concurrents : les processus sont en concurrence permanente pour l'accès au processeur. Cependant, le fait que les processus disposent de leur propre espace mémoire limite les possibilités d'interactions entre processus et ne permet pas une gestion fine de la concurrence entre processus.
Pour pallier ce problème, il existe la notion de processus léger (thread). Un processus léger est une séquence d'exécution du code d'un programme au sein d'un processus (lourd). L'avantage de ces processus légers est qu'ils partagent les ressources système (code, mémoire, fichiers). Ils ont cependant leur propre pile d'exécution. En fait, vous avez déjà utilisé des threads en Java, car le noyau Java s'appuie sur le multi-threading : le programme

est exécuté par un thread et la mémoire est gérée par un autre thread. La classe Thread et l'interface Runnable sont les bases pour le développement des threads en Java.
L'interface Runnable déclare une unique méthode **void run()**. Cette interface doit être implémentée par toute classe dont les instances sont destinées à être exécutées par un thread. Dans les classes qui implémentent cette interface, la méthode **run()** doit être redéfinie pour contenir le code des traitements qui seront exécutés dans le thread. Lors du démarrage du thread, c'est cette méthode **run(**) qui est appelée.

La classe Thread implémente l'interface Runnable.

Ainsi, la manière simple de créer un thread en Java est de créer une classe MonThread qui hérite de la classe Thread. Cette classe MonThread doit redéfinir la méthode **run()** afin de décrire le comportement du thread. Il est alors possible de créer des instances de MonThread avec l'opérateur new.
Pour démarrer l'exécution du thread, il faut appeler la méthode **start()** (définie dans la classe Thread).

Question 1.
Créez un nouveau projet chat. Créez un nouveau paquetage serveur dans lequel vous ajouterez une classe TestThread. Faites en sorte que cette classe soit un thread qui affiche le message "serveur en vie". Testez le bon fonctionnement de votre thread.

Question 2.
La classe Thread possède une méthode statique **sleep (long time)** qui permet d'interrompre l'exécution du thread pendant time millisecondes.
Faites en sorte que le thread TestThread affiche le message "serveur en vie" toutes les secondes.

1.2 États d'un thread et transitions

Les threads sont des objets ayant un état prédéfini : leur état d'exécution. Il s'agit dans la classe Thread d'un attribut avec une visibilité privée. Les objets de type Thread peuvent être :

- créés (nouveau) ;
- actifs (exécutables) ;
- endormis (en attente) : en attente de notification ;
- bloqués (après l'acquisition d'un verrou) ;
- temporisés ;
- morts (terminés).

La classe Thread propose un ensemble de méthodes, présentées dans la Figure 1, qui permettent de passer d'un état à un autre. Les objets de type Thread sont donc créés comme n'importe quel autre objet Java avec l'opérateur new().
Ceci permet d'allouer la mémoire nécessaire pour l'objet ; mais à ce moment là, le thread n'est pas encore vivant. La méthode **start()**, comme nous l'avons vu précédemment, permet d'allouer les ressources nécessaires à l'exécution du thread et lance la méthode **run()** : le thread devient alors actif.
Un thread peut être endormi, temporisé ou bloqué. Dans ces cas précis, il ne consomme pas de ressources machine, et une fois réveillé (après un délai d'attente, l'acquisition d'une ressource ou l'appel d'une méthode) il redevient actif. La mort d'un thread peut intervenir de différentes façons, en particulier à la fin de la méthode **run()**. Par ailleurs, on notera que la méthode *stop()* est maintenant dépréciée et on ne l'utilisera donc pas.
Pour arrêter proprement un objet de type Thread, on utilise la méthode **interrupt()** qui positionne un indicateur booléen à true. Un objet de type Thread doit alors vérifier régulièrement s'il a été interrompu en faisant appel à la méthode **Thread.currentThread().isInterrupted()**, et en terminant l'exécution de la méthode run() si la condition est vérifiée.

On notera cependant que si un objet de type Thread est dans un état bloqué, endormi ou temporisé, il ne peut pas tester s'il a été interrompu. Dans ce cas, si la méthode **interrupt** a été appelée sur ce thread, l'appel bloquant est brutalement terminé par une exception de type InterruptedException. La levée de cette exception refait passer automatiquement le flag "interrompu " à zéro. Il faut donc le re-positionner volontairement à 1 (utiliser la méthode interrupt) pour arrêter proprement le thread.
On veillera alors à traiter cette exception et à interrompre le thread.

