

INF111 Programmation orientée objet

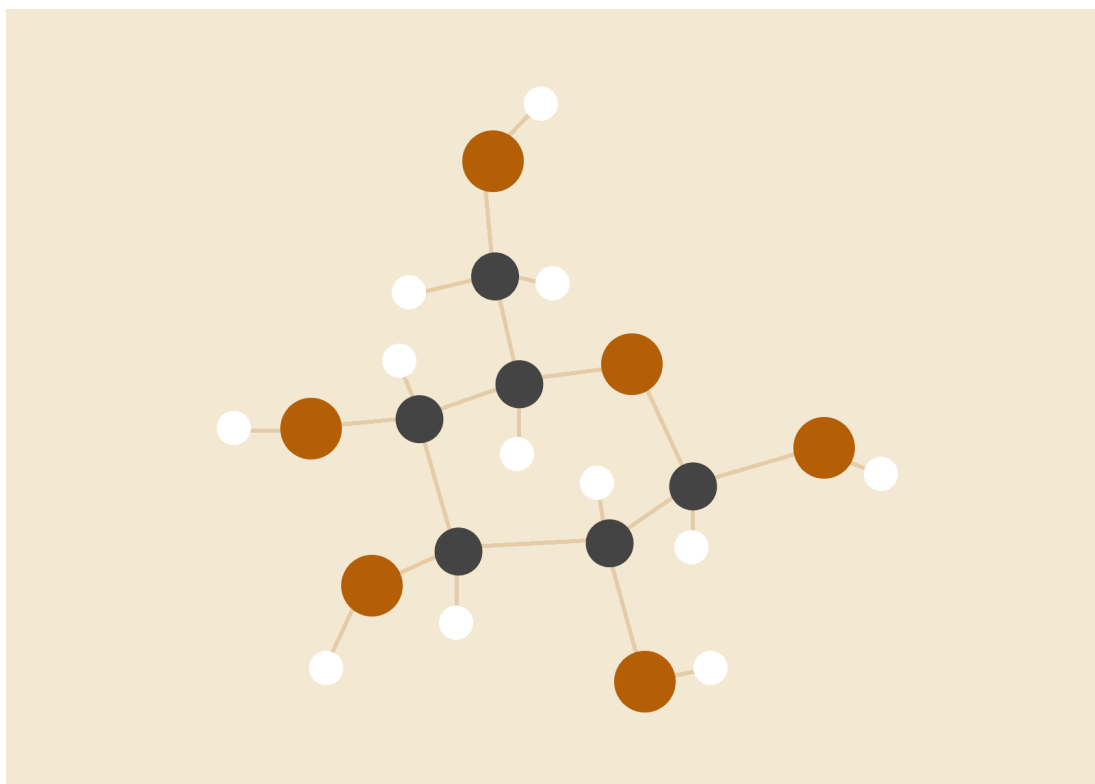
Travaux pratiques (TP1 et TP2)

Groupes : tous



Département  
des Enseignements  
Généraux

# Échecs et Chat en Réseau



**Abdelmoumène Toudeft, Professeur enseignant**

[Abdelmoumene.Toudeft@etsmtl.ca](mailto:Abdelmoumene.Toudeft@etsmtl.ca)

**Bureau B-1642**



*Table des matières*

<b>Introduction.....</b>	<b>1</b>
<b>Description et contexte.....</b>	<b>2</b>
<b>TP 1.....</b>	<b>3</b>
<b>Directives pour le TP 1.....</b>	<b>4</b>
<b>Introduction à la programmation réseau avec les sockets.....</b>	<b>5</b>
Adresse IP et port de communication.....	5
Serveur.....	6
Client.....	6
Connexion-Communication.....	6
Socket.....	6
Implémentation en Java.....	6
Exemple : un serveur simple et un client simple.....	8
Nécessité d'un protocole de communication.....	8
Code fourni.....	9
Classe Connexion.....	9
Classes Serveur et ServeurChat.....	10
Classes Client et ClientChat.....	10
<b>Programmation multithread.....</b>	<b>11</b>
Un mot sur les accès concurrents.....	11
<b>Introduction à la programmation événementielle.....</b>	<b>12</b>
Événement.....	12
Gestion d'événements.....	13
Implémentation.....	13
Classe Evenement.....	14
Interface GestionnaireEvenement.....	14
Classe utilitaire EvenementUtil.....	14
Classes GestionnaireEvenementServeur et GestionnaireEvenementClient.....	15
<b>Programmes de départ.....</b>	<b>16</b>
Démarrer les programmes.....	16
Lancer plusieurs clients simultanément.....	17
Traitement des commandes par le serveur.....	19
Traitement des commandes par le client.....	20

