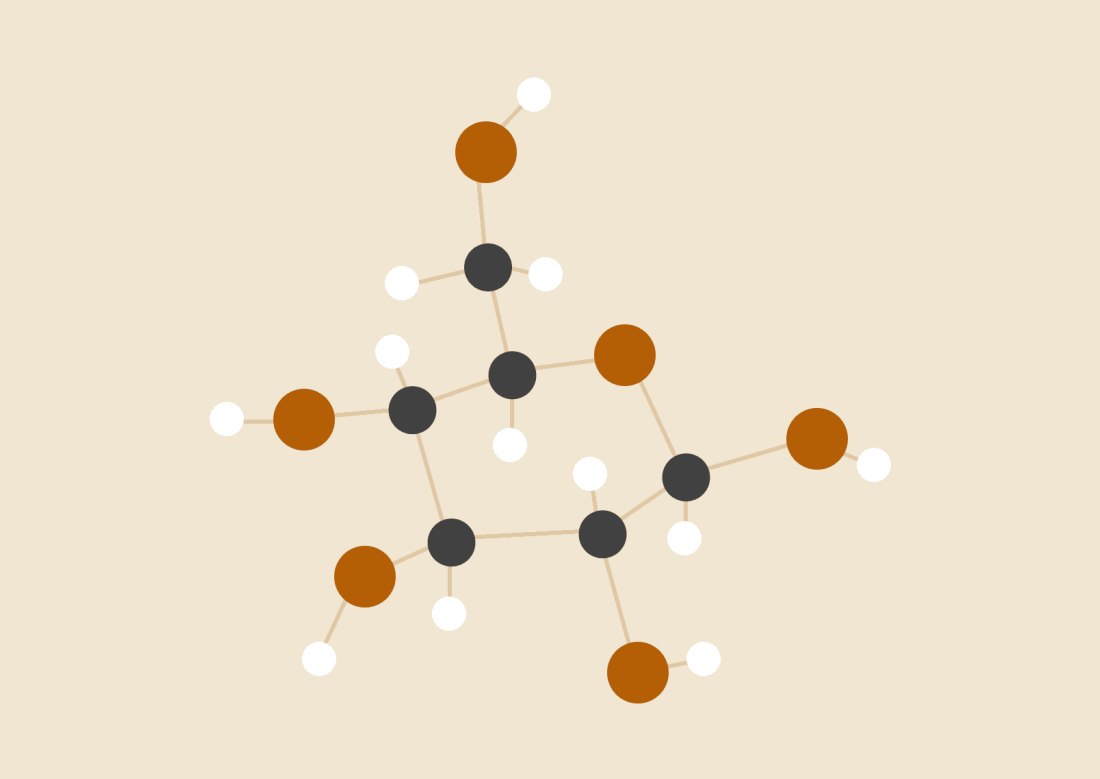
INF111 Programmation orientée objet

Travaux pratiques (TP1 et TP2)

Groupes : tous

Département des Enseignements Généraux

Tic-Tac-Tchat en Réseau



Abdelmoumène Toudeft, Professeur enseignant

Abdelmoumene.Toudeft@etsmtl.ca

Bureau B-1642

INF111 Programmation orientée objet

Travaux pratiques (TP1 et TP2)

Groupes : tous

Département des Enseignements Généraux

Table des matières

INF111 Programmation orientée objet

Travaux pratiques (TP1 et TP2)

Groupes : tous

Département des Enseignements Généraux

INF111 Programmation orientée objet

Travaux pratiques (TP1 et TP2)

Groupes : tous

Département des Enseignements Généraux

Introduction

Ce projet va vous permettre de mettre en œuvre les concepts et notions vu dans le cours **INF111 Programmation orientée objet**. Il sera réalisé en **2 TPs** :

- **TP1** : mise en oeuvre des concepts objets de base et programmation en mode console;
- **TP2** : ajout d'une interface graphique avec gestion d'événements et mise en œuvre d'autres concepts.

Mais pas que ...

ce projet va aussi ...

- vous introduire au monde de la programmation réseau avec les sockets où un serveur communique avec des clients ;
- vous montrer les mécanismes internes de la programmation événementielle;
- utiliser les bases de la programmation parallèle (*multithreading*).

Tous ces extras vous seront évidemment fournis et expliqués et vous n'aurez pas à les programmer vous-mêmes. Vous n'êtes même pas obligés de les comprendre pour faire les TPs.

Description et contexte

Nous désirons réaliser une architecture client-serveur qui permettra à des utilisateurs d'utiliser une application cliente pour se connecter à un serveur afin de clavarder dans un salon de chat public, de clavarder à 2 dans un salon de chat privé et de jouer des parties de tic-tac-toe (jeu de morpion).

Il y aura donc 2 applications (programmes) :

- Une application cliente que les utilisateurs utilisent pour accéder aux fonctionnalités ;
- Une application serveur pour héberger les salons de chats et les parties de jeu de tic-tac-toe et qui relayera les échanges entre les utilisateurs.

Lors de sa connexion au serveur, l'utilisateur devra fournir un **alias** qui n'est pas déjà utilisé par un des utilisateurs déjà connectés. L'alias doit aussi respecter certaines règles (ne peut pas être vide ou contenir des caractères spéciaux, ...). Si la connexion est acceptée, l'utilisateur recevra tous les messages qui ont été envoyés par les utilisateurs membres du salon public depuis le démarrage du serveur.

L'utilisateur pourra ensuite :

- chatter dans le salon public;
- chatter à 2 en privé;
- jouer à tic-tac-toe.

Il y aura donc 2 types de salons :

- **Salon privé** : 2 membres uniquement. Dès que 1 quitte, le salon disparaît.
- **Salon public** : tous les utilisateurs sont membres automatiquement.

INF111 Programmation orientée objet
TP 1
Auteur : Abdelmoumène Toudeft
Groupes : tous
Enseignant-e-s : El Hachemi Alikacem,
Frédéric Simard et
Abdelmoumène Toudeft

Département
des Enseignements
Généraux

TP 1

Dans ce premier TP, nous travaillerons en **mode console** (pas d'interface graphique).

Vous allez **compléter le programme serveur et le programme de tic-tac-toe** pour implémenter les fonctionnalités de chat et de jeu de tic-tac-toe.

Le programme client fourni est complet et vous n'avez pas à le modifier dans le cadre de ce TP 1. Vous l'utilisez tel quel dans le TP 1. Il sera complété dans le TP 2.

Ce TP comporte 4 questions contenant 18 sous-questions.

Avertissement

Vous devez absolument **respecter les spécifications** fournies dans cet énoncé (nom des classes, nom des attributs et méthodes publiques,...). Sans quoi, votre code risque de ne pas s'intégrer correctement avec le code fourni ou qui sera fourni dans le TP 2.

