



Cours « Systèmes distribués »

Tarak Chaari

**Maître assistant à l'institut supérieur d'électronique
et de communication**

tarak.chaari@isecs.rnu.tn

<http://www.redcad.org/members/tarak.chaari/cours/coursSD.pdf>

Votre interlocuteur

 **Tarak CHAARI**

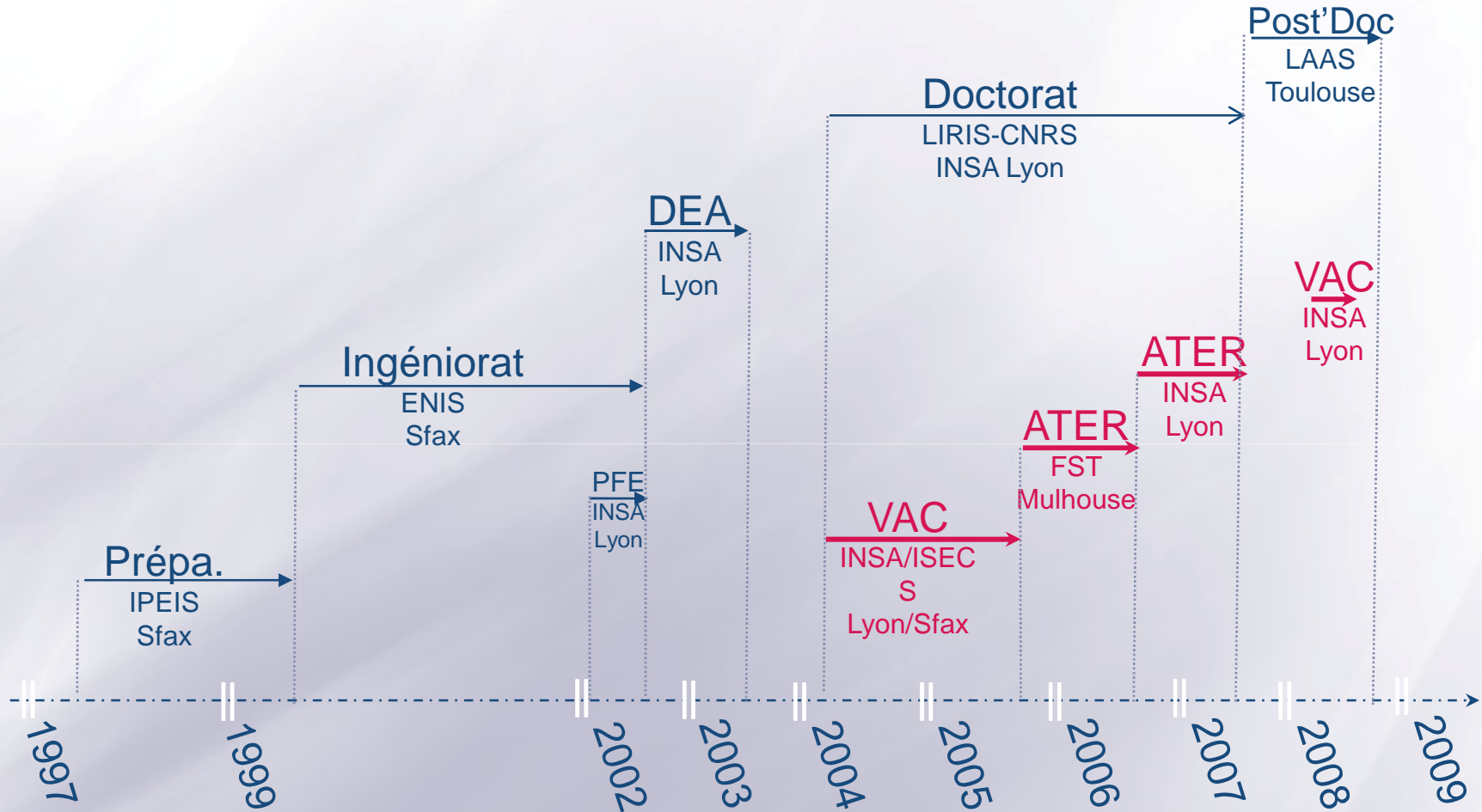
 **Maître assistant à l'ISECS**

 **Membre de l'unité de recherche RedCad**

 **Enseignement: Ingénierie des systèmes d'information**

 **Recherche: l'adaptation dans les environnements dynamiques**

Cursus universitaire



- Formation & Recherche
- Enseignement
- VAC Vacataire

Contenu du cours « systèmes distribués »

☰ Introduction sur les systèmes d'information

- Système d'information
- Architectures logicielles
- Apparition des systèmes distribués

☰ Pré-requis pour la programmation distribuée

- TCP/IP
- Les flux en Java

☰ Outils pour la programmation distribuée

- Les sockets en JAVA

En rodage !!!

Introduction sur les systèmes d'information



Définitions 1

Qu'est ce qu'un système d'information ?

un ensemble organisé de ressources (personnel, données, procédures, matériel, logiciel, ...) permettant d'acquérir, de stocker, de structurer et de communiquer des informations sous forme de textes, images, sons, ou de données au sein des organisations. Selon leur finalité principale, on distingue des systèmes d'information supports d'opérations (traitement de transaction, contrôle de processus industriels, supports d'opérations de bureau et de communication) et des systèmes d'information supports de gestion (aide à la production de rapports, aide à la décision...).

Un système d'information médical

Des bases de données

- Administratives
- Dossier médical
- Données médicales

Une infrastructure réseau

- Liaisons Ethernet
- Liaisons modem

Des postes de travail

- MacIntosh, PC (WIN98, PC...)
- Bureau du médecin, des infirmières...

Des applications médicale

Gestion :

- administrative
- du dossier médical
- des médicaments
- des lits
- des actes
- des examens
- du courrier électronique
- ...

Les besoins du SI

- Echange de données entre applications hétérogènes manipulant des données au format propriétaire
- Répartition des données sur des sites géographiques distants
- Interopérabilité des plates-formes de développement
- Portabilité des applications
- Gestion de la cohérence permanente des données
- Gestion des accès concurrents
- Persistance des données
- Intégration des systèmes legacy
- Ouverture
- Sécurité


 **D'autres exemple existent**

- **Domaine bancaire**
- **Domaine de la production automobile...**

Définitions 2

Qu'est ce qu'une application ?

une **application** est un outil qui permet de réaliser une ou plusieurs tâches ou fonctions. Un amalgame est courant avec le terme logiciel.

Exemple commande sur Internet

- Authentification sur le réseau local
- Connexion sur le serveur distant
- Passage de la commande
- Gestion du suivi
- Gestion de relance



Encore en rodage?