Par ailleurs, vous aurez noté l'appel à Thread.currentThread() pour récupérer la référence sur le thread qui est actuellement exécuté.
Lorsqu'un thread est actif, il ne s'exécute pas en permanence, mais il dispose régulièrement d'une tranche de temps. Les tranches de temps étant brèves, ceci donne à l'utilisateur l'impression d'une exécution simultanée (sur un processeur).

Question 3.
Modifier la méthode **run()** de TestThread de sorte que l'exécution se termine correctement lorsque le thread a été interrompu. Pour faire les tests, dans la méthode principale, après avoir démarré le thread, on peut attendre un certain temps en faisant appel à la méthode **sleep()**, puis on interrompt le thread.

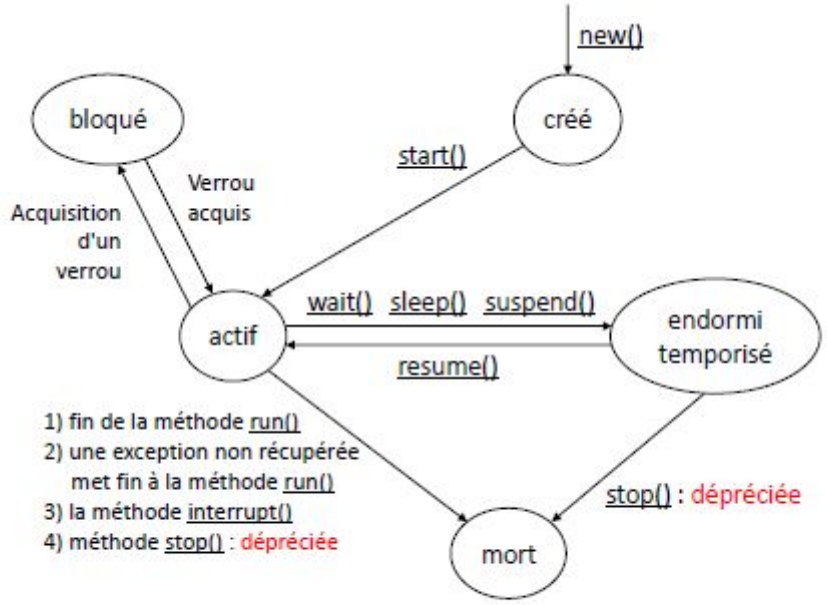

Figure 1: Les méthodes de la classe **Thread** qui permettent de faire les transitions entre les différents états.

1.4 Communication client/serveur

Le protocole TCP (Transmission Control Protocol) offre un service en mode connecté et fiable. La procédure d'établissement de connexion est dissymétrique. Un processus, appelé serveur attend des demandes de connexion qu'un processus, appelé client, lui envoie. Une fois l'étape d'établissement de connexion effectuée,le fonctionnement devient symétrique.

Les sockets (prises de raccordement) forment une API : ils offrent aux programmeurs une interface entre le programme d'application et les protocoles de communication. Un socket est une représentation logique du point de branchement d'un programme au réseau. Dans le modèle TCP/IP, les extrémités de communication sont identifiées par trois informations : une adresse IP, le protocole utilisé, et un numéro de port. Ce dernier est un entier entre 0 et 65535.

Dans le cadre des TPs, il est recommandé d'utiliser un numéro de port dynamique (privé), i.e. compris entre 49152 et 65535 (de 0 à 1023 ce sont les ports réservés, et de 1024 à 49151, ce sont les ports enregistrés). Nous nous intéressons ici aux sockets utilisant le protocole TCP, i.e. qui fonctionnent en utilisant le modèle client/serveur et offrent une communication par flux.

On notera que du côté du serveur, on utilise deux types de socket : l'un, appelé socket d'écoute reçoit les demandes de connexion, et l'autre, appelé socket de service, sert pour la communication. En effet, un serveur peut être connecté simultanément avec plusieurs clients, et dans ce cas on utilise autant de socket de service que de clients.