INF111 Programmation orientée objet
TP 1
Auteur : Abdelmoumène Toudeft
Groupes : tous
Enseignant-e-s : El Hachemi Alikacem,
Frédéric Simard et
Abdelmoumène Toudeft

Département
des Enseignements
Généraux

Directives pour le TP 1

- ➔ Travail en équipes de **2 ou 3**;
- ➔ Date limite de constitution des équipes : **21 octobre à 23h59**
- ➔ Date limite de remise du travail : **09 novembre à 23h59**
- ➔ Le travail doit être remis sur *Github* (tout le monde doit avoir un compte *Github*);
- ➔ Vous devez remettre uniquement l'application serveur et l'application du jeu de tic-tac-toe;
- ➔ Trois (3) programmes sous fournis sous forme de projets *IntelliJ* : ***ChatServer***, ***ChatClient*** et ***TicTacToe***. Ce sont les points de départ du *TP* :
  - Un membre de l'équipe doit **ouvrir les projets dans *IntelliJ*** puis **les partager dans *GitHub***. Les autres membres vont **importer les projets** dans *IntelliJ* à partir de *Github*.
  - Les noms des référentiels doivent **montrer clairement** les noms des membres de l'équipe.
  - Les référentiels doivent être **privés**.
  - Le document ***Github et IntelliJ.pdf*** explique cette démarche.
  - Vous devez mettre votre **enseignant comme collaborateur** à votre/vos référentiel(s).
- ➔ Le fichier ***docs.zip*** contient la documentation *Javadoc* des classes et interfaces fournies (ouvrez les fichiers ***index.html*** dans votre navigateur).

Introduction à la programmation réseau avec les sockets

Avertissement

Cette section vous présente des notions et concepts qui **ne font pas** partie du cours. **Vous n'avez pas à comprendre** tous ces concepts et toutes ces notions. L'objectif est de vous aider à comprendre le contexte dans lequel fonctionnent les programmes que vous aurez à modifier/compléter.

Ceci dit, connaître ces notions et concepts n'est pas une mauvaise chose pour un-e futur-e ingénieur-e 😉.

Adresse IP et port de communication

Un ordinateur doté d'une carte réseau peut être connecté à d'autres ordinateurs pour constituer un réseau. Chaque ordinateur est identifié sur le réseau par une **adresse IP** composée de 4 entiers entre 0 et 255. La forme de l'adresse est 192.68.23.4.

Pour communiquer sur un réseau, les programmes utilisent des **ports de communication**. Un port est représenté par un numéro sur 2 octets. Cela signifie que sur chaque ordinateur, il y a potentiellement 2^{16} = 65536 programmes qui peuvent accéder au réseau.

Pour se connecter à un programme sur un réseau, on a besoin de connaître :

1. L'adresse IP de l'ordinateur sur lequel s'exécute le programme ;
2. Le numéro de port utilisé par le programme.

L'adresse IP **127.0.0.1** est spéciale. Elle représente l'adresse de l'ordinateur sur lequel s'exécute le programme lui-même (***localhost***). Un programme peut l'utiliser pour se connecter à un autre programme qui s'exécute sur le même ordinateur.

Serveur

Un serveur est un programme qui **écoute sur un port de communication** et attend qu'un client s'y connecte. Par exemple, un serveur web écoute, par convention, sur le port 80.

Client

Un client est un programme qui utilise un port de communication pour se connecter à un serveur. Par exemple, un navigateur web est un client qui se connecte à des serveurs web.

Connexion-Communication

Pour se connecter, le client a besoin de connaître l'adresse IP et le port de communication du serveur.

Le client envoie une demande de connexion. Une fois que le serveur l'a acceptée, la connexion est établie et les deux parties peuvent communiquer.

Pour communiquer, les deux parties utilisent des mécanismes d'entrée/sortie identiques à ceux utilisés pour communiquer avec des fichiers (flux d'entrées/sorties).

Socket

Un socket est un point de communication. Il est fourni par le système d'exploitation comme une interface pour cacher tous les détails de la connexion et de la communication entre un client et un serveur.

Un socket désigne l'extrémité d'un canal de communication bidirectionnel entre un client et un serveur. Une fois la connexion établie, le client et le serveur disposent, chacun de son côté, d'un socket pour communiquer avec l'autre.

Implémentation en Java

Pour faciliter le développement d'applications client/serveur avec les sockets, l'API Java fournit les 2 classes **java.net.ServerSocket** et **java.net.Socket** pour établir des connexions :

- **ServerSocket** est utilisée par un serveur pour écouter sur un port de communication;
- **Socket** est utilisée par les clients et les serveurs pour créer des flux pour communiquer.

Le serveur doit d'abord créer un objet de type **ServerSocket** avec un port de communication (exemple : 3845) et se mettre en attente de connexion avec la méthode **accept()**.

Le client doit connaître l'adresse IP du serveur et le port sur lequel il attend. Il crée un objet de type **Socket**. Le constructeur de cette classe envoie une demande de connexion au serveur. Si celui-ci accepte, la connexion est établie et les 2 parties disposent chacune d'un objet de type **Socket** qui représente l'autre partie. Ces objets vont servir pour créer des flux d'entrée/sortie pour communiquer (envoyer et recevoir des données).

Serveur	Client

`ServerSocket ss = new ServerSocket(3845);`
`Socket s1 = ss.accept();`
 	
`//Connexion établie. s1 représente le client`
	`Socket s2 = new Socket(adresse_serveur,3845);`
	
	`//Connexion établie. s2 représente le serveur`

Pour la communication, on utilise les flux d'entrée/sortie, les mêmes qu'on utilise pour lire et écrire des données à la console ou dans des fichiers sur disques.

Comme dans ce travail les programmes ne s'échangent que du texte, on va se limiter aux 2 classes de flux **PrintWriter** et **BufferedReader**.

Un objet de type **PrintWriter** se comporte exactement comme le flux prédéfini **System.out**. On utilisera donc sa méthode **println()** pour envoyer du texte. Chaque appel à cette méthode sera suivi d'un appel à la méthode **flush()** pour "*flusher*" le *buffer* du flux.

Un objet de type **BufferedReader** fournit la méthode **readLine()** pour lire une ligne de texte.

