JAVA

Introduction à la programmation objet avec java – 3/4

SOMMAIRE

- Packages, Visibilités (15mn)
- les exceptions (50mn)
- java.io: entrées/sorties fichiers (15mn)
- java.net: entrées/sorties réseaux (20mn)

Partie 1

Packages, Visibilités

Les packages

- Un package regroupe un ensemble de classes sous un même espace de nommage
  - L'intérêt est de regrouper les classes par thème, lien logique, dépendance…
- un package équivaut à un répertoire contenant des fichiers classes
- un package peut contenir des sous-packages
- pour utiliser des classes contenues dans un package, il faut soit :
  - utiliser leur nom long `package.souspackage.classe`
  - utiliser la clause import (avec ou sans « * »)
  - le * n ’est pas récursif !

Packages personnalisés

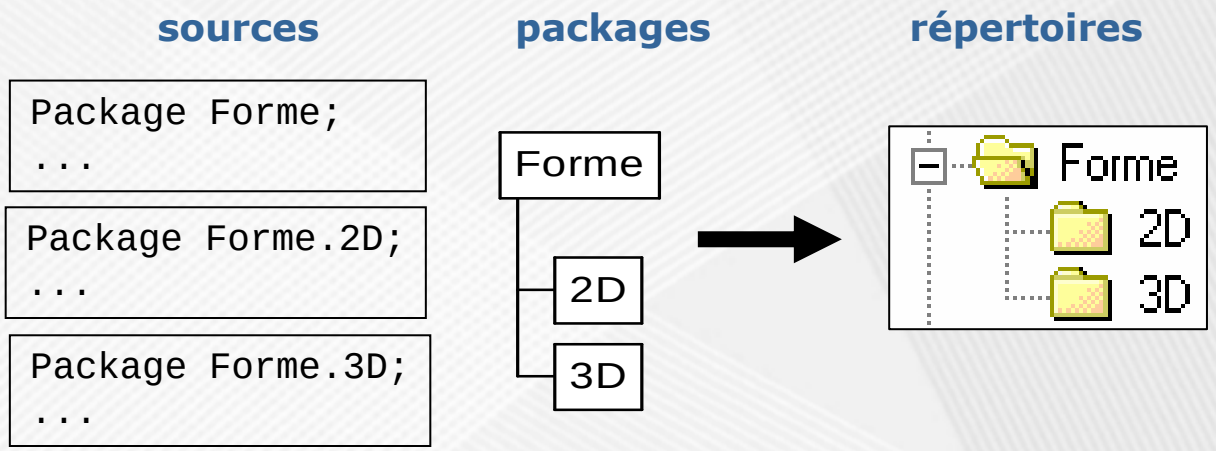
- L'instruction `package` indique à quel package appartient la ou les classe(s) du fichier.
- `package` doit être la première instruction du fichier source
- Par défaut (si pas d'instruction `package` dans le fichier) une classe appartient au package anonyme.
- Exemple :

```
package forme;

class Circle {
    double x, y, r;
    void Circle(double x, double y, double r) {
    this.x = x; this.y = y; this.r = r;
    }
    ...
```