Java fournit une implémentation réalisée grâce aux classes Socket et ServerSocket définies dans le paquetage java.net.

1.4.1 ServerSocket

La classe ServerSocket représente le socket d'écoute du côté serveur. Cette classe possède un constructeur auquel on passe en paramètre le numéro du port sur lequel on "écoute" les requêtes des clients (vous choisirez dans ce TP un numéro entre 49512 et 65535). Vous pourrez regarder avec la Javadoc les méthodes fournies par cette classe. On retiendra en particulier la méthode **accept()** qui attend qu'une connexion soit réalisée vers le socket d'écoute et l'accepte.

Cette méthode est bloquante jusqu'à ce qu'une connexion soit effectivement établie. Cette méthode renvoie une instance de la classe Socket correspondant à la socket de service qui permet de communiquer avec le client. La méthode **accept()** étant par défaut bloquante jusqu'à ce qu'une connexion soit établie, il est possible de définir un temps maximal d'attente avec la méthode **setSoTimeout(int ms)**. Lorsque le temps d'attente est dépassé et qu'il n'y a eu aucune connexion, alors une exception SocketTimeoutException est levée. Par ailleurs, la classe ServerSocket fournit une méthode **close()** qui ferme le socket, et qu'il est préférable d'appeler de manière explicite dans le code lorsqu'on a plus besoin de l'objet ServerSocket. L'appel à cette méthode est souvent réalisé dans un bloc finally après le traitement des exceptions.

1.4.2 Socket

La classe Socket représente un socket de flux : un socket côté client, ou encore un socket de service côté serveur (que l'on récupère par un appel à la méthode **accept()** de la classe ServerSocket).

Cette classe possède un constructeur **Socket(InetAddress address, int port**) qui crée un socket de flux et se connecte au numéro de port spécifié et à l'adresse IP spécifiée dans l'objet InetAddress. On peut obtenir un objet de type InetAddress en faisant appel aux méthodes statiques **getLocalHost()** ou **getByName(String host**) de la classe InetAddress.

La classe Socket possède également un constructeur **Socket(String host, int port)** qui crée un socket de flux et se connecte au numéro de port spécifié et à l'hôte nommé host.
La classe Socket possède également une méthode **close()** qu'il est préférable d'appeler de manière explicite lorsque l'on souhaite fermer le socket. On notera que tout thread actuellement bloqué sur une opération d'entrée/sortie sur ce socket provoquera la levée d'une exception SocketException. De plus, lorsqu'un socket a été fermé, il ne peut pas être utilisé pour réaliser une nouvelle connexion, il faut créer une nouvelle instance de Socket.

Question 4.
Dans le paquetage serveur, ajoutez une classe TestServeur qui hérite de Thread. Redéfinissez la méthode **run()** de sorte que TestServeur ait le comportement minimal d'un serveur : création d'un socket d'écoute, et tant qu'il n'est pas interrompu, attente et acceptation d'une nouvelle connexion (socket de service), puis fermeture de ce socket de service. Vous veillerez à fermer proprement les sockets, et à laisser des traces d'exécution (lors de l'attente d'une connexion, lorsque la connexion a été établie, dans le traitement des exceptions, lors de la fermeture des sockets).

Question 5.
Créez un nouveau paquetage client et ajoutez-y une classe TestClient. Dans cette classe, ajoutez une méthode public void **connect()** qui suit le comportement d'un client qui se connecte au serveur : création d'un socket de flux en précisant l'adresse du serveur et le port d'écoute, puis fermeture de ce socket.
Pour l'adresse du serveur, vous utiliserez dans un premier temps l'adresse locale de votre machine, puis vous pourrez ensuite tentez de vous connecter sur le serveur d'un de vos camarades.
Par ailleurs, vous veillerez ici aussi à laisser des traces d'exécution (lorsque la connexion a été établie, dans le traitement des exceptions, lors de la fermeture des sockets).
NB : Pour les tests, il faut que TestServeur et TestClient possèdent des méthodes principales (main). Il suffit alors d'exécuter TestServeur, puis d'exécuter autant de fois que vous le souhaitez TestClient. Vous pouvez également vérifier ce qui se passe si vous exécutez TestClient sans que le serveur ait été lancé ou avec une adresse ou un port incorrects.