Serveur

```
BufferedReader br1 = new BufferedReader(new
InputStreamReader(s1.getInputStream()));

PrintWriter pw1 = new
       PrintWriter(s1.getOutputStream());

//pw1 sert à envoyer du texte au client.
//br1 sert à lire le texte provenant du
//client
```

Client

```
BufferedReader br2 = new BufferedReader(new
InputStreamReader(s2.getInputStream()));

PrintWriter pw2 = new
       PrintWriter(s2.getOutputStream());

//pw2 sert à envoyer du texte au serveur.
//br2 sert à lire le texte provenant du
//serveur
```

Exemple : un serveur simple et un client simple

Le fichier ***SocketsExempleSimple.zip*** est un projet *IntelliJ* qui contient 2 programmes :

- ***ServeurSimple*** : un programme serveur qui renvoie l'inverse du message qu'il reçoit;
- ***ClientSimple*** : un programme client qui envoie des messages au serveur.

Le code des 2 programmes est fourni dans l'**annexe 2**.

Vous êtes invités à essayer ces programmes pour *palper* le fonctionnement d'applications client/serveur.

Nécessité d'un protocole de communication

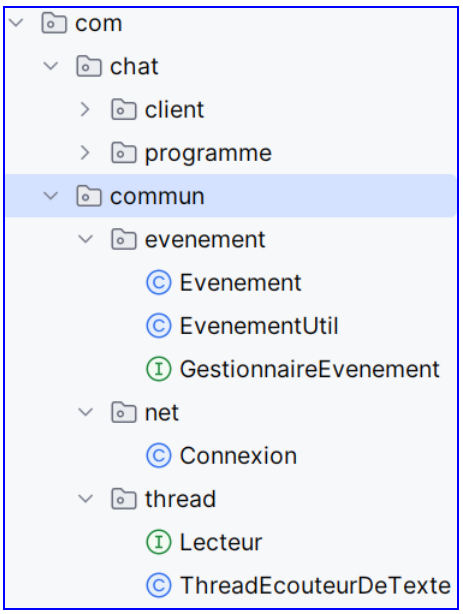
Pour se comprendre, le client et le serveur doivent se mettre d'accord sur un ensemble de commandes qu'ils pourront utiliser ainsi que la signification de ces commandes. Cet ensemble de commandes constitue ce qu'on appelle "**protocole de communication**". Par exemple, un serveur web et un navigateur utilisent le protocole standard HTTP (*HyperText Transfer Protocol*). Un serveur de courriel et un client de courriel (comme Outlook) utilisent le protocole standard SMTP (*Send Mail Transfer Protocol*).

Pour notre architecture client/serveur, on va définir **notre propre protocole de communication**. Les commandes de ce protocole sont fournies dans l'**annexe 1**.

Code fourni

Trois (3) programmes sont fournis sous forme de projets *IntelliJ* : ***ChatServer***, ***ChatClient*** et ***TicTacToe***. Ce sont les points de départ du *TP*.

Les classes et interfaces des sous-packages du package **`com.commun`** sont strictement identiques dans ***ChatServer*** et ***ChatClient*** :



La section **Programmes de départ** vous explique comment fonctionnent les programmes ***ChatServer*** et ***ChatClient***.

Dans le cadre de ce travail, vous **n'avez pas à manipuler directement** toutes les notions de programmation réseau utilisées par les programmes. Vous allez plutôt utiliser les 3 classes **Connexion**, **ServeurChat** et **ClientChat** qui encapsulent tous les détails de la programmation réseau.

Classe *Connexion*

Encapsule un objet ***Socket*** et les flux d'entrée/sortie permettant la communication avec le socket. Elle encapsule aussi l'alias de l'utilisateur. Un objet **Connexion** représente une extrémité d'une connexion entre un client et un serveur. Il en

découle que :

- Un client possède une instance de **Connexion** lui permettant de communiquer avec le serveur;
- Un serveur possède autant d'instances de **Connexion** qu'il y a de clients connectés au serveur. Ces instances sont stockées dans l'attribut **connectes** (de type Vector<Connexion>) de la classe **Serveur**.

Vous n'avez pas à connaître les détails d'implémentation de cette classe, même s'ils ne sont pas sorciers. Vous aurez plutôt à manipuler ces objets à travers les méthodes publiques, principalement :

- `public void envoyer(String texte)` : envoie un texte sur la connexion réseau.
- `public String getAlias()` : retourne l'alias de l'utilisateur à qui correspond la connexion.
- `public void setAlias(String alias)` : attribut un alias à la connexion.
- `public String getAvailableText()` : récupère le texte arrivé sur la connexion réseau. Retourne une chaîne vide si aucun n'est arrivé.
- `public boolean close()` : ferme la connexion réseau.

Classes *Serveur* et *ServeurChat*

Vous aurez à travailler avec et à compléter la classe **ServeurChat** pour implémenter les fonctionnalités côté serveur. La classe **ServeurChat** dérive de la classe **Serveur** qui contient le nécessaire pour interagir avec les clients.

Vous n'aurez pas à connaître ni toucher le contenu de la classe **Serveur**.

Classes *Client* et *ClientChat*

Le programme client fourni utilise les classes Client et ClientChat pour interagir avec le serveur. Le programme utilise essentiellement les méthodes suivantes de la classe Client :

- `public boolean connecter()` : se connecte au serveur.
- `public boolean deconnecter()` : se déconnecte du serveur.
- `public void envoyer(String s)` : envoie un texte sur la connexion vers le serveur.

Comme dans le TP 1 vous n'avez pas à modifier le programme client. Vous n'aurez ni à utiliser ni à comprendre ces 2 classes.

Programmation multithread

Comment le serveur arrive-t-il à écouter l'arrivée de nouvelles connexions tout en surveillant l'arrivée de commandes des clients connectés et en interagissant avec l'utilisateur ?

Réponse : Le serveur utilise la **programmation multithread** pour effectuer plusieurs traitements en parallèle. Chaque **thread** est un processus qui s'exécute en parallèle avec le reste du programme.

En *Java*, la méthode principale d'un thread est la méthode **run()**.

Concrètement, le serveur utilise :

- Le **thread par défaut** dans lequel s'exécute la méthode *main()*;
- Un thread de type **ThreadEcouteurDeConnexions** qui surveille constamment l'arrivée de demandes de connexion;
- Un thread de type **ThreadEcouteurDeTexte** qui surveille constamment l'arrivée de messages/commandes des clients connectés;
- Un thread *anonyme* qui interagit avec les nouveaux connectés pour récolter leur alias.

De son côté, le client utilise :

- Le **thread par défaut** dans lequel s'exécute la méthode *main()*;
- Un thread de type **ThreadEcouteurDeTexte** qui surveille constamment l'arrivée de messages/commandes du serveur.

Dans le cadre de ce TP, **vous n'aurez pas** à faire ni à comprendre la programmation multithread. **Tout le code est déjà en place**.

Un mot sur les accès concurrents

La gestion des accès concurrents **ne fait pas partie du cours** et ne reçoit pas toute l'attention nécessaire dans ce travail. Le client et le serveur fonctionnent sans problème sur un même poste de travail. Mais, **il n'est pas garanti** que des problèmes d'accès concurrents ne surviendraient pas lorsque ces programmes sont lancés dans un contexte réel où des dizaines de clients accèdent au serveur simultanément à partir d'ordinateurs différents.