Un peu d'histoire



L'Histoire

Le système centralisé (70)

- Calcul
- Consoles de connexion
- Liaison série

Le client/serveur (80)

- Gestion de données simples
- Distribution des données
- Emulation de terminaux
- Réseau propriétaires

Le "fat" client (90)

- Gestion interactive des données
- Clients graphiques
- Réseaux locaux

Le Web (95)

- Approche information
- Clients documentaires
- Réseau Internet

Les objets distribués (00)

- Approche métiers
- Clients hétérogènes / MV
- Réseau haut-débits

Le code mobile (05)

- Approche dynamique
- Collaboration de machines
- Réseaux intelligents et actifs

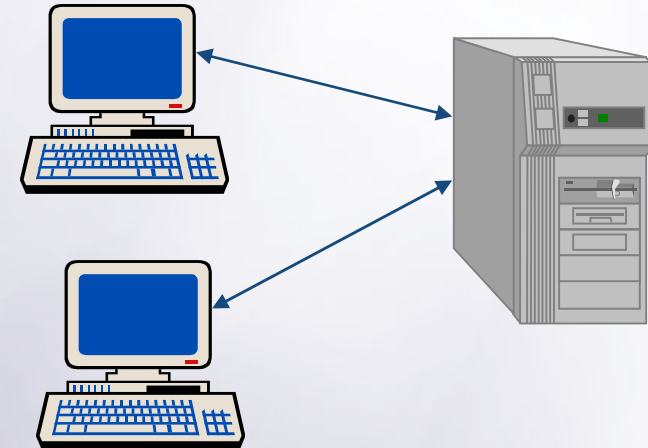


Systemes centralises

Systemes centralises

Terminaux

- Compatible IBM
- Unix



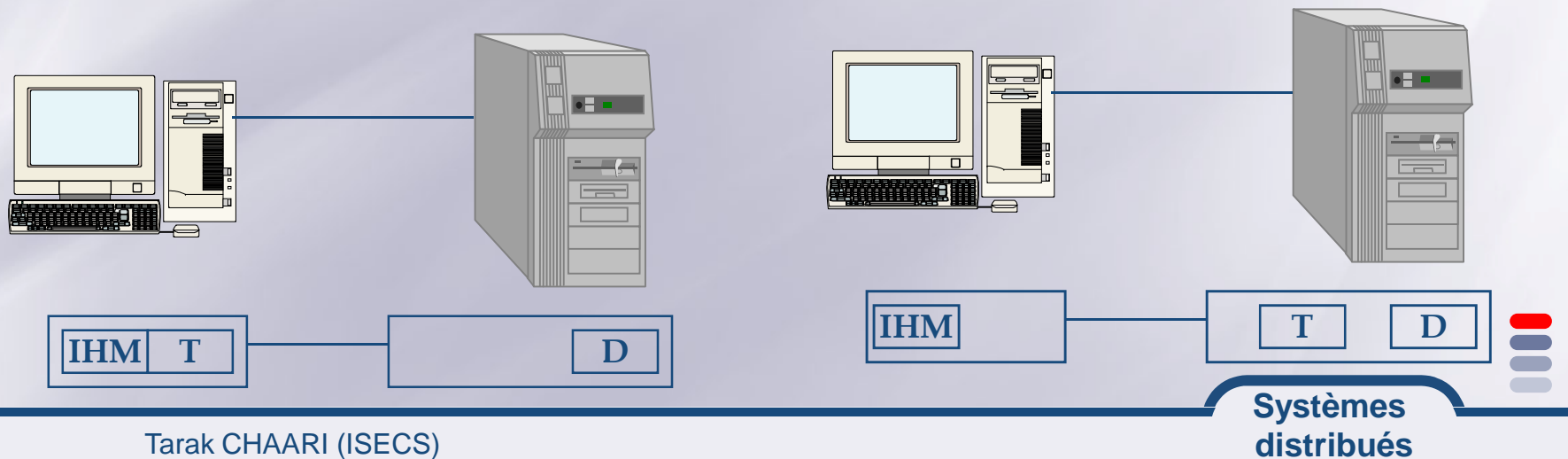
- Composants localises sur un site unique
- Centralisation des donnees, des traitements et de la presentation
- Historiquement sur systemes proprietaires
- Terminaux legers

Coûts ?

Client-serveur 1/4

Client-serveur 2 niveaux (2-tier)

- Le poste de travail héberge l'ensemble de la gestion d'interface homme-machine et le traitement,
- Le serveur est un serveur de base de données
- Architecture dénommée « client lourd »
- Le poste de travail n'héberge que l'interface homme-machine
- Le serveur héberge les données et les traitements
- Architecture dénommée « client léger »



Avantage

- Mise en œuvre
- Efficace pour un nombre réduit de clients

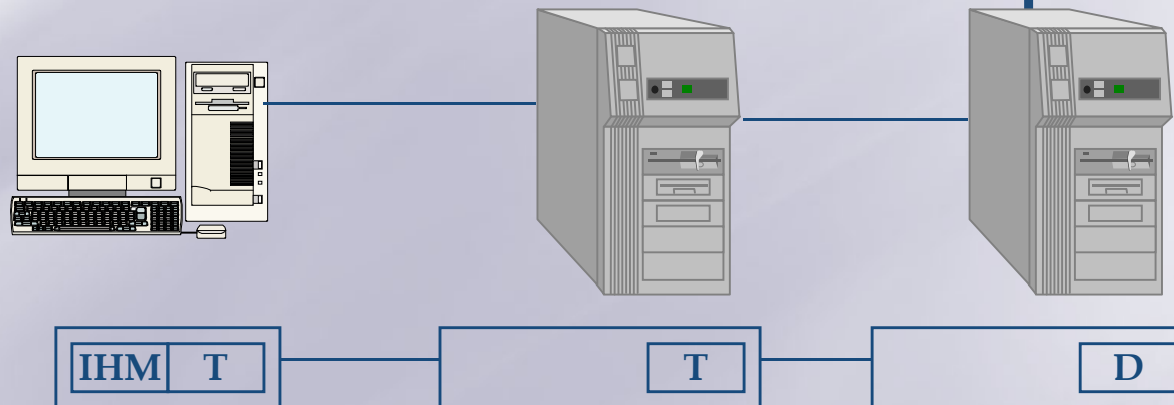
Inconvénients

- Coûts de déploiement ?
- Coûts de MAJ ?
- Accès concurrents ? (Limitation du nombre de clients)
- Hétérogénéité sur BD ?

Client-serveur 3/4

☰ Client-serveur 3 niveaux (3-tier)