1.5 Flux d'entrées/sorties

En Java, toutes les entrées/sorties sont gérées par des flux (lecture du clavier, affichage sur la console, lecture/écriture dans un fichier, échange de données réseau avec des sockets). Vous avez déjà utilisé au moins le flux standard pour écrire sur la console (System.out).

La paquetage java.io permet d'accéder à des flux de données sur des fichiers ou des sockets. Les flux binaires d'entrées héritent tous de la classe abstraite InputStream et les flux binaires de sortie héritent tous de la classe abstraite OutputStream.

La classe Socket fournit deux méthodes **getInputStream()** et **getOutputStream()** qui permettent de récupérer respectivement un flux d'entrée et un flux de sortie pour le socket. Les flux texte d'entrées héritent tous de la classe abstraite Reader et les flux texte de sortie héritent tous de la classe abstraite Writer. Il existe dans le paquetage java.io de nombreuses classes concrètes qui permettent de gérer les flux selon que l'on souhaite faire de l'impression, de la mise en tampon, de la sérialisation, de la conversion de données ... Nous proposons ici d'utiliser **BufferedReader** pour la lecture, et **PrintWriter** pour l'écriture de données au format texte. Vous restez libre de regarder la Javadoc et d'utiliser d'autres classes si nécessaire.

BufferedReader lit le texte à partir d'un flux de caractères en entrées, et utilise un tampon pour stocker les caractères et ainsi fournir une lecture plus efficace. Pour construire un objet BufferedReader à partir d'un InputStream (appelons le is), on peut procéder de la manière suivante :

```
1 InputStreamReader isr = new InputStreamReader(is);
2 BufferedReader in = new BufferedReader(isr);
```

On peut ensuite utiliser la méthode **readLine()** de BufferedReader qui renvoie un String contenant une ligne de texte (ou null si la fin du flux est atteinte).

PrintWriter écrit une représentation formatée des objets dans un flux de sortie texte. PrintWriter possède un constructeur **PrintWriter(OutputStream out, boolean autoFlush)** qui crée un objet PrintWriter à partir d'un objet de type OutputStream déjà existant ; et le booléen autoFlush permet de préciser si l'on souhaite vider automatiquement le tampon lors des appels aux méthodes **println()**, **printf()** et **format()**. Si le tampon n'est pas vidé de manière automatique, il faut alors penser à appeler la méthode **flush()** à chaque fois que l'on souhaite vider le tampon. Le flux en écriture n'est envoyé que lorsque le tampon est vidé. Il faut donc bien y penser.

Par ailleurs, les objets InputStream, OutputStream, BufferedReader et PrintWriter possèdent une méthode **close()** qui permet de libérer les ressources associées lorsqu'on en a plus besoin. Il est donc fortement recommandé de faire appel à la méthode **close()** sur ces objets. On notera par ailleurs que la fermeture d'un flux entraîne la fermeture des ressources qui lui sont associées. Par exemple, lorsque le flux associé à un socket est fermé, alors le socket est automatiquement fermé. Ou encore, lorsque le PrintWriter est fermé, le OutputStream associé est fermé.

Question 6.

Modifiez la méthode run() de la classe TestServeur, de sorte que lorsqu'un client vient de se connecter, on récupère à l'aide d'un BufferedReader une ligne de texte (qui sera envoyée par le client). On affichera ensuite ce message sur la console.

Question 7.

Modifiez la méthode connect() de la classe TestClient, de sorte qu'après s'être connecté au serveur, le client envoie un message en écrivant sur un objet de type PrintWriter. Testez que tout fonctionne correctement.

2 Tchat multi-clients

Nous cherchons à présent à mettre en place une application qui permette de faire un chat entre plusieurs clients. Cette discussion entre les clients se fait par l'intermédiaire d'un serveur. Lorsqu'un client envoie un message au serveur, ce dernier renvoie à tous les clients connectés le message. Tous les messages envoyés par le client seront préfixés par le nom du client afin d'identifier les propos des différents utilisateurs.
Comme il y a potentiellement plusieurs clients connectés au serveur, ce dernier doit conserver l'ensemble des connexions courantes. Cet ensemble doit être mis à jour à chaque nouvelle connexion d'un client, et à chaque départ d'un client de la discussion. Ainsi, dès qu'une des connexions reçoit un message de son client, elle doit accéder à l'ensemble des connexions pour envoyer ce message à tous les clients. Il y a ainsi plusieurs threads qui accèdent en lecture et en écriture à l'ensemble des connexions. Il faudra donc gérer convenablement les potentiels conflits d'accès simultanés à cette ressource partagée. Les Figures 2 et 3 présentent les diagrammes de classe pour les parties serveur et client respectivement. On rajoutera ces classes au fur et à mesure, mais n'hésitez à vous reportez à ces diagrammes pour avoir une vision d'ensemble.

