# Compte Rendu : Projet programmation multitâche

Dans le cadre de notre cours de programmation multitâche et temps réel, nous avons réalisé ce projet. L’objectif est de créer un serveur de chat et d’implémenter au fur et à mesure les clients.

Le projet peut être présenté sous trois versions :

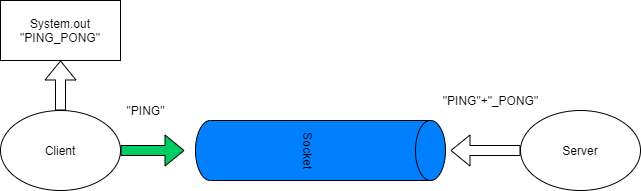
* V0 est composée du serveur et d’un client (le but étant de vérifier la communication à travers les sockets)
* V1 est composée du serveur et de deux clients, capable d’échanger entre eux
* V2 est composé du serveur et peut recevoir un nombre n de clients (+ GUI)

Le projet nous a permis de mieux comprendre le fonctionnement et les enjeux de la programmation multitâche. Nous avons appris comment faire communiquer client et server, mais aussi et surtout différents clients entre eux.



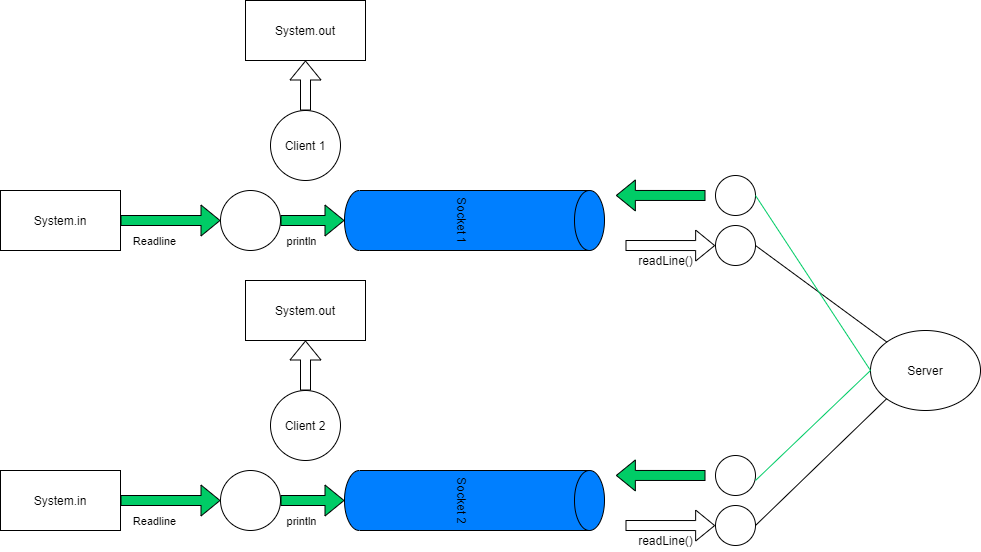
## Partie 1 : Modélisation

### Modélisation du projet : V0



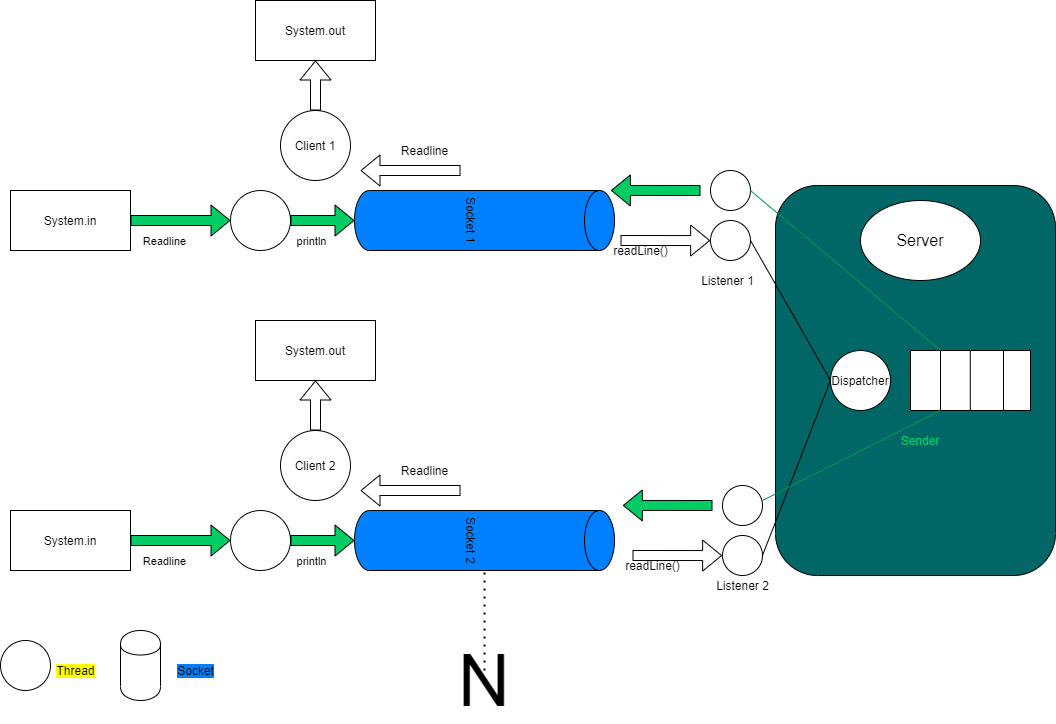
Dans ce schéma on peut voir comment fonctionne la V0. Lorsque le server est lancé, sur le port correspondant (par exemple 3333 dans cette V0) il est alors en attente de client. On viens alors simplement vérifier la communication entre les deux à l’aide d’un simple message « PING » du client, puis d’une réponde concaténé « PING »+ « \_PONG ».

### Modélisation du projet : V1



Dès la version 2, le schéma se complexifie. On peut voir ici la présence de plusieurs threads qui vont venir gérer les entrées et les sorties puis départager la réception et l’envoie au server. Dans ce cas-là, les deux clients peuvent communiquer. C’est un chat en duplex. Si l’on regarde l’exemple pour un seul client, on a tout d’abord une entrée (System.in) qui est traitée par **Readline** puis affiché par **println**. Ensuite du côté server on applique un **Readline** pour lire l’input et le renvoyer vers le thread destiné à l’autre client qui va alors parcourir le trajet annexe pour enfin être géré par le thread client qui va s’occuper d’afficher le message (**System.out**).

### Modélisation du projet : V2



La version 3 est donc celle qui est la plus aboutie. Avec cette version on peut faire communiquer plusieurs Clients, sans avoir de limite théoriquement (mise à part pour les ressources physique de la machine). L’ajout par rapport à la version précedente est notamment le dispatcher qui permet de traiter les différentes arrivés des thread. Dans ce cas il est plus simple de voir ça comme un trie effectif de chaque envoie fait par l’utilisateur. De cette manière le dispatcher sais à de quelle manière envoyer ce qu’il reçoit. Nous allons le voir plus tard mais ceci est particulièrement utile pour une des fonctionnalités supplémentaire que nous avons ajoutés (PM).

## Partie 2 : Fonctionnalités CLIENT/SERVER

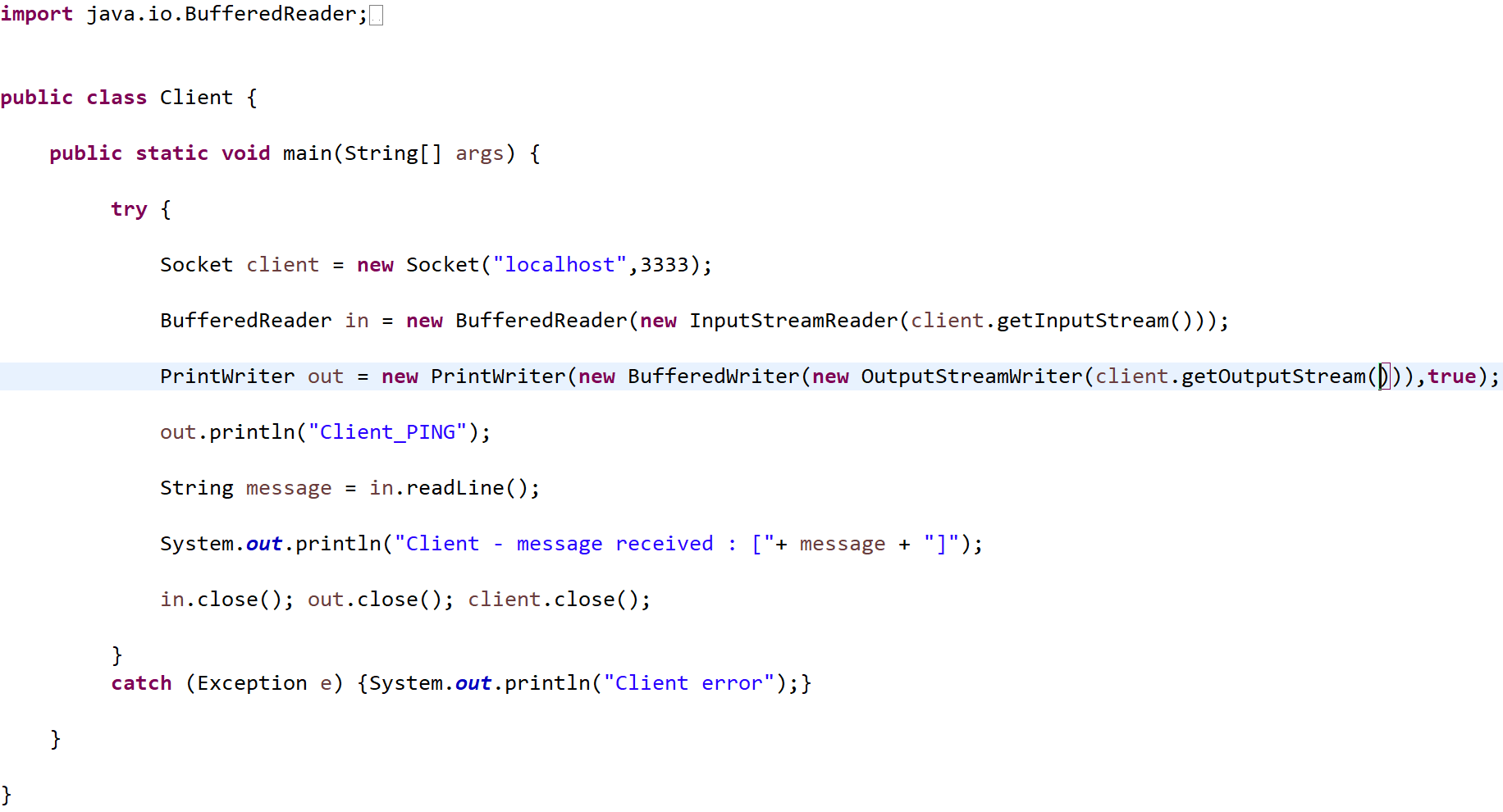
### Server V0



Server.java

Le premier Server représente le squelette pour toutes les version. On trouve le ServerSocket server et le Socket client. Le socket client est crée côté server après que le premier client accède au port du localhost. A l’aide du PrintWriter, le Server renvoie le message reçu par le client, permettant d’avoir la réponse « PING\_PONG » du côté client.