# INF111 Programmation orientée objet

## Travaux pratiques (TP1 et TP2)

Groupes : tous



Département  
des Enseignements  
Généraux

<b>Chatter en public et en privé.....</b>	<b>21</b>
Question 1 : Chatter dans le salon public.....	21
Question 2 : Historique des messages de chat.....	22
Question 3 : Chatter en privé.....	23
<b>Jeu d'Échecs.....</b>	<b>25</b>
Les classes.....	26
Question 4 : Compléter le jeu d'échecs.....	28
<b>Jouer en réseau.....</b>	<b>30</b>
Question 5 : Un état de partie pour le client.....	31
Question 6 : Mettre en place le jeu en réseau.....	32
<b>Annexe 1 - Exemple de serveur et client simples.....</b>	<b>34</b>
Un serveur simple qui renvoie l'inverse du message qu'il reçoit.....	34
Un client simple qui communique avec le serveur.....	35
<b>Annexe 2 - Protocole de communication.....</b>	<b>36</b>
Commandes utilisées par le client.....	36
Commandes utilisées par le serveur.....	37

**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 œuvre 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.



## 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.



## TP 1

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

Vous allez compléter 3 programmes pour implémenter les fonctionnalités de chat et de jeu d'échecs.

Ce TP comporte 6 questions contenant 27 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 les prochaines étapes ou dans le TP 2.

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 1

- Travail en équipes de **2 ou 3**;
- Date limite de constitution des équipes : **18 octobre à 23h59**
- Date limite de remise du travail : **17 novembre à 23h59**
- Le travail doit être remis sur *Github* (tout le monde doit avoir un compte *Github*);
- Trois (3) programmes sont fournis sous forme de projets *IntelliJ* : **ChatServer**, **ChatClient** et **JeuEchecs**. 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.
  - ◆ 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 (s'envoyer et recevoir des données).

Serveur	Client
<pre>ServerSocket ss = new ServerSocket(3845); Socket s1 = ss.accept();  //Connexion établie. s1 représente le client</pre>	<pre>Socket s2 = new Socket(adresse_serveur, 3845);  //Connexion établie. s2 représente le serveur</pre>

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 1**.

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*).

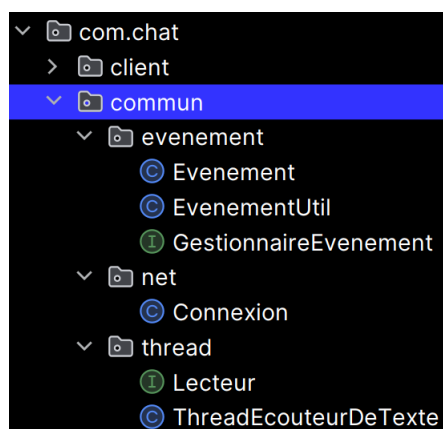
Pour notre architecture client/serveur, on va définir **notre propre protocole de communication**. Les commandes de ce protocole sont fournies dans l'**annexe 2**.



## Code fourni

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

Les classes et interfaces des sous-packages du package `com.chat.common` 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 présentées précédemment. 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. Il 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.

**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 String getAvailableText();`
- `public void envoyer(String texte);`
- `public boolean close();`
- `public String getAlias();`
- `public void setAlias(String alias).`

### 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*

Vous aurez à travailler avec et à compléter la classe **ClientChat** pour implémenter les fonctionnalités côté client pour le jeu d'échecs (question **5** et sous-questions). La classe **ClientChat** dérive de la classe **Client** qui contient le nécessaire pour interagir avec un serveur. Vous n'avez pas à connaître les détails d'implémentation de la classe **Client**. Vous allez plutôt utiliser ses méthodes publiques, principalement :

- `public boolean connecter();`
- `public boolean deconnecter();`
- `public void envoyer(String s)`



## 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 ?

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 fonctionneront 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 clic 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.