2.1 Partie serveur : mémoriser l'ensemble des connexions

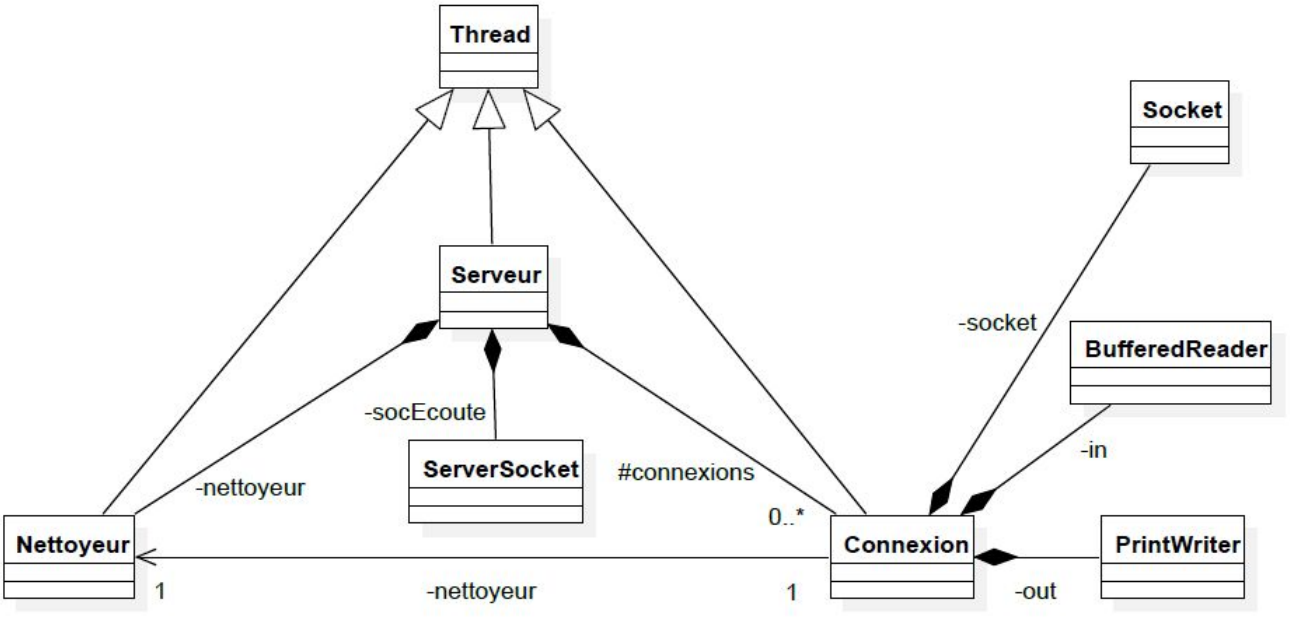

Figure 2: Diagramme de classe du paquetage **serveur**.

Question 8.

Dans le paquetage serveur, ajoutez une classe Connexion qui représente une connexion du client, avec ses attributs. Ajoutez-y un constructeur qui prend en paramètre un objet de type Socket et initialise tous les attributs (le constructeur peut lever une exception de type IOException).

Question 9.

Dans le paquetage serveur, ajoutez une classe Serveur qui représente le serveur, avec ses attributs. Ajoutez-y un constructeur par défaut qui initialise le socket d'écoute. Ce constructeur pourra lever une exception de type IOException.

Question 10.

Dans la classe Serveur, redéfinissez la méthode run() de la classe Thread. À chaque fois qu'un client se connecte sur le socket d'écoute, on crée un nouvel objet Connexion qu'on ajoute à l'ensemble des connexions.