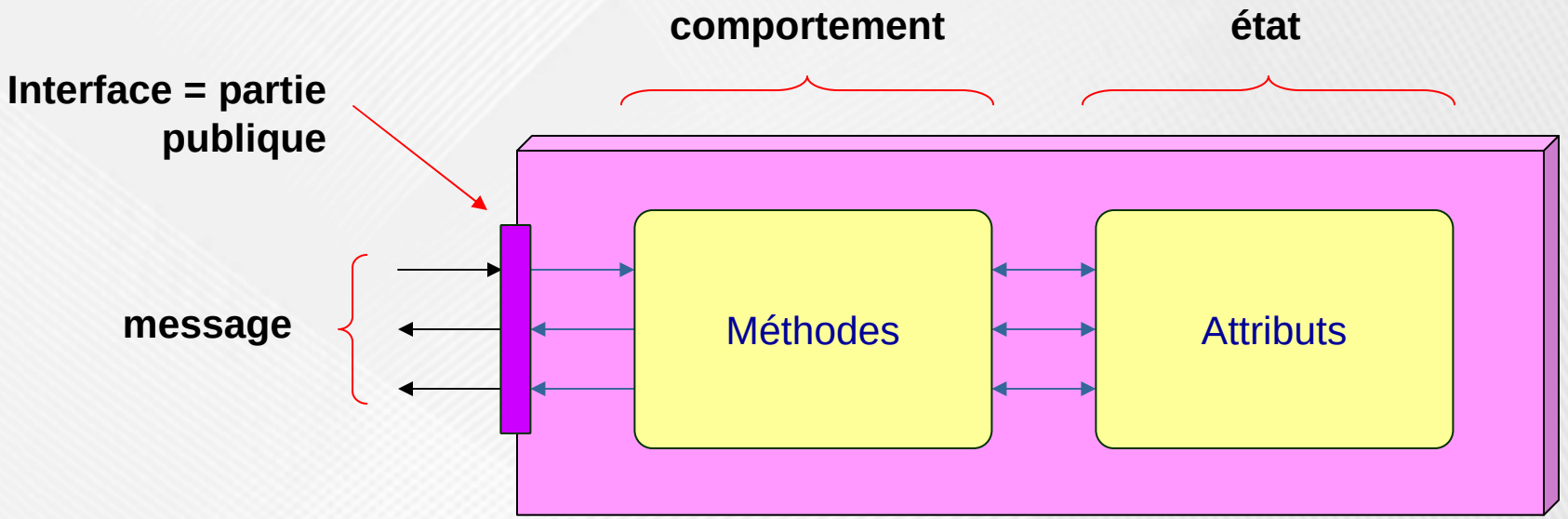
Packages personnalisés

- L'arborescence des répertoires contenant les classes doit être identique à l'arborescence des packages
  - les noms des packages/répertoires **sont** case sensitifs
  - par defaut le package anonyme est le repertoire courant
- Une classe `Circle` appartenant au package `Forme.2D` doit se trouver dans le fichier `Forme/2D/Circle.class`



- Le compilateur sait créer automatiquement les sous-répertoires pour y placer les fichiers classes
- Les répertoires contenant les packages doivent être dans la variable d'environnement CLASSPATH.

Rappel: Principe d'encapsulation



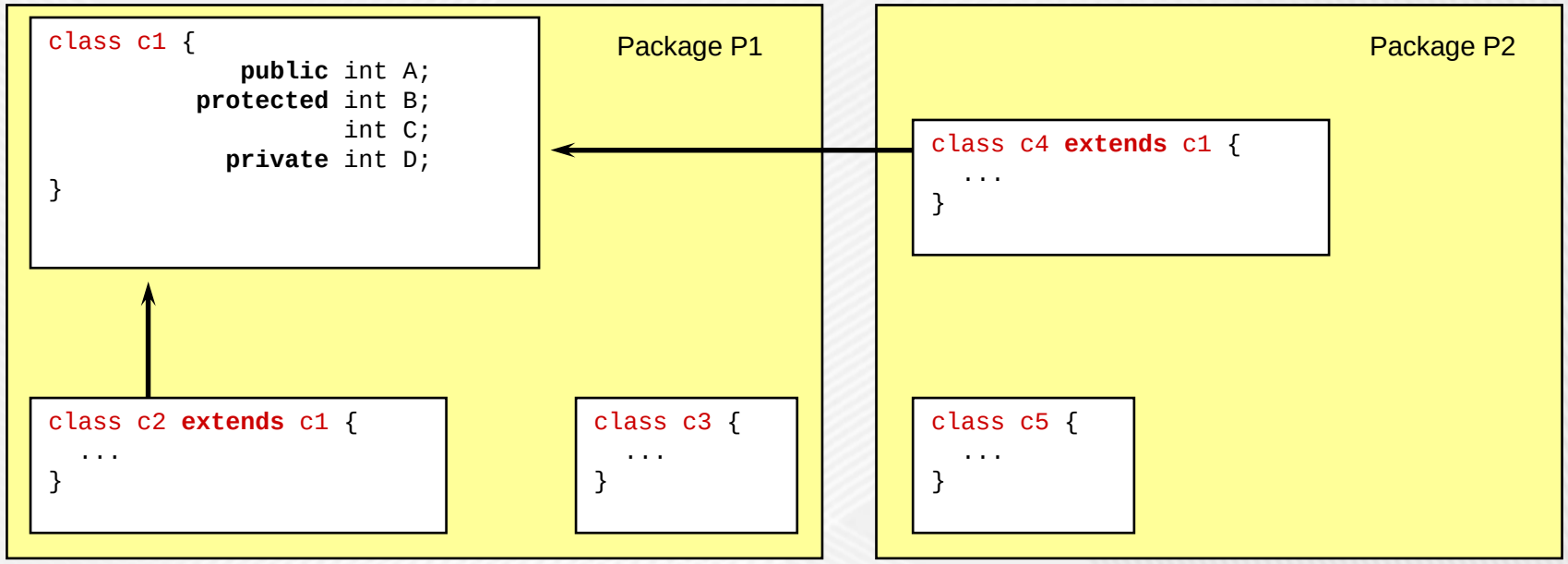
- Un objet est composé de 2 parties :
  - partie publique : opérations qu'on peut faire dessus
  - partie privée : partie interne (intime) = données sensibles de l'objet (les attributs et les autres méthodes)
- Les utilisateurs de l'objet ne voient (cad ne peuvent utiliser et ceci est contrôlé par le compilateur) que la partie publique (qui est un ensemble de méthodes)