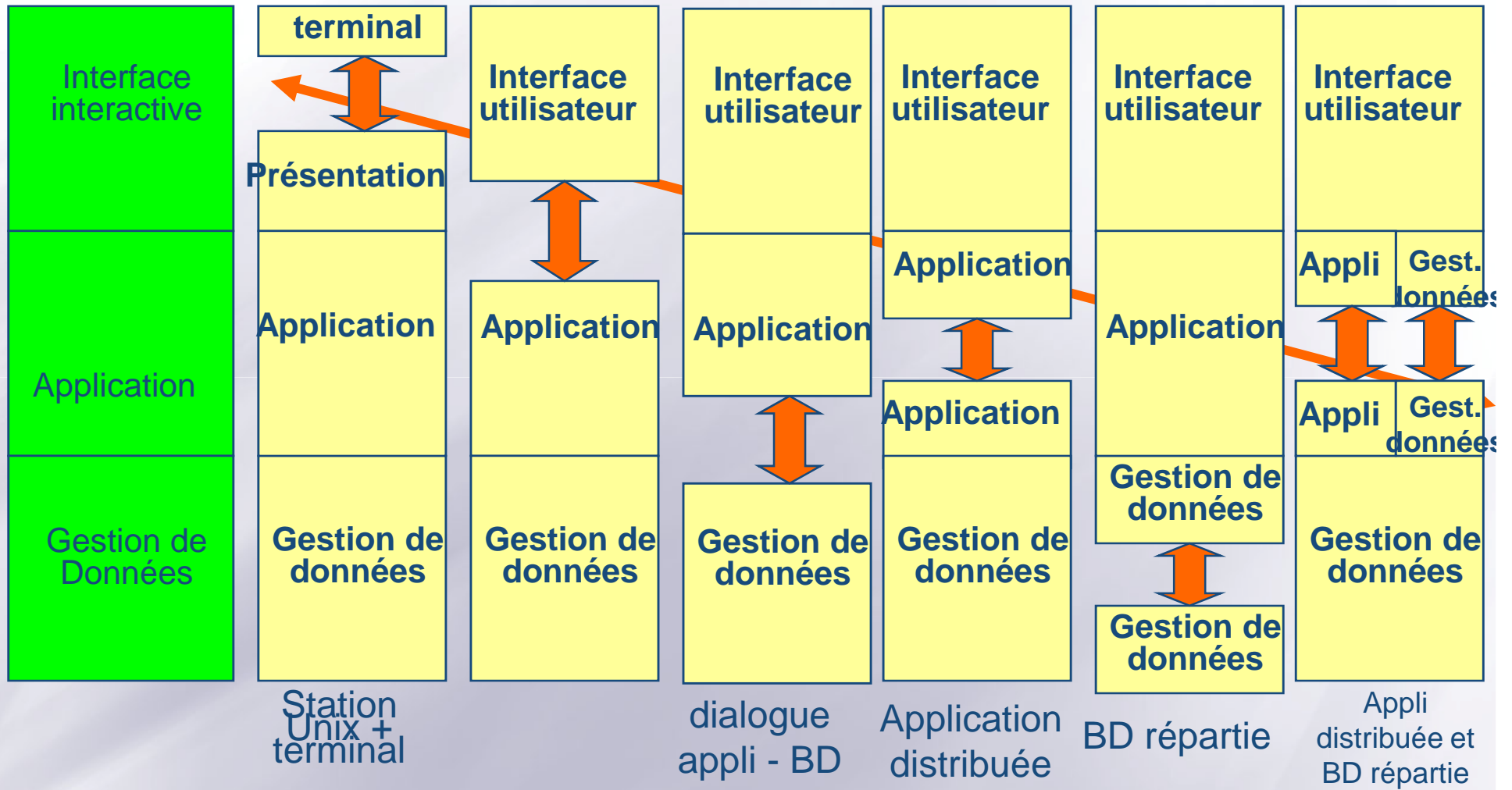
- Le poste de travail héberge la gestion d 'interface homme-machine et une partie des traitements,
- Le serveur d 'applications gère l 'autre partie des traitements
- Le serveur de données gère les accès aux données
- Architecture dénommée « traitements coopératifs »



Avantages

- Première infrastructure informatique pour un travail coopératif
- Centralisation des traitements au niveau du serveur
- Pas de duplication des données (état global observable)
- Gestion plus simple de la cohérence et de l'intégrité des données
- Maîtrise globale des processus (workflow) relativement élémentaire

Options d'architecture Client / Serveur



Nouveaux besoins

- La distribution et l'accès à l'information dans une entreprise sont des facteurs fondamentaux de succès
- L'informatique est par nature distribuée, évolutive et variée

==> Besoins pour de nouvelles architectures informatiques



Du réel au virtuel

Entreprise classique





- **Systeme informatique sur un site unique**
- **Maîtrise de l'ensemble des phases conduisant à la réalisation d'un produit**

Entreprise virtuelle

- **Regroupement d'entreprises localisées sur des sites géographiques distants**
- **Coopération en associant des compétences complémentaires**
- **Association limitée dans le temps**

==> Quelle infrastructure informatique peut supporter le système d'information de l'entreprise virtuelle ?




Facteurs économiques

-  Conjecture économique
-  Mondialisation du commerce et des marchés
-  Accroissement de la concurrence
-  Croissance de la complexité des produits

==> Nécessité d'augmenter la productivité et la compétitivité



Conclusion de la première partie

-  **Des technologies multiples**
-  **Différents modèles d'architectures**
-  **Différents logiciels indépendants**
-  **Une évolution vers l'intégration de l'hétérogène**
-  **Une première solution : Programmation distribuée**



Outils pour la programmation distribuée



Prérequis: Quelques rappels TCP/IP



Adresse IP

☰ Tient sur 32 bits (IPv4)

- un quadruplet de nombres de 8 bits chaque, séparés par des points.
- Exple: 127.0.0.1

☰ ICANN – *Internet Corporation for Assigned Names and Numbers*

- attribue les adresses IP du réseau public internet

☰ @IP 127.0.0.1

- adresse de rebouclage (ang. loopback)
- désigne la machine locale (ang. localhost)

☰ 5 classes

- Classe A: de 1.0.0.1 à 126.255.255.254
 - 127 réseaux de plus que 16 millions d'ordinateurs chaque
 - Réservées au grands réseaux
 - 1er octet réseau (NetID), 3 octets pour l'@ de l'hôte (hostID)

...Adresse IP

- **Classe B: de 128.0.0.1 à 191.255.255.254**
 - 16575 réseaux de plus que 6500 ordinateurs chaque
 - Réservées aux moyens réseaux
 - les deux 1ers octets réseau, 2 octets suivants pour l'@ de l'hôte
- **Classe C: de 192.0.0.1 à 223.255.255.254**
 - + que 2 millions de réseaux de 254 ordinateurs chaque
 - Réservées aux petits réseaux d'entreprise
 - les trois 1ers octets réseau, 4ème octet pour l'@ de l'hôte
- **Classe D: de 224.0.0.1 à 239.255.255.255**
 - groupes multicast
- **Classe E**
 - réservées pour futur usage

IPv6

- Résout le problème de pénurie d'adresses IP
- @IP sur 128 bits

Port d'écoute

- ☰ 16-bits unsigned integer,
 - de 1 à 65535.

- ☰ Ports utilisateur ≥ 1024 .

- ☰ Exemples de ports réservés,
 - FTP, 21/TCP
 - Telnet, 23/TCP
 - SMTP, 25/TCP
 - Login, 513/TCP
 - HTTP, 80/TCP



