Cours JAVA - FMS

Initiation à la programmation

avec le langage Java

Partie 4 : Les Threads et Réseaux (Socket) + TD

Stéphane MASCARON
Architecte Logiciels Libres
http://mascaron.net

Cours JAVA - FMS

- Les Threads Introduction :
  - La machine virtuelle Java permet l'exécution concurrente de plusieurs **Threads** : « **fils d'éxécution** » qui se caractérisent par :
    - Un état actif ou inactif qui sont eux même sous-divisés en :
      - Prêt à l'exécution
      - En exécution
      - En attente
    - Un Nom : une chaîne de catactères
    - Un contexte d'exécution (pile de mémorisation des appels de méthodes, instruciton courante, ...)

Cours JAVA - FMS

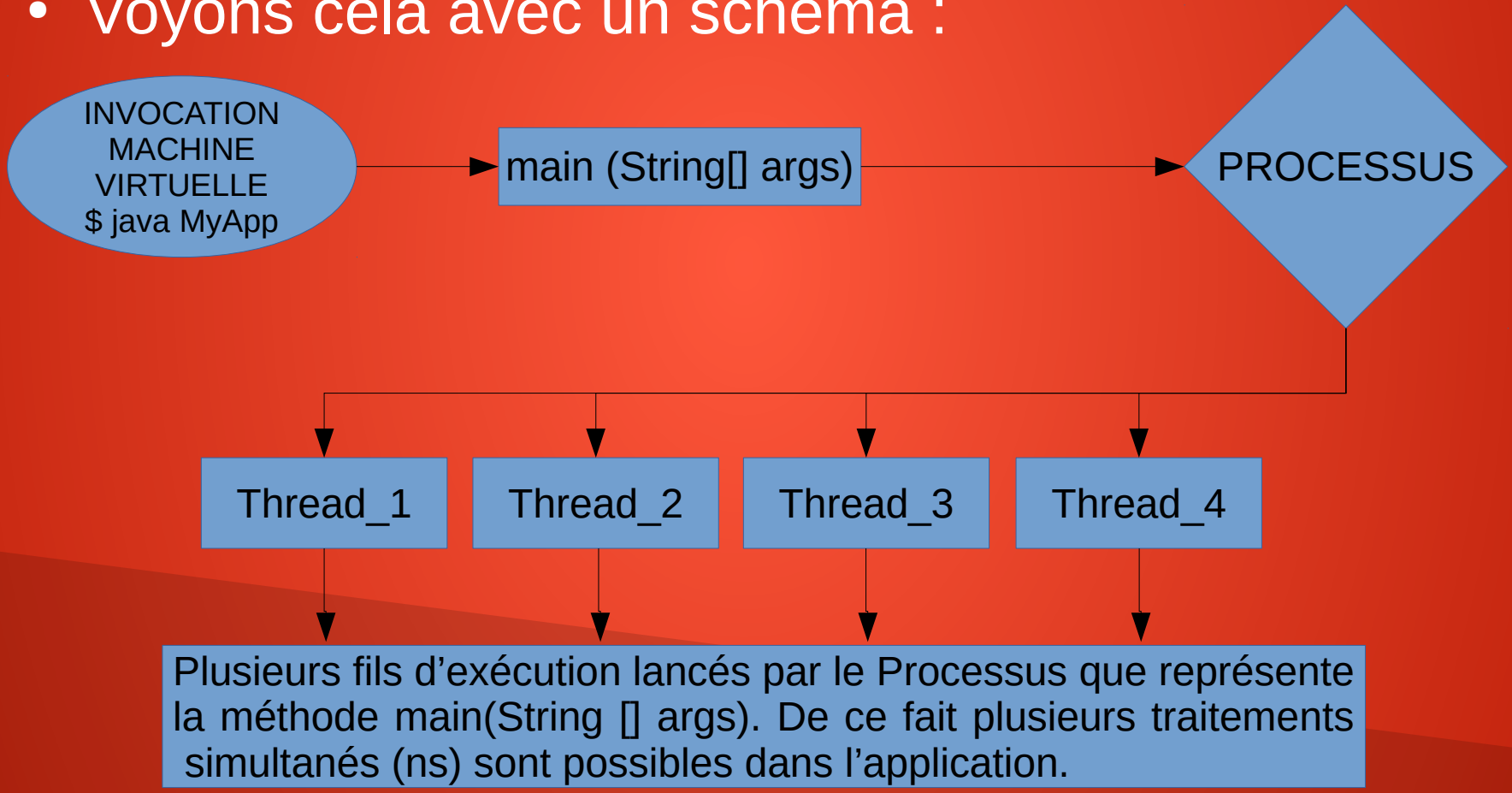
- Pourquoi utiliser plusieurs fils d'exécution ?
  - Pour par exemple effectuer des tâches qui doivent s'exécuter et se répeter à des intervalles de temps régulier et cela de façon indépendante de l'interface principale de l'application.
    - Enregistrement automatique dans les traitements de texte, animation à l'écran, gestion des connexions sur un serveur, …)
    - Permet de simplifier les systèmes qui fonctionnent en « temps réel », serveurs de chat, serveurs Web. On peut utiliser un thread par appel client (Apache, Tomcat, …).