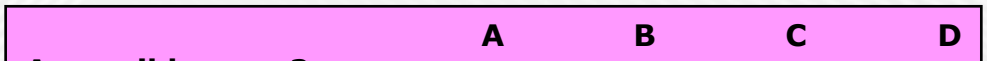
Encapsulation & Visibilité

Mots clés utilisable pour gérer les droits d'accès pour un membre (variable ou méthode) :

- **public**
  - visible par toutes les autres méthodes à l'intérieur de l'objet ou à l'extérieur.
- **protected**
  - visible uniquement dans sa classe ou sous classe (et dans le package)
- **private**
  - private sera visible uniquement dans sa classe.
- **<sans>**
  - visible uniquement dans le package en cours - Equivalent à public mais pour le package

Encapsulation & Visibilité





Partie 2

Les exceptions

- Définition
- Traiter une exception
- Types d'exceptions
- Méthodes de la classe Exception
- Propager une exception
- Définir ses classe d'Exception

Les exceptions

- Comment peut t'on gérer les erreurs ?
  - Etablir une convention sur le code retour
    - par exemple : 0 ou !0 en C (qui est ok?)
    - renvoyer un booleen ou un String ou autre
  - positionner la valeur d'une variable globale
    - les shells Unix (et C) font ca : "errno"
    - en Java on pourait utiliser un champs static
  - lancer (lever) une exception, l'attraper ailleurs dans le programme et faire une action corrective
- Java offre la possibilité des exceptions
  - une manière de gérer les erreurs plus orientée objet
  - plus puissant et plus flexible que des return
  - Une exception correspond à un événement anormal ou inattendu
  - mots clés : **`try`**, **`catch`**, **`throw`**, **`throws`**, **`finally`**

Les exceptions

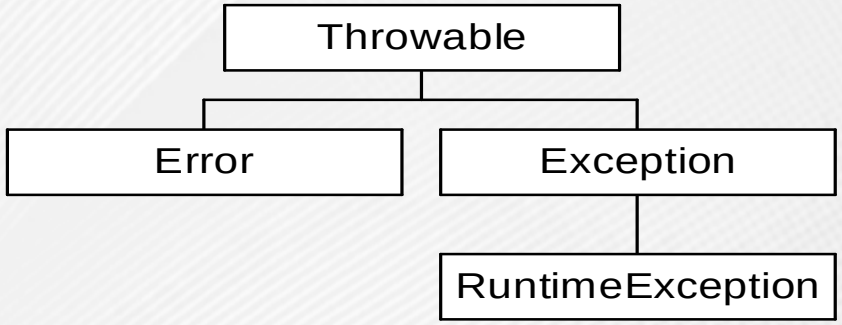
- Elles permettent de séparer un bloc d'instructions de la gestion des erreurs pouvant survenir dans ce bloc.
- Format:

```
try {
    // Code pouvant lever des Exception
    // (exemple : ouverture de fichier)
}
catch (Exception e) {
    // Gestion des Exceptions
    // on ecrit ici le code pour recupérer le programme
    // malgré cette erreur
    // on peut se contenter d'afficher un message et
    // poursuivre le programme
}
```

Les exceptions

- Principe d'utilisation:
  - Emetteur: code où il faut anticiper un problème:
    - détecter l'erreur, probablement avec un **`if`** …
      créer une nouvelle Exception et la lancer (**`throw`**)
    - ou laisser la JVM détecter l'erreur, créer et lancer une Exception
  - Recepteur: code dans l'appelant (quelque part dans la pile d'appel)
    - tenter un **« `try` »**, en espérant que ca marche comme prévu (ecrire le code comme si tout marchait)
    - se préparer à attraper (**`catch`**) une Exception (et recoller les morceaux si nécessaire!)

Les exceptions



- Une exception est toujours une instance d'une classe fille de Throwable
- `java.lang.Error:` Erreur fatale
  - erreur qui va entrainer la fin du programme
  - à ne pas gérer (inutile)
- `java.lang.Exception :` erreur que l'on peut traiter
  - sous Exception, il y a par exemple IOException qui n'est pas une *RuntimeException*
- `java.lang. RuntimeException:` erreur d'exécution
  - erreurs comme la division par zéro, débordement…