Prérequis: Les Flux en JAVA

***Le package java.io
La gestion de fichiers en java
Sérialisation d'objets***



Les Flots/Flux/Streams

☰ **Toutes les entrées/sorties en JAVA sont gérées par des flux (streams)**

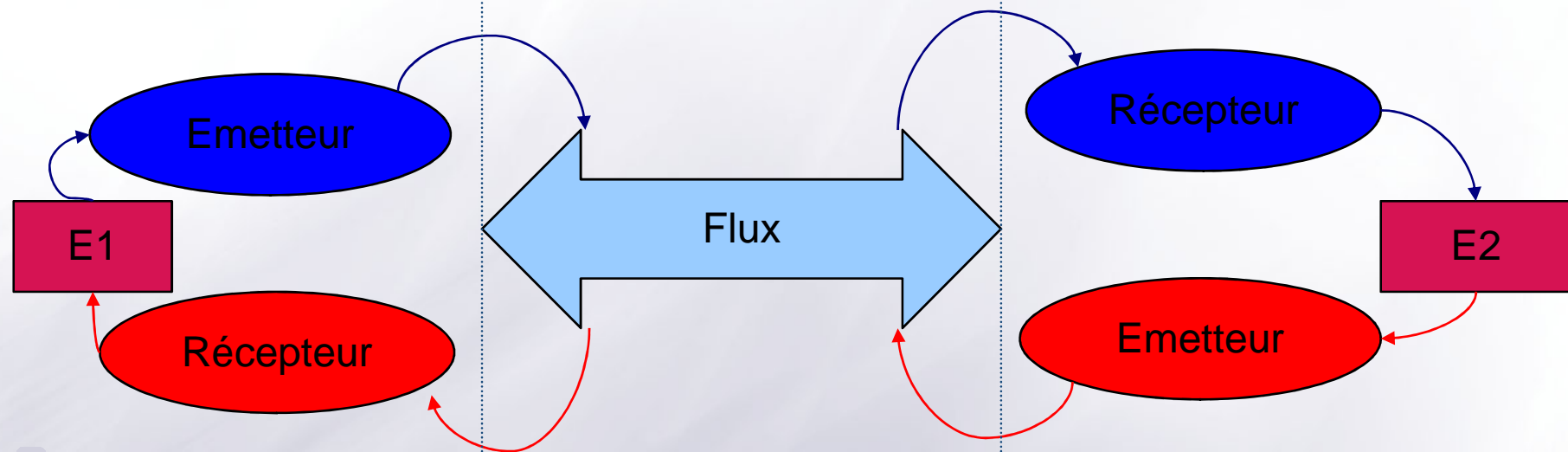
- Lecture du clavier
- Affichage sur la console
- Lecture/Ecriture dans un fichier
- Echange de données réseaux avec les Sockets

☰ **Stream: objet JAVA possédant la caractéristique de pouvoir envoyer ou recevoir des données**

Flux / Flots / Stream

- Un flux est une série d'informations envoyé sur un canal de communication
- C'est le paradigme utilisé dans le monde objet pour représenter une communication entre 2 entités qui ne sont pas des objets
- Un flux se comporte comme une rivière :
 - Il faut une source/émetteur
 - Et un receptr/consommateur
 - Un flux est bidirectionnel, par contre les points d'accès (émetteur/consommateur) ne peuvent travailler que dans un seul sens

Circulation d'information sur un flux



- Les flux informatiques ne véhiculent que des octets
- Les émetteurs et les récepteurs se chargent de transformer les octets sous des formes plus intéressantes pour les Entités (data, buffer, crypto...)
- Emetteur et récepteur doivent se mettre d'accord sur le format des structures envoyées (notion de protocole d'accord)
- La bibliothèque `java.io` définit l'ensemble des transformations que l'on peut désirer sur un ensemble d'octets arrivant dans le flux
- Il existe également des émetteurs et des récepteurs standards

Emetteurs et recepneur standards du système d'exploitation

Il existe 3 flux standards pour un système d'exploitation

- Les octets circulant entre une application (A) et l'écran (E) pour indiquer une information standard System.out
- Les octets circulant entre une application (A) et l'écran (E) pour indiquer une information d'erreur System.err
- Les octets circulant entre le clavier (C) et une application (A) System.in

Exercice: Représentez sur un schéma les flux et les points d'entrée qui vous interresserait en tant que programmeur

Exemple de saisie avec System.in

- Voici le code qui lit l'entrée du clavier et qui envoie le caractère sur la sortie standard (affichage du code ASCII du caractère)

```
import java.io.IOException;

public class MainClass {

public static void main(String[] args) throws IOException{
    System.out.println("Entrez un caractere");
    int inChar = System.in.read();
    System.out.print("Vous avez saisie: "+inChar);
    }

}
```

Intanciación des émetteurs / récepteurs

Il existe des entités classiques pouvant être la source ou la destination d'octets

● **Le fichier** : Il permet de stocker des octets

- Classe: `java.io.File` (nommage, droits d'accès et propriétés d'un fichier ou d'un répertoire)
- `File f=new File("/tmp/toto");`

● **La socket réseau** : elle permet d'envoyer des octets à une autre machine

- Classe: `java.net.Socket` (point d'entrée pour une connexion TCP/IP entre deux machines)
- `Socket s=new Socket("www.isecs.rnu.tn", 80)`

Instanciación de flujos de comunicación

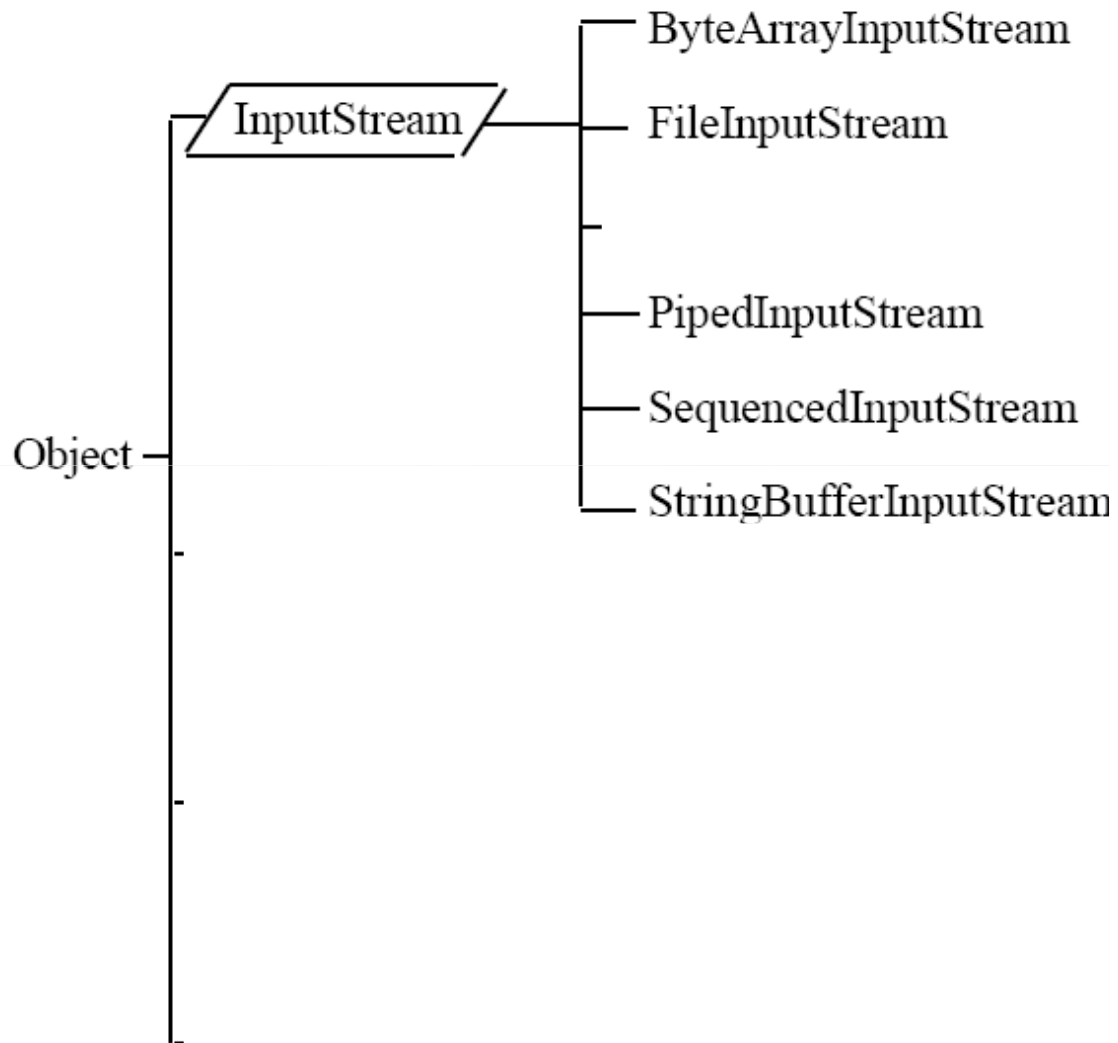
- Después de la creación de la fuente o destino, es necesario construir objetos de acceso al flujo para poder intercambiar los datos
- Los accesos en los puntos de entrada no son siempre homogéneos :

```
File f=new File("/tmp/toto");
FileInputStream fin=new FileInputStream(f);
fin.read();
Socket s=new Socket("www.isecs.rnu.tn", 80);
InputStream sin=s.getInputStream();
sin.read();
```

```
private void lecture (InputStream in){
    int i=in.read();
    while(i!=-1) {        System.out.println(i);
                        i = in.read();    }
}
```