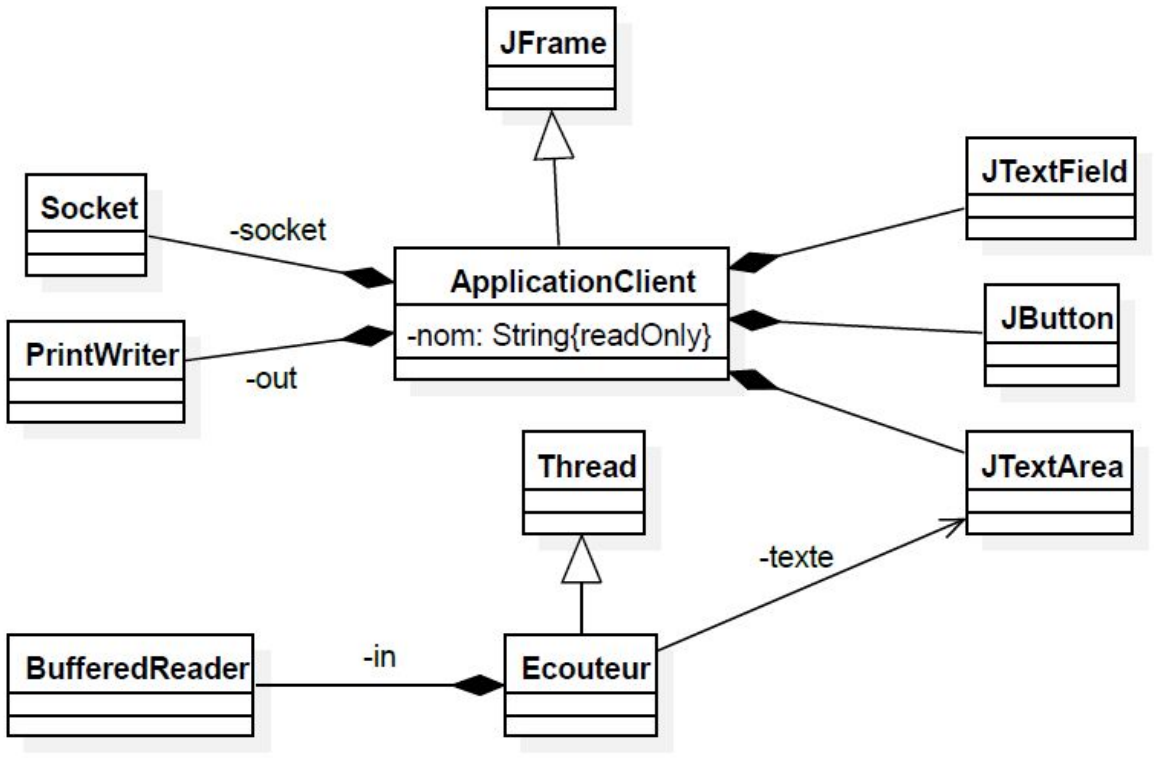

Figure 3: Diagramme de classe du paquetage **client**.

On pourra ajouter une trace d'exécution qui donne le nombre de connexions. On pourra aussi écrire une méthode private void fermerSocketEcoute() qui gère correctement la fermeture du socket d'écoute ; et cette méthode sera appelée dans le bloc finally de la méthode run().

Question 11.

Dans la méthode principale de la classe Serveur, écrivez quelques lignes de test pour vérifier que tout fonctionne correctement. Après avoir lancé le serveur, vous pouvez créer des objets de type Socket qui se connectent au serveur, comme dans l'exemple suivant :

```
1 Serveur serv = new Serveur();
2 serv.start();
3 for(int i=0; i<5; i++) {
4 Socket soc = new Socket(InetAddress.getLocalHost(), 49512);
5 }
```

2.2 Partie client : mise en place de l'interface graphique et connexion au serveur.

On s'intéresse ici à l'interface graphique pour l'utilisateur (du côté client donc). Un exemple est donné dans la Figure 4.

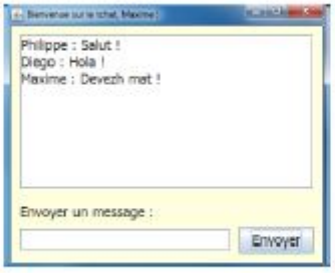

Figure 4: Exemple d'interface graphique pour l'utilisateur.