Cours JAVA - FMS

- Qu'est-ce qu'un processus ?
  - Attention il y a parfois abus de langage lorsque l'on parle de Processus et de Thread, cela dépend en général du système d'exploitation sur lequel on fait le développement.
  - Voyons les notions de Thread et Processus pour les programmes Java
    - Le Processus est le **fil d'exécution système** suite à l'appel de la méthode « main(String args[ ]) », cette méthode lance donc un et un seul processus.
    - Un Thread sera un **fil d'exécution applicatif** suite à l'instanciation d'un objet héritant de la classe **java.lang.Thread** dans la méthode « main » ou dans une des classes de l'application.

Cours JAVA - FMS

- Le Thread est étroitement lié à son processus parent, il ne peut pas s'exécuter par lui-même.
  - Voyons cela avec un schéma :



Cours JAVA - FMS

- Problématique : la synchronisation
  - Il est évident que la gestion de plusieurs fils d'exécution pose des problèmes, notamment celui de la synchronisation ds différentes tâches qui s'exécutent en « même temps ».
  - Surtout si ces tâches d 'exécution partagent une ou plusieurs références d'objets.
    - Nous allons mettre en évidence ces problèmes dans des exemples de codes Java. Puis nous les corrigerons grâce aux outils mis à la disposition du développeur Java.
    - Ci-après un premier exemple simple non sychronisé.

Cours JAVA - FMS

Exemple : Code du Thread qui réalise le traitement, ici un parcours de boucle for :

```
package fr.FMS.java.td;

public class FilExec extends Thread {

    private int valeur;

    public FilExec() {
        super();
        valeur = 0;
    }

    public void run() {
        for (int i=0;i<200;i++) {
            System.out.println(this.getName()+" / valeur : "+valeur);
            valeur ++;
        }
    }
}
```

Cours JAVA - FMS

Exemple : Code du Main qui instancie les Threads.

```
package fr.FMS.java.td;

public class ThreadTest {

    public static void main(String[] args) {
        FilExec th1 = new FilExec();
        FilExec th2 = new FilExec();
        th1.start();
        th2.start();
    }
}
```

Cours JAVA - FMS

Exemple : Résultat de l'exécution
Thread-1 / valeur : 0 Thread-0 / valeur : 0 Thread-1 / valeur : 1 Thread-0 / valeur : 1 Thread-1 / valeur : 2 Thread-0 / valeur : 2 Thread-1 / valeur : 3 Thread-0 / valeur : 3 Thread-1 / valeur : 4 Thread-0 / valeur : 4 Thread-1 / valeur : 5 Thread-0 / valeur : 5 Thread-1 / valeur : 6 Thread-0 / valeur : 6 Thread-1 / valeur : 7