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 défini les classes et interfaces suivantes dans le package **com.chat.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 ces 2 méthodes ***traiter()***, dans le cadre de ce TP.

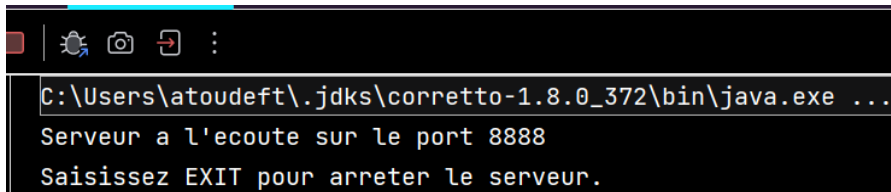
Le client et le serveur détiennent, dans un attribut de type ***GestionnaireEvenement***, une référence vers un gestionnaire d'événements qu'ils alertent chaque fois qu'ils reçoivent un message/commande (dans leurs méthodes ***lire()***).

## 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 :



```
C:\Users\atoudeft\.jdk\corretto-1.8.0_372\bin\java.exe ...  
Serveur a l'ecoute sur le port 8888  
Saisissez EXIT pour arreter le serveur.
```

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 perd 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**.



```
C:\Users\atoudeft\.jdk\corretto-1.8.0_372\bin\java.exe ...  
Vous etes connectes au serveur A l'adresse 127.0.0.1 sur le port 8888  
Saisissez vos textes (EXIT pour quitter) :  
      .WAIT_FOR alias  
Moumene  
      .OK
```

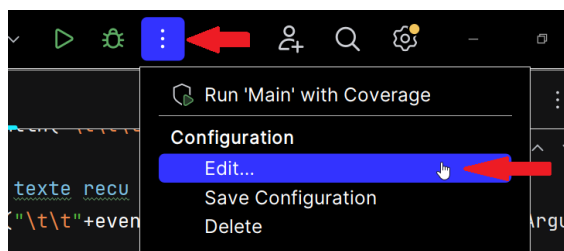
Une fois que l'alias est accepté, l'utilisateur est ajouté à la liste des connectés (attribut **connectes**, hérité par **ServeurChat** de la classe **Serveur**) et le serveur renvoie tous les messages qui ont déjà été envoyés au salon de chat public. Si aucun message n'a encore été envoyé, le serveur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

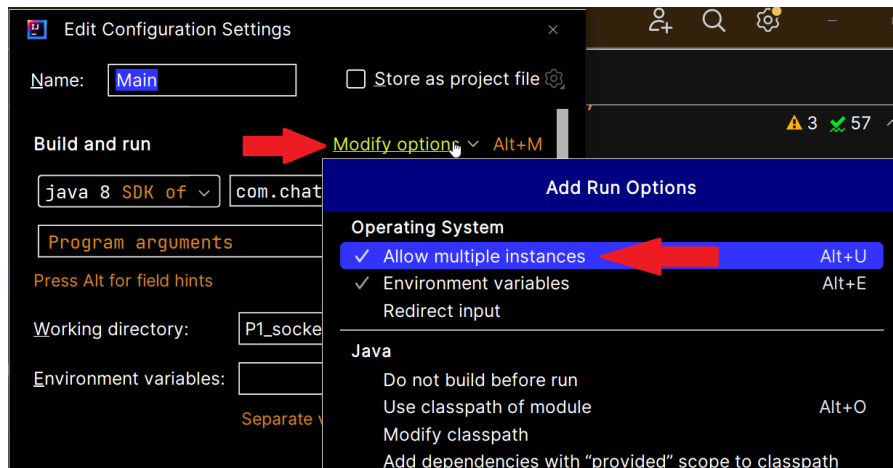
## Lancer plusieurs clients simultanément

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;

```
.WAIT_FOR alias
Carl
.OK
LIST
6 personnes dans le salon :
- Moumene
- Ali
- Lila
- Annie
- Pierre
- Carl
```