Question 12.

Dans le paquetage client, ajoutez une fenêtre graphique ApplicationClient qui sera l'interface de chat pour les utilisateurs. Placez-y les objets graphiques suivants : un TextArea pour afficher les messages de tous les clients, un TextField pour écrire un nouveau message, et un JButton pour envoyer le nouveau message.

Question 13.

Ajoutez les attributs de ApplicationClient (voir Figure 3). Modifiez le(s) constructeur(s) afin d'initialiser les attributs et de réaliser la connexion au serveur. N'hésitez pas à créer des méthodes intermédiaires si nécessaire, et à gérer convenablement les exceptions (il faut informer l'utilisateur des éventuels problèmes).

Question 14.

Faites en sorte que la méthode dispose() soit appelée à la fermeture de la fenêtre, et redéfinissez cette méthode dispose() afin de fermer convenablement le socket et les flux.

Question 15.

Testez le code développé jusqu'à présent : lancez le serveur, puis lancez plusieurs clients qui doivent donc se connecter au serveur. Vérifiez en particulier que le nombre de connexions dans le serveur est correct. Vérifiez aussi que vous gérez bien les cas où il y aurait un problème de connexion.

2.3 Partie serveur : retour sur les connexions

Question 16.

Dans la classe Connexion du paquetage serveur, ajoutez une méthode **private void envoyerMessage(String message)**. Cette méthode doit écrire sur le flux de sortie de chacune des connexions de l'ensemble des connexions du serveur. Jusqu'ici, nous n'avons pas évoqué le problème de conflit de ressources : plusieurs threads tentent de mettre à jour le même emplacement de mémoire en simultané. Que se passerait-il par exemple, si plusieurs threads Connexion appelaient en même temps la méthode envoyerMessage() ? Ou bien si lors de l'appel à envoyerMessage(), le serveur ajoutait une nouvelle connexion dans la liste ? Afin de verrouiller l'accès à certaines ressources communes, le langage Java propose le mot clé synchronized. Il est possible de verrouiller une méthode afin qu'elle ne soit utilisée que par un seul thread à la fois. Pour cela, il suffit d'ajouter le mot clé synchronized dans la signature de la méthode. Par exemple :

```
1 private synchronized void envoyerMessage (String message) {
2 ... // Un seul thread à la fois peut exécuter cette méthode
3 }
```

Il est aussi possible de verrouiller un objet dans une portion de code. Pour cela, à l'intérieur d'une méthode on déclare :

```
1 public void maMethode () {
```

```
2 ...
3 synchronized(nomObjet) {
4 ... // Je suis seul à accéder à nomObjet
5 }
6 ...
7 }
```

Question 17.

Modifiez le code des classes Serveur et Connexion de telle sorte que deux threads ne puissent pas accéder en même temps à l'ensemble des connexions. On veillera à faire cela à chaque fois que l'ensemble des connexions sera utilisé.

Question 18.

Dans la classe Connexion, redéfinissez la méthode run(). Le déroulement est le suivant : tant qu'il y a un message à lire sur le flux d'entrée, on envoie ce message à toutes les connexions. Lorsque la fin du flux est atteinte, on peut fermer toutes les ressources (on peut écrire une méthode spécifique pour cela), et terminer l'exécution de la méthode run(). N'oubliez pas dans la classe Serveur, de démarrer le thread Connexion à chaque fois qu'une nouvelle connexion est ajoutée à la liste.

2.4 Partie client : entrées/sorties

Question 19.

Dans la classe ApplicationClient, ajoutez la partie contrôle lorsque l'utilisateur appuie sur le bouton "Envoyer". Pour rappel, il faut envoyer le message saisi par l'utilisateur au serveur, en le préfixant avec le nom de l'utilisateur.

Question 20.

Dans le paquetage client, ajoutez une classe Ecouteur qui représente un écouteur en attente sur un flux de texte en entrée (de type BufferedReader), et qui affiche le texte reçu sur un objet de type JTextArea.
Cet objet Ecouteur hérite de Thread et a comme attributs un JTextArea et un BufferedReader. Ajoutez un constructeur par données. Redéfinissez la méthode run() de telle sorte que l'écouteur ajoute sur la zone de texte les lignes de textes qu'il reçoit sur le flux.