Exemple : exécution suite
Thread-0 / valeur : 7 Thread-0 / valeur : 8 Thread-0 / valeur : 9 Thread-0 / valeur : 10 Thread-0 / valeur : 11 Thread-0 / valeur : 12 Thread-0 / valeur : 13 Thread-0 / valeur : 14 …*Thread-0 jusqu'à* Thread-0 / valeur : 55 Thread-1 / valeur : 8 Thread-0 / valeur : 56 Thread-1 / valeur : 9 Thread-0 / valeur : 57 Thread-1 / valeur : 10 Thread-0 / valeur : 58 Thread-1 / valeur : 11 Thread-0 / valeur : 59 Thread-1 / valeur : 12 Thread-0 / valeur : 60

Exemple : exécution suite
… alternance jusqu'à Thread-0 / valeur : 86 Thread-1 / valeur : 39 Thread-0 / valeur : 87 Thread-0 / valeur : 88 Thread-0 / valeur : 89 Thread-0 / valeur : 90 … *Thread 0 jusqu'à* Thread-0 / valeur : 139 Thread-1 / valeur : 40 Thread-1 / valeur : 41 Thread-1 / valeur : 42 … *thread 1 jusqu'à* Thread-1 / valeur : 199 Thread-0 / valeur : 140 Thread-0 / valeur : 141 … *Thread 0 jusqu'à* Thread-0 / valeur : 199

Cours JAVA - FMS

- Explication sur l'exemple simple :
  - Cette exemple nous montre bien que les Threads ne sont pas synchrones. Puisque l'on commence par le Thread 1 puis le Thread 0 en alternance ensuite il affiche une série de Threads 0 puis une série de Threads 1.
  - Ce problème peut apparaître de manière plus ou moins importante en fonction du système d'exploitation sur lequel s'exécute votre application.
  - De ce fait lorsqu'un Thread produit une information et un autre Thread la consomme, l'information produite n'a pas 100 % de chance d'être consommée à cause des problèmes de synchronization.

Cours JAVA - FMS

- Voyons quelques exemples de codes qui peuvent poser des problèmes de synchronisation
  - Voici le code de la méthode « run » d'un Thread 1 :

Exemple : Code de la méthode run du Thread N° 1.

```
(…)
if ( x!= 0) { coef = 2 * y / x ; }
(…)
```

  - Mais juste après le test sur « x » un autre Thread 2 exécute le code suivant :

Exemple : Code de la méthode run du Thread N° 2.

```
(…)
  x = 0 ;
(…)
```

  - Puis enfin le Thread 1 exécute l'opération de calcul du coeff, on aura donc une dévision par zéro et une exception de levée malgrès le test sur « x ».

Cours JAVA - FMS

- Voyons un autre exemple

Exemple : Code accédé par deux threads en concurrence

```
if (valeurImportante > 0) {
   valeurImportante += 1 ;
}
```

- Si 2 Threads accèdent à ce code en même temps, l'une des 2 incrémentations risque d'être perdue.
- Ce qui peut avoir de fâcheuses conséquences dans votre application :
  - Mauvais affichage à l'écran
  - Base de données non mise à jour correctement
- Voyons avec notre exemple comment résoudre ce problème.

Cours JAVA - FMS

- Modifiez le code de notre exemple en partageant la variable valeur en la rendant « static » :

Exemple : Code de notre classe FilExec

```
package fr.FMS.java.td;

public class FilExec extends Thread {
    private static int valeur;

    public FilExec() { super(); valeur = 0; }
(…)
```

- Ré-exécutez le « main (…) » de notre exemple et regardez la sortie standard. On peut constater des pertes de données, le résultat est alléatoire.

Cours JAVA - FMS

- Voici les logs possibles (exécuter plusieurs fois)

Résultat de l'exécution
Thread-0 / valeur : 0
Thread-1 / valeur : 0
Thread-0 / valeur : 1
Thread-1 / valeur : 2
Thread-0 / valeur : 3
Thread-1 / valeur : 4
Thread-1 / valeur : 6
Thread-1 / valeur : 7
Thread-1 / valeur : 8
Thread-1 / valeur : 9
Thread-1 / valeur : 10
Thread-1 / valeur : 11
Thread-1 / valeur : 12
Thread-1 / valeur : 13
Thread-0 / valeur : 5