Introduction à la programmation événementielle

Avertissement

Cette section vous explique les **principes de la programmation événementielle**. Les mécanismes utilisés dans ce TP sont déjà fournis et en place.
La compréhension de ces principes n'est pas nécessaire pour ce TP mais est utile pour comprendre le fonctionnement des applications exploitant la programmation événementielle.

Événement

Un événement dans le monde de la programmation est quelque chose qui se produit lors de l'exécution d'un programme et auquel le programme peut éventuellement réagir. Voici quelques exemples d'événements :

- Dans le monde des applications graphiques :
  - L'utilisateur clique sur un bouton ou un item dans un menu graphique;
  - L'utilisateur clique sur un bouton de la souris;
  - L'utilisateur ferme une fenêtre à l'écran;
  - ...
- Dans le monde de la programmation en général :
  - Une connexion réseau est perdue;
  - Le téléchargement d'un fichier est terminé;
  - Un processus en tâche de fond a fini son travail;
  - ...

Un événement est caractérisé par :

- Sa **source** : l'objet sur lequel s'est produit l'événement (quel bouton a été cliqué, quelle fenêtre a été fermée,...);
- Son **type** : simple clic, double clic, ...
- Ses **arguments** : regroupe toutes les informations additionnelles qui décrivent l'événement et dont le programme peut avoir besoin pour **gérer l'événement** correctement.

Dans le contexte de notre travail, un événement correspond à la **réception d'un message** (commande) par un client ou un serveur. Le client reçoit des messages d'un seul serveur uniquement. Par contre, un serveur peut recevoir des messages de différents clients.

La source de l'événement sera l'objet qui a reçu le message. Par exemple, lorsque le serveur reçoit un message/commande, la source sera l'objet **Connexion** correspondant au client qui lui a envoyé ce message/commande.

Le type de l'événement sera la nature de la commande.

Les arguments de l'événement constituent toute autre information qui vient dans le message.

Exemple :

Si le serveur reçoit d'un client le message suivant :

`MSG Bonjour à tous`

alors :

- La **source** de l'événement est l'objet **Connexion** sur lequel le message est arrivé (cet objet permettra, entre autres, d'obtenir l'alias de la personne qui a envoyé le message);
- Le **type** de l'événement est ***MSG*** (qui signifie que c'est un message de chat public, voir le protocole en ***annexe 2***);
- Les **arguments** de l'événement est le texte ***Bonjour à tous***.

Gestion d'événements

Le programme met en place des traitements (méthodes) qui sont exécutés lorsque les événements se produisent. Les objets qui fournissent ces traitements sont des **gestionnaires d'événements**.

Implémentation

Pour concrétiser le mécanisme de gestion d'événements pour notre architecture client/serveur, nous avons définit les classes et interfaces suivantes dans le package ***com.commun.evenement*** :

Classe *Evenement*

La classe ***Evenement*** représente un événement. Son constructeur reçoit la source, le type et les arguments de l'événement et la classe fournit les accesseurs :

```
public class Evenement {
	private Object source;
	private final String type, argument;
	public Evenement(Object source, String type, String argument) {
		this.source = source;
		this.type = type;
		this.argument = argument;
	}
	public Object getSource() {
		return source;
	}
	public String getType() {
		return type;
	}
	public String getArgument() {
		return argument;
	}
}
```

Interface *GestionnaireEvenement*

L'interface ***GestionnaireEvenement*** représente un gestionnaire d'événement. Elle impose une méthode ***traiter()*** à tout gestionnaire d'événements pour gérer l'événement :

```
public interface GestionnaireEvenement {
	void traiter(Evenement evenement);
}
```

Classe utilitaire *EvenementUtil*

Cette classe utilitaire fournit la méthode ***extraireInfosEvenement()*** qui permet de séparer le type et les arguments d'un événement à partir d'un message/commande texte. La méthode retourne un tableau de 2 éléments de type *String*. Le premier

élément contient le type de l'événement et le second contient ses arguments :

```
public static String[] extraireInfosEvenement(String str) {
	str = str.trim();
	if ("".equals(str))
		return new String[]{"", ""};
	else {
		int i = str.indexOf(' ');
		if (i == -1)
			return new String[]{str, ""};
		else
			return new String[]{str.substring(0, i),
					str.substring(i).trim()};
	}
}
```

Classes *GestionnaireEvenementServeur* et *GestionnaireEvenementClient*

Ces 2 classes implémentent l'interface ***GestionnaireEvenement*** et fournissent, dans la méthode ***traiter()***, le code de gestion d'événement pour le serveur et pour le client.

Ces gestionnaires détiennent une référence vers le serveur/client.

Vous allez intervenir beaucoup dans la méthode ***traiter()*** du serveur, dans le cadre de ce TP.

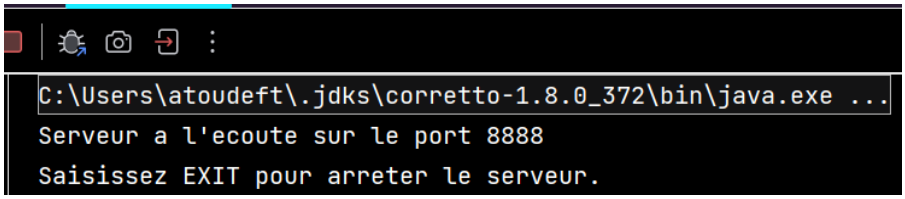
Le client et le serveur détiennent, dans un attribut de type ***GestionnaireEvenement***, une référence vers un gestionnaire d'événements qu'ils alertent chaque fois qu'ils reçoivent un message/commande (dans leurs méthodes ***lire()***).

Programmes de départ

Démarrer les programmes

Le programme serveur ***ChatServer*** doit être démarré en premier pour que le serveur soit à l'écoute avant de démarrer le programme client ***ChatClient***.

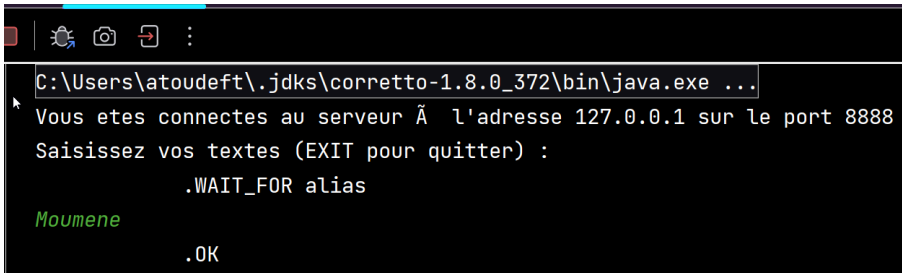
Au démarrage du programme serveur, le serveur démarre et écoute sur le port 8888 l'arrivée de demandes de connexions :



Le port 8888 est le port d'écoute par défaut et est déclaré dans l'interface ***Config***. Pour changer de port, il faut modifier cette valeur dans les interfaces ***Config*** à la fois dans l'application serveur et l'application cliente.

