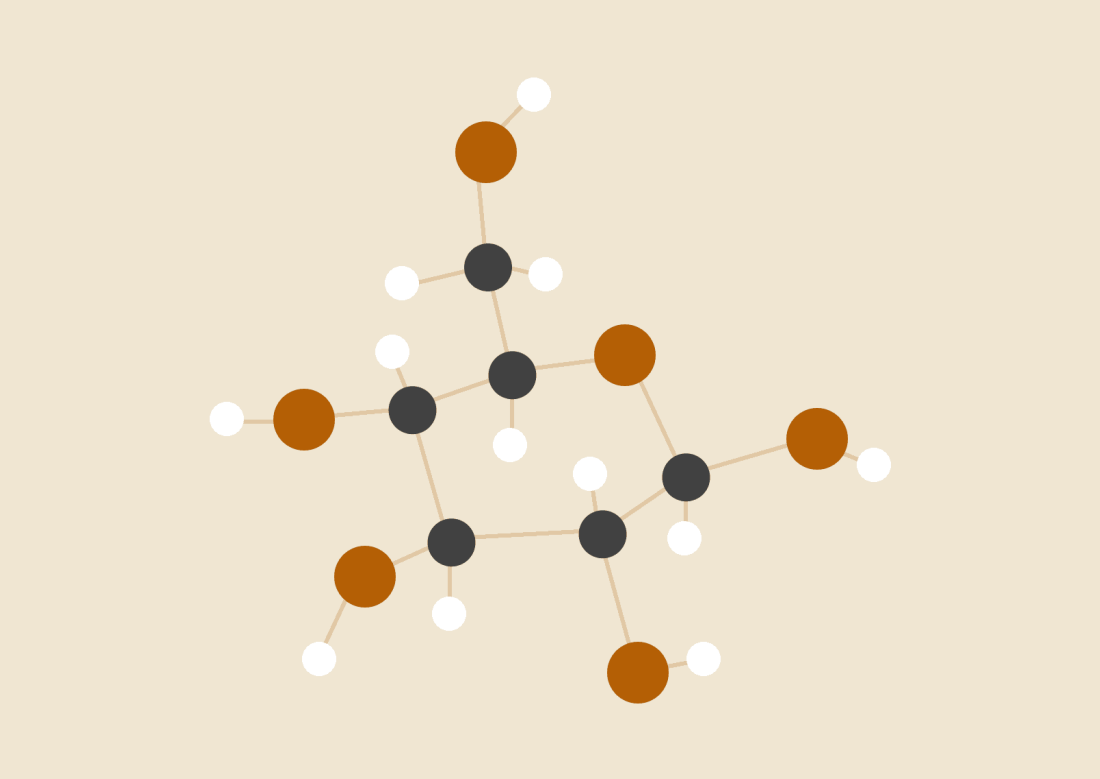
INF111 Programmation orientée objet

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

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 objets.

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.

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
et Abdelmoumène Toudeft

Département
des Enseignements
Généraux

TP 2

Ce deuxième TP fait suite au TP 1.

Nous travaillerons en **mode graphique** avec *Swing*.

Vous allez commencer à partir des programmes fournis.

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,...).

Directives pour le TP 2

- ➔ Travail avec **les mêmes équipes que le TP 1** (tout changement doit être motivé);
- ➔ Le travail individuel est refusé. Ceux qui n'ont pas fait d'équipe au TP1 doivent le faire pour ce TP2.
- ➔ Date limite de remise du travail : **14 décembre à 23h59**
- ➔ Le travail doit être remis sur *Github* (tout le monde doit avoir un compte *Github*);
- ➔ Deux (2) programmes sous fournis sous forme de projets *IntelliJ* : ***ChatServer2*** et ***ChatClient2***. Ce sont les points de départ du *TP 2*. Ces programmes incluent la solution du *TP 1* mais contiennent aussi des ajustements nécessaires pour le *TP 2*.
- ➔ Vous travaillerez **uniquement** sur le programme ***ChatClient2***. Vous n'aurez pas à toucher au programme serveur ***ChatServer2***. Ce dernier doit être en fonction avant de lancer le programme client.
- ➔ Vous devez ajouter le programme client à votre référentiel *Github* du TP 1. Vous **ne devez pas** utiliser un autre référentiel.

Ajustements par rapport au TP 1

Pour les besoins de ce TP, plusieurs ajustements ont été apportés au code du TP 1. Cette section résume les principales modifications apportées au serveur et au client.

Ajustements au serveur

Toutes les réponses du serveur sont précédées d'une commande (type d'événement). Par exemple, dans le TP 2, le client **a besoin** de savoir si un message de chat est public ou privé pour l'afficher au bon endroit dans l'interface graphique. Le serveur précède donc son envoi par une commande **`MSG`** ou **`PRV`**.