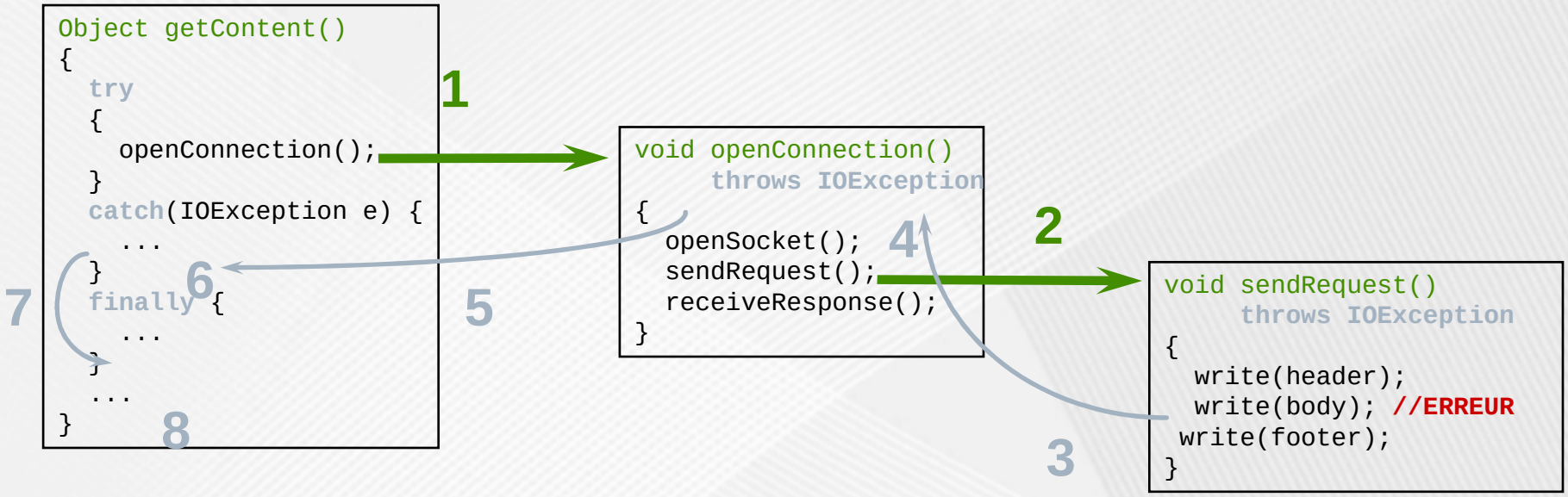
Traiter les exceptions

- Quant une exception est lancée, toutes ou une partie des instructions suivantes sont ignorées.
- La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc catch acceptant cette exception soit trouvé
- Si aucun bloc catch n'est trouvé, l'exception est capturée par l'interpréteur (JVM) qui l'affiche et le programme s'arrête.

- Un bloc optionnel finally peut-être posé à la suite des catch. Son contenu sera exécuté qu'il ait eu une exception ou pas.
  - Notamment après un catch ou après un break, un continue ou un return dans le bloc try

Les exceptions (3)

Exemple de séquencement :



Exemple : try

```
int[] tableau = new int[30] ;
// Initilialisation du tableau avec une boucle infinie
try {
     for (int i = 0 ; /* Tant que vrai */; i ++)
          tableau[i] = i ;

     System.out.println (i);
}
catch (ArrayIndexOutOfBoundsException e) {
     // ou e est une instance de la classe
     // ArrayIndexOutOfBoundsException
     System.out.println (e + " : Erreur : dépassement d'indice : " +
                          e.getMessage ( ) ) ;
}

// Les instructions suivantes sont exécutées
System.out.println ("Taille du tableau : " + tableau.length) ;
System.out.println ("Fin") ;
```

- Exécution :

```
Java.lang.ArrayIndexOutOfBoundsException : Erreur : dépassement d'indice
     : null
Taille du tableau : 30
Fin
```

Exemple : finally

- Programme :

```
int[] tableau = new int[30],i ;
// Initilialisation du tableau avec
// une boucle infinie d'où ArrayIndexOutOfBoundsException
try {
    for (i = 0 ; i<max; i ++)
    tableau[i] = i ;
}
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println ( "Erreur : dépassement d'indice" ) ;
}
finally {
    System.out.println ("Taille du tableau : " + tableau.length) ;
}
```

Exécution si max>30:

```
Erreur : dépassement d'indice
Taille du tableau : 30
```

Exécution si max=30:

```
Taille du tableau : 30
```

Traiter plusieurs exceptions

- Un même bloc peut générer plusieurs types d'exception. Plusieurs blocs catch peuvent co-exister :

```
try {
   instructions pouvant générer une ou
   plusieurs exceptions ;
}
catch (MonException1 e) {
   …
}
catch (MonException2 e) {
   …
}
```

Throw: lancer une exception

- Il est possible de lancer et propager une exception
- Mot clef `throw` qui permet lancer une instance d'exception sur une instruction donnée

```
int Stock ( ) throws PlusDeStockException {
    if (Quantite == 0)
    throw new PlusDeStockException ("Je n'ai plus de stock
    pour l'article " + Designation) ;
    return (Quantite) ;
}
```