- **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) :



```
C:\Users\atoudeft\.jdk\corretto-1.8.0_372
Vous etes connectes au serveur Ã l'adress
Saisissez vos textes (EXIT pour quitter) :
.WAIT_FOR alias
Pierre
.OK
Bonjour tout le monde !
.BONJOUR TOUT LE MONDE !
```

## 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***. C'est à cet endroit que vous allez gérer les futures réponses du serveur :

---

```
public void traiter(Evenement evenement) {  
    ...  
    switch (typeEvenement) {  
        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;  
        //Ajoutez ici d'autres case pour gérer d'autres commandes du serveur.  
        default: //Afficher le texte reçu :  
            System.out.println("\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.

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.

**2.4.** Ajoutez dans la structure **switch..case** du gestionnaire d'événements du client un **case "HIST"** dans lequel vous affichez l'historique des messages reçu du serveur. Vous devez afficher un message par ligne et chacun est précédé de 3 tabulations et d'un point (.).

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 **SalonPrivé** 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 ...).

**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**.

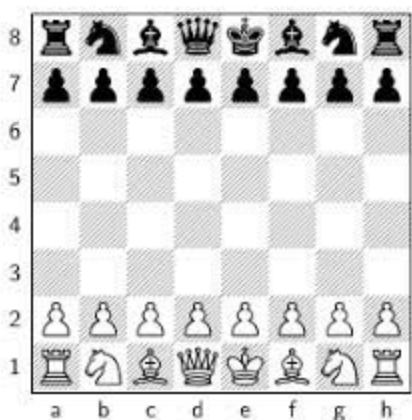
## Jeu d'Échecs

Vous allez créer les classes nécessaires pour jouer au jeu d'échecs. Ce n'est pas à vous de faire la conception. Elle vous est fournie.

Le programme de départ vous est déjà fourni ([JeuEchecs.zip](#)). Vous devez le compléter avant de l'intégrer dans l'architecture client/serveur pour jouer en réseau.

La conception utilisée ici est une conception parmi plusieurs possibles.

Le jeu d'échecs est un jeu de stratégie qui se joue à 2 sur une grille 8x8 appelé **échiquier**. Les 8 colonnes sont identifiées par les lettres **a** à **h** tandis que les 8 lignes sont identifiées par les nombres 1 à 8.



Échiquier (grille de jeu d'échecs) en position de début de partie

(source : <https://www.dicebreaker.com/games/chess>).

Un joueur joue avec les pièces blanches (*les blancs*) et l'autre avec les pièces noires (*les noirs*). C'est toujours les blancs qui commencent.

Chaque joueur possède 16 pièces : 8 pions, 1 roi, 1 dame, 2 tours, 2 fous et 2 cavaliers.

Initialement, les blancs occupent les lignes 1 et 2 et les noirs les lignes 7 et 8 (voir figure ci-dessus).

Chaque pièce se déplace selon des règles spécifiques.

Un joueur peut utiliser ses pièces pour éliminer (on dit "*manger*") les pièces de l'adversaire.

Le but du jeu est de mettre le roi de l'adversaire en **échec et mat**, c'est-à-dire dans une situation où il ne peut pas échapper à son élimination.

Voici une page qui explique les règles du jeu d'échecs :

<https://ecole.apprendre-les-echecs.com/regles-echecs/>



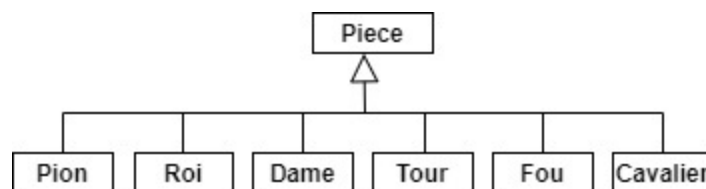
## Les classes

La classe **Position** représente une position/case sur l'échiquier. Rappelons que les lignes sont numérotées de **1** à **8** (type utilisé : **byte**) et les colonnes de **a** à **h** (type utilisé : **char**).

La classe **Position** fournit des méthodes non implémentées (déclenchent une **NotImplementedException**) et c'est à vous de les implémenter :

- public boolean **estVoisineDe**(Position p)
- public boolean **estSurLaMemeLigneQue**(Position p)
- public boolean **estSurLaMemeColonneQue**(Position p)
- public boolean **estSurLaMemeDiagonaleQue**(Position p)

Les classes des pièces constituent une famille (ou hiérarchie) de classes dont la racine est la classe **Piece** :



La classe **Piece** est fournie :

---