Lorsqu'un client d'alias *Annie* se connecte, le serveur doit informer tous les connectés pour qu'ils ajoutent ce client de la liste des connectés. Le serveur est donc ajusté pour envoyer la commande :

NEW Annie

à tous, sauf à *Annie*. Pour cela, l'instruction suivante est ajoutée à la méthode **ajouter()** de **ServerChat** :

```
envoyerATousSauf("NEW "+connexion.getAlias(), connexion.getAlias());
```

De manière similaire, lorsqu'un client d'alias *Annie* se déconnecte, en envoyant la commande **EXIT**, le serveur doit informer tous les connectés pour qu'ils retirent ce client de la liste des connectés. Pour cela, le serveur est ajusté pour envoyer la commande

EXIT Annie

à tous, sauf à Annie. Pour cela, l'instruction suivante est ajoutée dans le **`case "EXIT"`** du gestionnaire d'événements du serveur :

```
serveur.envoyerATousSauf("EXIT "+cnx.getAlias(), cnx.getAlias());
```

Lorsqu'un coup de Tic-Tac-Toe est invalide, le serveur envoie un message dans sa commande INVALID. Ce message sera affiché par le client.

Un attribut **alias** est ajouté dans la classe **ClientChat**, avec son *getter* et son *setter*, pour garder l'alias de l'utilisateur côté client (et l'afficher dans la barre de titre de la fenêtre).

L'**annexe 1** liste les commandes envoyées par le serveur (elles sont traitées dans les **`case`** du **`switch..case`** du gestionnaire d'événement du client).

Un nouveau gestionnaire d'événements pour le client

Le client dans le TP 1 crée lui-même le gestionnaire de ses événements, dès qu'il se connecte. Voici la méthode **connecter()** du client montrant la création du gestionnaire d'événements dans le TP 1 :

```
public boolean connecter() {
    boolean resultat = false;
    if (this.isConnecte()) //deja connecte
        return resultat;
    try {
        Socket socket = new Socket(adrServeur, portServeur);
        connexion = new Connexion(socket);
        this.setAdrServeur(adrServeur);
        this.setPortServeur(portServeur);

        //On cree l'ecouteur d'evenements pour le client :
        gestionnaireEvenementClient = new GestionnaireEvenementClient(this);

        //Démarrer le thread inspecteur de texte:
        vt = new ThreadEcouteurDeTexte( lecteur: this);
        vt.start();  //la methode run() de l'ecouteur de texte s'execute en
                     // parallele avec le reste du programme.
        resultat = true;
        this.setConnecte(true);
    } catch (IOException e) {
        this.deconnecter();
    }
    return resultat;
}
```

Pour le TP 2, nous avons besoin d'un gestionnaire d'événements capable d'interagir avec l'interface graphique. Cependant, le client ne connaît pas (et ne doit pas connaître) l'interface graphique. Dans le contexte d'une architecture

Modèle-Vue-Contrôleur (MVC), le client fait partie du modèle, l'interface graphique fait partie de la vue et le modèle **ne doit rien connaître** de la vue.

La conséquence de tout ça est que le client ne peut pas créer un gestionnaire d'événements qui a besoin de l'interface graphique.

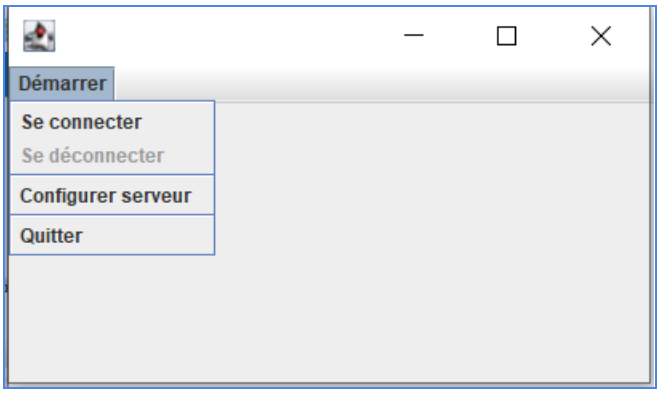
Dans ce TP 2, nous allons utiliser une nouvelle classe : **`GestionnaireEvenementClient2`**. Son constructeur reçoit le client et le panneau principal de l'application. Ce qui lui permet d'interagir avec le client et l'interface graphique.

Ce gestionnaire d'événements est instancié dans la méthode **`initialiserComposants()`** de la classe **`MainFrame`**.

D'autre part, nous avons besoin de réafficher la boîte de dialogue de saisie de l'alias aussi longtemps que l'alias fourni est refusé par le serveur. Nous modifions donc le serveur pour qu'il renvoie la réponse **`WAIT_FOR alias`** lorsque l'alias fourni n'est pas valide. De son côté, le gestionnaire d'événements du client va avoir un **`case "WAIT_FOR"`** pour traiter la réponse et demander à l'interface graphique d'afficher la boîte de dialogue.