Question 21. Dans la classe ApplicationClient, après avoir créé le socket et fait la connexion au serveur, créez un objet de type Ecouteur, et démarrez-le.

Question 22.

Effectuez à présent un test. Pour cela, vous pouvez définir, dans le paquetage client, une classe principale qui crée plusieurs instances de ApplicationClient avec des noms différents. Testez dans un premier temps que tout fonctionne bien en local. Si c'est le cas, vous pouvez testez à distance (vous choisissez une personne qui démarre son serveur (et vous indique l'adresse IP et le numéro de port), et tous les autres se connectent chez lui). Vérifiez que l'application tient la charge.

2.5 Echange d'objets

Échanger des messages texte, c'est bien; échanger des messages contenant les objets de notre choix, c'est mieux !

En effet, jusqu'ici nous n'avons envoyé que des objets de type **String**. Mais comment gérer l'envoi d'objets plus complexes : par exemple comment envoyer un message ainsi que la couleur à afficher ? On pourrait envisager de faire un encodeur et décodeur pour tout envoyer sous forme de **String**, mais ce serait fastidieux et source d'erreurs.

Pour envoyer des objets sur les flux d'entrée **InputStream** et les flux de sortie **OutputStream**, on peut utiliser des objets de type **ObjectInputStream** et **ObjectOutputStream** respectivement.

Pour faire les lectures et écritures, **ObjectInputStream** possède une méthode **readObject()**, et **ObjectOutputStream** possède une méthode **writeObject(Object o)**.

Pour l'écriture, il ne faut pas oublier de vider le tampon en appelant la méthode **flush()**. Par ailleurs, à la lecture, on récupère un objet de type **Object** que l'on peut être amené à caster si l'on souhaite faire appel à une de ses méthodes (il faut alors connaître le type de l'objet que l'on récupère ...).

Question 23.

Dans la classe **Connexion** modifiez les types des attributs qui servent à la lecture et à l'écriture pour qu'on puisse lire et écrire des objets. Modifiez en conséquence les méthodes de la classe **Connexion**.

Question 24.

Dans la classe **ApplicationClient**, modifiez le type de l'attribut qui sert à l'écriture pour qu'on puisse écrire des objets. Modifiez en conséquence les méthodes de la classe **ApplicationClient**. En particulier, modifiez la méthode **dispose()** de telle sorte qu'avant de fermer le flux, elle envoie au serveur un objet **null**, ce qui permet d'arrêter proprement le *thread* de la classe **Connexion**.

Question 25.

Dans la classe **Ecouteur**, modifiez le type de l'attribut qui sert à la lecture pour qu'on puisse lire des objets. Ajoutez une méthode **private boolean lireMessage()** qui fait la lecture d'un objet sur le flux d'entrée et traite l'objet lu. Cette méthode retourne **false** si l'objet lu est **null**, **true** sinon. Modifiez en conséquence la méthode **run()**.

Question 26.

Testez et vérifiez que l'application fonctionne comme précédemment. Et, à présent, nous sommes capables d'envoyer des objets de type autre que **String.**

Maintenant que l'on sait envoyer des objets, on souhaite qu'un utilisateur puisse appuyer sur un bouton sur sa fenêtre graphique. Ceci provoquera un mouvement (*wizz*) sur les fenêtres de tous les clients.
Pour que des objets puissent être écrits dans un fichier ou envoyés sur le réseau, il faut qu'ils soient sérialisables, i.e. qu'il soit possible de les convertir en un tableau d'octets qui va ensuite être écrit dans un fichier ou envoyé sur le réseau. Pour cela, il faut que les objets implémentent l'interface **Serializable**. C'est le cas de la classe **String** par exemple, et c'est pour cela que l'on a pu envoyer des objets de type **String** sur le réseau.

Question 27.

Ajoutez un paquetage **objetsenvoyes**.Ajouter une classe **Surprise**, qui contiendra les éléments que vous souhaitez communiquer (des informations complémentaires sur l'auteur du message, une image ...)

Modifier vos classes coté client pour être capable d'envoyer et de recevoir ces types d'objet, et d'en faire bon usage !