Si on saisit EXIT, le serveur s'arrête après avoir envoyé le message END à tous les clients connectés. Toutes les connexions sont fermées des côtés client et serveur. Notez que les programmes clients ne se terminent pas (une application ne se termine pas forcément lorsqu'elle ferme la connexion avec un serveur).

Lorsqu'on démarre l'application cliente, elle essaie de se connecter au serveur et lorsque la connexion réussit, elle reçoit du serveur le message ***WAIT_FOR alias*** pour dire qu'il faut saisir l'alias qui va identifier l'utilisateur sur le serveur de chat. L'alias ne doit pas être la chaîne vide, ne doit pas être déjà utilisé par un autre utilisateur sur le serveur et doit contenir uniquement les caractères ***a-z***, ***A-Z***, ***-***, ***_*** et ***0-9***.



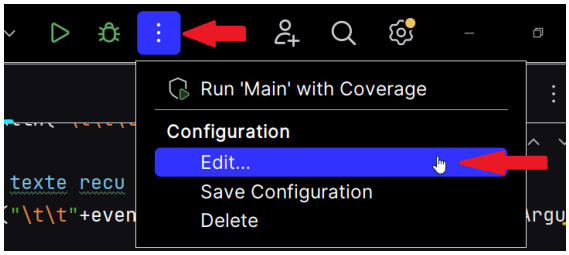
Une fois que l'alias est accepté, l'utilisateur est ajouté à la liste des connectés (attribut ***connectes***, hérité par ***ServeurChat*** de la classe ***Serveur***) et le serveur renvoie tous les messages qui ont déjà été envoyés au salon de chat public. Si aucun message n'a encore été envoyé, le serveur renvoie OK.

Tant que l'alias n'est pas accepté, le serveur ne renvoie pas OK et le client ne peut pas chatter (il peut uniquement saisir un autre alias).

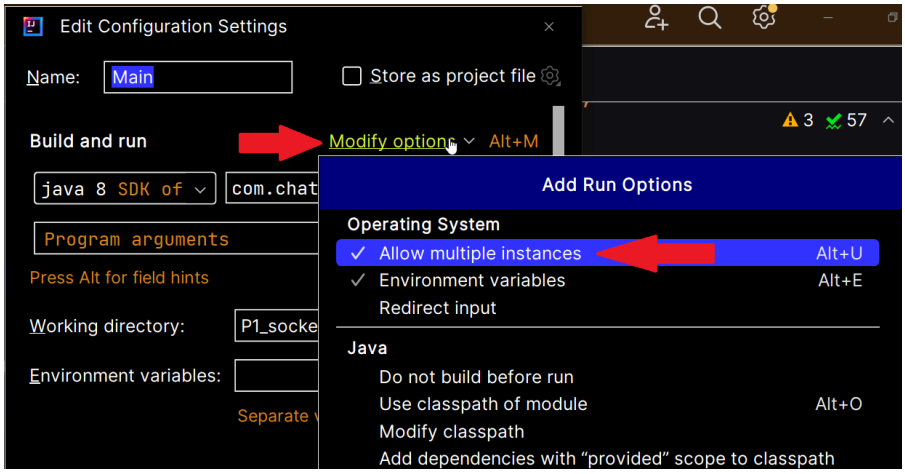
Lancer plusieurs clients simultanément

Le programme client doit être démarré plusieurs fois simultanément pour simuler plusieurs clients connectés au serveur.

Pour démarrer un programme plusieurs fois simultanément sous *IntelliJ*, il faut éditer la configuration d'exécution :



Ensuite, dans la fenêtre qui apparaît, dérouler l'item **Modify options** et cocher l'option **Allow multiple instances** :

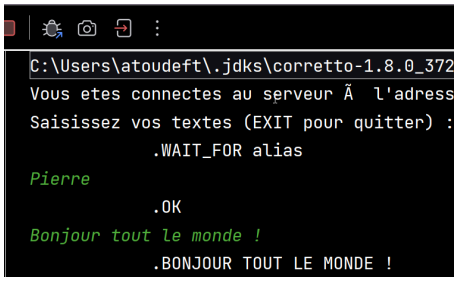


Démarrez plusieurs fois l'application cliente pour avoir plusieurs personnes connectées au serveur, chacune avec un alias différent.

À ce stade, l'utilisateur dispose des commandes suivantes :

- **LIST** : obtient la liste des alias des personnes connectées;

- **EXIT** : ferme la connexion avec le serveur et termine le programme client. Le serveur retire la personne de la liste des connectés;
- Tout autre texte saisi est renvoyé en majuscules par le serveur (traitement par défaut choisi au hasard) :



Traitement des commandes par le serveur

Le serveur traite les commandes des clients dans la structure **switch..case** de la méthode ***traiter()*** de la classe ***GestionnaireEvenementServeur***. C'est à cet endroit que vous allez gérer les futures commandes du client :

```
public void traiter(Evenement evenement) {
   ...
   switch (typeEvenement) {
       case "EXIT": //Ferme la connexion avec le client qui a envoyé "EXIT":
              cnx.envoyer("END");
              serveur.enlever(cnx);
              cnx.close();
              break;
       case "LIST": //Envoie la liste des alias des personnes connectées :
              cnx.envoyer("LIST " + serveur.list());
              break;
       //Ajoutez ici d'autres case pour gérer d'autres commandes du client.
       default: //Renvoyer le texte reçu convertit en majuscules :
              msg = (evenement.getType() + " "
                           + evenement.getArgument()).toUpperCase();
              cnx.envoyer(msg);
       }
   ...
}
```

Dans cette méthode :

- **cnx** (*Connexion*) est l'objet représentant l'utilisateur qui a envoyé la commande;

- **serveur** (*ServeurChat*) est le serveur de chat.
- **evenement** (*Evenement*) est l'événement correspondant à la réception de la commande par le serveur.
- **typeEvenement** (*String*) est le type de l'événement et correspond à la commande reçue.