- on peut aussi lancer les exceptions natives de l'API Java
- si l 'exception ne dérive pas de RuntimeException, on est obligé de la déclarer dans l'entête:
  - Mot clef `throws` dans l'entete de la fonction
  - on peut en déclarer plusieurs, séparés par des virgules

Définir une exception

- Les exceptions héritent toutes de la classe `Exception` définie dans `java.lang`.
- On peut définir une exception personnalisé en l'héritant de cette classe
- Elle peut contenir deux constructeurs qui appellent à l'aide du mot clef `super` le constructeur de la classe Exception.
- La `String` passée au 2ième constructeur est obtenue lors d'une exception (déclenchement) par la méthode `getMessage` définie dans la classe Exception
- Exemple :

```
class MonException extends Exception {
  public MonException ( ) {
    super ( ) ;
  }
  public MonException ( String S ) {
    super ( S ) ;
  }
}
```

On peut évidemment dériver MonException en Sous Exceptions

Les exceptions : résumé

- Les exceptions de manière générale doivent être traitées
- Traitement des exceptions :
  - Dans le bloc d'instructions en cours avec try, catch, [finally]
  - En répercutant l'exception dans la méthode appelante en ajoutant à la suite de la déclaration de la méthode en cours le mot clef throws
- Le traitement des exceptions est identique pour une exception native ou une exception personnalisée

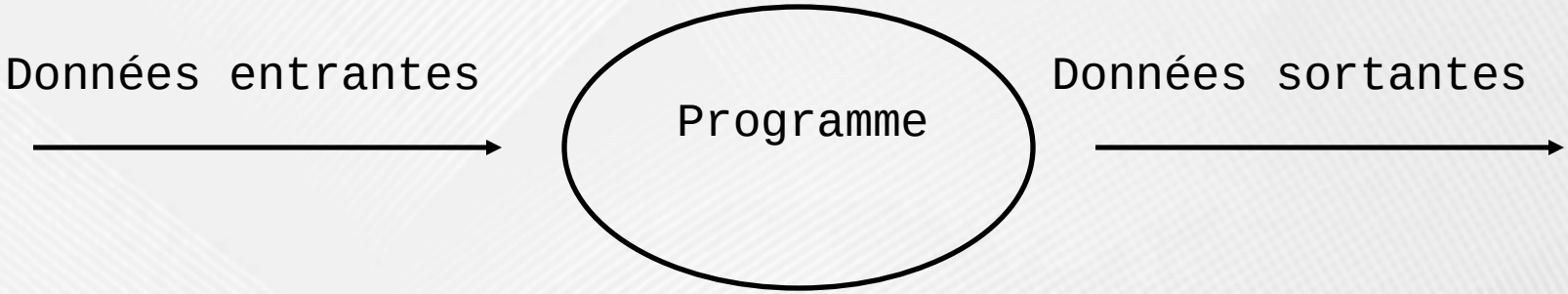
Partie 3

Entrées/sorties fichiers (java.io)