Résultat de l'exécution
Thread-0 / valeur : 15
Thread-0 / valeur : 16
Thread-0 / valeur : 17
Thread-0 / valeur : 18
Thread-0 / valeur : 19
Thread-0 / valeur : 20
Thread-0 / valeur : 21
Thread-0 / valeur : 22
Thread-0 / valeur : 23
Thread-0 / valeur : 24
Thread-0 / valeur : 25
Thread-0 / valeur : 26
Thread-1 / valeur : 14
Thread-1 / valeur : 28
Thread-1 / valeur : 29
Thread-1 / valeur : 30
Thread-1 / valeur : 31

Résultat de l'exécution
(…)
Thread-1 / valeur : 59
Thread-1 / valeur : 60
Thread-1 / valeur : 61
Thread-0 / valeur : 27
Thread-0 / valeur : 63
Thread-0 / valeur : 64
Thread-0 / valeur : 65
Thread-0 / valeur : 66
Thread-0 / valeur : 67
Thread-0 / valeur : 95
(…)
Thread-1 / valeur : 62
Thread-1 / valeur : 97
Thread-1 / valeur : 98
Thread-1 / valeur : 99

Cours JAVA - FMS

- La synchronisation
  - Voyons comment régler ce problème en ajoutant une instruction Java :
    - En fait avec notre variable statique partagée entre les 2 Threads on a le phénomène suivant :
      - Chaque Thread veut modifier la valeur de cette variable, or comme ils sont autonome du point de vue de l'accès à la ressource CPU, ils « se cognent » dans la boucle et parfois des valeurs peuvent disparaître (en fait elles ne sont pas affichées bien que calculées).
    - Comment empêcher ce genre de problèmes ?

Cours JAVA - FMS

- La synchronisation (suite)
  - Notre programme de test a pour but d'incrémenter une variable entière de 1 à 50 (boucle for) fois par Thread et d'afficher les valeurs de façon croissante, juste après quelles aient été calculées.
  - Pour éviter les problèmes de synchronisation, il existe un mot réservé en Java : « **synchronized** »
  - Il devra être placé devant une « **section critique** », c'est à dire du code qui peut engendrer une erreur de synchronisation de Thread. Elle peut être :
    - Une méthode, dans ce cas le mot synchronized est placé devant la déclaration de la méthode. (**public synchronized void methode ()**)
    - Une instruction, ou un ensemble d'instruction, dans ce cas il sera placé dans le code devant cette ou ces instructions suivi d'un début de bloc « {« et d'une fin de bloc « } » avec en paramètre la référence de l'objet à synchroniser, ici « System.out ».

Cours JAVA - FMS

- La synchronisation (suite)
  - Pour éviter le problème de concurrence avec cette instrcution « synchronized » il faut que le premier Thread empêche le second de commencer son traitement tant qu'il n'a pas fini le sien.
  - Pour ce faire, il faut verrouiller un objet, pour notre exemple des valeurs de « FilExec.java » c'est l'affichage qui est mélangé, donc le « System.out »

Exemple : Code de notre classe FilExec

```
package fr.FMS.java.td;
public class FilExec extends Thread {
    private static int valeur;
    public FilExec() { super(); valeur = 0; }

    public void run() {
       synchronized (System.out) {
          for (int i=0;i<50;i++) {
            System.out.println( ...)
          }
       }
    }}
```

Cours JAVA - FMS

- Vérification de la synchronisation
  - Exécutez le code modifié et regardez la sortie standard :