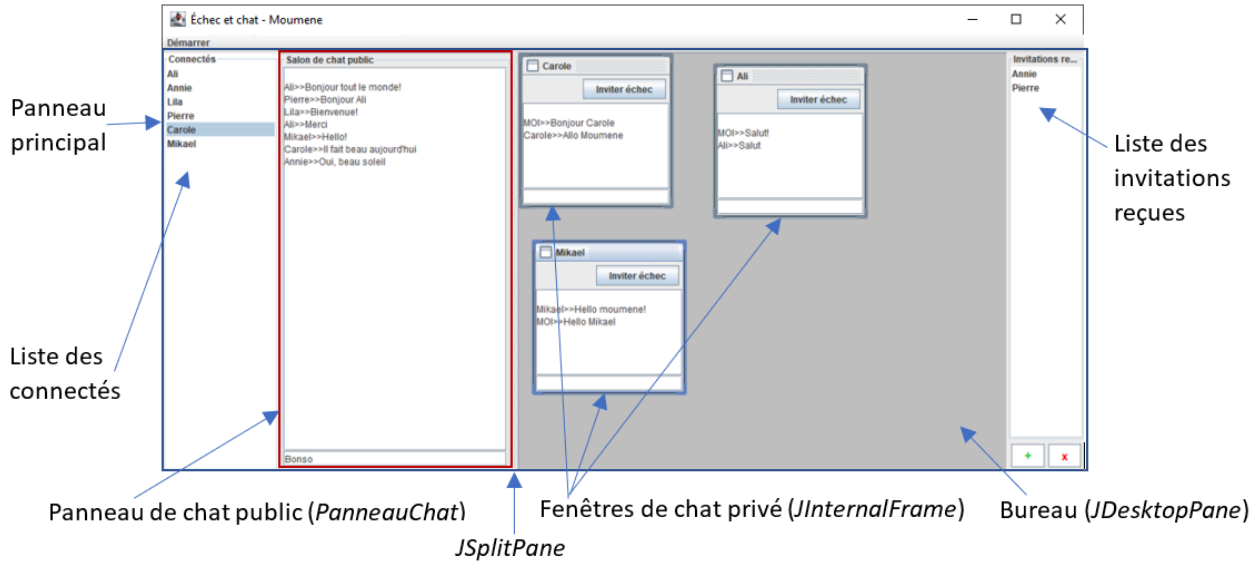
Interface graphique du client

La fenêtre principale du programme **ChatClient2** est un objet de la classe **MainFrame**. Au départ, la fenêtre est vide et présente un menu **Démarrer** qui permet à l'utilisateur de se connecter, de configurer le serveur ou de quitter (voir figure ci-contre).



Dans une des questions, on vous demandera de faire en sorte que les items de menu **Se connecter** et **Se déconnecter** s'activent et se désactivent selon que l'utilisateur est connecté ou non.

Une fois la connexion au serveur acceptée, le panneau principal de l'application apparaît. La figure ci-dessous montre la fenêtre principale de l'application **ChatClient2** à laquelle on désire arriver à la fin du TP :



Le panneau principal, qui montre toute l'interface, apparaît dès que l'utilisateur est connecté au serveur et disparaît dès qu'il se déconnecte.

Le panneau principal, géré par un *BorderLayout*, montre :

- à l'ouest, la liste des connectés (dans un composant *JList*);
- à l'est, un panneau d'invitations (**PanneauInvitations**). Ce panneau montre en haut la liste des invitations reçues (*JList*) et en bas, 2 boutons pour accepter ou refuser une invitation;
- au centre, séparé en 2 par un *JSplitPane*, le panneau de chat public et un **bureau** (*JDesktopPane*) contenant des fenêtres de chat privé (*JInternalFrame*). Chaque fenêtre interne affiche un panneau de chat privé (**PanneauChatPrive**).

Pour voir des exemples sur les composants *Swing*, vous pouvez consulter :

https://www.demo2s.com/java/java-swing-introduction.html

http://www.java2s.com/Code/Java/Swing-JFC/CatalogSwing-JFC.htm

http://www.java2s.com/Code/Java/Swing-Components/CatalogSwing-Components.htm

Programmes de départ

Le programme serveur **ChatServer2** est déjà programmé. Vous n'aurez pas à le modifier.

Le programme client **ChatClient2**, fourni, permet de se connecter et de voir la liste des connectés. Aussi longtemps que l'alias fourni n'est pas valide, la boite de connexion se ré-affiche.

L'alias du client apparaît dans la barre de titre de la fenêtre et la liste des autres connectés apparaît dans le composant *JList* situé à gauche. La liste se rafraîchit automatiquement à l'arrivée ou au départ de connectés.