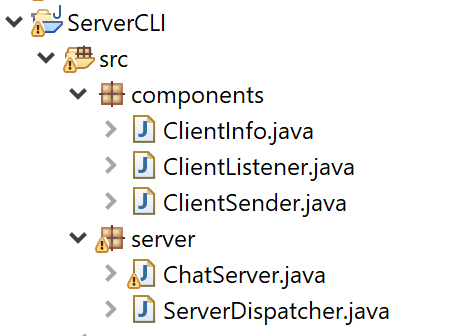
### Client V0



Client.java

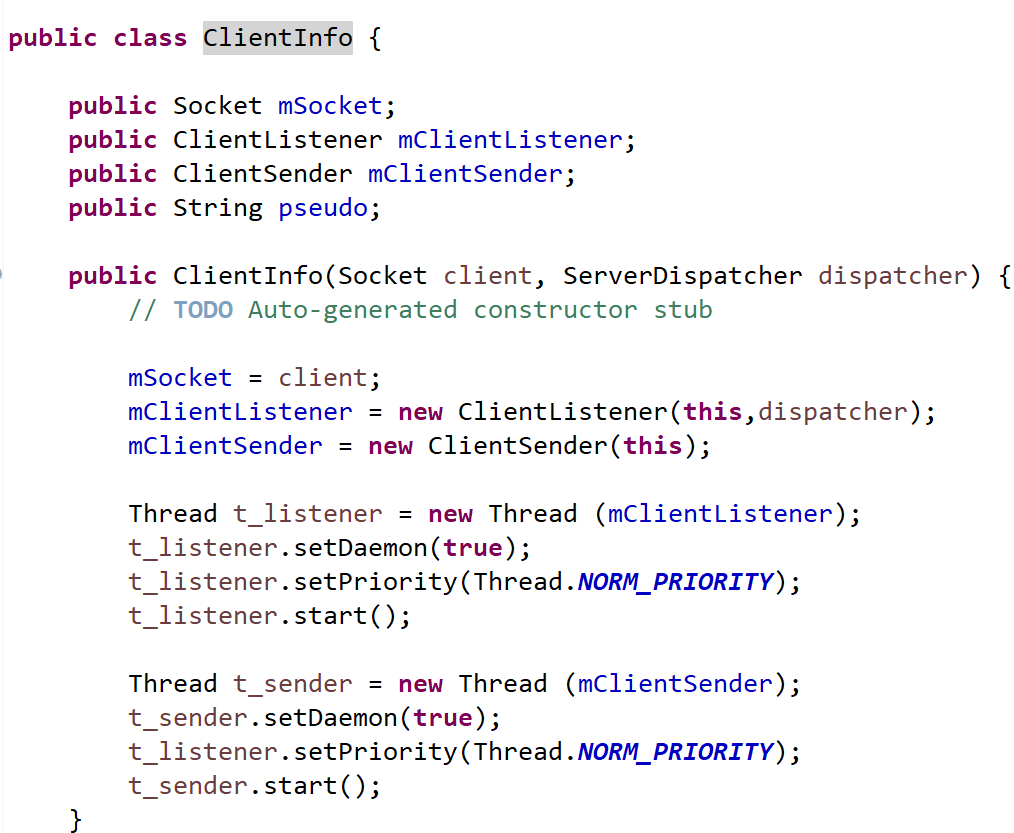
La classe Client est composé d’un socket (localhost :3333). La lecture en entrée est faite grâce au BufferedReader **in** tandis que la sortie est faite grâce à PrintWriter **out.** De ce fait il est possible d’envoyer « Client\_PING » au Serveur. L’exception permet de gérer le cas où le client ne trouve pas de Serveur associé au port correspondant.

### Server V1

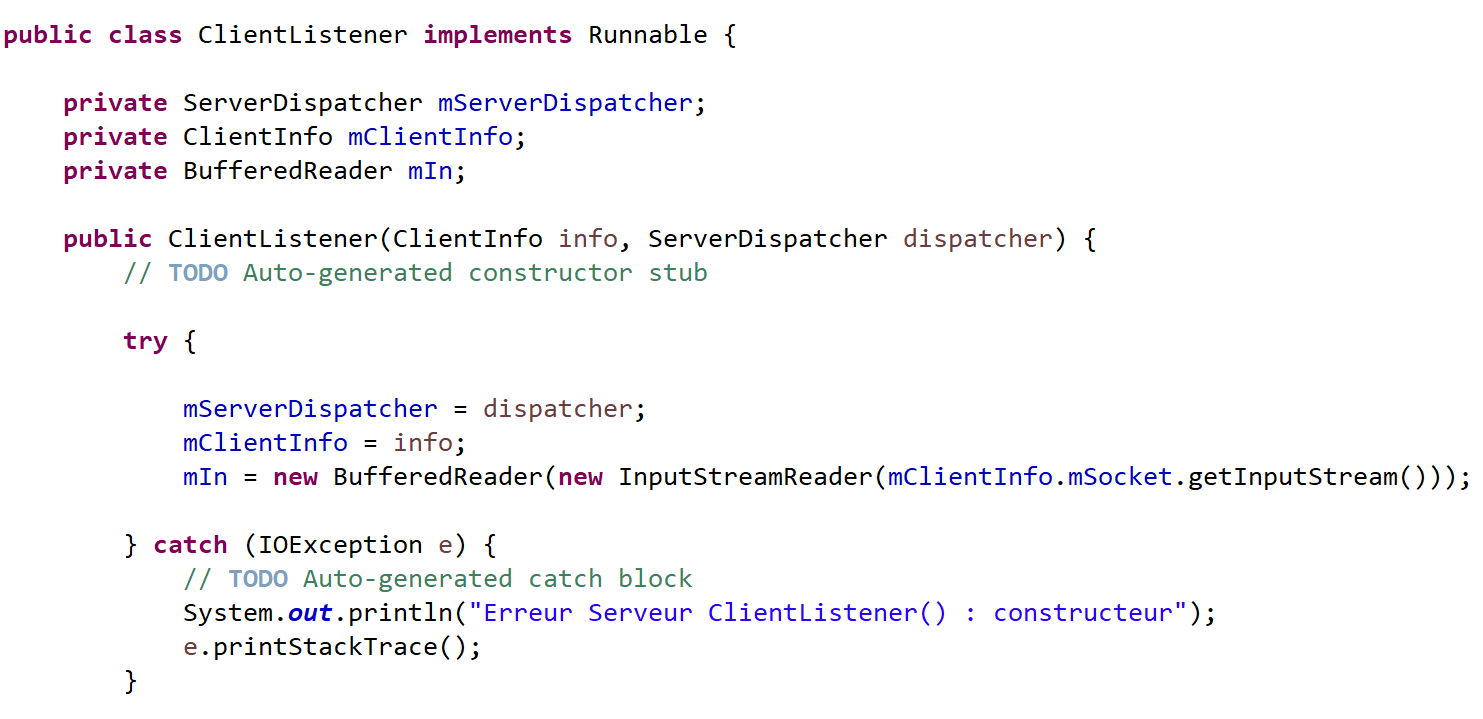


Voici la structure générale du ServerCLI qui est la version améliorée de la V1 du serveur. Ici nous avons la capacité de faire communiquer deux clients différents. Il n’y a pas de dispatcher pour gérer de potentiels clients en plus.