Traitement des commandes par le client

De manière symétrique, le client traite les réponses du serveur dans la structure **switch..case** de la méthode ***traiter()*** de la classe ***GestionnaireEvenementClient***. Dans ce TP1, vous n'aurez pas à modifier l'application cliente. Le *switch..case* traite déjà toutes les commandes reçues du serveur :

```
public void traiter(Evenement evenement) {
   ...
   switch (typeEvenement) {
      /******************* COMMANDES GÉNÉRALES *******************/
      case "END" : //Le serveur demande de fermer la connexion
            client.deconnecter(); //On ferme la connexion
            break;
      case "LIST" : //Le serveur a renvoyé la liste des connectés
            arg = evenement.getArgument();
            membres = arg.split(":");
            System.out.println("\t\t"+membres.length+" personnes dans le salon :");
            for (String s:membres)
               System.out.println("\t\t\t- "+s);
            break;
      /******************* CHAT PUBLIC *******************/
      case "HIST" : //Le serveur a renvoyé l'historique des messages du chat public
            arg = evenement.getArgument();
            membres = arg.split("\n");
            for (String s:membres)
               System.out.println("\t\t\t."+s);
            break;
      /******************* CHAT PRIVÉ *******************/
      case "JOIN" :
            arg = evenement.getArgument();
```

```
            System.out.println(arg + " vous a envoyé une invitation à un chat "
                               + "privé (JOIN/DECLINE alias pour accepter ou "
                               + "refuser)");
            break;
      case "JOINOK" :
            arg = evenement.getArgument();
            System.out.println(arg + " Vous êtes en chat privé avec "+arg
                               +" (PRV alias msg pour lui envoyer "
                               +"un message en privé)");
            break;
      //...etc
      default: //Afficher le texte recu du serveur :
            System.out.println("\t\t\t."+evenement.getType()
                               +" "+evenement.getArgument());
   }
```

Notez bien :

- Dans le serveur de chat (classe ***ServeurChat***), vous disposez de l'attribut ***connectes*** (***Vector<Connexion>***) contenant les objets ***Connexion*** représentant tous les utilisateurs actuellement connectés au serveur. L'attribut ***connectes*** est hérité de la classe ***Serveur***.
- Un objet de type ***Connexion*** fournit la méthode ***envoyer()*** qui permet d'envoyer du texte à l'utilisateur représenté par cet objet ***Connexion***.

Chatter en public et en privé

Question 1 : Chatter dans le salon public

Vous allez faire en sorte que lorsqu'un client envoie la commande suivante :

MSG message

le serveur de chat renvoie un texte de la forme :

alias>>message

à toutes les personnes connectées sauf celui qui l'a envoyé (***alias*** est l'alias de la

personne qui a envoyé le message).

Par exemple, si l'utilisateur d'alias ***Annie*** envoie la commande :

```
MSG Bonjour à tous
```

le serveur de chat envoie le texte :

```
Annie>>Bonjour à tous
```

à tout le monde sauf à ***Annie***.

Pour cela, vous devez :

1.1. Ajouter au serveur de chat la méthode ayant l'en-tête suivante :

```
public void envoyerATousSauf(String str, String aliasExpediteur)
```

Cette méthode va envoyer la chaîne ***str*** à tous les utilisateurs connectés sauf à celui qui a l'alias ***aliasExpediteur***.

1.2. Traiter les commandes **MSG** des clients. Pour cela, ajoutez dans la structure **switch..case** du gestionnaire d'événements du serveur un `case "MSG"` dans lequel :

- Vous récupérez l'alias de l'expéditeur du message;
- Vous récupérez le texte du message (c'est l'argument de l'événement);
- Vous appelez la méthode précédente pour envoyer le message à tous sauf à l'expéditeur.

Testez vos programmes. Lorsqu'un client envoie un message avec la commande MSG, tous les autres devraient le recevoir précédé de l'alias de l'expéditeur et la paire de symboles **>>**

Question 2 : Historique des messages de chat

Vous allez faire en sorte que :

- Le serveur mémorise tous les messages envoyés par les personnes connectées ;
- Le serveur envoie à toute personne nouvellement connectée les messages

mémorisés.

Pour cela :

2.1. Ajouter au serveur de chat un attribut ***historique*** de type ***Vector*** de *String* pour stocker tous les messages envoyés au salon de chat public. Ces messages seront stockés sous forme de chaînes de caractères ayant la forme :

alias>>message

où ***message*** est le message envoyé par l'utilisateur ***alias***.

2.2. Complétez la méthode ***historique()*** du serveur de chat de manière à retourner une chaîne de caractères contenant tous les messages de l'historique à raison de un message par ligne (les messages sont séparés par le caractère ***'\n'***).

2.3. Ajoutez au serveur de chat une méthode ***ajouterHistorique()*** qui reçoit une chaîne de caractères (qui représente un message) et qui l'ajoute à l'historique des messages.

Appelez cette méthode dans le `case "MSG"` du gestionnaire du serveur pour ajouter chaque message reçu d'un client dans l'historique des messages.

Le serveur de chat contient déjà, dans sa méthode ***ajouter()***, le code qui permet d'envoyer à toute personne nouvellement connectée l'historique des messages dans le format :

HIST historique

historique est la chaîne de caractères retournée par la méthode *historique()* du point 2.2.

Notez que dans la structure **switch..case** du gestionnaire d'événements du client, un `case "HIST"` contient déjà le code qui affiche l'historique des messages reçu du serveur.

Testez vos programmes. Lorsqu'un client se connecte, il reçoit l'historique des messages postés dans le salon public. S'il n'y a pas encore de messages, le client reçoit OK.

Question 3 : Chatter en privé

On désire maintenant permettre à des personnes connectées au serveur d'envoyer des invitations à d'autres personnes pour chatter en privé. Une personne peut refuser l'invitation qui lui a été envoyée ou l'accepter. Si l'invitation est acceptée, un salon privé est créé pour permettre aux 2 personnes de chatter en privé. Si l'invitation est refusée, elle est supprimée.

Notez bien que :

- Une personne qui chatte en privé ne quitte pas le salon public (elle peut donc continuer à chatter dans le salon public).
- Une personne peut chatter dans plusieurs salons privés.
- Une personne ne peut pas chatter avec elle-même.
- Un salon privé ne garde pas l'historique des messages échangés.
- On ne peut pas avoir simultanément plusieurs invitations identiques (même hôte et même invité).
- On ne peut pas avoir simultanément plusieurs salons privés avec les mêmes personnes (même si l'hôte et l'invité sont inversés).
- Un salon privé disparaît dès qu'une des 2 personnes quitte le salon ou se déconnecte du serveur.

Vous allez donc, dans l'application serveur :

3.1. Créer une classe **Invitation** pour représenter des invitations à chatter en privé. La classe aura 2 attributs de type *String* contenant les alias de l'hôte et de l'invité.

3.2. Créer une classe **SalonPrive** pour représenter des salons de chat privés. La classe sera identique à la classe **Invitation** (on aurait pu avoir une seule des 2 classes mais en prévision d'une évolution future ...).