Si l'utilisateur est connecté, l'item de menu **Quitter** demande une confirmation de déconnexion avant de quitter. Sinon, le programme se termine sans confirmation.

La fenêtre principale (de type **MainFrame**) encapsule :

- Un **ClientChat**;
- Un **PanneauPrincipal**;
- Les items du menu principal (des **JMenuItem**);
- Le gestionnaire d'événements client (de type **GestionnaireEvenementClient2**);
- L'écouteur des événements du menu principal (de type **EcouteurMenuPrincipal**) qui gère les items du menu dans sa méthode **actionPerformed()**.

Le panneau principal (qui apparaît lorsque le client s'est connecté) encapsule :

- Un **ClientChat**;
- Un composant liste (**JList**) affichant les alias des connectés;
- Un modèle de liste (**DefaultListModel**) contenant les alias des connectés (voir ci-après);
- Un panneau de chat public;
- Un panneau pour les invitations;

- Un bureau (**JDesktopPane**) pour afficher les fenêtres de chat privé;
- Un dictionnaire (**Map**) contenant des références vers les panneaux de chat privé (voir ci-après).

Le composant liste (*JList*) et les modèles de listes (*ListModel*)

Nous utiliserons le composant **JList** pour afficher la liste des connectés et la liste des invitations reçues.

Ce composant affiche des données provenant d'une source qui peut être un tableau, un *Vector* ou un modèle de liste.

Lorsque le composant **JList** est alimenté par un tableau ou un **Vector**, chaque changement dans la source (ajout, suppression ou modification de donnée) nécessite qu'on rafraichisse explicitement le composant **JList**.

Par contre, et c'est l'avantage, lorsque le composant **JList** est alimenté par un modèle de liste, les changements provoquent automatiquement le rafraîchissement du **JList** (en réalité, le **JList** observe le modèle et réagit à ses changements).

Un modèle de liste est un objet de type **ListModel**, qui est une interface. L'API *Java* fournit une implémentation par défaut : la classe **DefaultListModel**. Elle est suffisante pour la plupart des utilisations, dont la nôtre dans ce TP.

```
JList jListe1<String>;

DefaultListModel<String> donnees;

donnees = new DefaultListModel<>();

jListe1 = new JList<>(donnees);
```

Tout changement dans `donnees` (ajout, suppression ou modification) va provoquer le rafraîchissement automatique du composant *`jListe1`*.

Les 2 méthodes de *ListModel* que nous utiliserons sont :

- `addElement()`
- `removeElement()`

Nous utiliserons 2 modèles de liste pour stocker les alias des personnes connectées

et les alias des personnes qui ont envoyé une invitation à chatter en privé :

Dans la classe **PanneauPrincipal** :

```
private DefaultListModel<String> connectes;
```

et dans la classe **PanneauInvitations** :

```
private DefaultListModel<String> invitationsRecues;
```

Les dictionnaires (Map)

Pour trouver facilement un panneau de chat privé lors de l'arrivée d'un message privé ou d'une invitation à jouer à Tic-Tac-Toe, nous gardons des références vers ces panneaux dans une structure de données. Nous utilisons à cette occasion un dictionnaire ou carte (**Map**). Un dictionnaire stocke chaque donnée en lui attribuant **une clé** afin de la retrouver facilement.

Map est une interface. On ne peut donc pas l'instancier. **HashMap** et **TreeMap** sont 2 classes concrètes qui implémentent l'interface **Map**. Sans raison particulière, nous choisissons **HashMap**.

Voici un exemple d'un dictionnaire qui stocke des objets *Voiture* avec des clés de type chaîne de caractères :

```
Map<String, Voiture> mesAutos = new HashMap<>();
//Insérer de voitures dans le dictionnaire :
mesAutos.put("ma corolla", new Voiture(...));
Voiture honda = new Voiture(...);
mesAutos.put("ma belle honda", honda);
//Accéder aux voitures du dictionnaire :
Voiture v = mesAutos.get("ma corolla");
if (v!=null) ...
//Vérifier si une clé existe :
if (mesAutos.containsKey("ma corolla")) …
```

```
//Supprimer une donnée à partir de sa clé :
mesAutos.remove("ma corolla");
```

Les principales méthodes de *Map* que nous utiliserons sont donc :

- `put()`
- `get()`
- `remove()`
- `containsKey()`

Nous stockerons chaque panneau de chat privé avec l'alias correspondant comme clé, dans la classe **PanneauPrincipal** :

```
private Map<String, PanneauChatPrive> panneauxPrives;
```

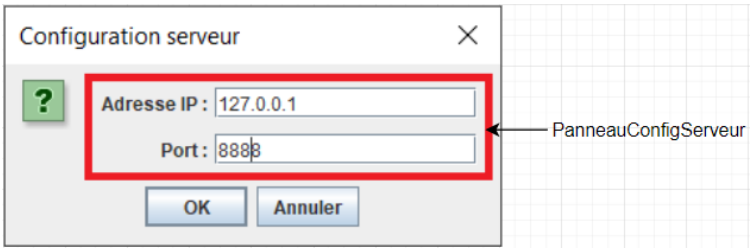
Question 1 : États des items de menu et configuration serveur

Le client est un objet *observable* qui notifie ses observateurs lorsqu'il se connecte ou se déconnecte (méthode **setConnecte()**).