*ClientInfo :*

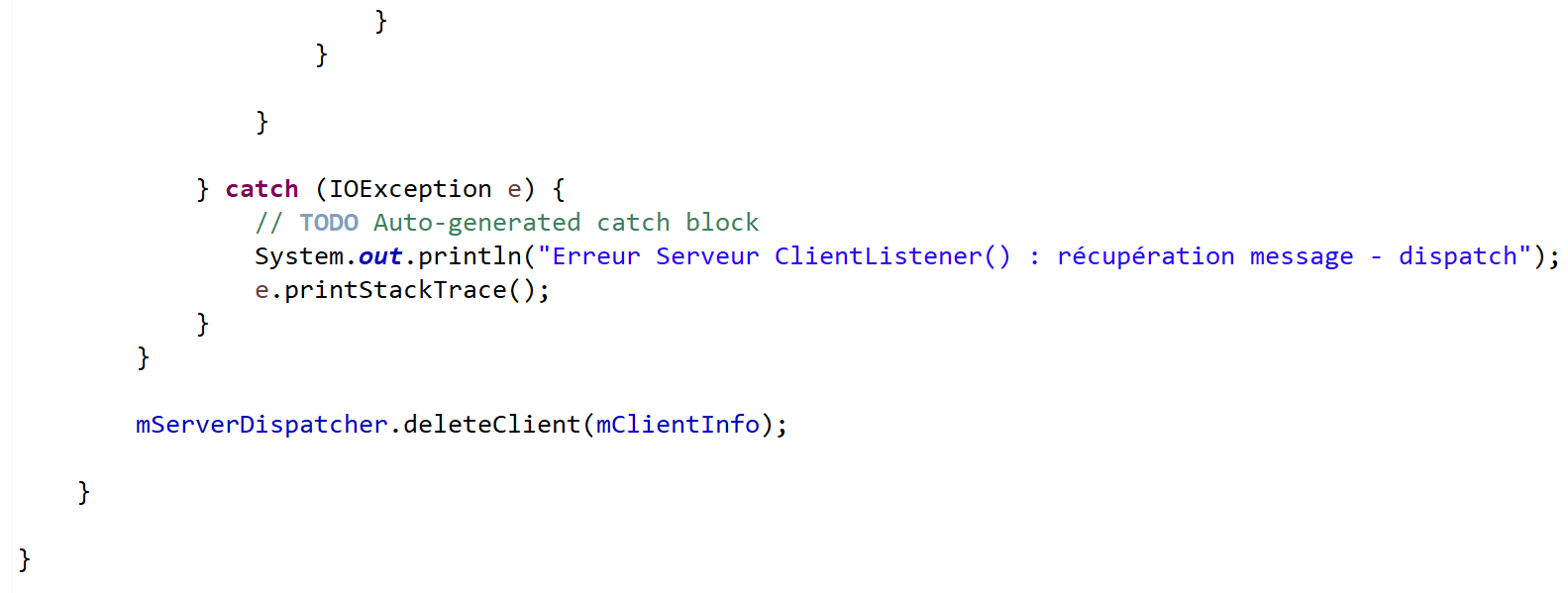


*ClientListener :*

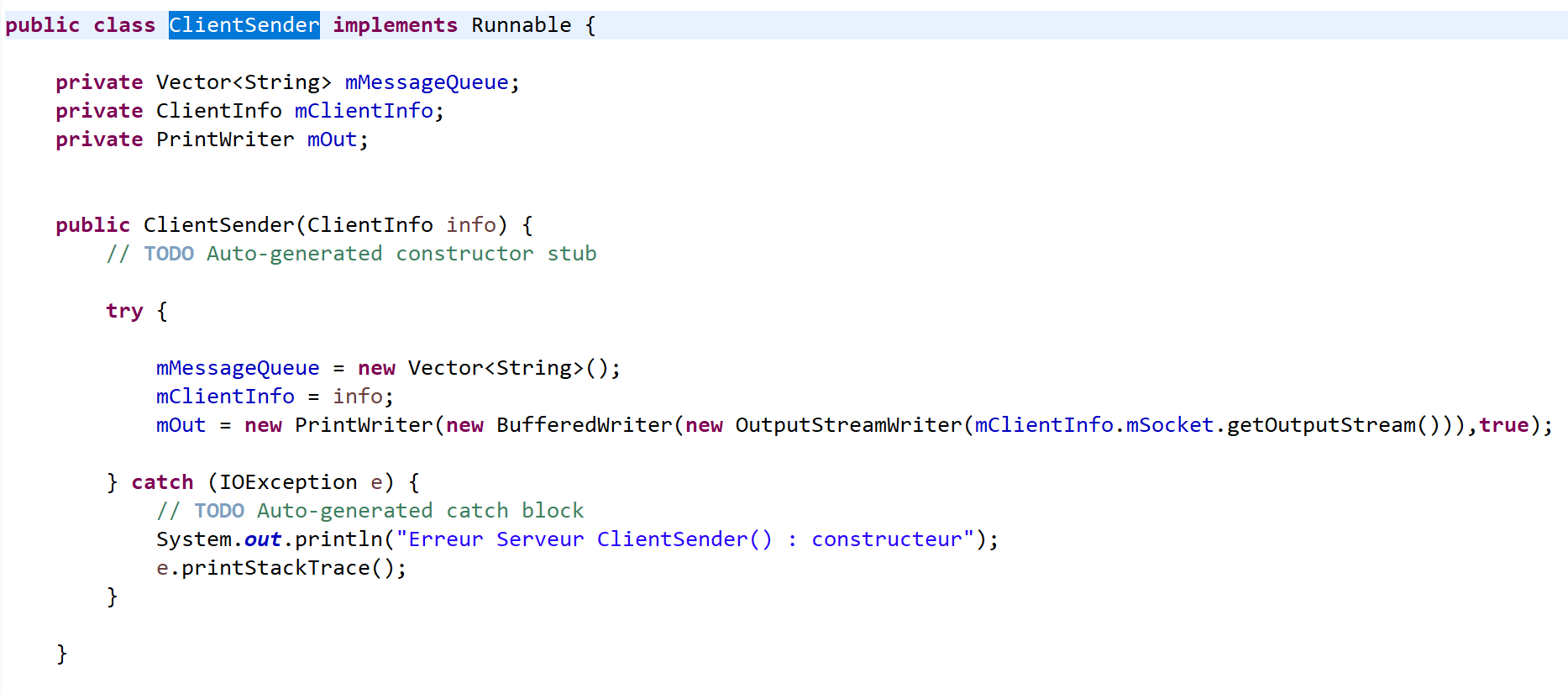


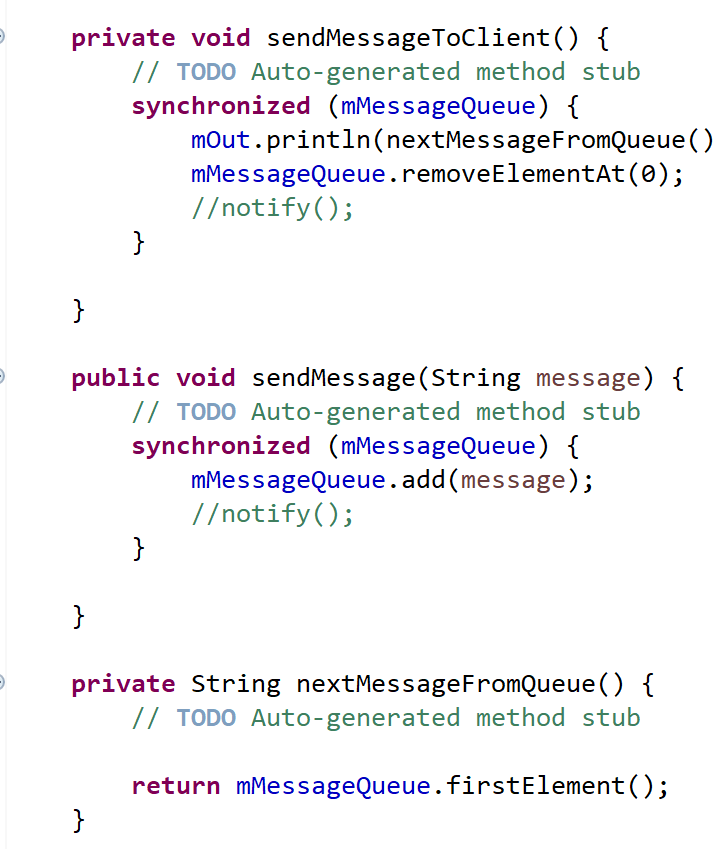


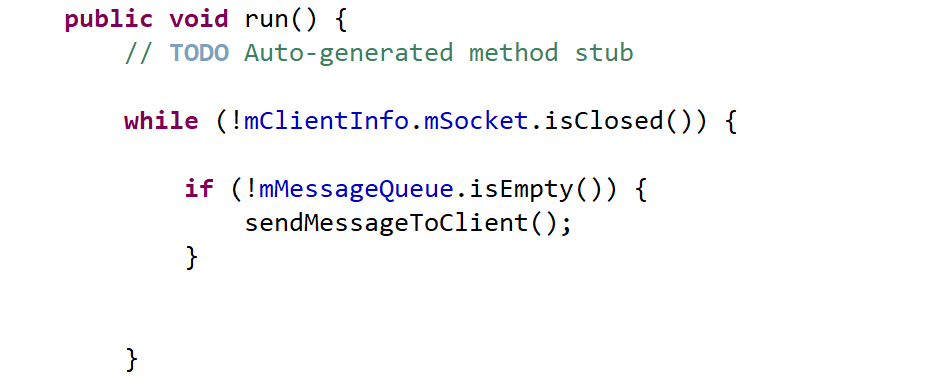




*ClientSender :*





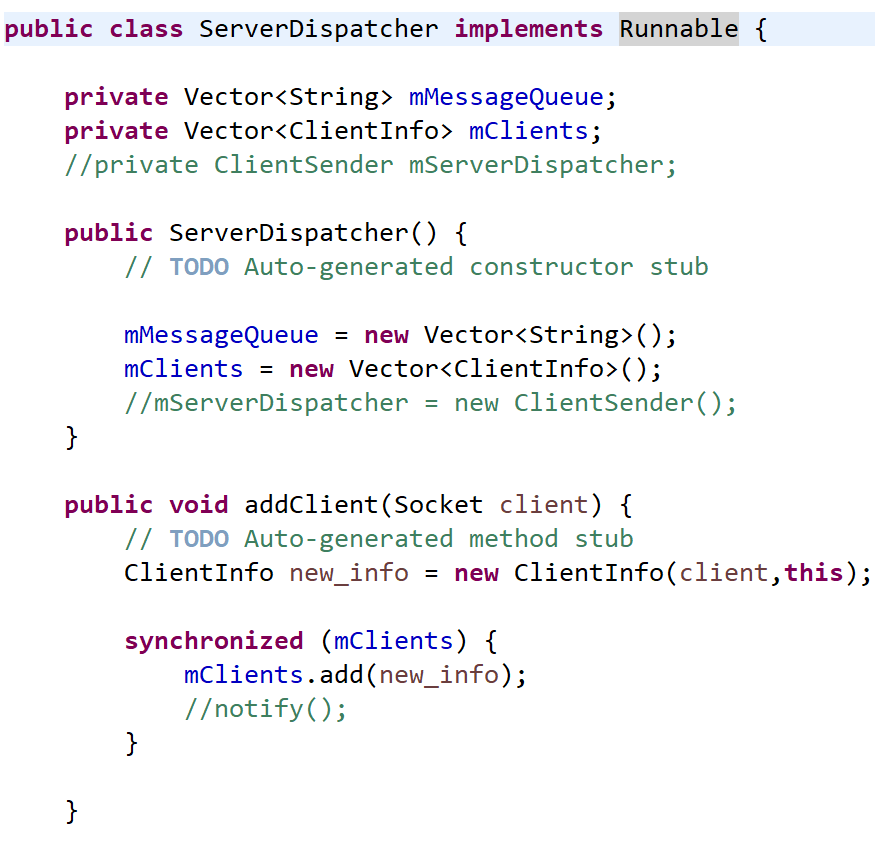


*ChatServer : Main*

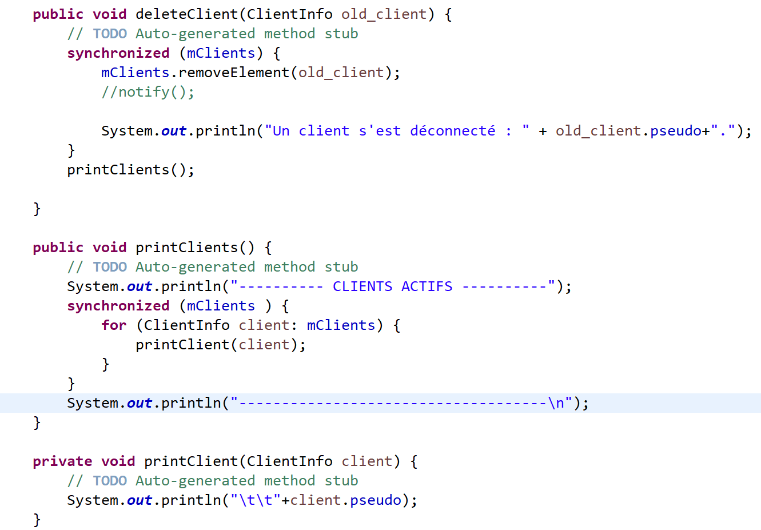
Seul attribut : **public** **static** **final** **int** ***LISTENING\_PORT*** = 5555;



*ServerDispatcher* :



ServerDispatcher.java – screenshot 1



ServerDispatcher.java – screenshot 2

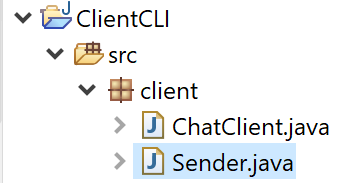


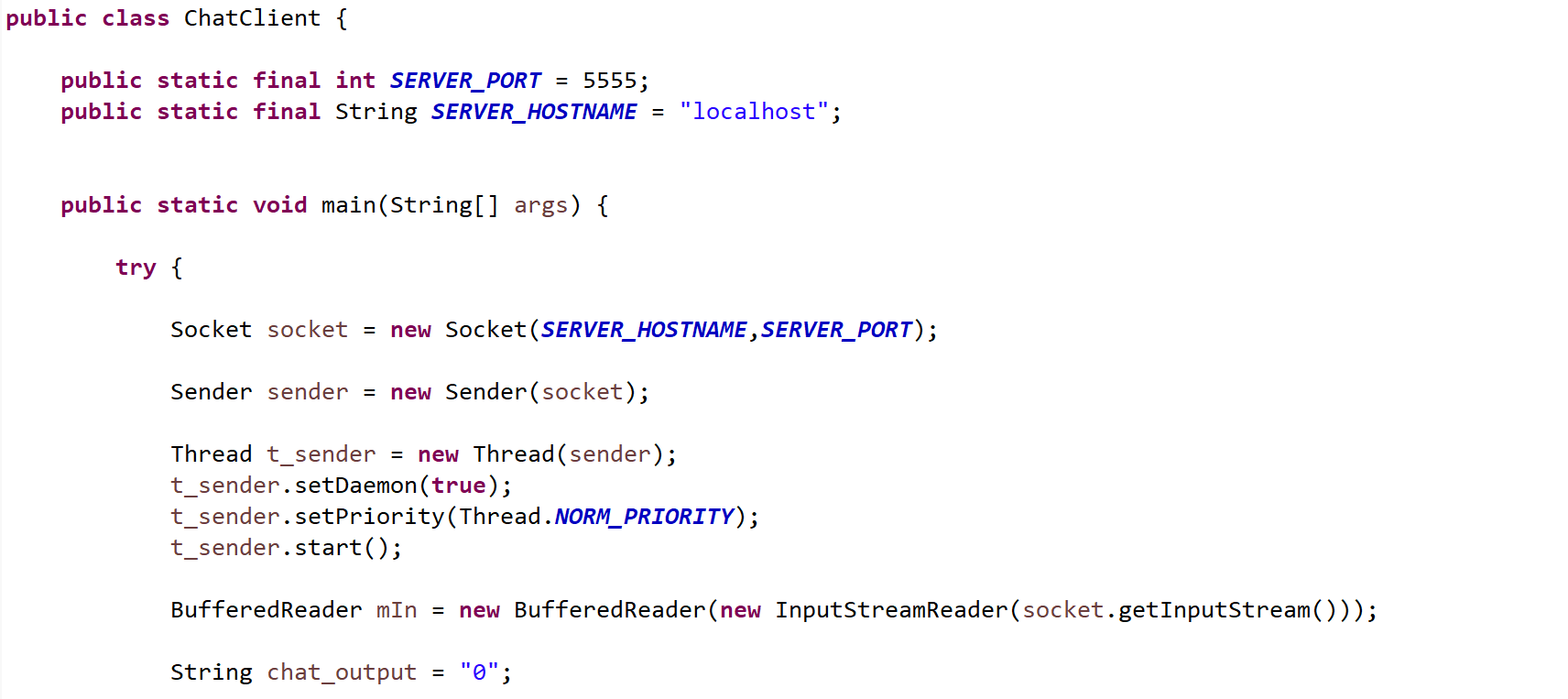
ServerDispatcher.java – screenshot 3

ServerDispatcher.java – screenshot 1

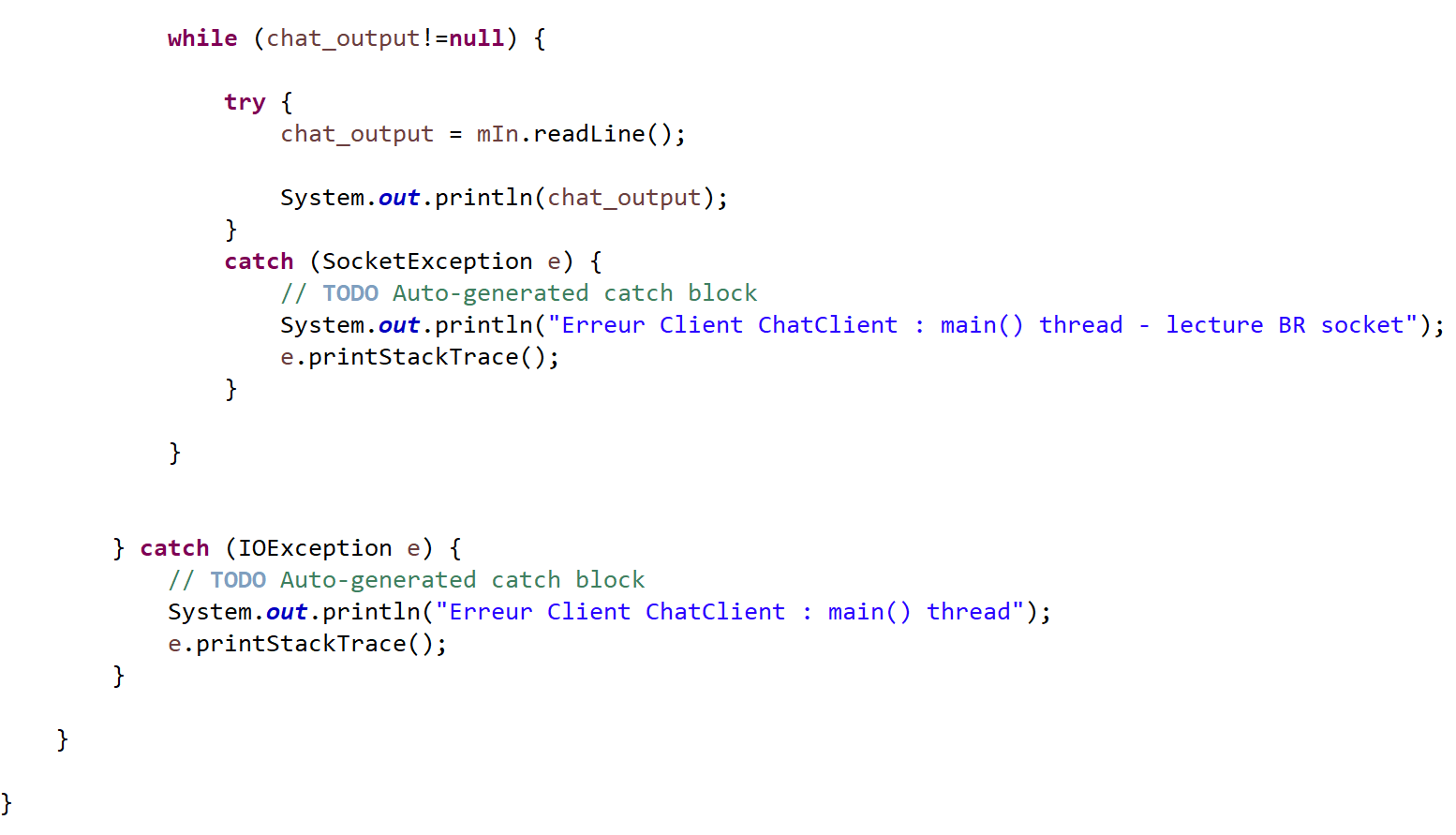
### Client V1

ChatClient :