Résultat de l'exécution	Résultat de l'exécution	Résultat de l'exécution
Thread-1 / valeur : 0 Thread-1 / valeur : 1 Thread-1 / valeur : 2 Thread-1 / valeur : 3 Thread-1 / valeur : 4 Thread-1 / valeur : 5 Thread-1 / valeur : 6 Thread-1 / valeur : 7 Thread-1 / valeur : 8 Thread-1 / valeur : 9 Thread-1 / valeur : 10 Thread-1 / valeur : 11 Thread-1 / valeur : 12 (…)	Thread-1 / valeur : 49 Thread-0 / valeur : 50 Thread-0 / valeur : 51 Thread-0 / valeur : 52 Thread-0 / valeur : 53 Thread-0 / valeur : 54 Thread-0 / valeur : 55 Thread-0 / valeur : 56 Thread-0 / valeur : 57 Thread-0 / valeur : 58 Thread-0 / valeur : 59 Thread-0 / valeur : 60 Thread-0 / valeur : 61 (…)	Thread-0 / valeur : 87 Thread-0 / valeur : 88 Thread-0 / valeur : 89 Thread-0 / valeur : 90 Thread-0 / valeur : 91 Thread-0 / valeur : 92 Thread-0 / valeur : 93 Thread-0 / valeur : 94 Thread-0 / valeur : 95 Thread-0 / valeur : 96 Thread-0 / valeur : 97 Thread-0 / valeur : 98 Thread-0 / valeur : 99

Cours JAVA - FMS

- **Exercice** : Une autre solution de synchronisation est possible, c'est à dire que l'on peut verrouiller un autre objet que « **System.out** » ? Quels sont les objets qui entrent en jeu dans ce petit programme ?
  - Trouvez l'objet et testez par vous même la synchronisation afin de vérifier votre hypothèse dans Eclipse en modifiant le code source.
  - 25 minutes.

Cours JAVA - FMS

- Programmation réseau en java, le package « **java.net** »
  - Ce paquetage fournit un ensemble complet de Classes pour communiquer sur le réseau Internet ou Intranet.
  - Ces classes permettent de télécharger des URL, définir des types MIME, utiliser des protocoles standards de l'Internet (HTTP)
  - Voyons quelques unes de ces classes dans des exemples concrets.

Cours JAVA - FMS

- La classe **URL**
  - Cette classe permet de télécharger des informations situées à une adresse notée par une URL.
    - Elle offre 3 possibilités de téléchagement
      - Directement par la méthode « **getContent ()**». Cette méthode retourne une référence objet dont la classe correspond à un type MIME pour lequel on a un gestionnaire.
      - Indirectement via une URLConnection obtenue grâce à la méthode « **openConnection()** », et comme précédemment par la méthode « **getContent()** ».
      - Via la création d'un canal de type « **InputStream** » obtenu par la methode « **openStream()** ». Cette façon de faire est la seule si on ne possède pas de type MIME donc de gestionnaire pour les données associées.

Cours JAVA - FMS

- Voyons un exemple avec la classe URL

Exemple : Code de la classe URLReader contenant une simple méthode « main »

```
package fr.FMS.java.td;

import java.net.* ;
import java.io.* ;

public class URLReader {

    public static void main(String[] args) throws IOException {
        URL pageWeb = new URL("http://mascaron.net"); // inscrivez l'url que vous voulez
        BufferedReader in = new BufferedReader(new InputStreamReader(pageWeb.openStream())) ;
        String inputLine ;
        while ((inputLine = in.readLine())!= null){
            System.out.println(inputLine) ;
        }
        in.close() ;
    }
}
```

- Si vous codez cet exemple dans Eclipse vous devriez voir à l'exécution sur la console le code source de la page index.html du serveur appelé.

Cours JAVA - FMS

- La class **URLConnection**
  - Cette classe abstraite a pour utilité de proposer un meilleur contrôle des informations téléchargées depuis une URL.
  - Voici les informations complémentaires qui sont récupérables
    - Son type
    - Son en-tête
    - La Date de dernière modification
  - Voyons ci-après un exemple de code

Cours JAVA - FMS

Exemple : Code de la classe **InfosURL** permettant d'obtenir des infos sur le document de l'URL.

```
package fr.FMS.java.td;

import java.io.*;
import java.net.*;
import java.util.Date;

public class InfoURL {
    public static void afficheInfos(URLConnection urlc, BufferedReader in) throws
IOException {
        System.out.println(urlc.getURL().toExternalForm() + " :");
        System.out.println("Type MIME : " + urlc.getContentType());
        System.out.println("Longueur : " + urlc.getContentLength());
        System.out.println("Date dernière modif : " + new Date(urlc.getLastModified()));
        System.out.println("Contenu du document : \n ");
        String inputLine;
        while ((inputLine = in.readLine())!= null){ System.out.println(inputLine); }
    }
    public static void main(String[] args) throws IOException {
        URL url = new URL("http://mascaron.net");
        URLConnection connection = url.openConnection();
        BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));
        InfoURL.afficheInfos(connection, in);
       in.close();
    }
}
```