- ¿Qué podemos decir de las clases `FileInputStream`, y `InputStream` ? ¿Cuáles son las funciones que se pueden aplicar sobre un flujo de entrada ?

java.io: arborescence de classes



API : InputStream

 **int available()**

- Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.

 **void close()**

- Closes this input stream and releases any system resources associated with the stream.

 **abstract int read()**

- Reads the next byte of data from the input stream.

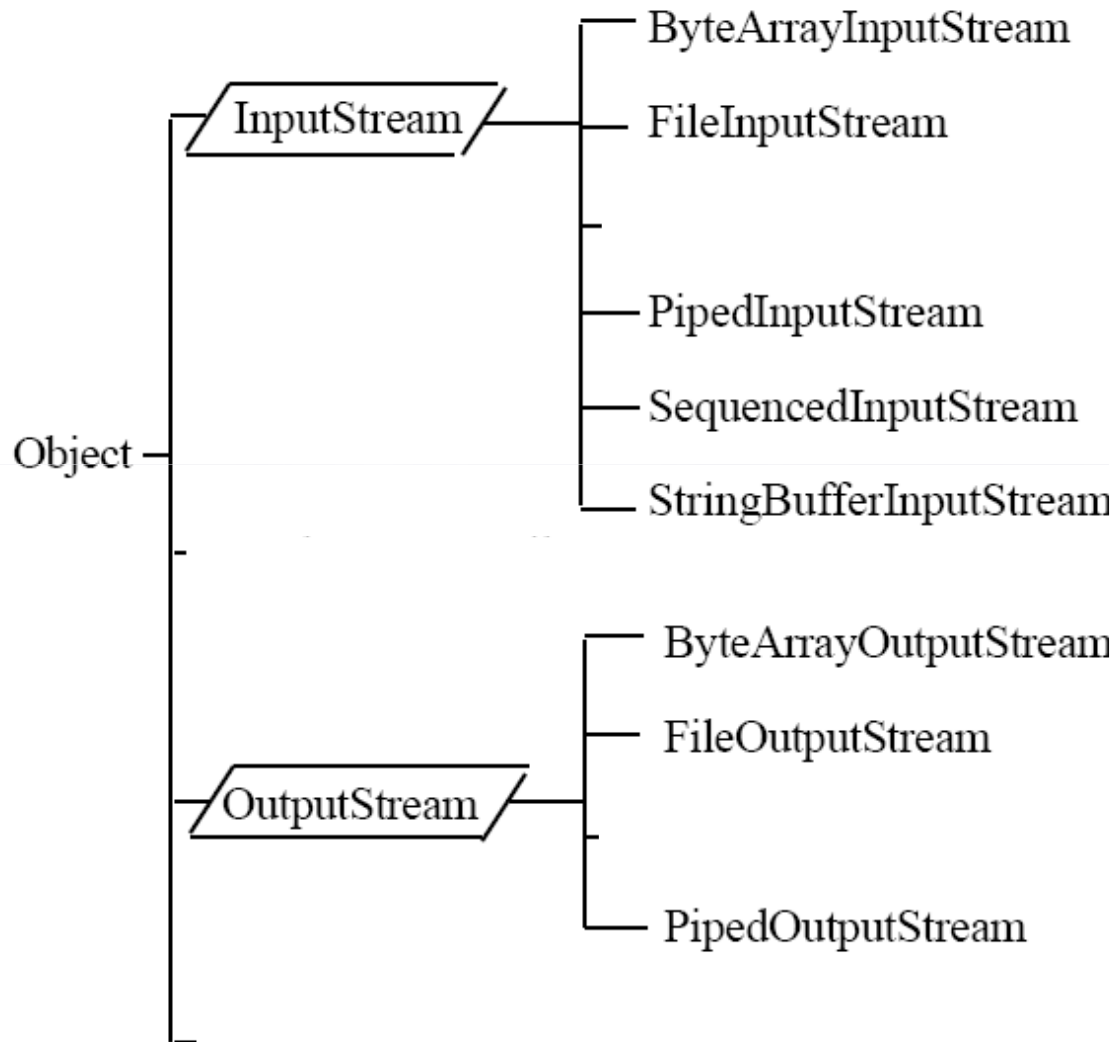
 **int read(byte[] b)**

- Reads some number of bytes from the input stream and stores them into the buffer array b.

 **int read(byte[] b, int off, int len)**

- Reads up to len bytes of data from the input stream into an array of bytes.

java.io: arborescence de classes



Exercice: copie de fichiers

- ☰ Ecrire une classe `Copy` qui réalise la copie d'un fichier dans un autre.
- ☰ Exemple d'utilisation:

```
java Copy sourceFile.txt destFile.txt
```

Que se passe-t'il si ?

- ☰ Que doit faire une entité lorsque qu'elle veut envoyer un long, double... ?
- ☰ Que doit faire une entité lorsqu'elle lit le byte, celui-ci n'est pas encore disponible ?

==> Il faut “décorer” nos flux



Décoration de flux

☰ La décoration est l'art de traiter une série d'octets avant de les transmettre à l'application demandeuse.

☰ Il existe 4 familles de décorations

- Data : permet de convertir les bytes en structure primitive (long, boolean, double...)
- Character : permet de convertir les bytes en texte
- Object : permet de convertir les bytes en structure objet
- Service : buffer, crypto, gzip permet d'appliquer des fonctions spécifiques sur la lecture

☰ Instanciation de décorateur:

```
DatInputStream dis=new DatInputStream((new  
FileInputStream("/tmp/toto"));
```

Flux de données: Data(Input|Output)Stream

- ☰ Ces classes permettent de lire et d'écrire des types primitifs sur des flux.

```
FileInputStream fis = new FileInputStream("source.txt");  
DataInputStream dis = new DataInputStream(fis);
```

```
int i    = dis.readInt();  
double d = dis.readDouble();  
String s = dis.readLine();
```

```
FileOutputStream fos = new FileOutputStream("cible.txt");  
DataOutputStream dos = new DataOutputStream(fos);
```

```
dos.writeInt(123);  
dos.writeDouble(123.456);  
dos.writeChars("Une chaine");
```

