JAVA
Les entrées-sorties

Les entrées-sorties

- Un programme a souvent besoin d'échanger des informations :
  - Recevoir des données d'une source
  - Envoyer des données vers un destinataire
- La source et la destination de ces échanges peuvent être de natures multiples :
  - un fichier, une socket réseau, un autre programme, etc ...
- De la même façon, la nature des données échangées peut être diverse :
  - du texte, des images, du son, etc ...

Les flux (Streams)

- Les flux (Streams) permettent d'encapsuler le processus d'envoi et de réception de données. Les flux traitent toujours les données de façon séquentielle.
- En Java, les flux peuvent être divisés en plusieurs catégories :
  - les flux d'entrée (input stream) et les flux de sortie (output stream)
  - les flux de traitement de caractères et les flux de traitement d'octets
- Java définit des flux pour lire ou écrire des données mais aussi des classes qui permettent de faire des traitements sur les données du flux.
  - Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des filtres.
  - Par exemple, il existe des filtres qui permettent de mettre les données traitées dans un tampon (buffer) pour les traiter par lots.
- Toutes ces classes sont regroupées dans le package **java.io**

Les flux

- **Reader** : types de flux en lecture sur des ensembles de caractères
- **Writer** : types de flux en écriture sur des ensembles de caractères
- **InputStream** : types de flux en lecture sur des ensembles d'octets
- **OutputStream** : types de flux en écriture sur des ensembles d'octets

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

Les flux

Pour le préfixe, il faut distinguer les flux et les filtres. Pour les flux, le préfixe contient la source ou la destination selon le sens du flux.

Préfixe du flux	**Source ou destination du flux**
ByteArray	Tableau d'octets en mémoire
CharArray	Tableau de caractères en mémoire
File	Fichier
Object	Objet
Pipe	Pipeline entre deux threads
String	Chaîne de caractères

Les flux

La package **java.io** définit ainsi plusieurs classes :

	Flux en lecture	**Flux en sortie**
Flux de caractères	BufferedReader CharArrayReader FileReader InputStreamReader LineNumberReader PipedReader PushbackReader StringReader	BufferedWriter CharArrayWriter FileWriter OutputStreamWriter PipedWriter StringWriter
Flux d'octets	BufferedInputStream ByteArrayInputStream DataInputStream FileInputStream ObjectInputStream PipedInputStream PushbackInputStream SequenceInputStream	BufferedOutputStream ByteArrayOutputStream DataOuputStream FileOutputStream ObjetOutputStream PipedOutputStream PrintStream