Rappel : Les *core* API

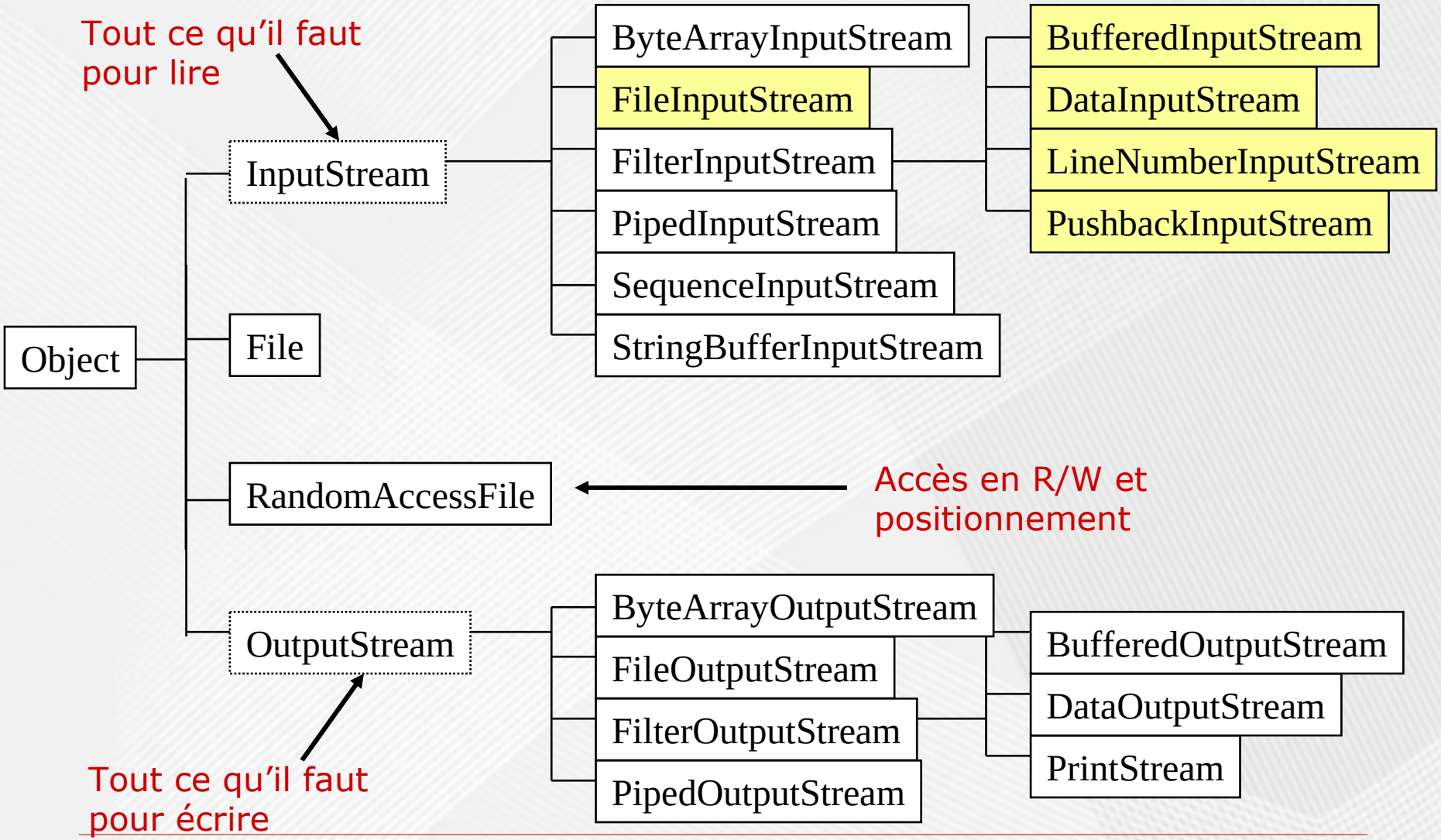
- **Equivalent des librairies standards de C/C++:**
  - **java.lang** : Types de bases, Threads, Exception, Math, ...
  - **java.util** : Hashtable, Vector, Stack, Date, ...
  - **java.applet**
  - **java.awt** : Interface graphique portable
  - **java.io** : accès aux i/o par flux (fichiers, stdin, stdout,..)
  - **java.net** : Socket (UDP, TCP, multicast), URL, ...
  - **java.lang.reflect** : introspection sur les classes et les objets
  - **java.beans** : composants logiciels
  - **java.sql** (JDBC) : accès homogène aux bases de données
  - **java.security** : signature, cryptographie, authentification

Le modèle des Stream



- Un stream doit être vu comme un tuyau
- la data peut etre de type characters, lines, bytes, objects,…
- les Streams se connectent à des fichiers , des terminaux, des chaines de caractères, le réseau, …
- utilise toujours les même méthodes : read, write (polymorphisme)

Les API java: java.io.*



Les API java: java.io.File

- Information sur les fichiers (taille, chemin..)
- Constructeurs

```
File(String path) ou (String path, String name) ou
    (File dir, String name)
```

- Methods

```
boolean exists(), isFile(), isDirectory(),
         canRead(), canWrite();
long    length(), lastModified();
boolean delete(), mkdir(), mkdirs(),
             renameTo(File dest);
String  getName(), getParent(), getPath(),
          getAbsolutePath()
```
: retourne le nom du fichier, le nom du répertoire père

Les API java: java.io.File

- Cette classe fournie une description *plateforme-independante* des fichiers et des répertoires.

```
File f = new File("/etc/passwd");
System.out.println(f.exists());   // --> true
System.out.println(f.canRead());  // --> true
System.out.println(f.canWrite()); // --> false
System.out.println(f.getLength()); // --> 11345

File d = new File("/etc/");
System.out.println(d.isDirectory()); // --> true

String[] files = d.list();
for(int i=0; i < files.length; i++)
   System.out.println(files[i]);
```

Les API java: java.io.File(Input|Output)Stream

- Ces classes permettent d'accèder en lecture et en écriture à un fichier.

```
FileInputStream fis = new FileInputStream("source.txt");
byte[] data = new byte[fis.available()];
fis.read(data);
fis.close();

FileOutputStream fos = new FileOutputStream("cible.txt");
fos.write(data);
fos.close();
```

Les API java: java.io.Data(Input|Output)Stream

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

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

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

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

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

Les API java: java.io.PrintStream

- Cette classe permet de manipuler un `OutputStream` au travers des méthode `print()` et `println()`.

```
PrintStream ps = new PrintStream(new FileOutputStream("cible.txt"));

ps.println("Une ligne");
ps.println(123);
ps.print("Une autre ");
ps.print("ligne");
ps.flush();
ps.close();
```