Flux de lecture / écriture avec buffer

- ☰ Cette classe permet de renvoyer, sur le flux de sortie *theOutput*, la chaîne de caractère lue sur le flux d'entrée *theInput*.

```
private void echoServer(InputStream theInput, OutputStream theOutput){
    // Un BufferedReader permet de lire par ligne.
    BufferedReader plec = new BufferedReader(
        new InputStreamReader(theInput));
    // Un PrintWriter possède toutes les opérations print classiques.
    PrintWriter pred = new PrintWriter(    new BufferedWriter(
        new OutputStreamWriter(theOutput,    true);

    while (true)
    {
        String str = plec.readLine(); // lecture du message
        if (str.equals("END")) break;
        System.out.println("ECHO = " + str); // trace locale
        pred.println(str); // renvoi d'un écho
    }
}
```

Flux de caractères

☰ Ajout de la notion de flux de caractères (charater Streams)

- Les `InputStream` et `OutputStream` se basent sur la lecture et écriture d'octets (bytes)
- Les caractères sont codés sur 2 octets
 - Utilisations de deux classes (`writer` et `reader`) à la place de `InputStream` et `OutputStream`
 - Ecriture facile des caractères `UNICODE` (avec accents par exemple)
 - Pour lire et écrire directement des caractères unicode dans un fichier, il est conseillé d'utiliser `FileReader` et `FileWriter` à la place de `FileInputStream` et `FileOutputStream`

Flux d'objets: Sériáisation

- La sériáisation (Serialization) est le processus de transformation d'un objet dans un flux d'octets, et la désériáisation le processus inverse.
- La sériáisation permet à un objet d'être facilement sauvé sur un fichier ou transféré sur le réseau
- Les classes doivent implanter l'interface Serializable et éventuellement surcharger les méthodes readObject() et writeObject()



Flux d'objets :Syntaxe

- La classe de l'objet qu'on veut sérialiser doit implémenter l'interface `java.io.Serializable`
- Généralement, il n'y a pas de méthodes à implémenter sauf dans le cas où le programmeur veut définir des méthodes de sérialisation ou désérialisation spécifiques à l'aide de:

```
-private void writeObject (java.io.ObjectOutputStream  
    out) throws IOException  
-private void readObject(java.io.ObjectInputStream in)  
    throws IOException, ClassNotFoundException;
```


Exemple de sérialisation d'un objet dans un fichier

```
import java.util.*;
import java.io.*
public class EcritObjPers {
    Date d = new Date();
    FileOutputStream f;
    ObjectOutputStream s;
    try {
        f = new FileOutputStream(« date.ser »);
        s = new ObjectOutputStream(f);
        s.writeObject(d);
        s.close();
    } catch (IOException e) {}
}
}
```



Exemple de désérialisation d'un objet à partir d'un fichier

```
import java.util.*;
import java.io.*
public class LiitObjPers {
    Date d = null;
    FileInputStream f;
    ObjectInputStream s;
    try { f = new FileInputStream(« date.ser »);
        s = new ObjectInputStream(f);
        d=(date)s.readObject();
        s.close();
        System.out.println(« date : »+d);
    } catch (IOException e) {}
}}
```

Exercice: Ecriture de logs

- **Ecrire un utilitaire de Gestion de logs « WriteLog ».** Chaque log représente une ligne dans un fichier texte. Sur chaque ligne on trouve la date d'écriture, l'identifiant de son écrivain et son commentaire. Utilisez une classe `Entry` ayant pour attributs: la date, l'identifiant de l'écrivain et son commentaire. Cette classe doit définir la méthode de sérialisation de ses attributs dans un flux de sortie `OutputStream` ouvert.

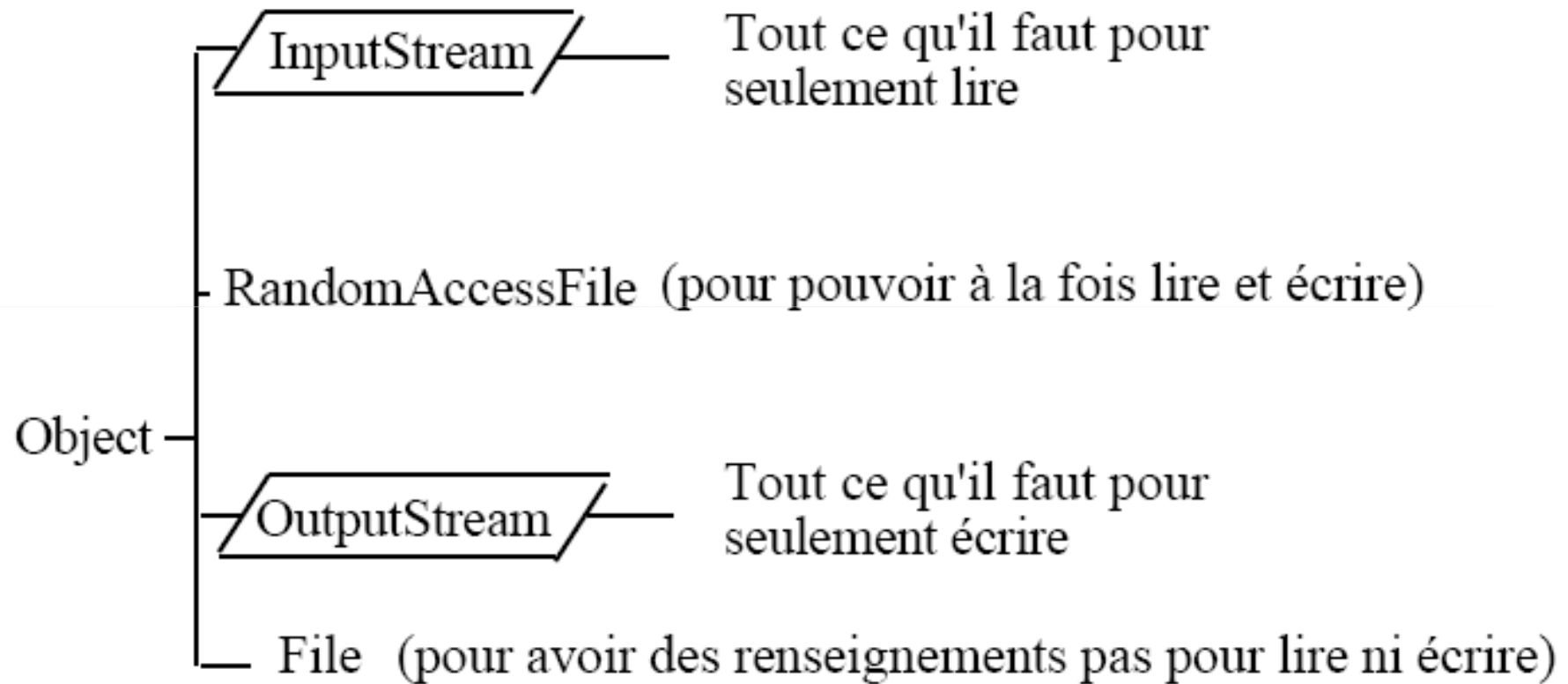


Exercice: Lecture de logs

☰ Ecrire un utilitaire « ScanLog » de lecture des logs enregistrés dans l'exercice précédent. Cet utilitaire doit permettre un parcours sélectif en utilisant les options suivantes:

- after d* : pour afficher les enregistrements écrits après la date *d*
- before d* : pour afficher les enregistrements écrits avant la date *d*
- user u* : pour afficher que les enregistrements écrits par l'écrivain *u*

java.io: ce qu'il faut retenir



LES SOCKETS

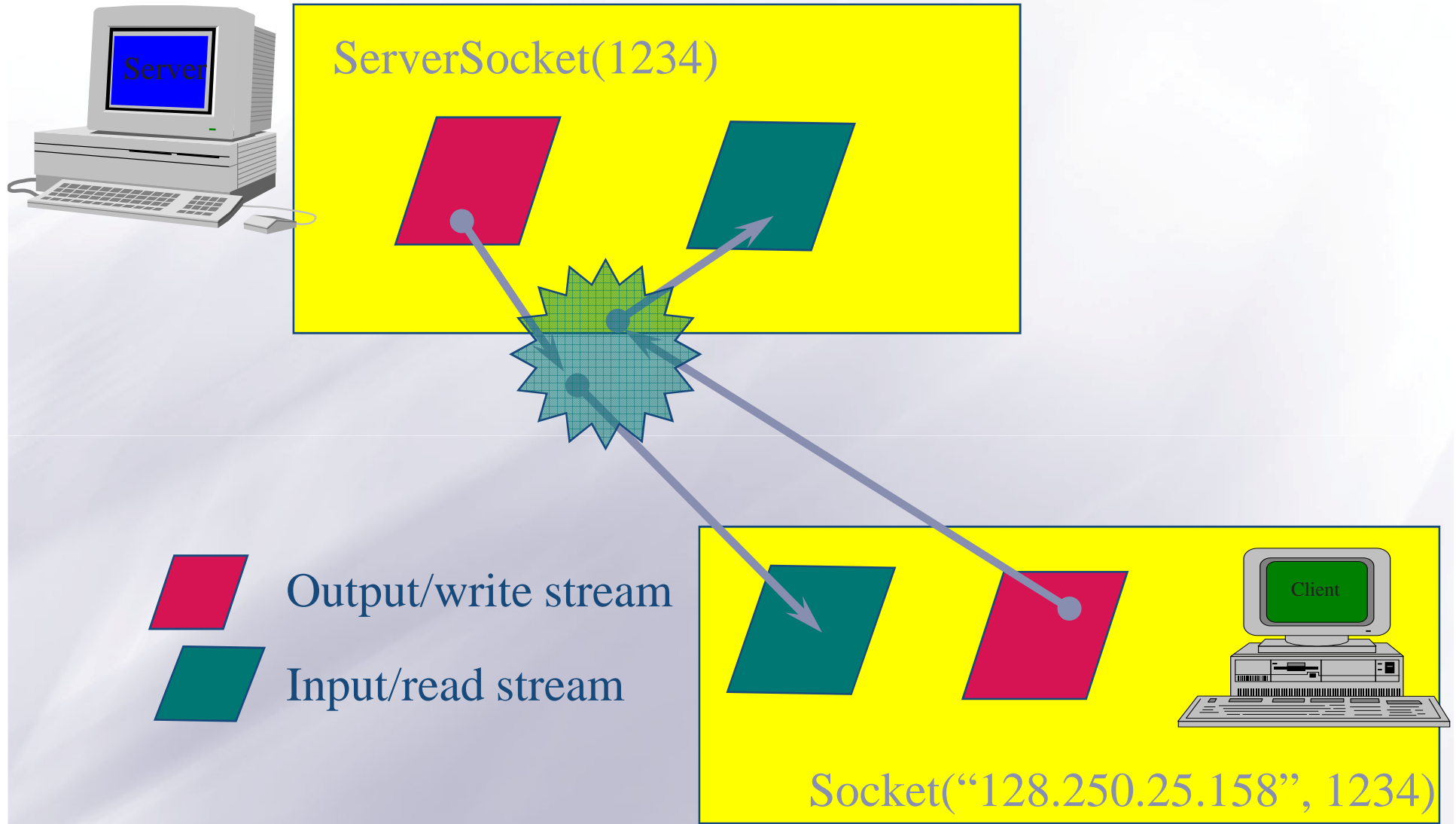
Un Socket est un point de communication bidirectionnelle entre applications



Définitions

- Un Socket est un point de communication bidirectionnelle entre applications
- Un socket est associé à un port de communication pour différencier les applications réseau sur une même machine
- L'API java fourni un package `java.net` pour l'utilisation des packages
 - `java.net.Socket` pour l'implémentation coté client
 - `java.net.ServerSocket` pour l'implémentation coté serveur

Schéma de communication des sockets



Peut être un nom complet "www.isecs.rnu.tn"

Java.net.InetAddress : nommage

☰ Cette classe permet de manipuler les adresses Internet.

☰ Pour obtenir une instance de cette classe :

- `public static InetAddress getLocalHost()`
throws `UnknownHostException`

Elle retourne un objet contenant l'adresse Internet de la machine locale.

- `public static synchronized InetAddress getByName(String host_name)`
throws `UnknownHostException`

Elle retourne un objet contenant l'adresse Internet de la machine dont le nom est passé en paramètre.

☰ Les méthodes principales de cette classe sont :

- `public String getHostName ()`

Retourne le nom de la machine dont l'adresse est stockée dans l'objet.

- `public String toString ()`

Retourne une chaîne de caractères qui liste le nom de la machine et son adresse.

☰ Exemple :

```
InetAddress adresseLocale = InetAddress.getLocalHost ();
```

```
InetAddress adresseServeur = InetAddress.getByName("www.google.fr");
```

```
System.out.println(adresse);
```

Fonctions d'un sockets

4 fonctions principales:

- Se connecter à une machine distante
- Envoyer des données
- Recevoir des données
- Fermer la connexion

 Un socket ne peut se connecter qu'à une seule machine

 Un socket ne peut pas être se reconnecter après la fermeture de connexion



Instanciación d'un socket

☰ La création d'un socket se fait à l'aide de l'un de ces constructeurs

- `public Socket(String host, int port) throws UnknownHostException, IOException`
- `public Socket(InetAddress address, int port) throws IOException`
- `public Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException`
- `public Socket(InetAddress address, int port, InetAddress localAddr, int localPort) throws IOException`

☰ Ces constructeurs créent un objet Socket et ouvre une connexion réseau avec la machine distante 'host'

☰ Tous ces constructeurs lèvent une exception en cas d'impossibilité de connexion

Connexion d'un socket

- Pour qu'un socket puisse se connecter à une machine distante, cette dernière doit écouter les requêtes des clients sur le port spécifié
- Il faut définir au minimum l'adresse de la machine distante et son port d'écoute pour pouvoir se connecter
- On peut utiliser les constructeurs pour déterminer les ports d'écoute d'une machine



Exercice

- Ecrire un programme Java qui affiche sur l'écran les ports d'écoute d'une machine distante donnée (en ligne de commandes)
- Rappel: les ports TCP d'une machine peuvent aller de 0 à 65535



Envoi et réception de données

- Les données sont envoyées et reçues à travers les flux d'entrée/sortie

```
public    InputStream    getInputStream()    throws  
    IOException  
public    OutputStream    getOutputStream()    throws  
    IOException
```

- Il y a aussi une méthode qui permet de fermer le flux d'envoi et de réception de données

```
public    synchronized    void    close()    throws  
    IOException
```