TD5

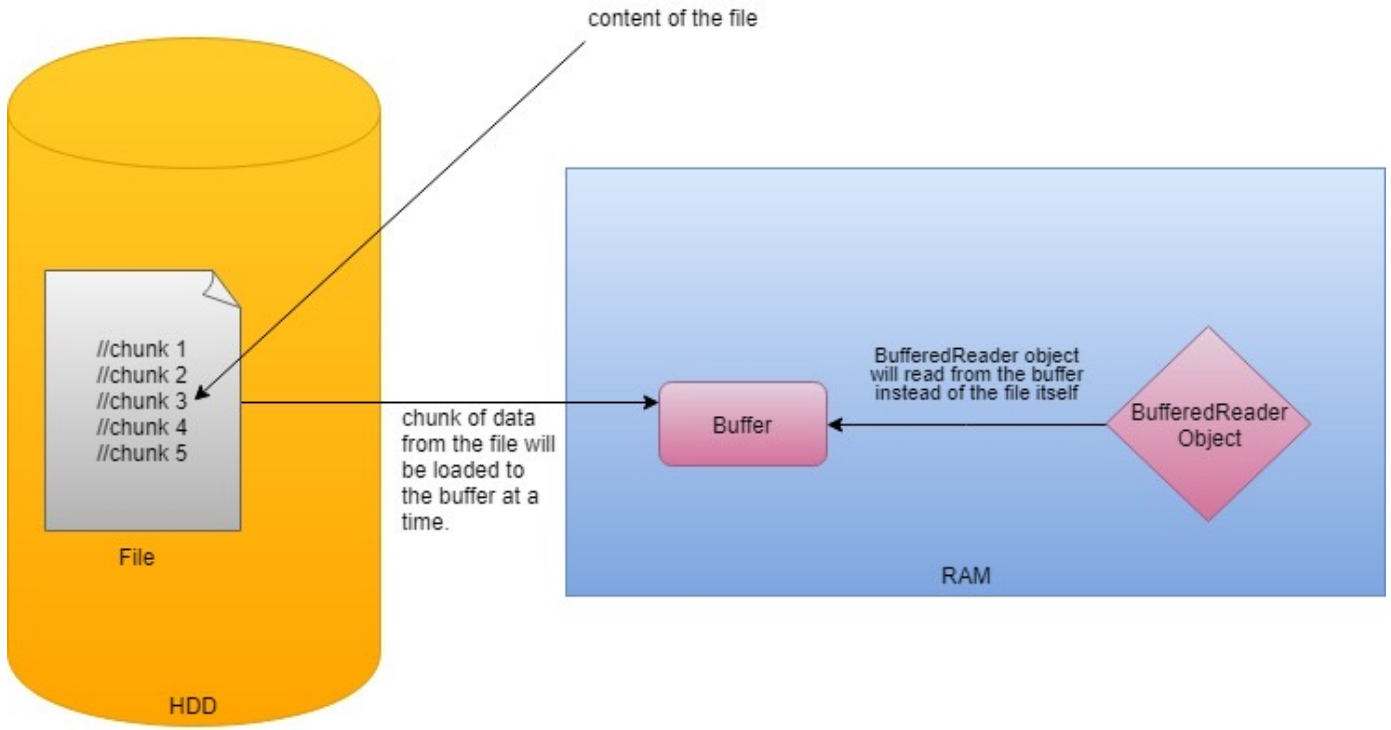
Exercice 1

Lire ce fichier **texte.txt** en utilisant la classe **java.io.FileReader**.

Les tampons de flux

- Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble de caractères représentant une ligne plutôt que de traiter les données caractères par caractères.
- Le nombre d'opérations est ainsi réduit.
- Les classes **BufferedReader** et **BufferedWriter** permettent de gérer des flux de caractères tamponnés avec des fichiers.
-
```
BufferedReader fichier = new BufferedReader(new
                         FileReader("monfichier.txt"));
```

Les tampons de flux



Working of BufferedReader Class

TD5

Exercice 2

Lire ce fichier **texte.txt** en utilisant la classe **java.io.BufferedReader**.

java.nio

- **java.nio** (New IO) a été introduit dans Java 7
- **Objectif** : améliorer les performances des fichiers, du réseau et des buffers.
  - Les données ne sont plus traitées octet par octet comme dans java.io, mais paquet par paquet
- 2 nouveaux objets :
  - Les channels : se sont les flux
  - Les buffers : tampons de flux dont on peut définir la taille

java.nio

```
//Création d'un nouveau flux de fichier
fis = new FileInputStream(new File("test.txt"));
//On récupère le canal
fc = fis.getChannel();
//On en déduit la taille
int size = (int)fc.size();
//On crée un buffer correspondant à la taille du fichier
ByteBuffer bBuff = ByteBuffer.allocate(size);
```

TD5

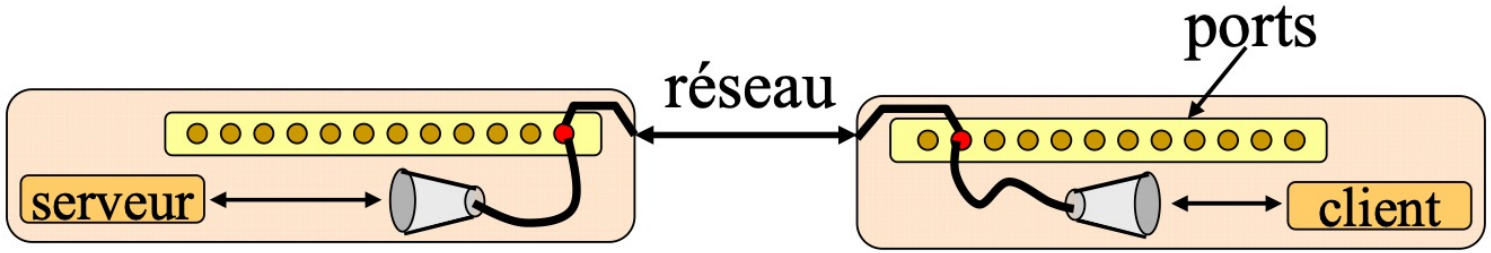
Exercice 3

Comparez le temps d'exécution de la lecture du fichier **texte.txt** en utilisant d'abord la librairie **java.io**, puis la librairie **java.nio**.