La fenêtre principale (*MainFrame*) est un observateur qui observe le client et réagit à ses changements dans sa méthode **seMettreAJour()**.

1.1. Faites en sorte que l'item de menu **Se connecter** soit désactivé lorsque le client est connecté et ré-activé lorsque le client est déconnecté. Faites en sorte que l'item **Se déconnecter** se comporte inversement.

1.2. Complétez le constructeur de la classe **PanneauConfigServeur** pour qu'il affiche l'adresse IP et le port d'écoute du serveur dans les 2 champs de texte (voir encadré rouge de la figure ci-contre).



1.3. Complétez le `case "CONFIGURER"` de la méthode **actionPerformed()** de la classe **EcouteurMenuPrincipal** pour afficher le panneau de configuration du serveur dans une boîte de confirmation (voir figure précédente). La boite doit se ré-afficher aussi

longtemps que le numéro de port saisi n'est pas un entier entre 1 et 65735 (utilisez une gestion d'exception). Les données saisies sont fournies au client qui les stocke dans ses attributs.

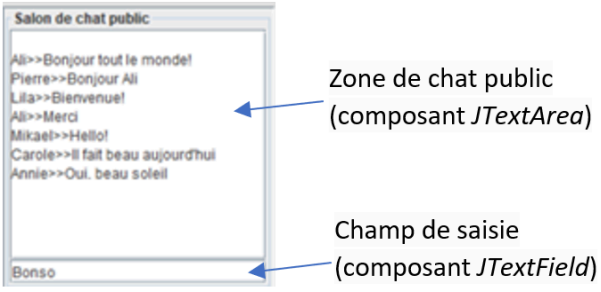
La boîte s'affiche lorsqu'on active l'item de menu ***Configurer serveur***.

Question 2 : Chat public

Vous allez faire en sorte que les messages de chat public apparaissent dans le panneau de chat public.

Le panneau de chat public (classe **PanneauChat**) est un panneau qui affiche :

- Une zone de texte qui doit être non éditable (on ne peut pas saisir du texte directement dedans) qui affiche les messages du salon;
- Un champ de texte où l'utilisateur va saisir ses messages publics.



2.1. Complétez le constructeur de la classe **PanneauChat** pour que :

1. il soit géré par un **BorderLayout**;
2. il contienne le champ de saisie au sud;
3. il place la zone de chat dans un **JScrollPane** placé au centre;
4. la zone de chat soit non éditable.

2.2. Un écouteur de type **EcouteurChatPublic** encapsule des références vers le client et le panneau de chat.

Complétez la méthode **actionPerformed()** de la classe **EcouteurChatPublic** pour vérifier si la source de l'événement est un champ de texte (*JTextField*), alors :

1. récupérer le texte saisi dans le champ et, s'il n'est pas vide,
2. l'envoyer au serveur précédé de la commande **MSG** (comme dans le TP 1),
3. et l'ajouter au panneau de chat précédé de **MOI>>** (comme dans le TP 1),
4. vider le champ de texte (effacer le texte saisi).

2.3. Dans la méthode **setEcouteur()** de **PanneauChat**, enregistrez l'écouteur auprès du champ de saisie.

Dans le constructeur de **PanneauPrincipal**, un écouteur de chat public est déjà créé et fourni au panneau de chat public.

Le gestionnaire d'événements du client **GestionnaireEvenementClient2** contient dans le *switch..case* de la méthode *traiter()* un `case "MSG"` qui affiche les messages publics envoyés par le client.

À ce stade-ci, les messages de chat public devraient apparaître dans le panneau de chat public. Testez le programme avec 2 ou 3 clients pour vous en assurez.

Question 3 : Chat privé

Vous allez faire en sorte que les invitations au chat privé apparaissent et que les utilisateurs puissent les refuser ou les accepter et chatter en privé.

La classe **EcouteurListeConnectes** représente des écouteurs d'événements de souris. Cette classe dérive de la classe abstraite **MouseAdapter** qui implémente l'interface **MouseListener** (voir les adaptateurs ci-après).

Les événements de clic de souris sont gérés par la méthode **mouseClicked()**.

Cet écouteur est utilisé pour gérer les double-clics sur le composant **JList** affichant les alias des connectés. Il est enregistré dans le constructeur de **PanneauPrincipal**.