Cours JAVA - FMS

- Les classes java.net.**Socket** et java.net.**ServerSocket**
  - **ServerSocket** :
    - Cette classe finale va permettre de construire des serveurs, il s'agit **de boucles infinies** qui attendent une **connexion** en provenance d'un client sur **un numéro de port logique**.
    - Grâce à la méthode « **accept()** » on récupère une référence sur une instance du socket ouvert lors de la réception d'une demande.

Cours JAVA - FMS

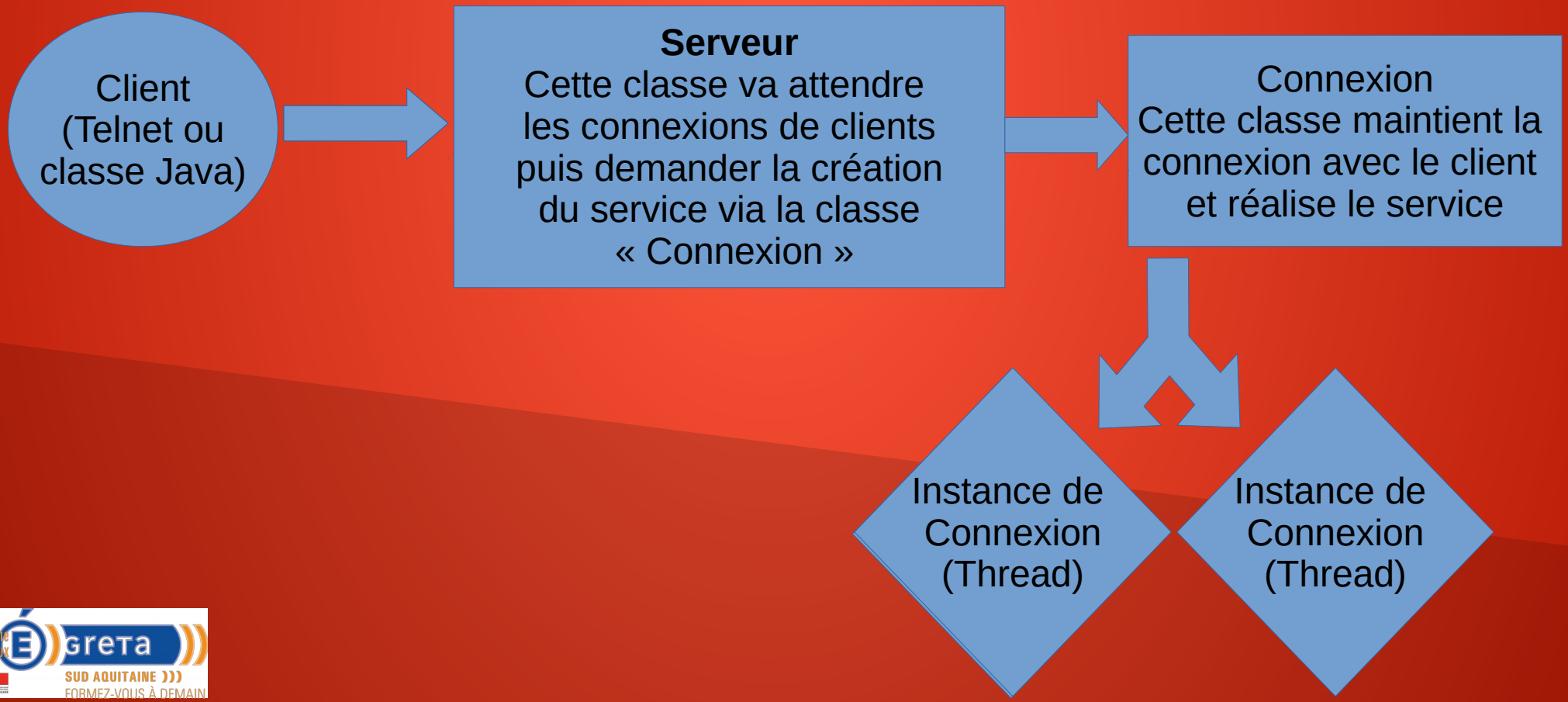
- **Socket** :
  - Cette classe finale va autoriser la communication inter processus sur le réseau.
  - Le constructeur prend en paramètres, l'adresse de la machine distante et le numéro de port logique, pour construire un socket.
  - Il est par la suite possible de lire et d'écrire sur des canaux de communication grâce aux méthodes :
    - **getInputStream**()
    - **getOutputStream**()
  - Ces canaux peuvent être des descendants de n'importe quelle sous-classe de **InputStream** et **OutputStream** du package java.io.