```
public abstract class Piece {  
    protected char couleur; //b ou n  
    public Piece(char couleur) {  
        this.couleur = couleur;  
    }  
    public char getCouleur() {  
        return couleur;  
    }  
    public abstract boolean peutSeDeplacer(Position pos1, Position pos2,  
                                           Piece[][] echiquier);  
}
```

---

La classe est **abstraite** et constitue une base commune à toutes les classes de



pièces. La méthode abstraite **peutSeDeplacer()** va être implémentée différemment dans chacune des classes filles.

C'est à vous de fournir les classes concrètes des pièces **Pion, Roi, ...** (voir questions suivantes). Chaque classe doit fournir un constructeur qui reçoit la couleur de la pièce sous forme d'un caractère. Le constructeur appelle celui de la classe mère (avec **super()**).

La méthode **peutSeDeplacer()** va vérifier si la pièce peut se déplacer selon ses propres règles et en tenant compte de la position des autres pièces sur l'échiquier. Par exemple, mise à part le cavalier, les pièces ne peuvent pas passer par-dessus d'autres pièces.

Les règles communes à toutes les pièces sont vérifiées par la méthode **deplace()** de **PartieEchecs**. Par exemple, une pièce ne peut pas se déplacer vers une position où il y a déjà une pièce de même couleur. Voir question **4.2** ci-dessous.

La classe utilitaire **EchecsUtil** fournit quelques méthodes utilitaires :

- `public static Position getPosition(byte i, byte j)`
- `public static byte getLigne(byte i)`
- `public static char getColonne(byte j)`
- `public static boolean positionValide(Position p)`
- `public static byte indiceLigne(byte ligne)`
- `public static byte indiceColonne(char colonne)`

Ces méthodes sont déjà implémentées. Elles sont statiques. On les appelle à l'aide de la classe :

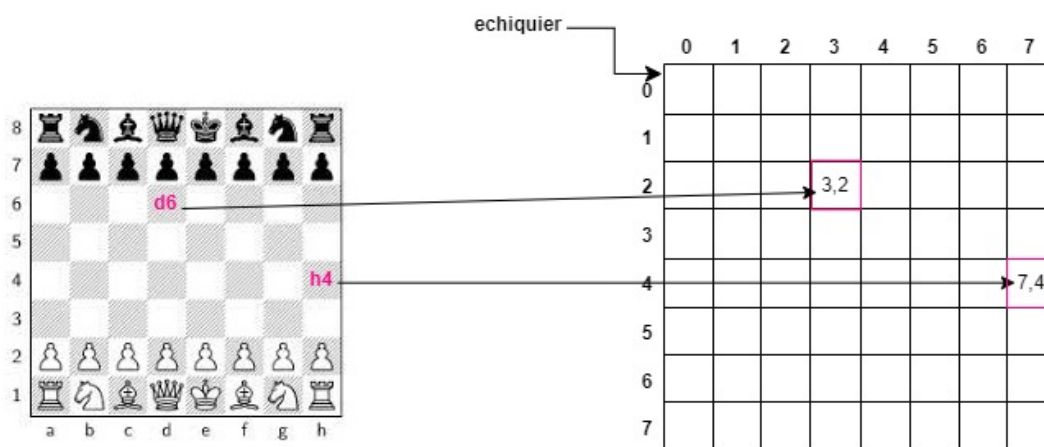
```
EchecsUtil.getPosition(...)  
EchecsUtil.getLigne(...)  
...
```

Un constructeur privé empêche l'instanciation de la classe **EchecsUtil**.

La classe **PartieEchecs** représente une partie d'échecs. Elle encapsule un échiquier sous forme d'un tableau 2D de pièces. Cette classe va gérer l'essentiel du déroulement d'une partie.

Les indices des lignes et colonnes de l'attribut **echiquier** en mémoire sont entre **0**

et 7. Vous devez utiliser les méthodes de la classe **EchecsUtil** pour établir les correspondances entre ces indices et les numéros des lignes (8 à 1) et colonnes (a à h) d'un vrai jeu d'échecs.



La correspondance entre les cases d'un vrai échiquier et le tableau 2D de la partie d'échecs dans le programme est fournie par la classe *EchecsUtil*.

## Question 4 : Compléter le jeu d'échecs

**4.1.** Implémentez les méthodes non implémentées de la classe **Position**.

**4.2.** Implémentez la méthode **deplace()** de la classe **PartieEchecs**. Cette méthode doit faire les vérifications de base nécessaires avant d'appeler la méthode **peutSeDeplacer()** pour vérifier si la pièce peut se déplacer d'une position à une autre.

Voici les principales conditions que la méthode **deplace()** doit vérifier avant d'appeler **peutSeDeplacer()** :

- Les positions initiale et finale sont valides;
- Il y a bien une pièce à déplacer à la position initiale;
- La couleur de la pièce à déplacer possède bien la couleur correspondant au joueur qui a la main;
- Il n'y a pas à la position finale une pièce de même couleur que la pièce à déplacer;



- Si le joueur veut faire un roque, les conditions sont bien réunies. Par exemple, si la tour ou le roi ont déjà bougé, le roque n'est pas permis. Vous devez ajouter à la classe **PartieEchecs** les attributs nécessaires à ce genre de vérification.

Pour exprimer qu'il veut faire un roque, le joueur va déplacer son roi de 2 cases vers la gauche ou la droite.