Les adaptateurs

Un adaptateur, tel que **MouseAdapter**, est une classe qui nous fournit une implémentation par défaut d'une interface afin de nous éviter d'implémenter toutes les méthodes alors qu'on n'a besoin que d'une seule ou de quelques unes.

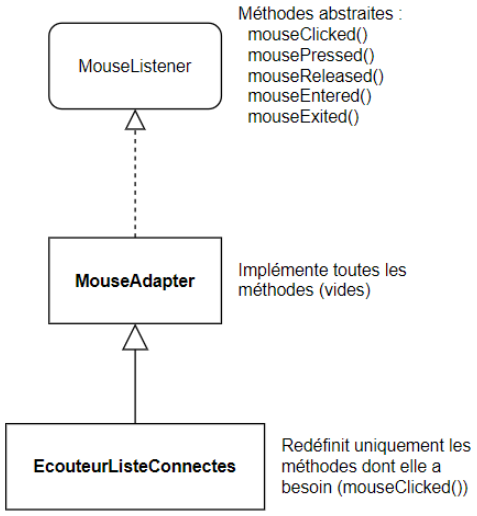
Par exemple, **MouseListener** est une interface qui contient les 5 méthodes de gestion d'événements de la souris :

- `mouseClicked();`
- `mousePressed();`
- `mouseReleased();`
- `mouseEntered();`

- `mouseExited()`.

Nous, on n'a besoin que de **mouseClicked()**.

Au lieu que la classe **EcouteurListeConnectes** implémente **MouseListener** et implémente les 5 méthodes vides sauf **mouseClicked()**, elle dérive plutôt de **MouseAdapter** qui implémente déjà les 5 méthodes mais vides. Comme ça, nous aurons juste à redéfinir **mouseClicked()**.



Pour toutes les interfaces écouteurs qui ont plus d'une méthode (c'est à dire, toutes sauf **ActionListener**), il y a une classe adaptateur correspondante (**WindowAdapter**, **KeyAdapter**, **FocusAdapter**,...).

3.1. Complétez la méthode **mouseClicked()** pour vérifier si l'événement est un double-clic alors, récupérer l'alias sur lequel a eu le double-clic et l'envoyer au serveur avec une commande **JOIN** (comme dans le TP 1).

Le serveur informe l'invité en lui envoyant une commande `JOIN` qui est traitée dans le `case "JOIN"` du côté client :

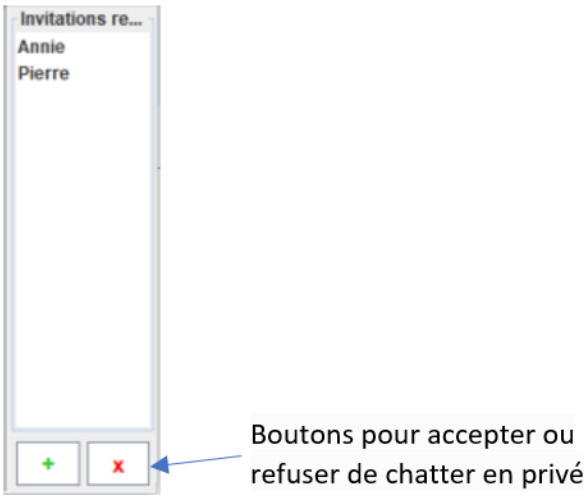
```
case "JOIN":
    alias = evenement.getArgument();
    panneauPrincipal.ajouterInvitationRecue(alias);
    break;
```

À ce stade-ci, un utilisateur peut inviter une personne connectée à chatter en privé, en double-cliquant sur son alias dans la liste des connectés. La personne invitée devrait voir apparaître dans le panneau des invitations reçues les alias des personnes qui l'ont invité.

Il faut maintenant gérer les clics sur les boutons pour accepter ou refuser des invitations.

Le panneau des invitations (classe **PanneauInvitations**) encapsule :

- La liste des invitations reçues (**DefaultListModel**) qui alimente un *JList*;
- Deux boutons pour accepter ou refuser des invitations;
- Un écouteur d'événements d'action (à fournir avec **setEcouteur()**).