Cours JAVA - FMS

- Un exemple de Client / Serveur en Java.
  - Et si on écrivait un Serveur ?
    - Pour écrire un serveur on va devoir écrire un processus qui écoute indéfiniment une connexion éventuelle sur un numéro de port logique donné.
    - Pour écouter les connexions on utilise une instance de la classe « **ServerSocket** »**.**
    - Ce programme hérite du comportement de la classe « Thread », et possède donc une méthode « run () » qui contiendra une boucle sans fin while(true).
    - Lorsqu'un appel est accepté par la méthode « accept() », on passe la référence du Socket à un objet qui sera chargé de réaliser le traitement. On va appeler cette classe « Connexion » elle hérite aussi de « Thread ».

Cours JAVA - FMS

- Voyons comment nous allons organiser notre code :
  - Il nous faut une classe « **Serveur** » dans un fichier « **Serveur.java** », une classe « **Connexion** » dans un fichier « **Connexion.java** » :



Cours JAVA - FMS

Exemple : Code de la classe **Serveur** qui écoute sur un PORT (ici 2222)

```
package fr.FMS.java.td;

import java.io.IOException;
import java.net.*;

public class Serveur extends Thread {
    protected static final int PORT=2222;
    protected ServerSocket ecoute;
    public Serveur() {
        try { ecoute = new ServerSocket(PORT); }
        catch (IOException ex) { ex.printStackTrace();System.exit(1); }
        System.out.println("Serveur en écoute sur le PORT : " + PORT);
        this.start(); // on execute le Thread Serveur
    }
    public void run () {
        try {
            while(true) {
                Socket client = ecoute.accept();
                System.out.println("Connexion de:" + client.getInetAddress().toString());
                Connexion connect = new Connexion(client);
                connect.start(); // execution du Thread Connexion
            }
        } catch (IOException ex) { ex.printStackTrace(); System.exit(1); }
    }
    public static void main (String [] args) {
        new Serveur();
    }
}
```

Cours JAVA - FMS

Exemple : Code de la classe **Connexion** qui réalise le traitement pour chaque requêtes de chaque clients

```
package fr.FMS.java.td;
import java.io.*;
import java.net.Socket;
public class Connexion extends Thread {
     protected Socket client;
     protected BufferedReader in;
     protected PrintWriter out;
     public Connexion(Socket client_soc)  {
          client = client_soc;
          try {
               in = new BufferedReader(new InputStreamReader(client.getInputStream()));
               out = new PrintWriter(client.getOutputStream(), true);
          } catch (IOException e) {
               try { client.close(); } catch (IOException e1) {
                    e1.printStackTrace();
                    return;
               }
          }
     }
     public void run() {
          String ligne = null;
          try {while(true) {
                    ligne = in.readLine();
                    if (ligne.toUpperCase().compareTo("EXIT") == 0 ) break;
                    out.println(ligne.toUpperCase()); } // réalisation du traitement MAJ
          } catch (IOException ex) {
               System.out.println("connexion : " + ex.toString());
          } finally { try { client.close(); } catch (IOException ex) {ex.printStackTrace(); }}
     }
}
```

Cours JAVA - FMS

- Testez le Serveur dans Eclipse, lancer l'exécution de la classe Serveur elle a le main.
  - Il affiche dans la console :

```
Serveur en écoute sur le PORT : 2222
```

  - Si vous êtes sous Linux, vous pouver installer **telnet** et tapez dans un terminal :

```
telnet 127.0.0.1 2222
Trying 127.0.0.1…
Connected to 127.0.0.1.
Escape character is '^]'.
titi
TITI
```

  - Tapez ensuite une chaîne de caractères en minuscule, le serveur vous la renvoie en Majuscule.