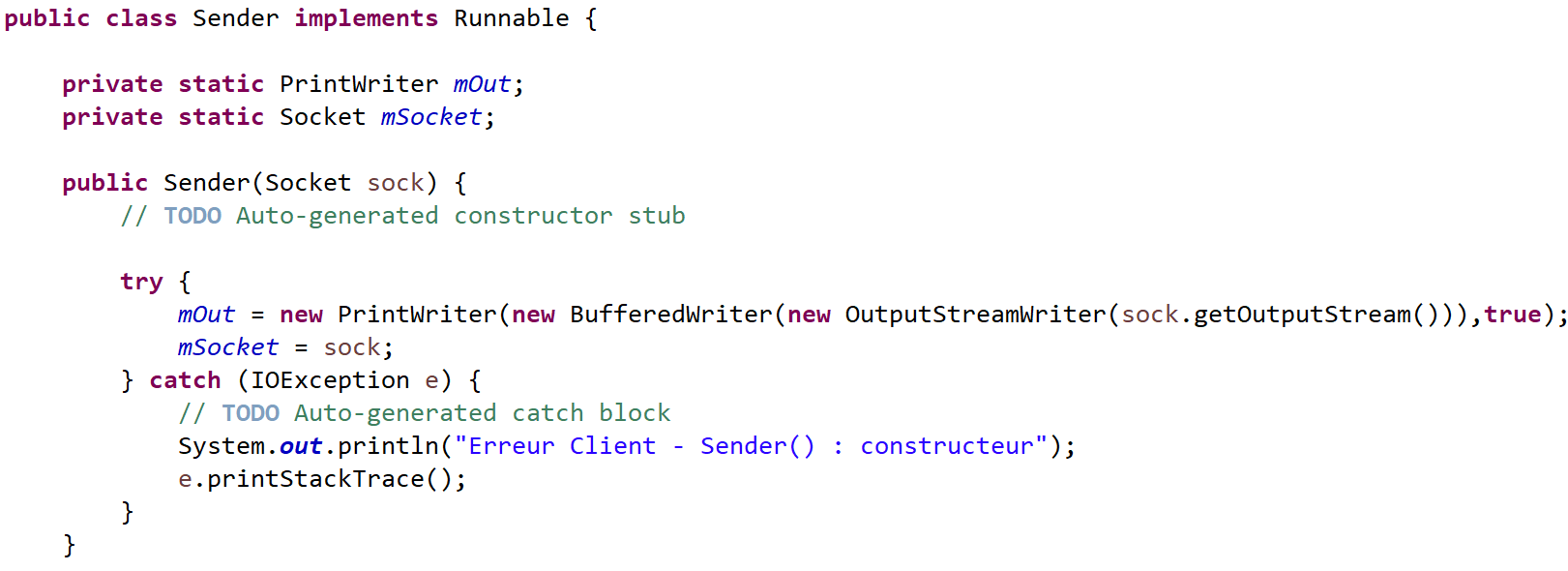


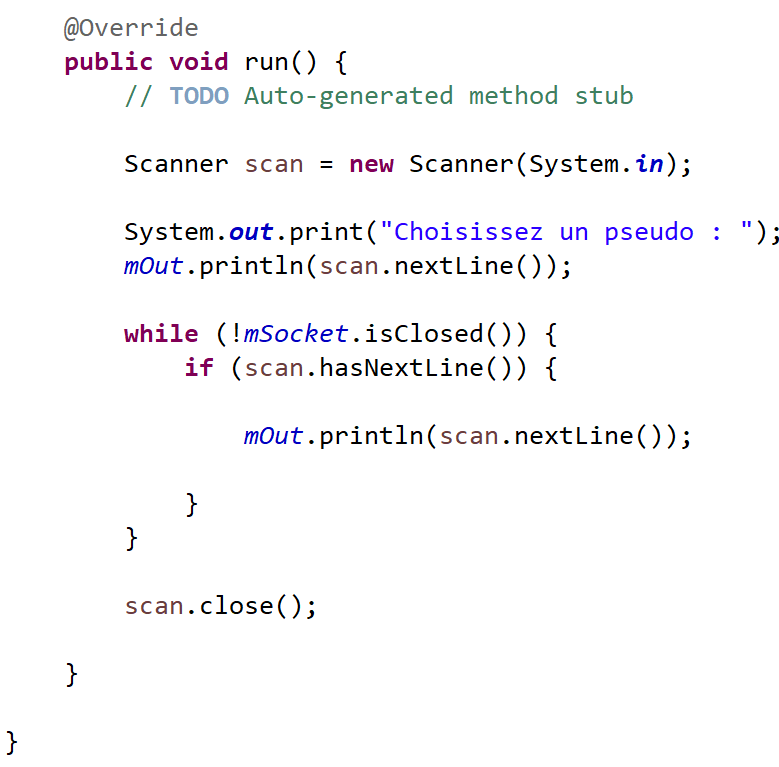
ChatClient.java – screenshot 1



ChatClient.java – screenshot 2

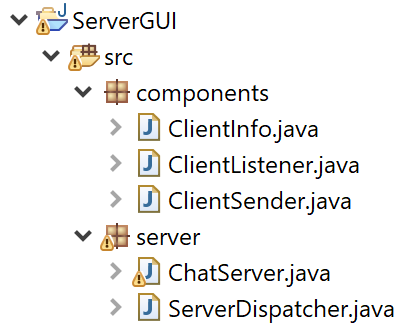
Sender :