Redéfinir la méthode equals() pour définir l'égalité entre 2 salons privés. Deux salons privés sont égaux s'ils regroupent les mêmes alias, peu importent leurs rôles (hôte ou invité).

3.3. Permettre à un utilisateur ***alias1*** d'envoyer une commande ***JOIN alias2*** afin d'inviter la personne ***alias2*** à chatter en privé ou d'accepter l'invitation qui lui a été préalablement envoyée par ***alias2***. Le gestionnaire d'événement du serveur gère

cette commande en :

1. Vérifiant si ***alias2*** a déjà envoyé une invitation à ***alias1***, alors un salon privé est créé pour les 2 utilisateurs;
2. Sinon, une invitation est créée et ***alias2*** est informé de l'arrivée d'une invitation de ***alias1***.

3.4. Permettre à un utilisateur ***alias1*** d'envoyer une commande ***DECLINE alias2*** afin de refuser une invitation provenant de la personne ***alias2***. Le gestionnaire d'événement du serveur gère cette commande en :

1. Supprimant l'invitation;
2. Informant ***alias2*** que alias1 a refusé son invitation.

La commande DECLINE servira aussi à un utilisateur ***alias1*** d'annuler une invitation qu'il a préalablement envoyée à ***alias2***.

3.5. Permettre à un utilisateur ***alias1*** d'envoyer une commande ***INV*** afin d'obtenir la liste de tous les alias des personnes qui lui ont envoyé des invitations.

3.6. Permettre à un utilisateur ***alias1*** d'envoyer une commande ***PRV alias2 message*** afin d'envoyer un message privé à ***alias2***, s'il y a un salon privé contenant ***alias1*** et ***alias2***.

3.7. Permettre à un utilisateur ***alias1*** d'envoyer une commande ***QUIT alias2*** afin de quitter le salon privé dans lequel il se trouve avec ***alias2***.

INF111 Programmation orientée objet
TP 1
Auteur : Abdelmoumène Toudeft
Groupes : tous
Enseignant-e-s : El Hachemi Alikacem,
Frédéric Simard et
Abdelmoumène Toudeft

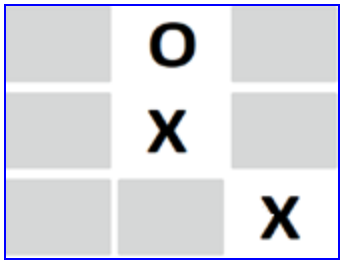
Département
des Enseignements
Généraux

Jeu de Tic-Tac-Toe

Le programme de tic-tac-toe incomplet vous est fourni (***TicTacToe.zip***). Vous devez le compléter. Dans le TP 2, vous allez l'intégrer dans l'architecture client/serveur pour jouer en réseau.

Vous n'avez pas à concevoir le jeu. La conception vous est fournie et imposée. Vous avez juste à compléter les classes en répondant aux questions ci-dessous.

Le jeu de tic-tac-toe se joue à 2. Chacun des 2 joueurs utilisent un symbole (X ou O) qu'ils placent à tour de rôle sur une grille 3x3. Le premier joueur qui arrive à aligner 3 de ses symboles, en ligne, en colonne ou en diagonale, a gagné la partie.



Exemple de grille de tic-tac-toe en cours de partie.
L'exemple montre une interface graphique de la grille.

Code fourni

L'énumération **Symbole** définit les 2 symboles utilisés par les 2 joueurs :

```
public enum Symbole { X, O }
```

L'énumération **StatutPartie** définit les 4 états possibles d'une partie de jeu :

```
public enum StatutPartie { EN_COURS, NULLE, X_GAGNE, O_GAGNE }
```

La classe **Position** représente une cellule de la grille de Tic-Tac-Toe. Cette classe est incomplète et vous devez la compléter (voir questions ci-dessous).

La classe **Plateau** représente une grille de Tic-Tac-Toe. Un objet de ce type encapsule un tableau 3x3 de symboles. Cette classe est incomplète et vous devez la compléter (voir questions ci-dessous).

La classe **Coup** représente un coup réalisé par un des joueurs. Il décrit le placement d'un symbole dans une position de la grille de Tic-Tac-Toe. Cette classe est incomplète et vous devez la compléter (voir questions ci-dessous).

La classe **Partie** représente une partie de jeu de Tic-Tac-Toe. Cette classe est incomplète et vous devez la compléter (voir questions ci-dessous).

La classe **Demo** est un programme de démonstration qui vous permet de tester le jeu afin de vous assurer qu'il fonctionne correctement.

La classe **MethodeNonImplementeeException** est utilisée dans le cadre du TP pour vous indiquer quelle méthode vous n'avez pas encore implémentée. Lors de l'exécution du programme **Demo**, si une méthode non encore complétée est appelée, une exception de ce type est déclenchée et un message du format suivant est affiché dans la console de l'environnement :

******* Vous n'avez pas encore implemente la methode : jouer() de la classe com.atoudeft.tictactoe.Partie**

Les méthodes à compléter contiennent l'instruction suivante qui déclenche l'exception :

```
throw new MethodeNonImplementeeException(
            "***** Vous n'avez pas encore implemente la methode : "
            +Thread.currentThread().getStackTrace()[1].getMethodName()
            +"() de la classe "+this.getClass().getName());
```

Vous devez supprimer l'instruction une fois que vous avez complété la méthode.

Question 4 : Compléter le jeu de tic-tac-toe

On vous demande de compléter les classes fournies pour le jeu de Tic-Tac-Toe.

Vous n'avez pas le droit de modifier le programme **Demo**. Celui-ci doit fonctionner correctement à la fin de votre implémentation.

4.1. Faites en sorte que lorsqu'on affiche une partie de tic-tac-toe avec une instruction comme celle-ci :

```
System.out.println(partie);
```

que le plateau, le joueur courant et l'état de la partie s'affichent dans le format suivant :

```
O . O
. X .
. . X
Joueur Courant : X
Etat : EN_COURS
```

4.2. Définir l'égalité entre des positions. Deux positions sont égales si elles correspondent à la même ligne et à la même colonne.

4.3. Complétez la méthode **placer()** de la classe **Plateau**. Cette méthode tente de réaliser le coup reçu en paramètre sur le plateau. Si la case spécifiée dans le coup est pleine, la méthode retourne *false*. Sinon, elle place le symbole dans la case et incrémente le nombre de case remplies.

4.4. Complétez la méthode **ligneGagnante()** de la classe **Plateau**. Cette méthode vérifie si un des joueurs a gagné, c'est-à-dire qu'il a aligné son symbole sur une ligne, une colonne ou une diagonale. Cette méthode retourne une liste (ArrayList) contenant les 3 positions gagnantes. Si aucun joueur n'a gagné, la méthode retourne une liste vide (qu'on peut obtenir avec l'appel `Collections.emptyList()`).