3.2. Dans le package **controleur**, ajoutez une classe d'écouteur d'événements d'action. Le constructeur doit recevoir un client de chat (*ClientChat*) et un panneau d'invitations qu'il va sauvegarder dans 2 attributs. La méthode de gestion d'événements doit faire ce qui suit :

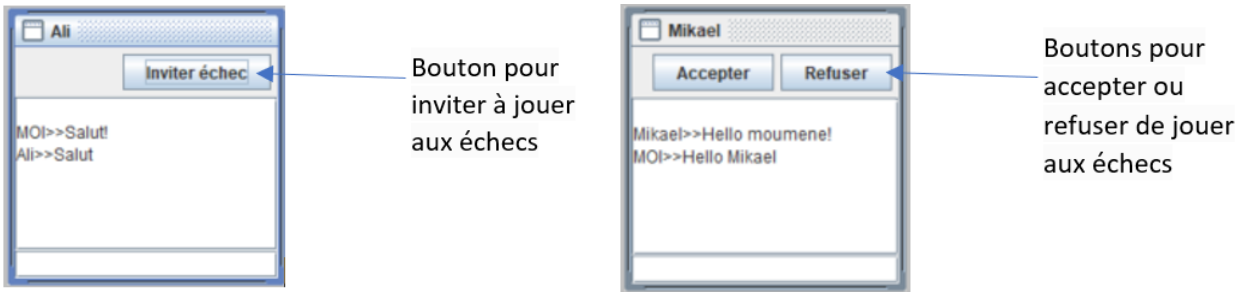
1. Récupérer la liste des invitations sélectionnées dans la liste (notez que l'utilisateur peut sélectionner plusieurs invitations pour les accepter ou refuser en bloc). Utilisez la méthode **getElementsSelectionnes()** du panneau d'invitations;
2. Vérifier si on a cliqué sur le bouton **+** (accepter) ou **x** (refuser) et, selon le cas, envoyer au serveur des commandes `JOIN` ou `DECLINE` pour chaque invitation sélectionnée;
3. Retirer les invitations sélectionnées de la liste.

3.3. Dans le constructeur de **PanneauPrincipal**, instanciez votre classe d'écouteur précédente et fournissez-là au panneau des invitations (avec **setEcouteur()**).

Lorsqu'une invitation est acceptée, le serveur envoie la commande **`JOINOK`** aux 2 personnes. Chacun des clients, dans `case "JOINOK"`, appelle la méthode **creerFenetreSalonPrive()** du panneau principal. Cette méthode crée un panneau de chat privé, l'enregistre dans le dictionnaire **panneauxPrives** et le place dans une nouvelle fenêtre interne (**JInternalFrame**) qui est ajoutée au bureau.

3.4. La classe **PanneauChatPrive** étend la classe **PanneauChat** pour ajouter 2 boutons pour inviter/accepter une partie de Tic-Tac-Toe et refuser de jouer. Complétez le constructeur de cette classe pour ajouter au nord les 2 boutons (le second bouton doit être caché, pour le moment).

3.5. Complétez les méthodes **invitationAJouerRecue()** et **invitationAJouerAnnulee()**. La première change le texte du premier bouton à **"Accepter"** et fait apparaître le bouton pour refuser. La seconde remet le texte du premier bouton à **"Inviter TTT"** et cache le bouton pour refuser.



La classe **EcouteurChatPrive** étend la classe **EcouteurChatPublic** pour stocker l'alias de la personne avec qui on chatte en privé.

3.6. Redéfinir la méthode **actionPerformed()** dans la classe **EcouteurChatPrive** pour vérifier la source de l'événement :

- Si c'est le bouton pour accepter de jouer à Tic-Tac-Toe, envoyer au serveur la commande **`TTT`**;
- Si c'est le bouton pour refuser de jouer à Tic-Tac-Toe, envoyer au serveur la commande **`DECLINE`**;

- Si c'est le champ de saisie alors :
  - Si le texte saisi est QUIT, envoyer au serveur la commande `QUIT` pour quitter le salon privé;
  - Si le texte saisi est ABANDON, envoyer au serveur la commande `ABANDON` pour abandonner la partie de Tic-Tac-Toe;
  - Tout autre texte est envoyé comme message privé avec la commande `PRV` (comme dans le TP 1).

3.7. Le gestionnaire d'événement client (**GestionnaireEvenementClient2**), dans le case "PRV", appelle la méthode **ajouterMessagePrive()** du panneau principal pour afficher le message privé dans le bon panneau de chat privé.

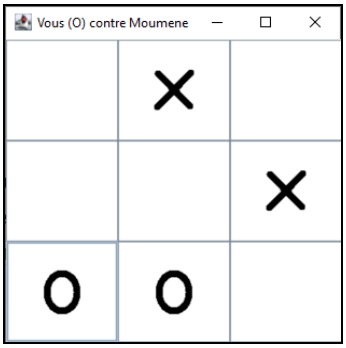
Complétez la méthode **ajouterMessagePrive()** pour trouver le bon panneau de chat privé et ajouter le message à sa zone de chat.