Cours JAVA - FMS

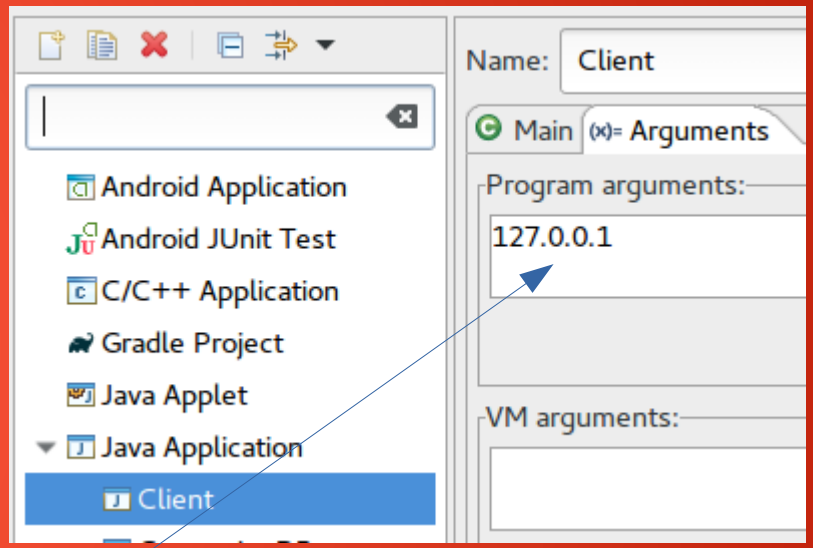
Exemple : Code de la classe **Client** qui réalise la connexion avec le serveur et propose une interface de saisie

```
package fr.FMS.java.td;

import java.io.*;
import java.net.*;

public class Client {
     protected static final int PORT = 2222;
     public static void main(String[] args) {
     Socket s = null;
     if (args.length !=1) {
          System.err.println("Usage : java Client <hote ip ou domaine>" );
          System.exit(1);
     }
     try { s = new Socket(args[0], PORT);
       BufferedReader lecture = new BufferedReader(new InputStreamReader(s.getInputStream()));
       BufferedReader console = new BufferedReader(new InputStreamReader (System.in));
       PrintWriter ecriture = new PrintWriter(s.getOutputStream(), true);
       System.out.println("Connexion établie: "  + s.getInetAddress() + "/ port : " + PORT);
        String ligne ;
       while (true) { System.out.print("? "); System.out.flush();
          ligne = console.readLine(); ecriture.println(ligne);
          ligne = lecture.readLine();
          if (ligne == null) { System.out.println("Connexion terminée ! "); break; }
          System.out.println("! " + ligne);
       }
     } catch (IOException e) {System.err.println("Error : " + e.getMessage());}
     }
}
```

Cours JAVA - FMS

- Résultat de l'exécution du programme Client :

```
Console
<terminated> Client [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java
Connexion établie: /127.0.0.1/ port : 2222
?fsfsd
! FSFSD
?sdfsdsd
! SDFSDSD
?dvv
! DVV
?fvf
f! FVF
?vgr
er! FVGR
?gerg
e! ERGERG
?rgergergerg
! ERGERGERGERG
?22222
! 22222
?exit
Connexion terminée !
```



NB : Pensez a mettre une IP dans le menu « Run configuration ... » d'éclipse, celle ou le serveur tourne.

Cours JAVA - FMS

- **Exercices réseau + Thread** : Ecrire un programme Java permettant de réaliser de multiple connexion sur un serveur et d'échanger des messages Textes.
- Bref écrire un mini Tchat :-) en partant du code précédent.