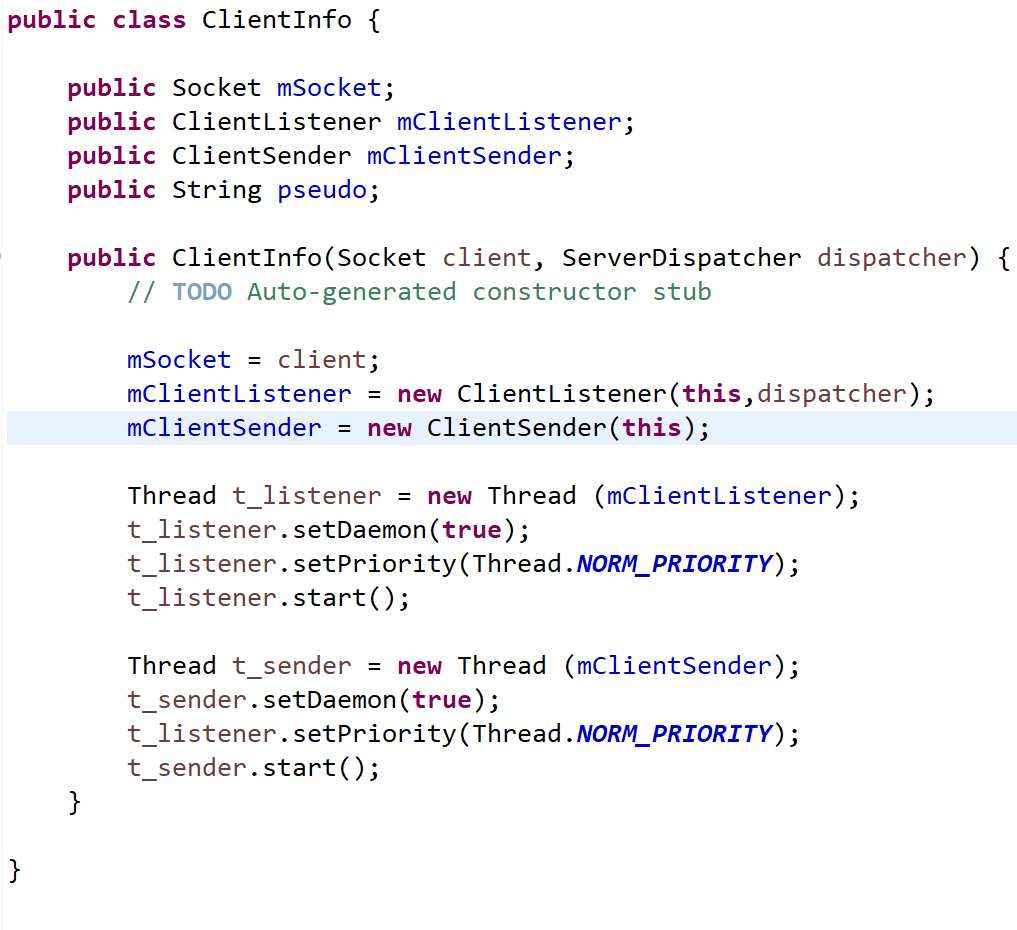
  
Sender.java – screenshot 1



Sender.java – screenshot 2

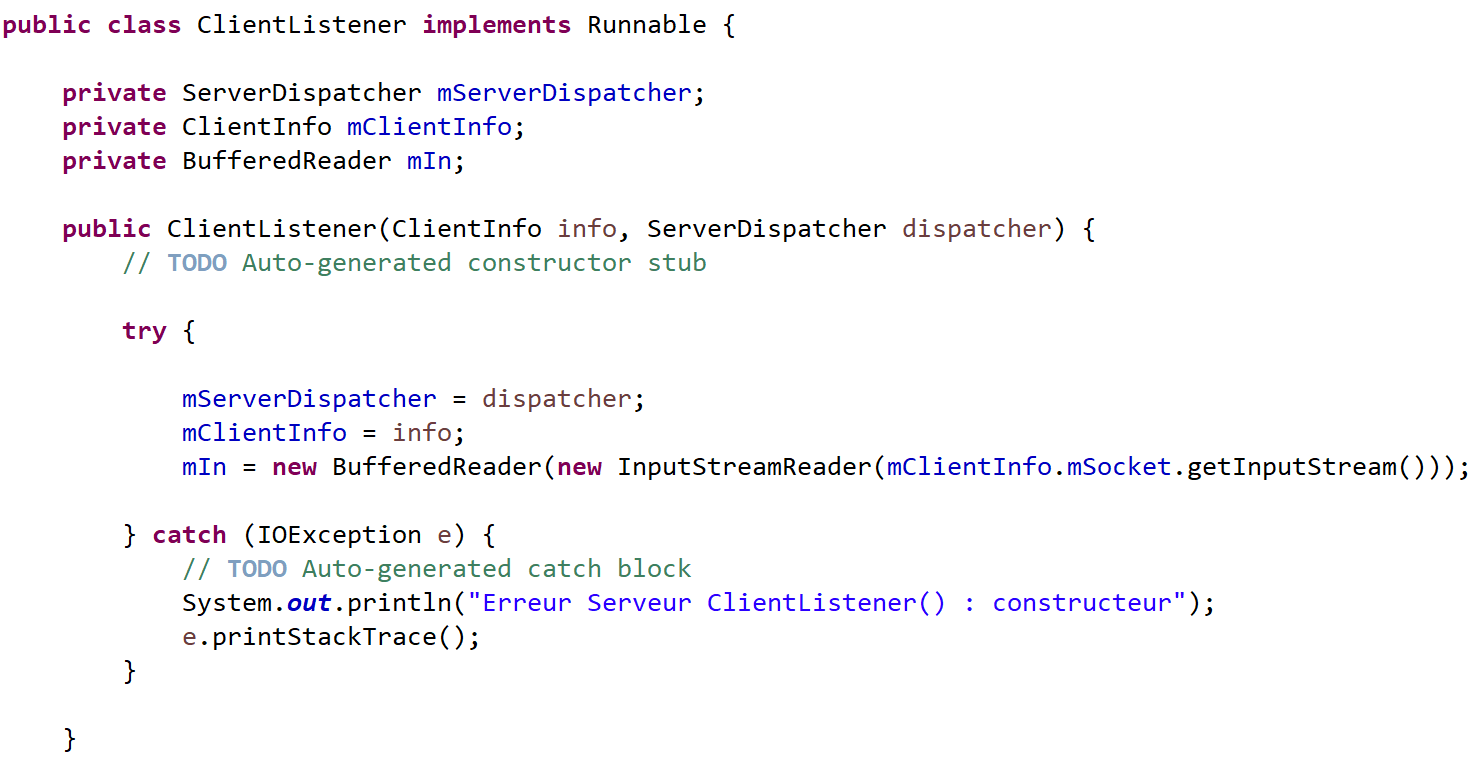
### Server V2



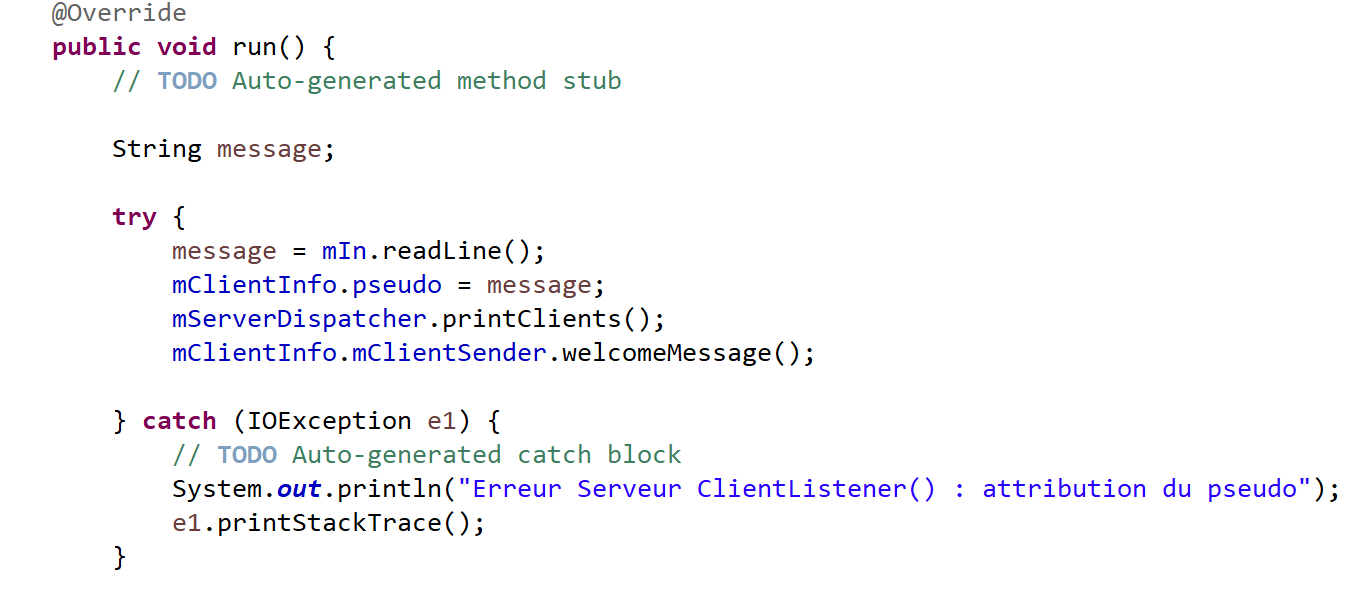


ClientInfo.java – screenshot 2

ClientListener :



ClientListernet.java – screenshot 1



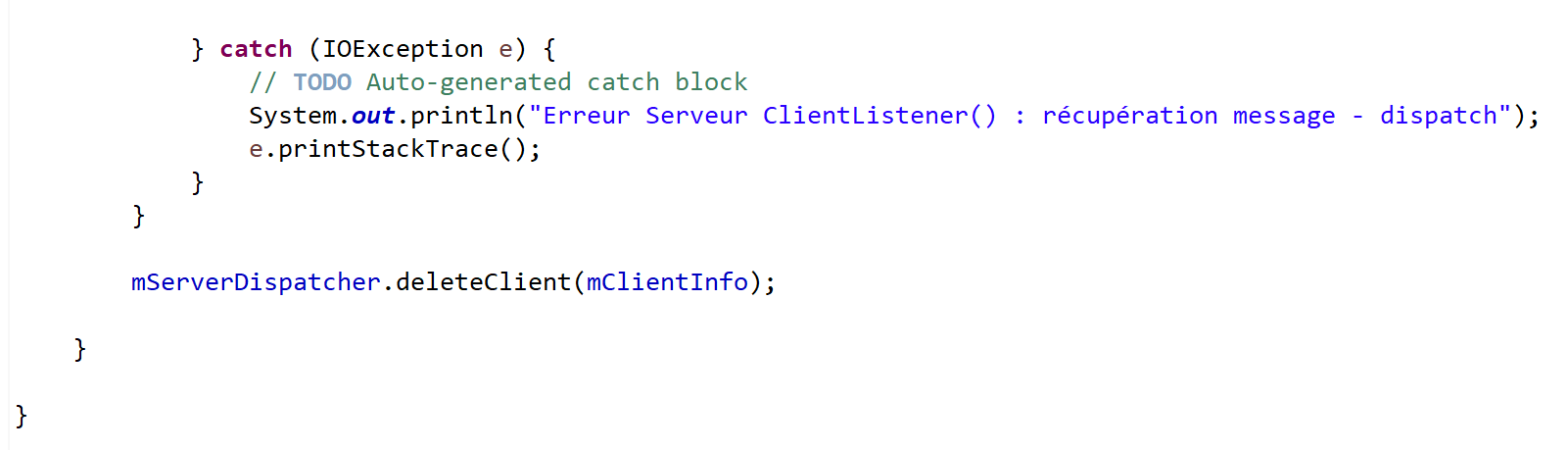
ClientListener.java – screenshot 2



ClientListener.java – screenshot 3



ClientListener.java – screenshot 4



ClientListener.java – screenshot 4

*ClientSender :*



ClientSender.java – screenshot 1

**

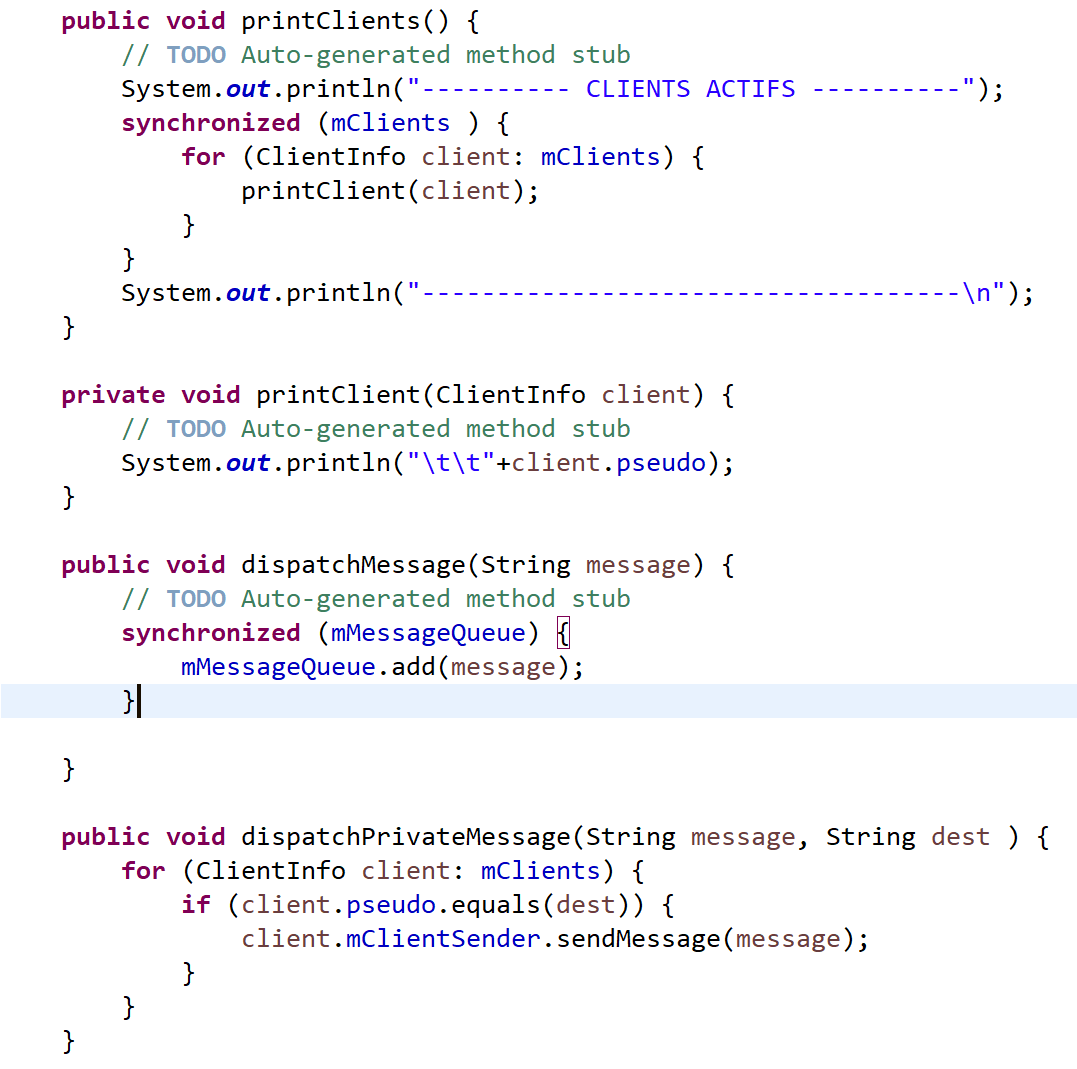
ClientSender.java – screenshot 2

ChatServer : Le code est ici le même

ServerDispatcher :