Réception de données

- La méthode `getInputStream()` permet d'obtenir le flux d'entrée pour la réception des données
- On peut utiliser les méthodes classiques de `InputStream` pour récupérer les données entrantes
- Généralement, on utilise les décorateurs pour faciliter la manipulation des données reçues

```
DataInputStream input;  
try {  
    input = new DataInputStream(mySocket.getInputStream());  
    String textFromServer = input.readLine();  
    System.out.println(textFromServer);  
}  
catch (IOException e) {System.out.println(e);}
```


Exemple d'un Socket coté client

- ☰ Ce programme permet de se connecter à un serveur qui fourni le temps sur le port TCP 13

```
try {
    Socket s = new Socket("metalab.unc.edu", 13);
    InputStream in = s.getInputStream();
    InputStreamReader isr = new
    InputStreamReader(in);
    BufferedReader br = new BufferedReader(isr);
    String theTime = br.readLine();
    System.out.println(theTime);
}
catch (IOException e) {
    return (new Date()).toString();
}
```

Envoi de données

- La méthode `getOutputStream()` permet d'obtenir le flux de sortie pour l'envoi des données
- On peut utiliser les méthodes classiques de `OutputStream` pour envoyer des données à travers les sockets
- Généralement, on utilise les décorateurs pour faciliter l'envoi de données

```
PrintStream output;  
try {  
    output = new PrintStream(mySocket.getOutputStream());  
    output.print("Coucou serveur, je suis le client 😊");  
}  
catch (IOException e) {System.out.println(e);}
```


Quelques autres méthodes des Sockets

Informations générales sur la connexion :

```
public InetAddress getInetAddress()  
public InetAddress getLocalAddress()  
public int getPort()  
public int getLocalPort()
```

La méthode usuelle toString() :

```
public String toString()
```

La partie Serveur

- Dans une communication client/serveur, on a le programme client qui initialise la connexion d'un coté et la partie serveur qui répond aux requettes des clients
- Les clients et les serveurs sont connectés par des sockets
- Contrairement au client qui se connecte à un poste distant, le serveur attend les connections d'autres postes pour répondre à leurs requêtes

Les sockets Serveur

- Un socket serveur est attaché à un port particulier sur la machine locale
- Ensuite, il écoute sur ce port pour attendre les requête des clients
- Quand un serveur détecte une tentative de connexion, il lance une méthode *accept()*. Ceci crée automatiquement une socket entre le client et le serveur qui représente un canal de communication entre eux

Files d'attente

- Les connexions entrantes sont mises en file d'attente jusqu'à ce que le serveur puisse les accepter
- En général, la taille de la file d'attente est entre 5 et 50
- Si la file d'attente est pleine, le socket serveur refuse les connexions supplémentaires

La classe `java.net.ServerSocket`

- La classe `java.net.ServerSocket` représente un socket serveur
- Un socket serveur est affecté à un port TCP dès son instantiation. Ensuite, il suffit d'appeler sa méthode `accept()` pour écouter les connexions entrantes
- `accept()` bloque le programme serveur jusqu'à ce qu'une connexion est détectée. Dans ce cas, la méthode `accept()` retourne un objet socket (`java.net.Socket`)

Instanciation

☰ Il y a 3 constructeurs principaux qui permettent:

- l'affectation de la socket à un port TCP
- La longueur de la file d'attente
- L'adresse IP sur laquelle la socket sera affectée

```
public ServerSocket(int port) throws IOException
```

```
public ServerSocket(int port, int backlog) throws  
IOException
```

```
public ServerSocket(int port, int backlog,  
InetAddress bindAddr) throws IOException
```

Instanciation classique :

```
try {  
    ServerSocket serverSocket = new ServerSocket(80);  
}  
catch (IOException e) {  
    System.err.println(e);  
}
```

Affectation aux ports

- Quand un `ServerSocket` est crée, il essaye de s'attacher au port défini dans le constructeur
- S'il existe un autre socket serveur qui écoute déjà sur le port spécifié, l'exception `java.net.BindException`, (sous classe de `IOException`) est levée
- Aucun processus serveur ne peut être affecté à un port déjà utilisé par un autre processus serveur qu'il soit développé en java ou non.

Méthodes principales d'un server socket

☰ Les méthodes principales d'un server socket sont :

- `public Socket accept() throws IOException`
- `public void close() throws IOException`

☰ Un socket serveur ne peut plus être réinitialisé une fois qu'il est fermé

Lecture de données à l'aide d'un socket serveur

`ServerSocket` utilise la méthode `accept()` pour se connecter à un client

```
public Socket accept() throws IOException
```

Il n'y a pas des méthodes comme `getInputStream()` ou `getOutputStream()` pour un `ServerSocket`

`accept()` retourne un objet de type `java.net.Socket` avec lequel on peut obtenir les flux d'entrée et de sortie à l'aide des méthodes `getInputStream()` et `getOutputStream()`

Exemple d'un socket serveur

```
try {  
    ServerSocket ss = new ServerSocket(2345);  
    Socket s = ss.accept();  
    PrintWriter pw = new  
        PrintWriter(s.getOutputStream());  
    pw.print("Hello There!\r\n");  
    pw.print("Really I have nothing to say...\r\n");  
    pw.print("Goodbye now. \r\n");  
    s.close();  
}  
catch (IOException e) {  
    System.err.println(e);  
}
```

Un autre exemple

```
try {
    ServerSocket ss = new ServerSocket(port);
    while (true) {
        try {
            Socket s = ss.accept();
            PrintWriter pw = new PrintWriter(s.getOutputStream());
            pw.print("Hello " + s.getInetAddress() + " on port "
                + s.getPort() + "\r\n");
            pw.print("This is " + s.getLocalAddress() + " on port "
                + s.getLocalPort() + "\r\n");
            pw.flush();
            s.close();
        }
        catch (IOException e) {}
    }
}
catch (IOException e) { System.err.println(e); }
```

Techniques recommandées d'un serveur socket

- Il est recommandé d'ajouter du multithreading dans le serveur
- Il faut avoir une boucle continue qui traite les requêtes des clients
- Au lieu de traiter les requêtes directement, chaque socket devrait être passé à un Thread

```
while (true) {  
    try {  
        Socket s = ss.accept();  
        ThreadedEchoServer tes = new ThreadedEchoServer(s);  
        tes.start();  
    }  
    catch (IOException e) {}  
}
```

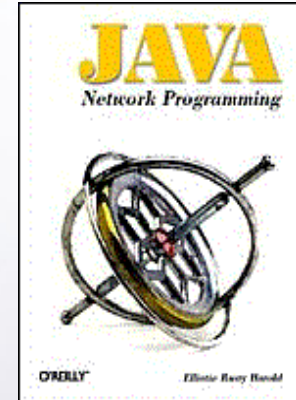
Exercice

- ☰ Réaliser deux versions d'un serveur d'écho:
 - une sans les Thread
 - une en utilisant les Thread
- ☰ Développer un client pour tester les serveurs d'écho
- ☰ Développer un serveur HTTP simple qui renvoi le contenu du fichier demandé dans une requête *Get*

Pour apprendre plus sur les sockets

☰ Java Network Programming

- O'Reilly & Associates, 1997
- ISBN 1-56592-227-1



☰ Java I/O

- O'Reilly & Associates, 1999
- ISBN 1-56592-485-1

