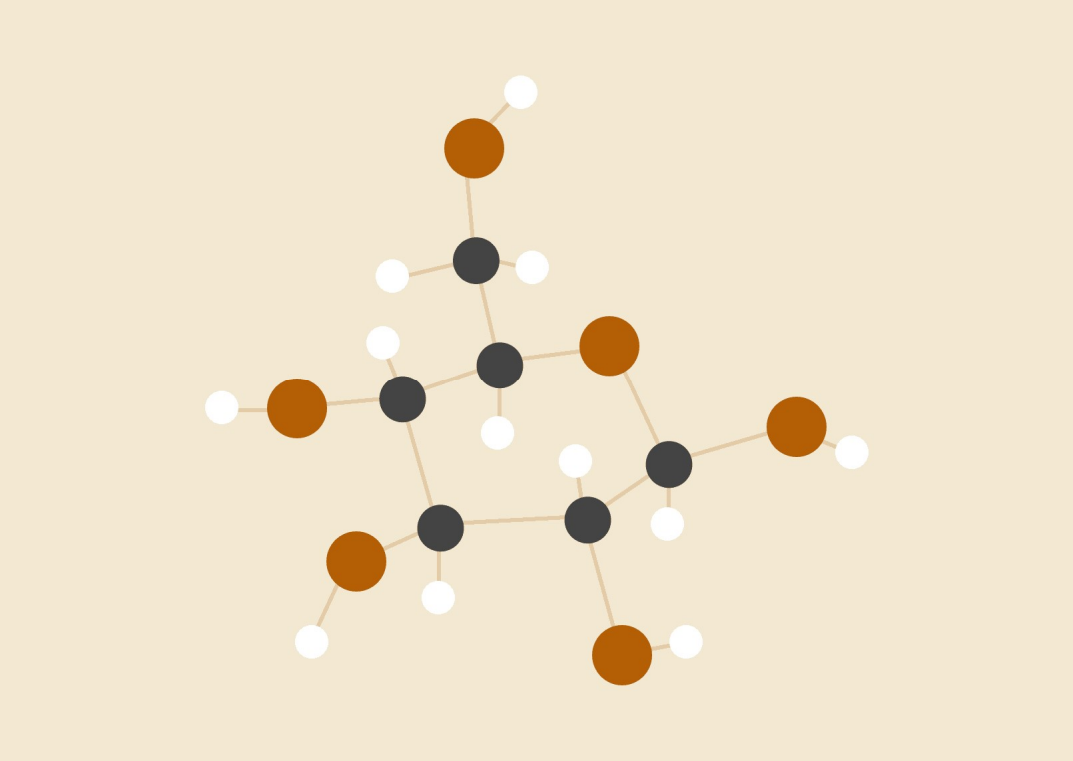
INF111 Programmation orientée objet

Travaux pratiques (TP1 et TP2)

Groupes : tous

Département des Enseignements Généraux

Échecs et Chat 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.

INF111 Programmation orientée objet

Travaux pratiques (TP1 et TP2)

Groupes : tous

Département des Enseignements Généraux

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 jeu d'échecs.

Il y aura donc 2 applications (programmes) :

- Une application cliente que les utilisateurs utiliseront pour accéder aux fonctionnalités ;
- Une application serveur pour héberger les salons de chats et les parties de jeu d'échecs 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 aux échecs.

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,...).

Aller à la table des matières

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

Département des Enseignements Généraux

Directives pour le TP 2

- ➔ Travail avec **les mêmes équipes que le TP 1** (tout changement doit être motivé);
- ➔ Date limite de remise du travail : **13 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 **essentiellement** sur le programme ***ChatClient2***. Nous allons éviter de toucher au programme serveur ***ChatServer2***.
- ➔ Vous devez ajouter le programme client à votre référentiel *Github*.