Lire au clavier

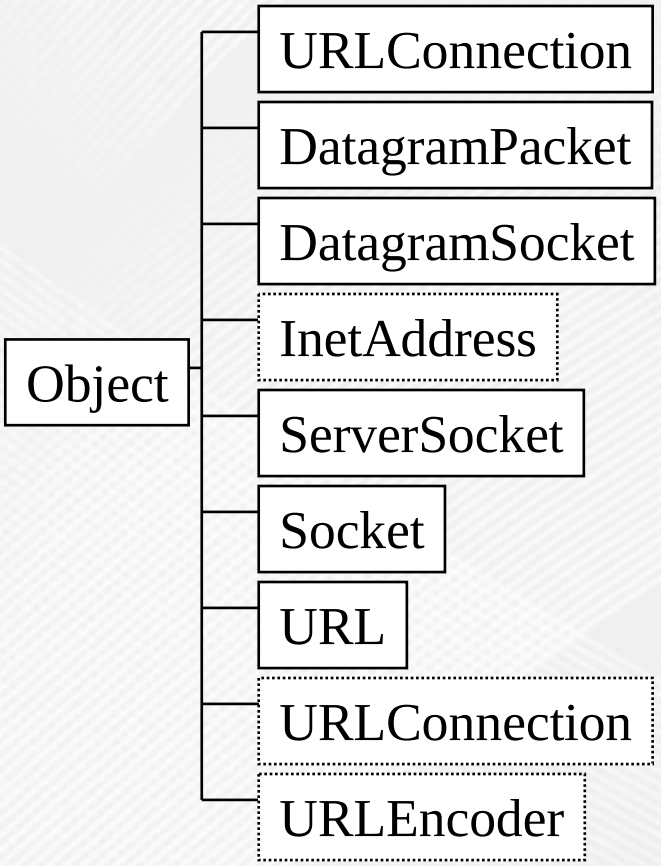
```
// méthode la plus recommendée

import java.io.*;
try {
    InputStreamReader isr = new InputStreamReader (System.in);
    BufferedReader br = new BufferedReader (isr);
    String ligne = br.readLine();
}
catch (IOException e) {…}
```

Partie 4

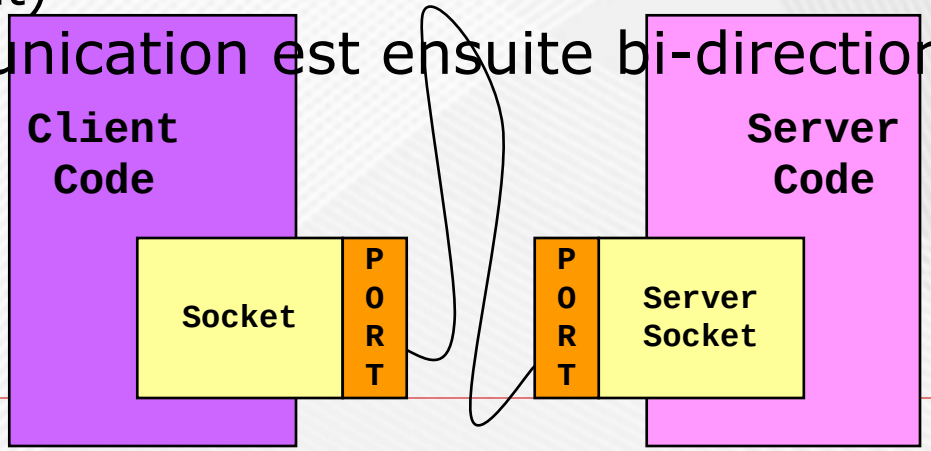
Entrées/sorties réseaux (java.net)

Entrées/sorties réseaux (java.net)



Socket

- Java gère les protocoles UDP et TCP
- Requis coté Serveur:
  - Attente d'une connexion cliente :
    - Utilisation d'un port local sur lequel les connexions sont attendues
- Requis coté Client:
  - Connexion à un serveur donné :
    - Connaissance du nom ou de l'adresse IP du serveur
    - Connaissance du port ouvert par le serveur
    - (+ allocation dynamique par le système d'un port sur le client)
- La communication est ensuite bi-directionnelle



Class java.net.InetAddress

Methodes statiques (création)

- InetAddress **getByName**(String host)
- InetAddress[] **getAllByName**(String host)
- InetAddress **getLocalHost**()
- Exemples
  - `InetAddress host = InetAddress.getLocalHost();`
  - `InetAddress host = InetAddress.getByName("www.iut.fr");`

Methodes sur une instance

- byte[] **getAddress**()
- String **getHostAddress**()
- String **getHostName**()
- boolean **isMulticastAddress**()