La variable **`lignes`** déjà déclarée dans la méthode contient toutes les lignes gagnantes possibles.

4.5. Complétez la méthode **mettreAJourStatutApresCoup()** de la classe **Partie**. Cette méthode vérifie s'il y a une ligne gagnante, elle met le gagnant dans le statut de la partie. Si le plateau est plein, elle indique que la partie est nulle. Dans tous les autres cas, la méthode ne fait rien.

4.6. Complétez la méthode **jouer()** de la classe **Partie**. Cette méthode reçoit un symbole et une position et essaie de placer le symbole dans la case correspondant à la position sur le plateau. La méthode retourne *false* si la partie est terminée ou si le symbole ne correspond pas à celui du joueur courant ou si la case est déjà pleine. Dans tous les autres cas, la méthode place le coup demandé, met à jour le statut de la partie, change de joueur courant et retourne true.

Annexe 1 - Protocole de communication

Commandes utilisées par le client

Ces commandes sont gérées par le gestionnaire d'événements du serveur.

Commande	Arguments	Description	Exemple
EXIT		Se déconnecte du serveur	EXIT
	Le serveur ferme la connexion avec le client et le retire de la liste des connectés.		
LIST		Demande la liste des connectés	LIST
	Le serveur renvoie la liste des alias des personnes connectées dans le format *alias1:alias2:alias3...*		
MSG	message	Envoie un message au salon de chat public	MSG Bonjour tout le monde
	Le serveur diffuse le message à toutes les autres personnes connectées dans le format *alias>>message*		
JOIN	alias	Envoie ou accepte une invitation à chatter en privé.	JOIN Annie
	Si l'utilisateur a déjà reçu une invitation de l'alias, le serveur crée un salon de chat privé entre les 2 personnes (envoie *JOINOK* aux 2). Sinon, le serveur crée une nouvelle invitation et informe le destinataire (en lui envoyant JOIN). Ensuite : - Si le destinataire de l'invitation l'accepte, un salon privé est créé; - Sinon, l'invitation est supprimée.		
DECLINE	alias	Refuse l'invitation de l'alias.	DECLINE Annie
	Le serveur supprime l'invitation envoyée par l'alias (ou envoyée à alias - optionnel).		
PRV	alias message	Envoie un message privé.	PRV Annie Allo toi
	Le serveur envoie le message à l'alias.		
INV		Demande au serveur la liste des invitations reçues.	INV
	Le serveur envoie la liste des invitations reçues par l'utilisateur avec la commande INV.		
QUIT	alias	Quitte un salon privé.	QUIT Annie
	Le serveur détruit le salon privé et informe alias en lui envoyant QUIT.		

Commandes utilisées par le serveur

Ces commandes sont gérées par le gestionnaire d'événements du client.

Commande	Arguments	Description	Exemple
END		Demande au client de se déconnecter	END
	Le client ferme la connexion avec le serveur (note : le programme client continue de s'exécuter).		
LIST	liste des alias séparés par **:**	Envoie la liste des connectés	LIST Moumene:Annie
	Le client reçoit la liste des alias des personnes connectées dans le format *alias1:alias2:alias3...*		
HIST	historique des messages	Envoie les messages déjà envoyés au salon de chat public, séparés par '\n'.	HIST msg1\nmsg2\nmsg3
	Le client reçoit la liste des messages du salon public dans le format *msg1\nmsg2\nmsg3...*		
JOIN	alias	Informe un client de la réception d'une invitation à un chat privé.	JOIN Annie
INV	liste des alias séparés par **:**	Envoie la liste des invitations à un chat privé.	INV Moumene:Annie
JOINOK	alias	Valide le démarrage d'un chat privé avec alias.	JOINOK Annie
DECLINE	alias	Informe le client que alias a refusé son invitation.	DECLINE Annie
QUIT	alias	Informe le client que alias a quitté le salon privé.	QUIT alias

Annexe 2 - Exemple de serveur et client simples

Voici un exemple d'un serveur simple et un client simple. Le serveur accepte un seul client. Il renvoie l'inverse du message envoyé par le client jusqu'à ce que ce dernier envoie le mot FIN.

Un serveur simple qui renvoie l'inverse du message qu'il reçoit

```
final int PORT_ECOUTE_SERVEUR = 1234;
ServerSocket serverSocket=null;
Socket socket=null;
BufferedReader br = null;
PrintWriter pw=null;
String messageRecu, messageInverse;

serverSocket = new ServerSocket(PORT_ECOUTE_SERVEUR);

// Attente d'une connexion :
System.out.println("SERVEUR : Attente de connexion...");
socket = serverSocket.accept();
System.out.println("SERVEUR : Connexion acceptee.");

//Création des flux d'entrée/sortie pour la communication avec le client :
br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
pw = new PrintWriter(socket.getOutputStream());

//Lecture des messages provenant du client :
messageRecu = br.readLine();
while (!"FIN".equals(messageRecu)) {
	System.out.println("SERVEUR: recue : " + messageRecu);

	//Création de la réponse au client (messageRecu inversé) :
	messageInverse = new StringBuilder(messageRecu).reverse().toString();

	//Envoi de la réponse au client :
	pw.println(messageInverse);
	pw.flush();

	//Attente et lecture du message suivant :
	messageRecu = br.readLine();
}
```

Un client simple qui communique avec le serveur

```
final String ADRESSE_IP_SERVEUR = "127.0.0.1";
final int PORT_ECOUTE_SERVEUR = 1234;

Scanner clavier = new Scanner(System.in);

Socket socket = null;
BufferedInputStream bis;
BufferedReader br = null;
PrintWriter pw = null;
String message;
String reponseRecu;

//Connexion au serveur :
socket = new Socket(ADRESSE_IP_SERVEUR,PORT_ECOUTE_SERVEUR);
System.out.println("Connexion établie avec le serveur...");

//Création des flux d'entrée/sortie pour la communication avec le serveur :
br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
pw = new PrintWriter(socket.getOutputStream());

//Envoi des messages au serveur :
System.out.print("Saisissez votre message (FIN pour terminer) : ");
message = clavier.nextLine();
pw.println(message);
pw.flush();

while (!"FIN".equals(message)) {
	//Attente et lecture de la réponse du serveur :
	reponseRecu = br.readLine();
	System.out.println("Réponse du serveur : "+reponseRecu);

	//Envoi du message suivant :
	System.out.print("Saisissez votre message (FIN pour terminer) : ");
	message = clavier.nextLine();
	pw.println(message);
	pw.flush();
}
```