ServerDispatcher.java – screenshot 1

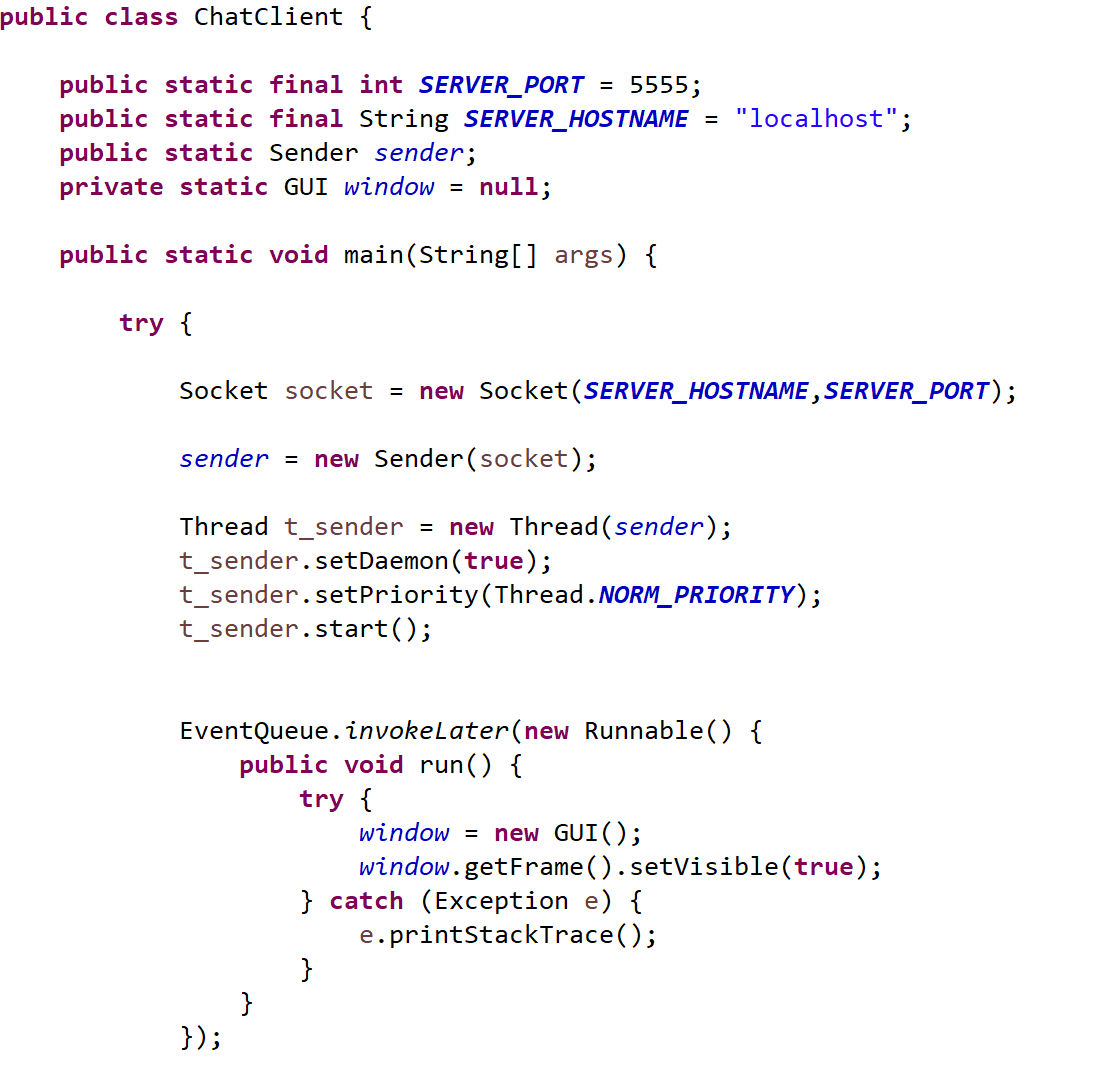


ServerDispatcher.java – screenshot 2

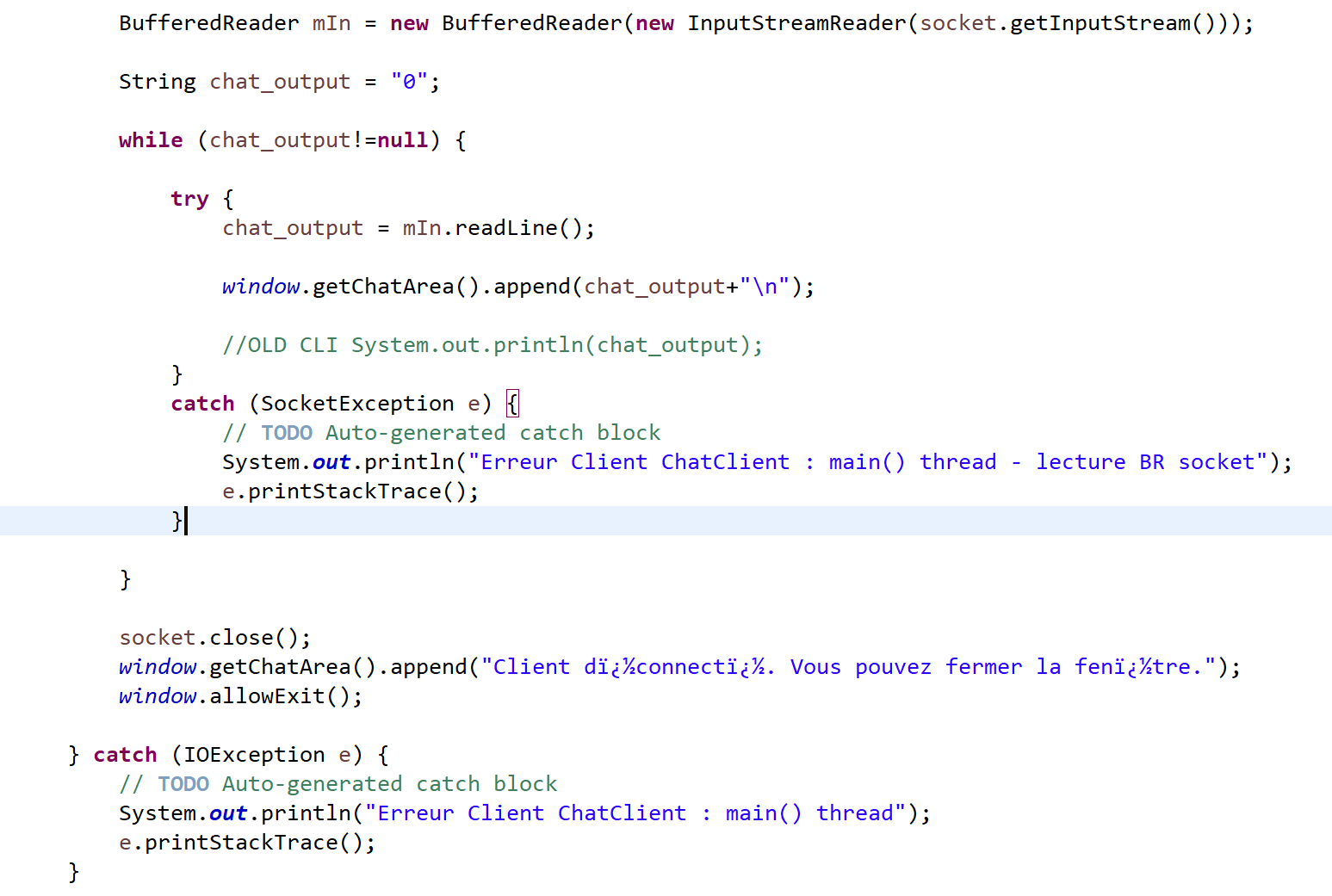


ServerDispatcher.java – screenshot 3

### Client V2



ChatClient.java – screenshot 1

**

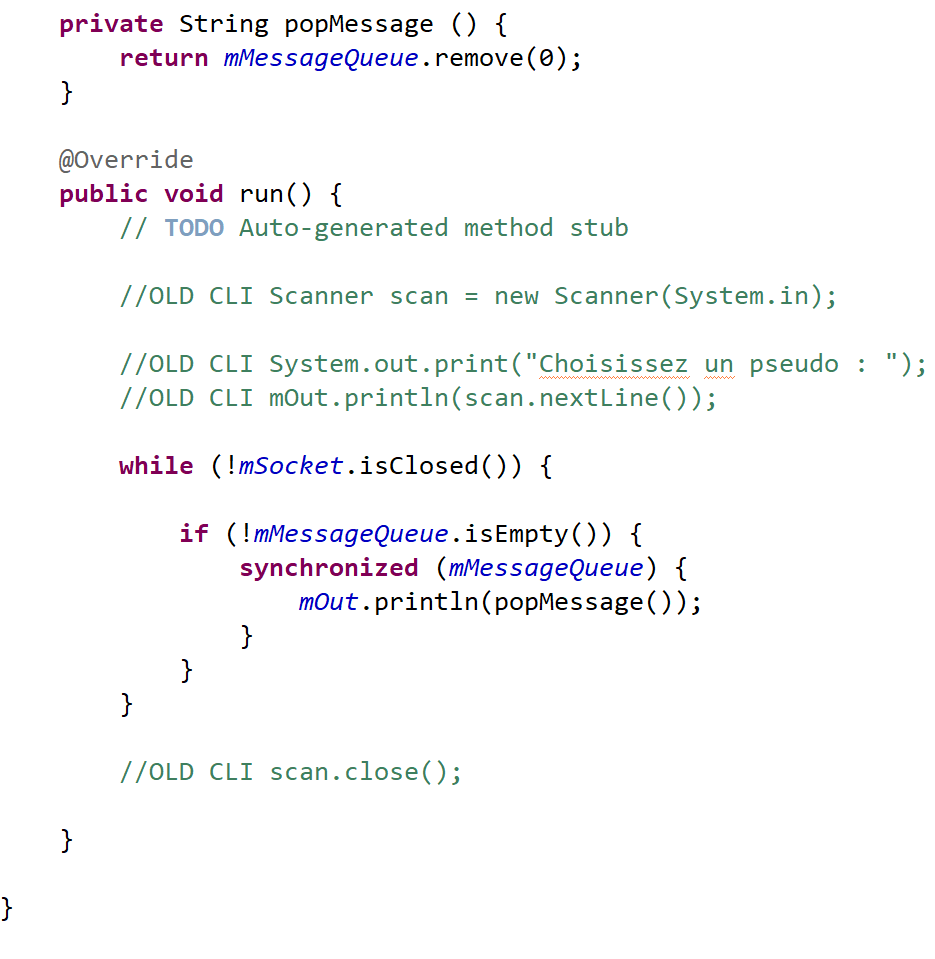
ChatClient.java – screenshot 2

Dans *ChatClient.java* on trouve aussi l’interface graphique dans la classe GUI. Je vous laisserais aller la voir, nous avons utilisés Window Builder, qui permet de créer simplement l’architecture d’une interface par widget.

*Sender* :