InetAddress : exemple

```
import java.net.*;

public class WhereAmI {
    public static void main( String[] argv ) {
                try {
                    System.out.println(InetAddress.getLocalHost());
                }
                catch(UnknownHostException e) {
                    System.out.println(e);
                }
    }
}

public static void Details(InetAddress adresse)
{
  String chaine = adresse.toString();
  String nom = adresse.getHostName();
  String ip = adresse.getHostAddress();
  boolean  multicast = adresse.isMulticastAddress())
}
```

TCP: java.net.ServerSocket

- socket TCP coté serveur

Sans le code de traitement des exceptions pour rentrer dans le slide !

```
...
int port_d_ecoute = 1234;
ServerSocket socket_d_ecoute = new ServerSocket(port_d_ecoute);

while(true){
    Socket socket_de_connexion = socket_d_ecoute.accept();
    Traitement(socket_de_connexion);
}
...

void Traitement(Socket s){
    PrintStream ps = new PrintStream(s.getOutputStream());
    DataInputStream dis = new DataInputStream(s.getInputStream());

    ps.println(”hello");

    String line = dis.readLine(); //ou readUTF()
    System.out.println(line);
}
```

TCP: java.net.Socket

- socket TCP coté client.

Sans le code de traitement des exceptions pour rentrer dans le slide !

```
String serveur = "www.serveur.fr";
int port = 80;

Socket s = new Socket(serveur, port);
// ou Socket s = new Socket("10.1.26.73", 80);

PrintStream ps = new PrintStream(s.getOutputStream());
DataInputStream dis = new DataInputStream(s.getInputStream());

ps.println("GET /index.html");
String line;
while((line = dis.readLine()) != null)
   System.out.println(line);
```

Lecture / Ecriture

- Tout socket possède un flux de lecture et un flux d'écriture:
  - getInputStream() dans java.io.Socket
  - getOutputStream() dans java.io.Socket
- Il est recommandé d'utiliser des procédés faisant usages de buffers pour la lecture et l'écriture (optimisation)
- utiliser les classes suivantes pour y accéder:
  - java.io.BufferedReader
    - BufferedReader (Reader in) où le Reader est connecté au java.io.InputStream provenant de la socket
    - Utiliser la classe InputstreamReader pour faire « la conversion »: BufferedReader depuisClient = new BufferedReader(new InputStreamReader(client.getInputStream()));
  - java.io.BufferedWriter
    - BufferedWriter (Writer out) ou PrintWriter(Writer out) le Writer est connecté au java.io.OutputStream provenant de la socket
    - Utiliser la classe OutputStreamWriter pour faire « la conversion »:
    - PrintWriter versClient = new PrintWriter(new OutputStreamWriter(client.getOutputStream()),true);
- On peut aussi utiliser DataInputStream et DataOutputStream comme pour les fichiers

Lecture / Ecriture

- Lecture
  - BufferedReader.readLine()
  - DataInputStream.readUTF()
  - DataInputStream.readInt/Long/Double()…
- Ecriture
  - BufferedWriter.write(String msg)
  - BufferedWriter.flush()
  - DataOutputStream.writeUTF()
  - DataInputStream.writeInt/Long/Double()…

Socket TCP: résumé

- Ouverture d'une socket :
  - Serveur : java.net.ServerSocket
    - `ServerSocket ( int port ) throws IOException`
    - `Socket accept() throws IOException`
  - Client : java.net.Socket
    - `Socket ( String Host, int port ) throws UnknownHostException, IOException`
      où Host peut être un nom de machine "serveur" ou une Adresse IP "194.167.106.245"
- Fermeture des sockets:
  - `Socket.close ( ) throws IOException`
  - `ServerSocket.close ( ) throws IOException`
  - *il est recommandé de vider et fermer les buffers de lecture et d'écriture avant !*

UDP: java.net.DatagramSocket(1)

- implémente une socket UDP

```
// Client

Byte[] data = "un message".getBytes();

InetAddress addr = InetAddress.getByName("www.site.fr");

DatagramPacket packet = new DatagramPacket(data, data.length, addr,
    1234);

DatagramSocket ds = new DatagramSocket();

ds.send(packet);
ds.close();
```

UDP: java.net.DatagramSocket(2)

```
// Serveur

DatagramSocket ds = new DatagramSocket(1234);

while(true)
{
    DatagramPacket packet = new DatagramPacket(new
    byte[1024], 1024);
    ds.receive(packet);
    System.out.println("Message: " + packet.getData());
}
```

java.net.URL

```
URL url = new URL("http://www.site.fr/index.html");

DataInputStream dis = new
   DataInputStream(url.openStream());

String line;
while ((line = dis.readLine()) != null)
   System.out.println(line);
```