Ajustements par rapport au TP 1

Pour les besoins de ce TP, quelques 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 mouvement de pièce d'échec est invalide, le serveur inclut maintenant un message dans sa commande INVALID. Ce message sera affiché en dessous de l'échiquier.

Un attribut **alias** dans la classe **ClientChat**, avec son *getter* et son *setter*, pour

garder l'alias de l'utilisateur côté client.

L'**annexe 2** contient (**en vert**) les commandes qui ont été ajoutées au serveur (des **`case`** ont été ajoutés au **`switch..case`** du gestionnaire d'événement du client pour traiter ces commandes).

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.

Pour permettre de fournir au client un gestionnaire d'événement autre que celui qu'il crée par défaut, nous avons :

1. Modifier le type de l'attribut dans la classe **Client** :

   ~~`private GestionnaireEvenementClient gestionnaireEvenementClient;`~~

   devient :

   ```
   private GestionnaireEvenement gestionnaireEvenementClient;
   ```

2. Ajouté un mutateur (*setter*) pour cet attribut :

   ```
   public void setGestionnaireEvenementClient(GestionnaireEvenement gec) {
      this.gestionnaireEvenementClient = gec;
   }
   ```

Ces changements vont nous permettre d'injecter un autre gestionnaire d'événements à la place de celui utilisé dans le TP 1. En particulier, dans ce TP 2, nous allons utiliser une nouvelle classe **GestionnaireEvenementClient2**.

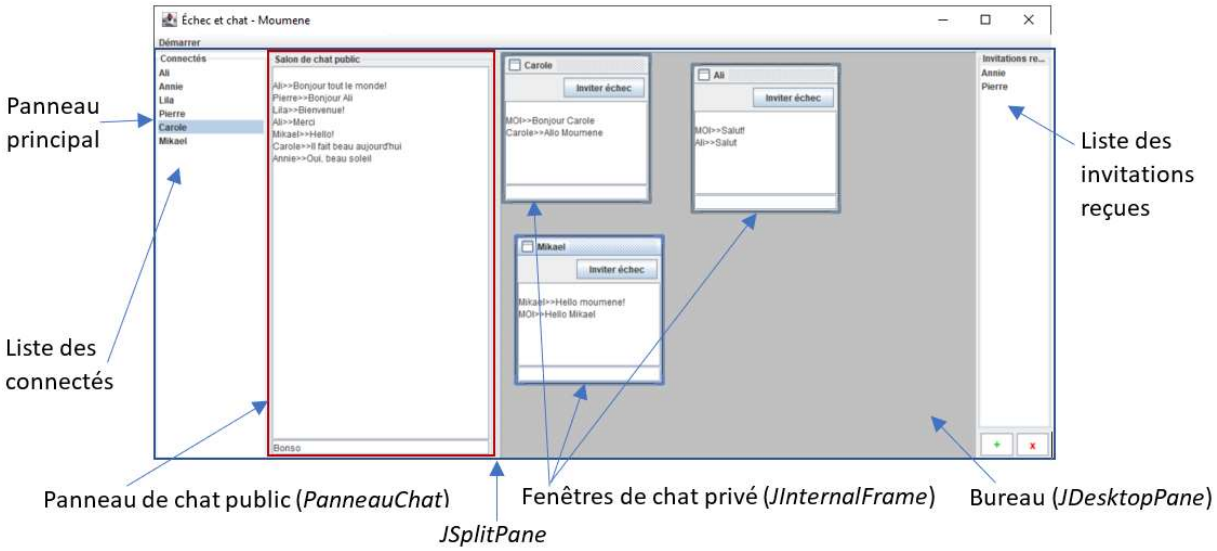
D'autre part, nous avons besoin de réafficher la boîte de dialogue de saisie de l'alias aussi longtemps que l'alias fourni est refusé par le serveur. Nous modifions donc le serveur pour qu'il renvoie la réponse **WAIT_FOR alias** lorsque l'alias fournit n'est pas valide. De son côté, le gestionnaire 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 **ChatClient** 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* que 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 **Se déconnecter** demande une confirmation avant de se déconnecter. Sinon, la déconnexion s'effectue 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 aux échecs, 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 (utilisez une gestion d'exception). Les données saisies sont fournies au client de chat.