Vous pouvez utiliser la fonction `System.currentTimeMillis()` pour calculer le temps d'exécution.

Les sockets

- Les sockets servent à communiquer entre deux hôtes : le client et le serveur à l'aide d'une adresse IP et d'un port
- Choix du protocoles :
  - TCP (Transport Control Protocol) : vérification que les paquets ne manquent pas ou qu'ils ne sont pas corrompus
  - UDP (User Datagram Protocol) : pas de contrôle d'erreurs. Plus rapide, mais moins sécurisé.
- Principe de base :
  - Chaque machine crée une socket
  - Chaque machine lit et/ou écrit dans sa socket
  - Les données vont d'une socket à l'autre à travers le réseau



Les classes de sockets

- UDP : DatagramSocket
  - Client : DatagramSocket()
  - Serveur : DatagramSocket(ServerPort)
- TCP :
  - Client : Socket(ServerName, ServerPort)
  - Serveur : ServerSocket(ServerPort)

Socket TCP en Java

- Création de notre serveur TCP :

```
// Création du serveur sur le port 2000

ServerSocket server = new ServerSocket(2000);
Socket socket = server.accept();
System.out.println("Nouvel utilisateur connecté");
socket.close();
server.close();
```

Socket TCP en Java

- Création de notre client TCP

```
// On crée la socket sur l'IP de localhost sur
    le port 2000
socket = new
    Socket(InetAddress.getLocalHost(),2000);
socket.close();
```

TD5

Exercice 4

Créer une classe Client et une classe Serveur avec permettant de faire une connexion sur le port 2000

Echange de messages

- Une fois la connexion établie et les sockets en possession, il est possible de récupérer le flux d'entrée et de sortie de la connexion TCP vers le serveur.
- Il existe deux méthodes pour permettre la récupération des flux :
  - `getInputStream()` de la classe `InputStream`. Elle nous permet de gérer les flux entrant ;
  - `getOutputStream()` de la classe `OuputStream`. Elle nous permet de gérer les flux sortant.
- En général le type d'entrée et sortie utilisé est `BufferedReader` et `InputStreamReader` pour la lecture, `PrintWriter` pour l'écriture. Mais on peut utiliser tous les autres flux qu'on a vu précédemment.

TD5

Modifiez les classes Client et Serveur afin d'échanger un message entre le client et le serveur.

- Côté serveur :
  - Le serveur est en attente d'un flux :
    `out = PrintWriter(socket.getOutputStream);`
  - Dès que la connexion est établie, le serveur prépare l'envoi d'un message aux clients :
    `out.println("Message");`
  - Le serveur envoie le message et on vide le buffer : `out.flush();`
- Côté client :
  - Le client se met en attente de reception du message : `BufferedReader(new InputStreamReader(socket.getInputStream());`
  - Le client lis le message et l'affiche dans la console : `in.readLine();`

Utilisation des Threads

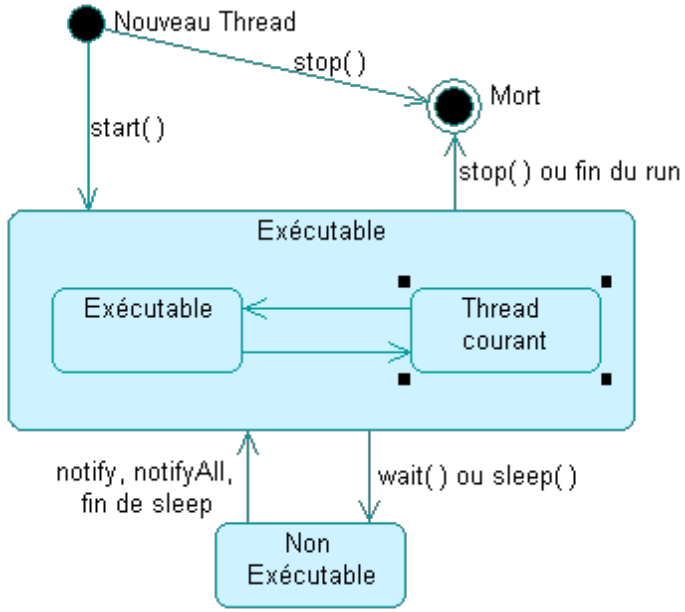
- Notre client/serveur fonctionne bien !
- Mais le problème, c'est que le serveur se ferme dès qu'il a reçu une connexion (pas très pratique…)
- La solution : utiliser les Threads (processus) afin de laisser au serveur la possibilité d'accepter plusieurs connexions à la fois