Sender.java – screenshot 1

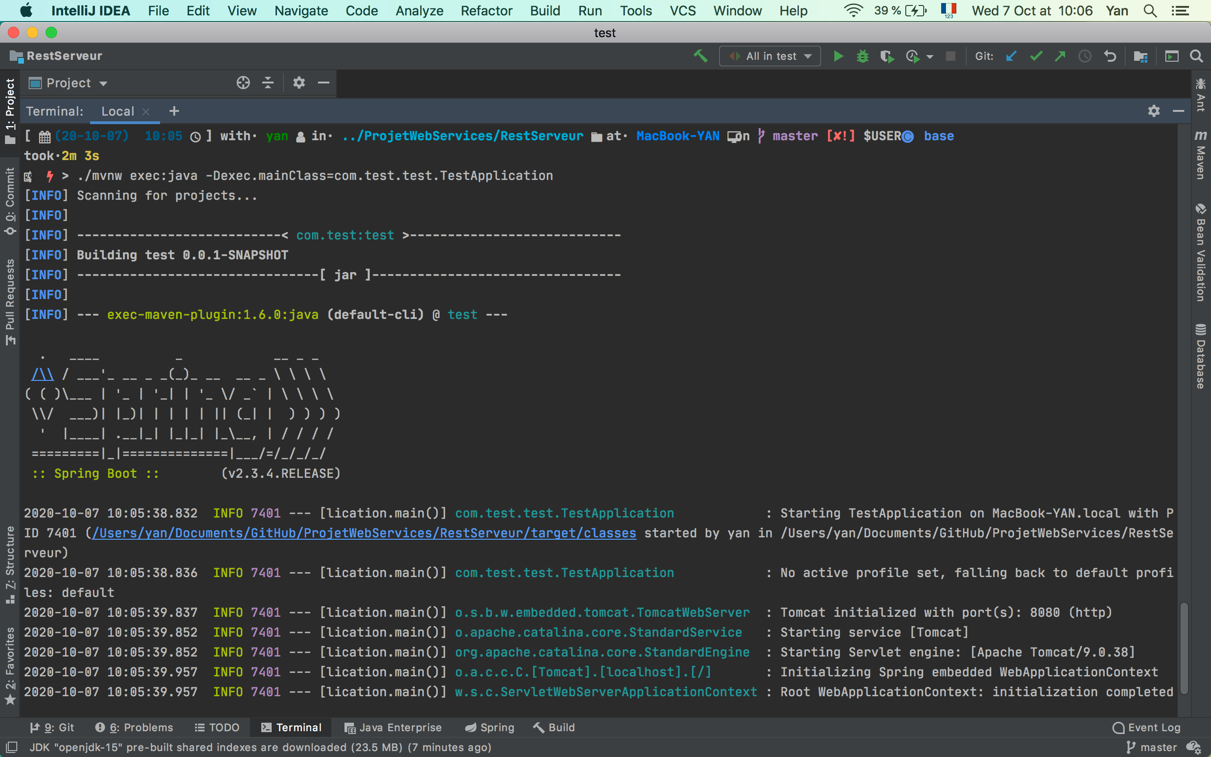


Sender.java – screenshot 2

## Partie 3 : Exécution

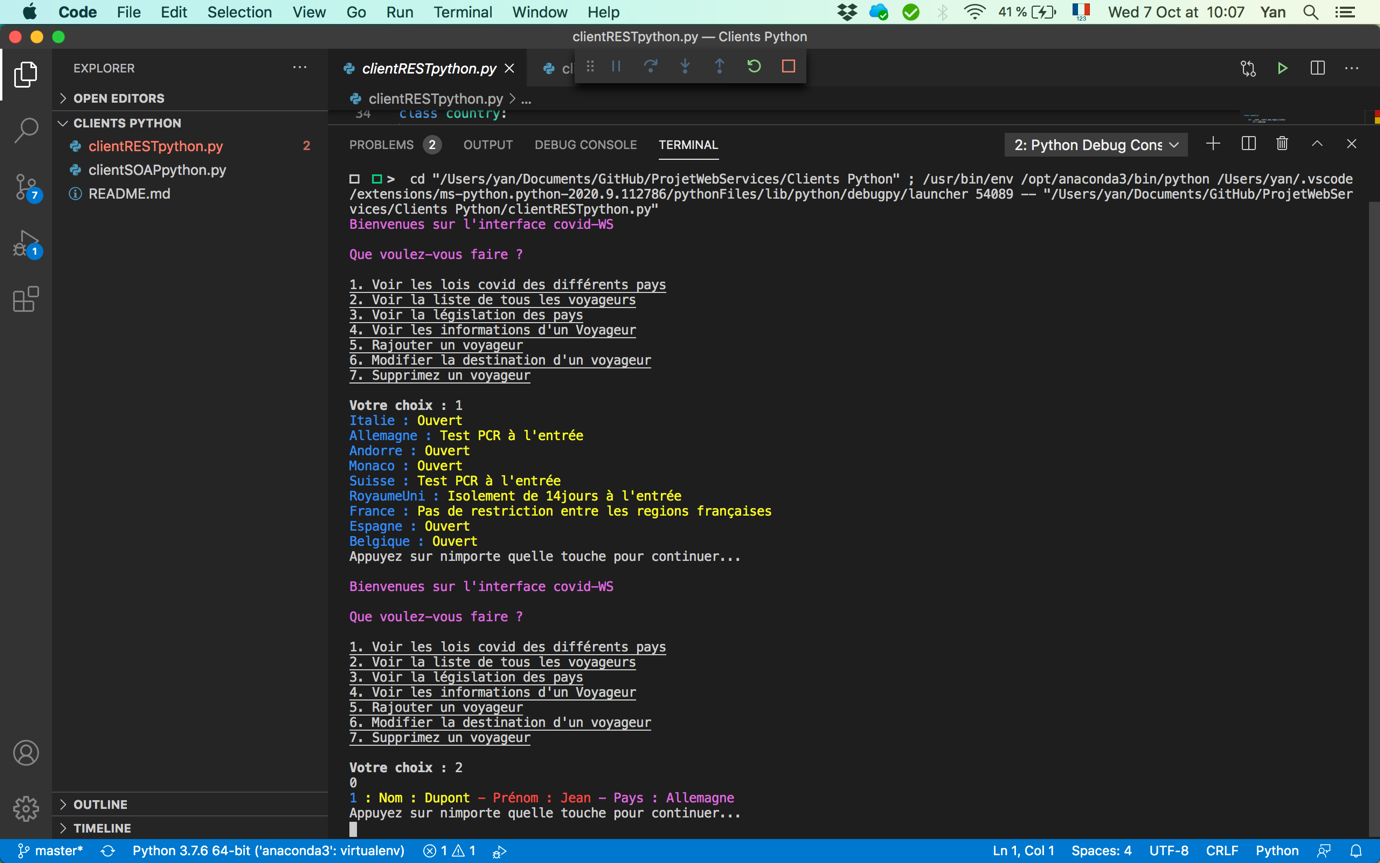
### Serveur REST – Java 12+

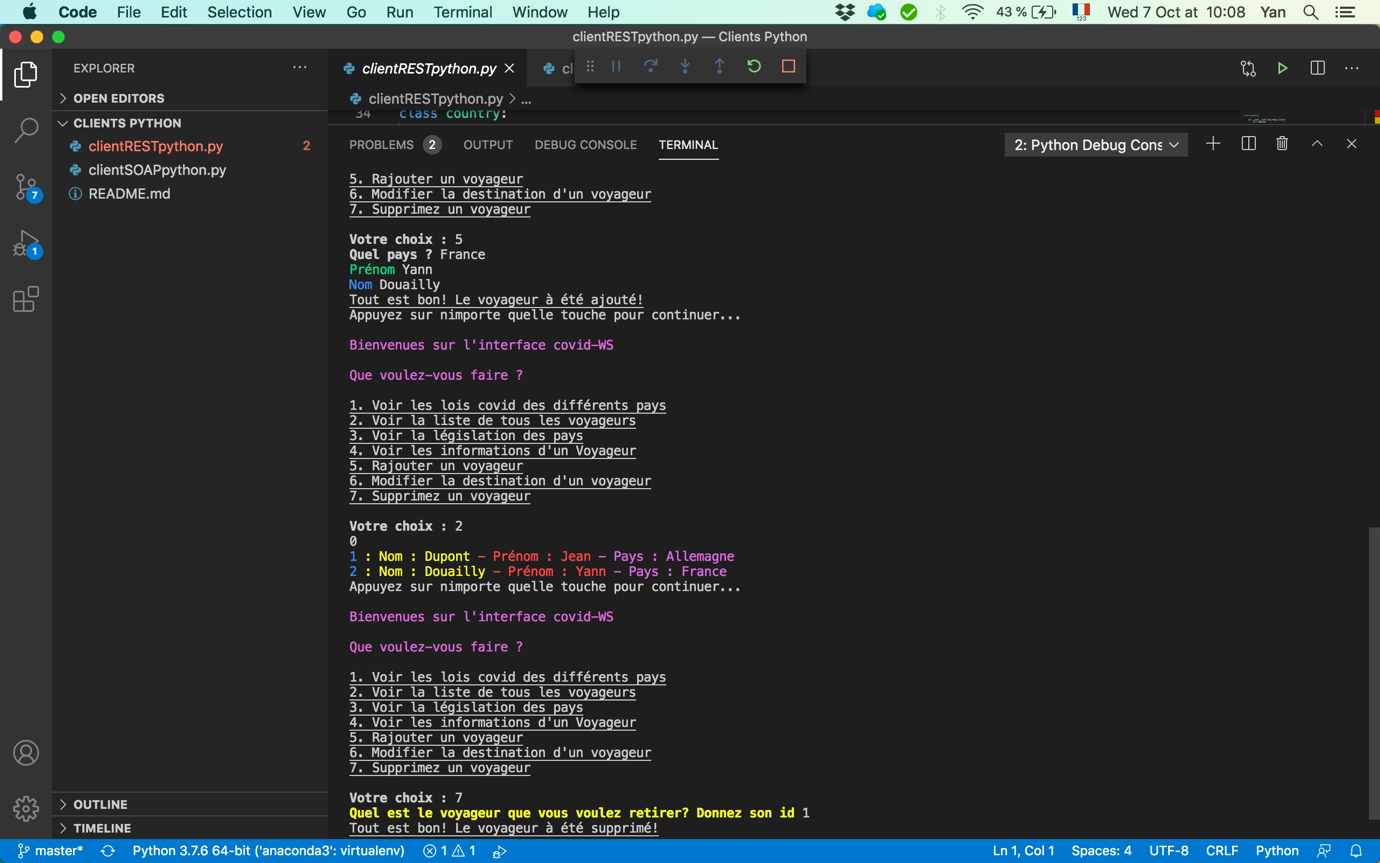
Un README est fourni avec le serveur REST.  
Il est lancé grâce à l’invite de commandes, intégré à l’IDE ou directement dans l’OS.



### Client REST - Python

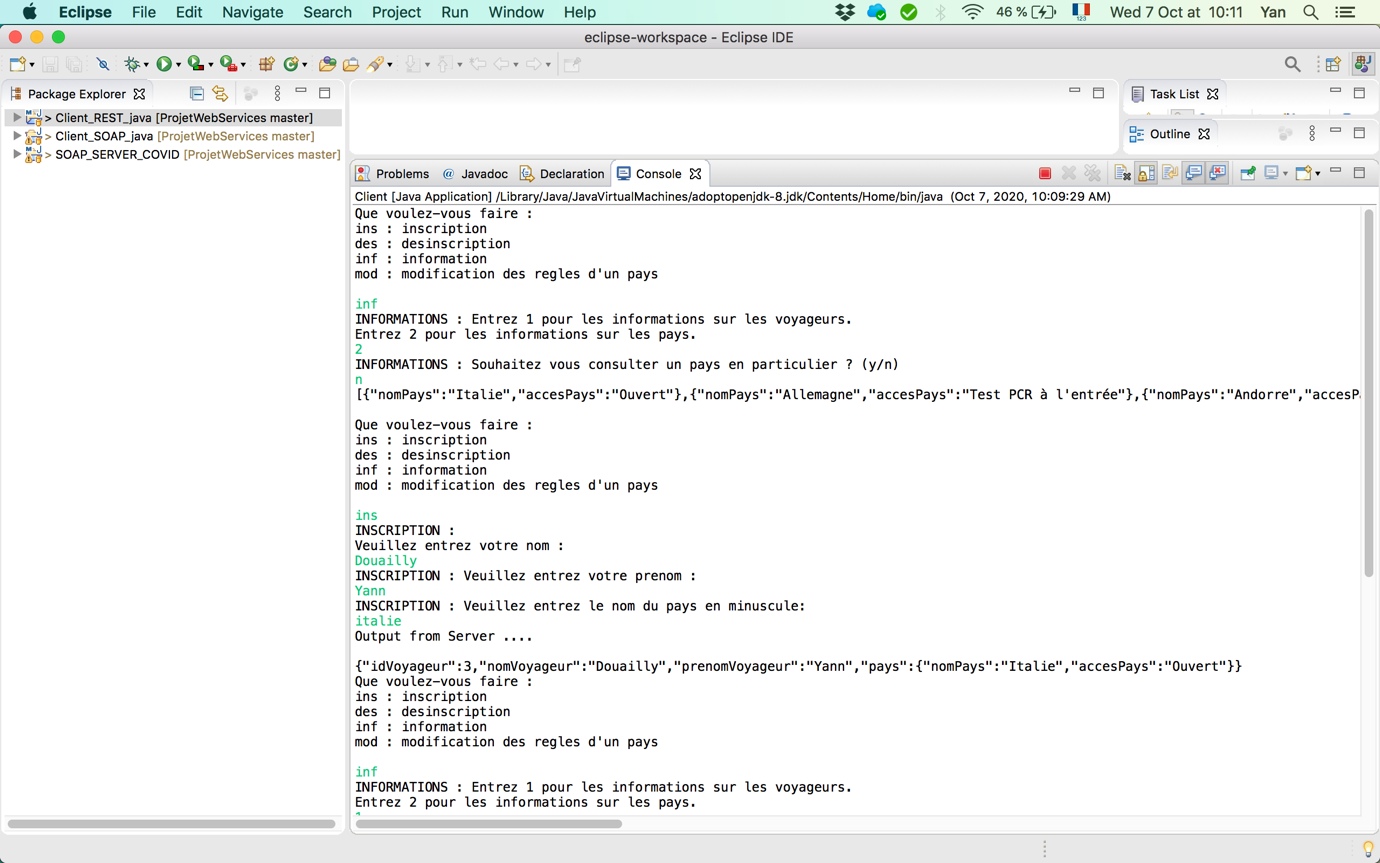
IDE conseillé : Visual Studio Code (latest)  
OS Compatible : Windows / macOS / Linux  
Lancement : “Run without debugging”

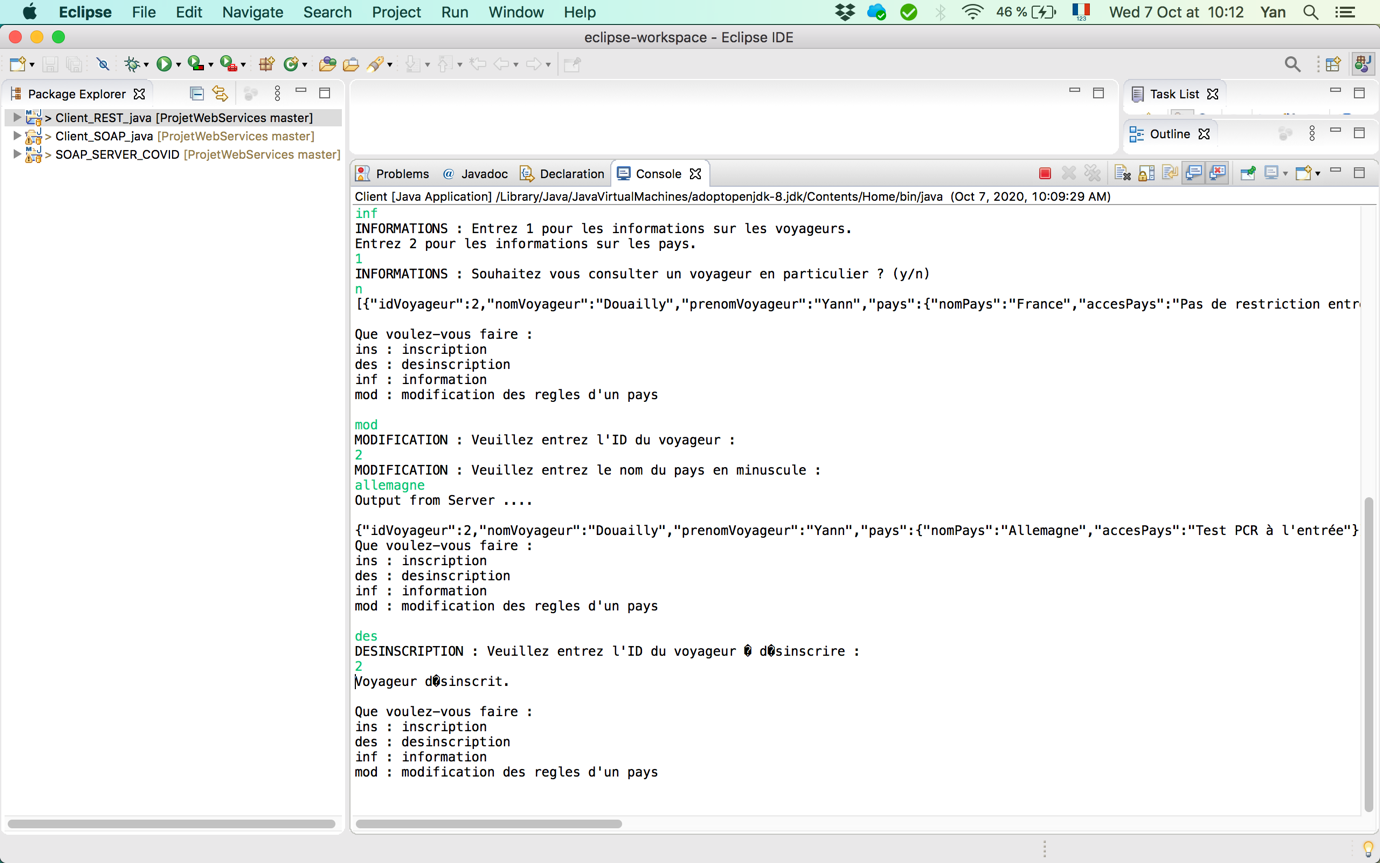




### Client REST – Java 8+

IDE conseillé : Eclipse IDE (2020.06 ou inférieure pour compatibilité Java 8)  
OS Compatible : Windows / macOS / Linux