Question 4 : Jeu de Tic-Tac-Toe en réseau

Lorsqu'une invitation à jouer à Tic-Tac-Toe est acceptée, le serveur envoie la commande TTTOK aux 2 clients pour les informer qu'une partie de Tic-Tac-Toe est démarrée entre eux. Chacun des 2 clients, dans le `case "TTTOK"` du gestionnaire d'événement client (**GestionnaireEvenementClient2**), crée une **EtatPartieTicTacToe** avec :

```
client.nouvellePartie();
```

Vous allez faire le nécessaire pour que l'état de la partie soit affiché graphiquement dans une grille et que le symbole du joueur (X ou O) et le nom de l'adversaire apparaissent dans la barre de titre de la fenêtre de Tic-Tac-Toe dans le format suivant : ***Vous (X) contre Annie***



4.1. Rendez les objets **EtatPartieTicTacToe** observables.

4.2. Complétez la méthode **coup()** de la classe **EtatPartieTicTacToe**. Cette méthode reçoit le coup tel que reçu du serveur (argument de la commande COUP, donc dans la forme **X 2 1**). La méthode décortique le coup et, s'il est valide, modifie l'état de

l'échiquier pour refléter le déplacement, notifier les observateurs et retourner ***true***. Si le déplacement n'est pas valide, la méthode retourne ***false***.

La classe **PanneauTicTacToe** est un panneau qui encapsule un **EtatPartieTicTacToe** et qui affiche l'état de la partie sous forme d'une grille de boutons.

4.3. Rendez les objets **PanneauTicTacToe** observateurs et, dans la méthode **seMettreAJour()**, modifiez les icônes des boutons pour refléter l'état de la partie (inspirez-vous du constructeur pour afficher les icônes). Pour effacer l'icône d'un bouton, appelez **setIcon(null)** sur le bouton. Dé-commentez la dernière ligne du constructeur pour connecter l'observateur sur l'observable.

À ce stade, le panneau de Tic-Tac-Toe se rafraîchit lorsque le client est informé d'un coup par le serveur. Mais, il reste à permettre au client de réaliser les coups en cliquant sur les boutons du panneau d'échiquier.

4.4. Complétez la méthode **actionPerformed()** de la classe **EcouteurTicTacToe**. La méthode détermine la position du clic (ligne et colonne) puis envoie au serveur le coup avec une commande `COUP`. Ajoutez à la classe les attributs nécessaires pour faire le travail.

4.5. Complétez le `case "TTTOK"` du gestionnaire d'événement client (**GestionnaireEvenementClient2**) pour :

1. Créer un écouteur de jeu de Tic-Tac-Toe et le fournir au panneau de TicTacToe;
2. Créer dans la variable `fenetreTicTacToe` une fenêtre de jeu Tic-Tac-Toe avec un titre de la forme ***Vous (symbole) contre aliasAdversaire***. Ensuite, rendre la fenêtre visible.

Annexe 1 - Protocole de communication

Voici les commandes utilisées par le serveur et que le client doit gérer dans son gestionnaire d'événements :

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
COUP	symb x y	Valide un coup de Tic-Tac-Toe envoyé par un client.	COUP X 0 1
	Le client prend acte et modifie sa grille de Tic-Tac-Toe.		
INVALID	message	Invalide un coup de Tic-Tac-Toe envoyé par un client.	INVALID pas votre tour
	Le client prend acte et essaie un autre coup.		
TTT	alias	Réception d'une invitation à jouer.	TTT Annie
TTTOK	alias symbole	Valide le démarrage d'une partie de Tic-Tac-Toe pour les clients	TTTOK Annie X
	Le client initie un nouvel objet *EtatPartie*TicTacToe et commence à jouer, s'il a le symbole X.		

Commande	Arguments	Description	Exemple
WAIT_FOR		Informe le client nouvellement connecté qu'il attend son alias.	WAIT_FOR
	Le client affiche la boîte de dialogue de saisie de l'alias.		
NEW	alias	Informe les clients de l'arrivée d'un nouveau connecté.	NEW Annie
	Le client affiche l'alias dans la liste des connectés.		
OK		Informe les clients que l'alias est accepté.	OK
	Le client affiche le panneau principal de l'application.		
EXIT	alias	Informe les clients du départ d'un connecté.	EXIT Annie
	Le client retire l'alias de la liste des connectés.		
QUIT	alias	Informe le client que alias a quitté le salon privé.	QUIT Annie
	Le client ferme la fenêtre de chat privé avec alias.		
MSG	message	Informe les clients de l'arrivée d'un message de chat public.	MSG Annie>>Bonjour à tous
	Le client affiche le message dans la zone de chat public.		
PRV	alias message	Informe les clients de l'arrivée d'un message de chat privé	PRV Annie Allo toi
DECLINE_TTT	alias	Informe les clients du refus d'une invitation à jouer à Tic-Tac-Toe	DECLINE_TTT Annie
ABANDON	alias	Informe les clients de l'abandon d'une partie de Tic-Tac-Toe	ABANDON Annie
TTT_END	etat_partie	Informe les clients de la fin de partie de Tic-Tac-Toe et donne le gagnant	TTT_END nulle ou TTT_END nom_gagnant