C'est quoi les Threads ?

- En Java, un programme s'exécute via un seul processus de manière procédural (un tâche après l'autre)
- Les Threads permettent de lancer plusieurs processus à la fois et ainsi de profiter de toute la puissance de la machine en faisant tourner en parallèle des tâches sur plusieurs cœurs processeurs.
- Pour créer un Thread, nous avons 2 solutions :
  - Créer une classe qui implémente l'interface Runnable
  - Créer une classe qui hérite de la classe Thread
- Dans les deux cas, il faut redéfinir la méthode `run()`

Les Threads

- Quelques méthodes utiles :
  - `thread.start()` : démarre le processus = fait appel à la méthode `run()`
  - `thread.stop()` : arrête le processus
  - `Thread.sleep(long millis)` : arrête le processus pendant un temps donné
- Cycle de vie d'un Thread :



TD5

Exercice 6

Écrire un programme Java qui génère 10 processus. Chaque processus devra patienter aléatoirement entre 0 et 10 secondes, puis afficher un message indiquant que le processus a bien été exécuté.

TD5

Maintenant que nous avons compris comment fonctionnent les Threads, retournons à notre client/serveur

Exercice 7

Modifier notre serveur afin que chaque requête d'un client soit implémentée dans un processus.
Pour ceci, nous implémenterons une classe `AcceptClient` dont le constructeur prend en paramètres une Socket.

Le serveur restera en activité en permanence grâce à un `while(true)` et devra retourner le nombre de clients qui se sont connectés.

TD5

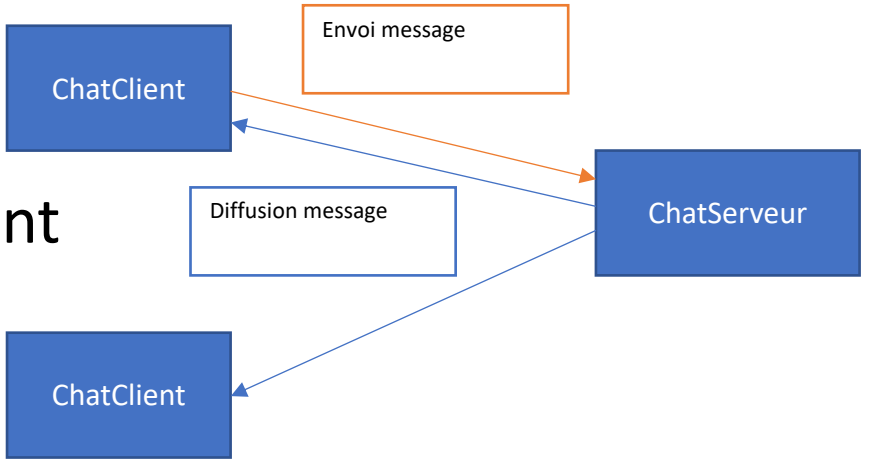
Exercice 8 : ChatApp

En s'inspirant des classes Client et Serveur précédentes, faire un service de messagerie.

La classe `AcceptClient` permet de gérer la connexion avec un nouvel utilisateur.

La classe `ReceiveMessage` permet au serveur de récupérer les messages des clients et de les afficher dans la console.

La classe `BroadcastMessage` permet au serveur de redistribuer à l'ensemble des clients le message reçu.



Bonus : faire un système de nom d'utilisateur permettant de savoir qui a envoyé le message.