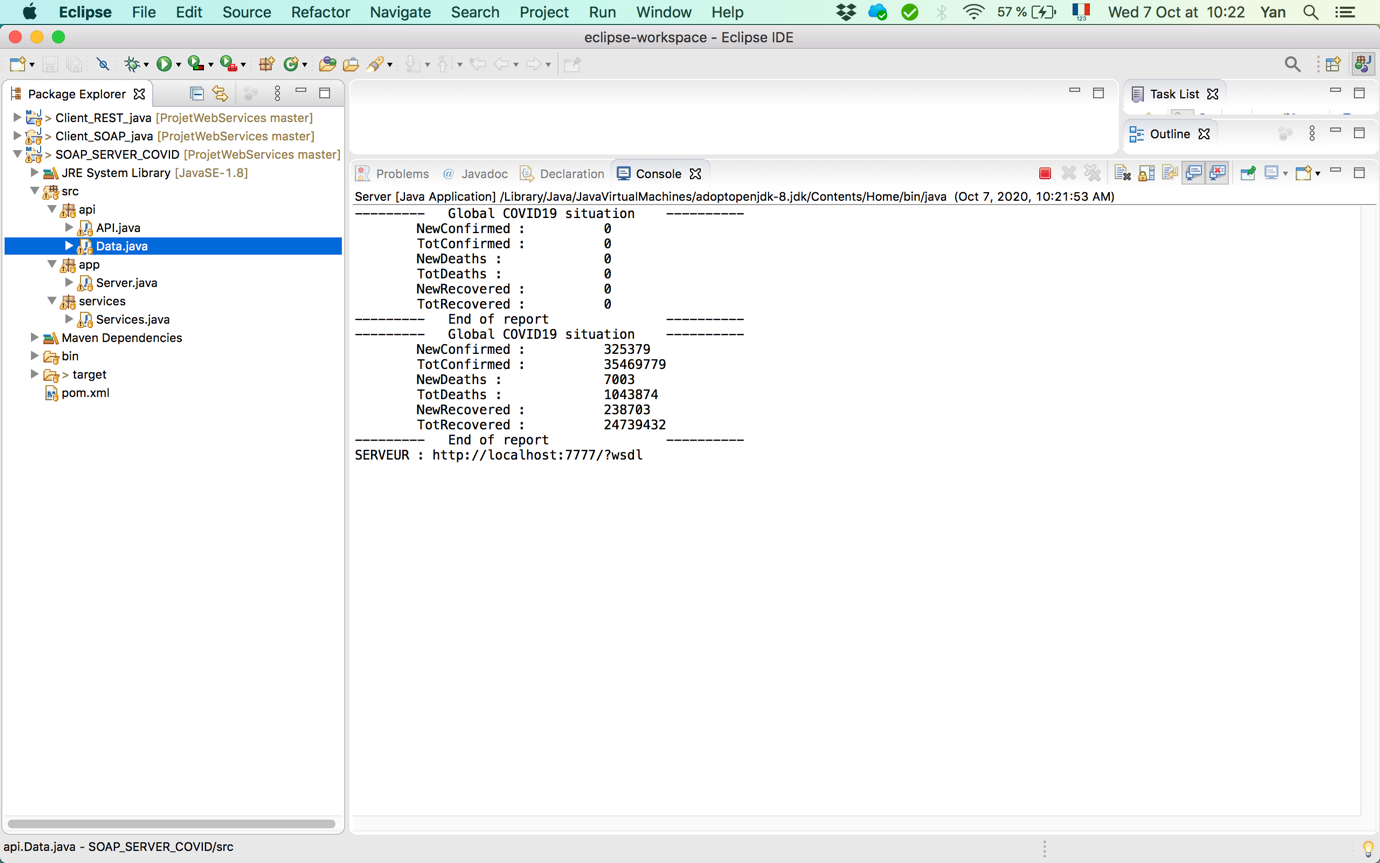




### Serveur SOAP – Java 8+

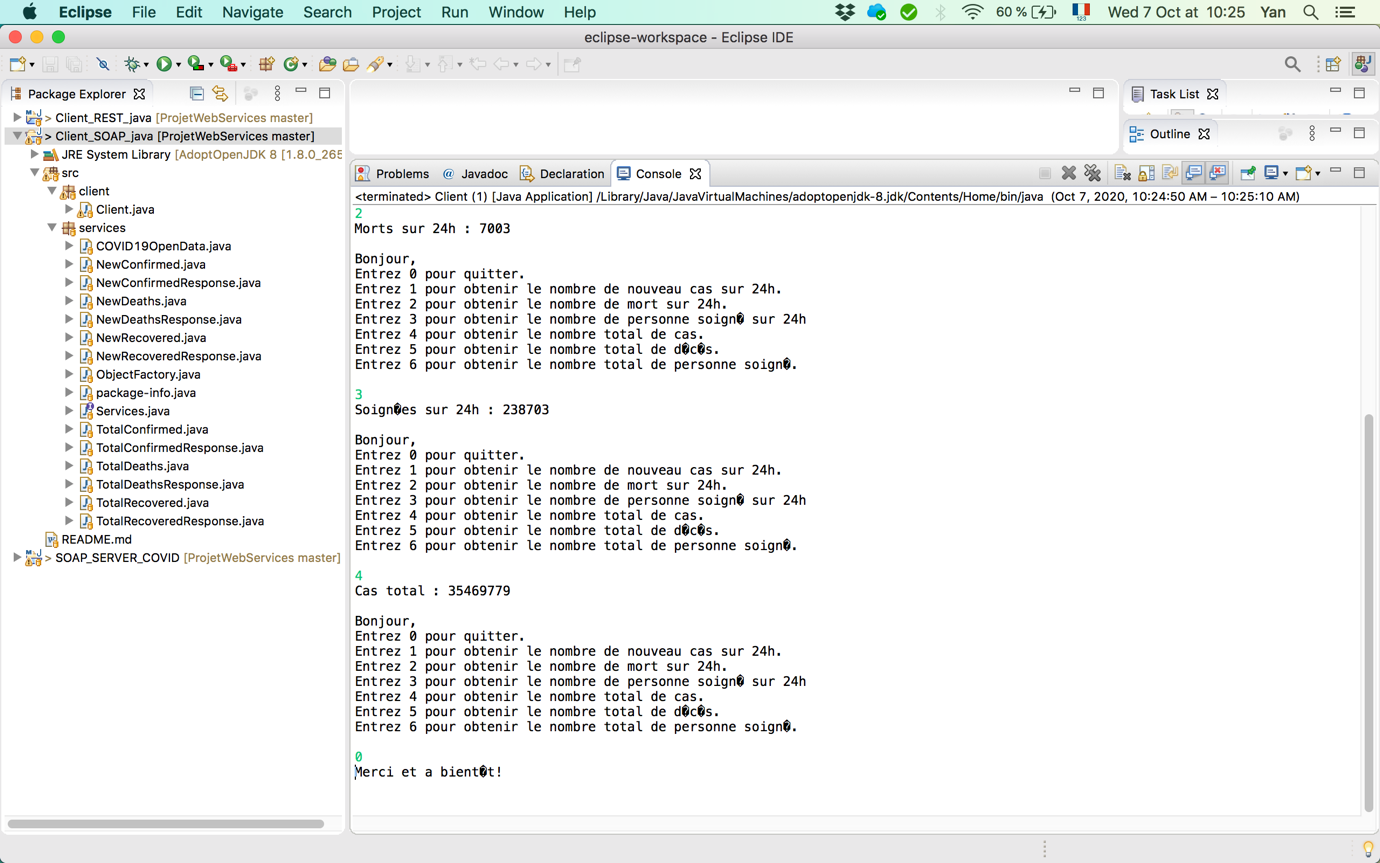
IDE conseillé : Eclipse IDE (2020.06 ou inférieure pour compatibilité Java 8)  
OS Compatible : Windows / macOS / Linux

Lancement : « Run project »



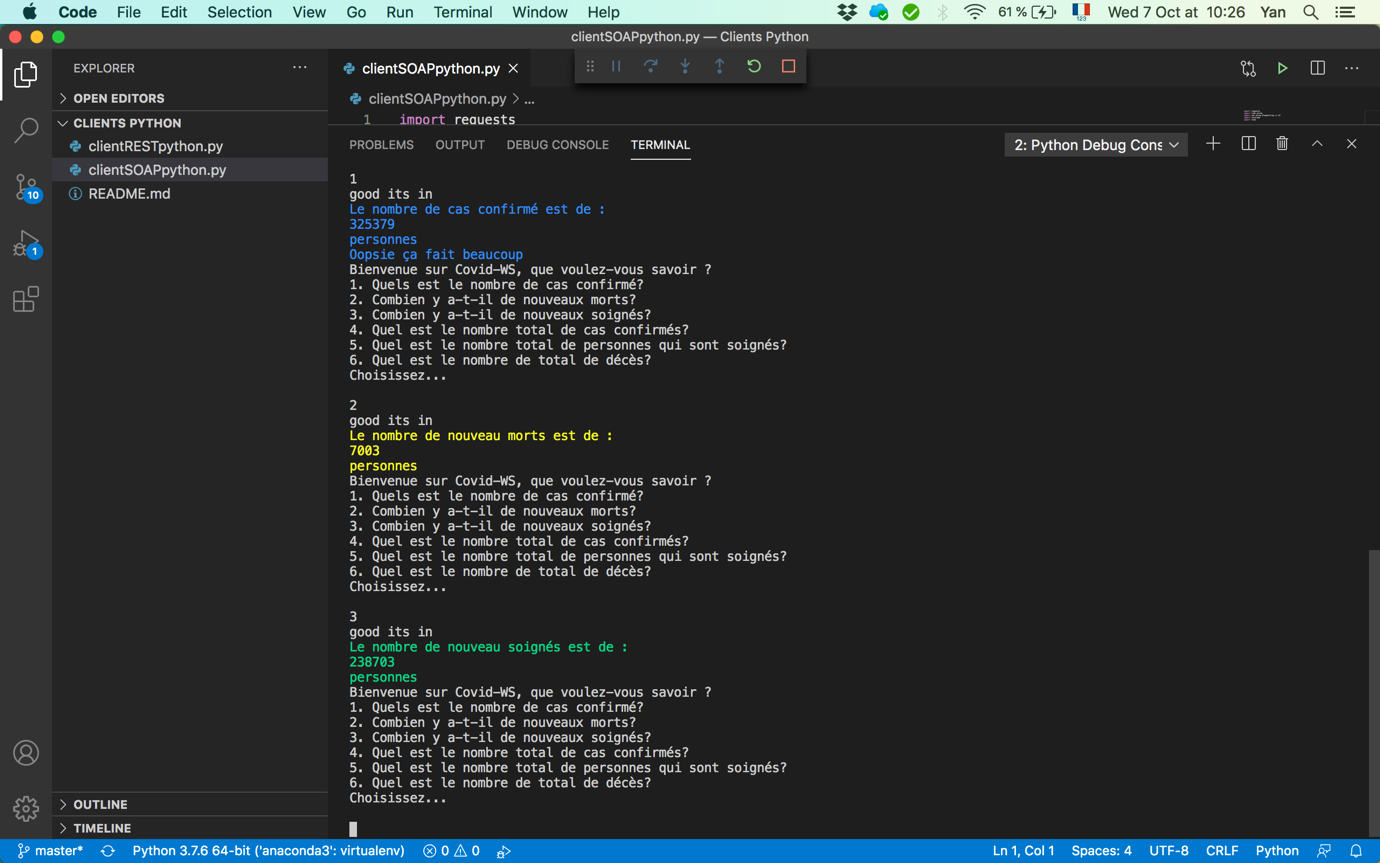
### Client SOAP – Java 8+

IDE conseillé : Eclipse IDE (2020.06 ou inférieure pour compatibilité Java 8)  
OS Compatible : Windows / macOS / Linux



### Client SOAP - Python

IDE conseillé : Visual Studio Code (latest)  
OS Compatible : Windows / macOS / Linux  
Lancement : “Run without debugging”



## Partie 4 : Conclusion

Nous avons réussi dans ce projet à mettre en œuvre un serveur REST d’application CRUD, ainsi que deux clients dans des langages différents.

Notre deuxième serveur, utilisant la technologie SOAP, dispose-lui aussi de deux clients fonctionnels dans des langages distincts.

Nous avons implémenté une application couche métier qui permet de recueillir des informations sur l’état de la crise sanitaire et des statistiques de celle-ci.

Avec plus de temps, nous aurions probablement intégrer une couche DOA avec une base de données minimale mais fonctionnelle pour stocker les informations des deux serveurs.

Cependant nous sommes déjà satisfaits de l’avancée du projet et de l’ensemble de ses fonctionnalités.