**4.3.** Définissez les classes de pièces concrètes **Pion, Roi, Dame, Tour, Fou** et **Cavalier**, toutes dérivant de la classe **Piece** et implémentez dans chacune d'elles la méthode abstraite **peutSeDeplacer()** pour vérifier si la pièce peut se déplacer d'une position à une autre.

**4.4.** Complétez le constructeur de la classe **PartieEchecs** de manière à placer les pièces sur l'échiquier dans la position de départ d'une partie de jeu d'échecs. Assurez-vous que le tour est bien au blancs (attribut **tour** à **'b'**).

**4.5.** Implémentez la méthode **estEnEchec()** de la classe **PartieEchecs**.

La méthode sera appelée par la méthode **deplace()** après chaque déplacement valide d'une pièce. La méthode vérifie si un des rois est en échec. Si le roi en échec est de la même couleur que la pièce qui vient d'être déplacée, le déplacement doit être invalidé.

### 3 simplifications :

- **On ne s'intéresse pas au Pat.** Le pat se produit lorsque le joueur, sans que son roi ne soit en échec, ne peut déplacer aucune pièce sans mettre son roi en échec. Dans ce cas, le pat est déclaré. Ce qui veut dire que la partie est finie sans qu'il y ait de gagnant.
- **Promotion automatique en dames :** lorsqu'un pion atteint la dernière ligne (8 pour les blancs et 1 pour les noirs), le joueur peut le transformer en une pièce de son choix. Pour simplifier, les pions seront automatiquement promu en dames.
- **On n'implémente pas la prise en passant.**



## Jouer en réseau

Nous allons maintenant intégrer le jeu d'échecs dans l'architecture client/serveur afin de permettre aux personnes qui chattent en privé de jouer des parties d'échecs.

### Quelle application va héberger les parties d'échecs ?

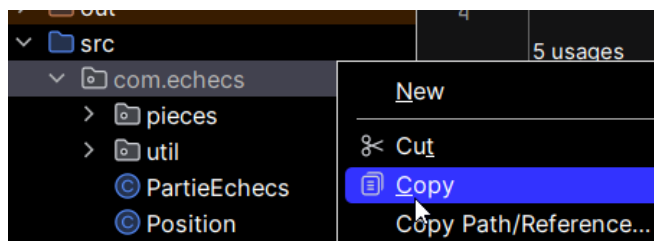
Il y a 2 approches, avec chacune ses avantages et ses inconvénients :

- La partie d'échecs est hébergée au niveau du serveur et les 2 clients vont juste afficher l'**état de la partie**;
- La partie est hébergée au niveau des 2 clients et le serveur ne fait que relayer les coups (déplacements des pièces) de chacun des joueurs.

**Nous optons pour la première approche.**

**Vous devez donc copier les packages** du jeu d'échecs dans l'application serveur uniquement.

Pour intégrer les packages du jeu d'échecs dans l'application serveur, sous *IntelliJ*, vous avez juste à copier le package **com.echecs** (et ses sous-packages) :



et le coller dans le dossier **src** de l'application serveur.

### Simplification :

Pour simplifier, on va supposer qu'un client **ne peut pas jouer plus d'une partie** d'échecs en même temps.



## Question 5 : Un état de partie pour le client

Puisque c'est le serveur qui héberge les parties de jeu d'échecs, c'est lui qui est responsable du déroulement de la partie (validation des déplacements, les tours de jeu,...).

Le client ne fait qu'envoyer les mouvements décidés par l'utilisateur et afficher l'état de la partie, c'est-à-dire l'état de l'échiquier. Il a donc besoin d'un objet qui garde l'état de la partie.

On va utiliser une classe **EtatPartieEchecs** pour cela.

**5.1.** Ajoutez dans l'application cliente une classe **EtatPartieEchecs** qui encapsule un attribut **etatEchiquier** de type tableau 2D de *char* de taille 8x8.

Mettez cet attribut en lecture et écriture (*getter* et *setter*).

Ce tableau 2D va stocker l'état de l'échiquier avec un *char* par case :

- Le caractère blanc ' ' pour une case vide;
- Une lettre par type de pièce (**p** pour pion, **t** pour tour, **c** pour cavalier, **f** pour fou, **d** pour la dame et **r** pour le roi);
- Des lettres majuscules pour les blancs et minuscules pour les noirs.

**5.2.** Ajoutez à la classe **EtatPartieEchecs** un constructeur sans arguments qui va initialiser l'échiquier à l'état initial d'une partie d'échecs.

**5.3.** Redéfinissez dans la classe **EtatPartieEchecs** la méthode **toString()** pour retourner une chaîne contenant l'état de la partie sous la forme matricielle (tableau 2D) suivante :