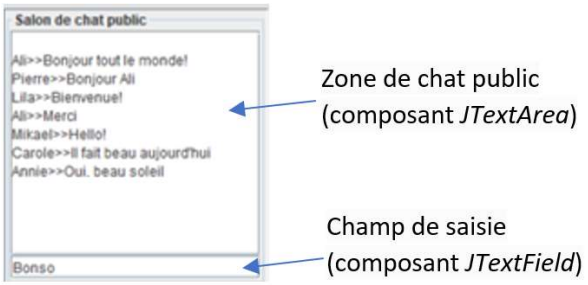
La boîte s'affiche lorsqu'on active l'item de menu ***Configurer serveur***.

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.

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

2.2. 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.

Question 3 : Chat privé

Vous allez faire en sorte que les invitations au chat privé apparaissent et que les utilisateurs puissent 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 **ActionListener**.

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**.

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 :

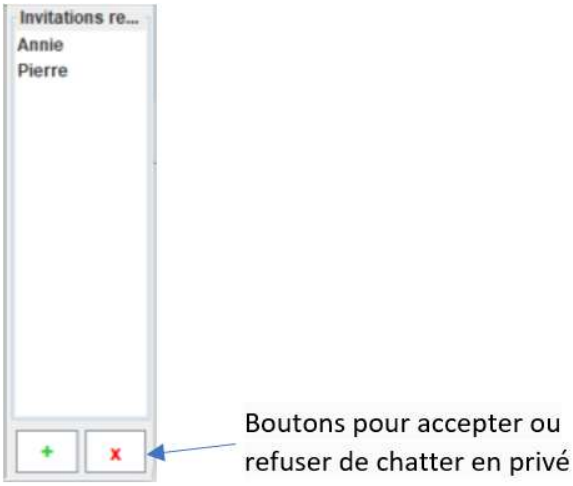
```
case "JOIN":
    alias = evenement.getArgument();
    panneauPrincipal.ajouterInvitationRecue(alias);
    break;
```

À ce stade-ci, un utilisateur peut inviter une personne connectée à chatter en privé. La personne invitée doit 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 d'échecs et refuser de jouer. Complétez le constructeur de cette classe pour ajouter au nord les 2 boutons et cacher le bouton qui sert à refuser une partie d'échecs.

3.5. Complétez les méthodes **invitationEchecRecue()** et **invitationEchecAnnulee()**. 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 échec"** et cache le bouton pour refuser.

La classe **EcouteurChatPrive** étend la classe **EcouteurChatPublic** pour stocker l'alias de la personne avec qui on chat 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 aux échecs, envoyer au serveur la commande **`CHESS`**;
- Si c'est le bouton pour refuser de jouer aux échecs, 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 d'échecs;
  - 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.

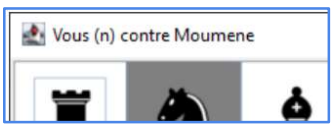
Complétez la méthode **ajouterMessagePrive()** pour trouver le bon panneau de chat privé et ajouter le message à sa zone de chat.

Question 4 : Jeu d'échecs en réseau

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

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

Vous allez faire le nécessaire pour que l'état de la partie soit affiché graphiquement dans une grille et que la couleur du joueur (n ou b) et le nom de l'adversaire apparaissent dans la barre de titre de la fenêtre d'échecs :



Source des images : https://commons.wikimedia.org/wiki/Category:PNG_chess_pieces/Standard_transparent

4.1. Rendez les objets **EtatPartieEchec** observables.

4.2. Complétez la méthode **move()** de la classe **EtatPartieEchec**. Cette méthode reçoit le déplacement tel que reçu du serveur (argument de la commande MOVE, donc dans la forme **e2-e4**, **e2e4** ou **e2e4**). La méthode décortique le déplacement 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***.