```
1 t c f d r f c t
2 p p p p p p p
3 . . . . .
4 . . . . .
5 . . . . .
6 . . . . .
7 P P P P P P P
8 T C F D R F C T
 a b c d e f g h
```



**5.4.** Ajoutez à la classe **ClientChat** un attribut de type **EtatPartieEchecs**.

Mettez cet attribut en lecture et écriture (*getter* et *setter*).

Tous les déplacements de pièces validés par le serveur vont provoquer une mise à jour de l'état de la partie d'échecs chez le client.

**5.5.** Ajoutez à la classe **ClientChat** une méthode **nouvellePartie()** qui va initialiser un nouvel objet **EtatPartieEchecs** dans son attribut.

## Question 6 : Mettre en place le jeu en réseau

**6.1.** Ajoutez à la classe **SalonPrive** un attribut de type **PartieEchecs** pour stocker la partie d'échecs entre les 2 personnes qui chattent en privé. Cet attribut doit être en lecture et écriture (*getter* et *setter*).

Modifiez les applications cliente et serveur pour :

**6.2.** Permettre à un utilisateur **alias1** d'envoyer une commande **CHESS alias2** afin d'inviter **alias2**, avec qui il chatte en privé, à jouer une partie d'échecs. La commande servira aussi à accepter par **alias1** une invitation envoyée par **alias2**.

Un utilisateur ne peut pas accepter une invitation s'il est déjà en partie d'échecs (puisque'il ne peut pas jouer plusieurs parties en même temps). L'acceptation est ignorée par le serveur.

S'il n'y a pas de chat privé entre les 2 personnes, la commande est ignorée par le serveur.

Une fois l'invitation acceptée :

- Le serveur crée une partie d'échecs dans le salon privé;
- Le serveur répartit au hasard les couleurs blanc et noir entre les 2 joueurs et les informe en leur envoyant le message **CHESSOK** couleur (CHESSOK b et CHESSOK n);
- Le client initialise une nouvelle partie (appel de la méthode **nouvellePartie()** de **ClientChat**).

Le joueur qui a les blancs commence la partie.

À partir de là, chacun des deux clients joue à tour de rôle et :



- Le joueur ne peut pas déplacer les pièces si ce n'est pas son tour;
- Le joueur ne peut pas déplacer les pièces qui ne sont pas de sa couleur;
- Tous les déplacements sont validés au niveau du serveur avant de mettre à jour l'état de la partie chez les 2 clients;
- La partie se termine lorsqu'il y a échec et mat ou lorsqu'un joueur abandonne la partie (question 6.4 ci-dessous).

**6.3.** Permettre à un utilisateur d'envoyer une commande **MOVE** afin de déplacer une pièce sur l'échiquier. Les formats de la commande acceptées sont :

**MOVE c3-e4**

ou

**MOVE c3 e4**

ou

**MOVE c3e4**

où les 2 premiers caractères constituent la position initiale et les 2 derniers la position finale du mouvement.

La commande est envoyée au serveur qui va la valider sur la partie d'échecs :

- Si la commande est valide, le serveur renvoie la même commande aux 2 clients et, à sa réception, chacun des 2 clients met à jour son échiquier. Chacun des clients affiche le nouvel état de l'échiquier (utilisez **toString()**);
- Si la commande est invalide, le serveur renvoie la commande **INVALID** et le client doit essayer un autre mouvement.

Après chaque déplacement valide, le serveur vérifie s'il y a échec, il informe les 2 joueurs avec la commande **ECHEC alias\_en\_echec**.

S'il y a échec et mat, il informe les 2 joueurs avec la commande **MAT alias\_gagnant**.

**6.4.** Permettre à un utilisateur d'envoyer une commande **ABANDON** afin d'abandonner une partie de jeu d'échecs.



## Annexe 1 - 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 acceptée.");

//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();
}
```



## Annexe 2 - 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 <i>alias1:alias2:alias3...</i>		
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 <i>alias&gt;&gt;message</i>		
JOIN	alias	Envoie ou accepte une invitation à chatter en privé.	JOIN Annie
	<p>Si l'utilisateur a déjà reçu une invitation de l'alias, le serveur crée un salon de chat privé entre les 2 personnes (envoie <i>JOINOK</i> aux 2).</p> <p>Sinon, le serveur crée une nouvelle invitation et informe le destinataire (en lui envoyant <i>JOIN</i>). Ensuite :</p> <ul style="list-style-type: none"> <li>- Si le destinataire de l'invitation l'accepte, un salon privé est créé;</li> <li>- Sinon, l'invitation est supprimée.</li> </ul>		
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.		
CHESS	alias	Invite ou accepte une invitation pour une partie de jeu d'échecs.	CHESS Annie
	<p>Si les 2 utilisateurs ne sont pas en chat privé, le serveur ignore la commande.</p> <p>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 et informe les 2 personnes en leur communiquant leur couleur.</p> <p>Sinon, le serveur crée une nouvelle invitation dans le salon de chat privé et informe le destinataire. Ensuite :</p>		



Commande	Arguments	Description	Exemple
	<ul style="list-style-type: none"> <li>- Si le destinataire de l'invitation l'accepte, une partie d'échecs est créé;</li> <li>- Sinon, l'invitation est supprimée. <b>L'utilisateur utilisera la même commande DECLINE pour refuser l'invitation aux échecs.</b></li> </ul>		
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 <i>alias1:alias2:alias3...</i>		
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 <i>msg1\nmsg2\nmsg3...</i>		
JOIN	alias	Informe un client de la réception d'une invitation à un chat privé.	JOINOK 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	<b>Informe le client que alias a quitté le salon privé.</b>	<b>QUIT alias</b>





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		Invalide un déplacement de pièce envoyé par un client.	INVALID
Le client prend acte et essaie un autre déplacement.			
CHESSOK	alias couleur	Valide le démarrage d'une partie d'échecs pour les clients	CHESSOK Annie n ou : CHESSOK Annie b
Le client initie un nouvel objet <i>EtatPartieEchecs</i> 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