N'oubliez pas de vérifier si c'est un pion qui a atteint la dernière ligne, il faut le transformer en dame, car le serveur n'informe pas les clients de la promotion d'un pion.

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

4.3. Rendez les objets **PanneauEchiquier** 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.

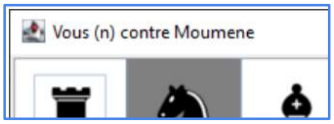
À ce stade, le panneau d'échiquier se rafraîchit lorsque le client est informé d'un déplacement de pièce par le serveur. Mais, il reste à permettre au client de déplacer des pièces en cliquant sur les boutons du panneau d'échiquier.

4.4. Complétez la méthode **actionPerformed()** de la classe **EcouteurJeuEchecs**. La

méthode doit enregistrer la position d'un premier clic (première position d'un déplacement) de manière à ce qu'au deuxième clic (deuxième position du déplacement), on envoie au serveur le déplacement avec une commande `MOVE`. Ajoutez à la classe les attributs nécessaires pour faire le travail.

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

1. Créer un écouteur de jeu d'échecs et le fournir au panneau d'échiquier;
2. Créer dans la variable `fenetreEchecs` une fenêtre de jeu d'échecs avec un titre de la forme ***Vous (couleur) contre aliasAdversaire***. Ensuite, rendre la fenêtre visible.



Annexe 1 - Protocole de communication

Commandes utilisées par le client

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.		
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.		
CHESS	alias	Invite ou accepte une invitation pour une partie de jeu d'échecs.	CHESS Annie
	Si les 2 utilisateurs ne sont pas en chat privé, le serveur ignore la commande. Si l'utilisateur a déjà reçu une invitation de l'alias, le serveur crée une partie d'échecs dans le salon de chat privé entre les 2 personnes.		

Commande	Arguments	Description	Exemple
	Sinon, le serveur crée une nouvelle invitation dans le salon de chat privé et informe le destinataire. Ensuite : - Si le destinataire de l'invitation l'accepte, une partie d'échecs est créé; - Sinon, l'invitation est supprimée.		
MOVE	pos1 pos2	Effectue un déplacement de pièce dans une partie de jeu d'échecs.	MOVE e2 e4 ou : MOVE e2-e4 ou : MOVE e2e4
	Le serveur valide le mouvement dans la partie d'échec et renvoie la même commande aux 2 utilisateurs, si valide. Sinon, il envoie INVALID à l'utilisateur.		
ABANDON		Abandonne une partie d'échecs.	ABANDON
	Le serveur déclare l'autre joueur gagnant, informe les 2 joueurs et détruit la partie.		

Commandes utilisées par le serveur

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

Commande	Arguments	Description	Exemple
MOVE	pos1 pos2	Valide un déplacement de pièce envoyé par un client.	MOVE e2 e4 ou : MOVE e2-e4 ou : MOVE e2e4
	Le client prend acte et modifie son échiquier.		
INVALID	message	Invalide un déplacement de pièce envoyé par un client.	INVALID pas votre tour
	Le client prend acte et essaie un autre déplacement.		
CHESSOK	couleur	Valide le démarrage d'une partie d'échecs pour les clients	CHESSOK n ou : CHESSOK b
	Le client initie un nouvel objet *EtatPartieEchecs* et commence à jouer, s'il a les blancs.		
ECHEC		Informe un client que son roi est en échec.	ECHEC
MAT	alias	Informe le client qu'il y a échec et mat et donne l'alias du gagnant.	MAT Annie
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.		
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
CHESS	alias	Informe les clients de la réception d'une invitation à jouer aux échecs	CHESS Annie

Commande	Arguments	Description	Exemple
DECLINE_CHESS	alias	Informe les clients du refus d'une invitation à jouer aux échecs	DECLINE_CHESS Annie