I/O
Entrées/Sorties

Problèmes classiques

- La représentation des fichiers doit être indépendante de la machine hôte
  - Par ex: le séparateur de fichier est / ou \
- Les données
  - “Binaires” ne doivent pas être transformées
  - “Caractères” doivent être lues/écrites avec le bon encodage

Abstraction !

En Java, quelle que soit la provenance des données (fichier, réseau, clavier), les données sont représentées par des flux

- Les flux binaires InputStream/OutputStream
- Les flux de caractères Reader/Writer

Il est possible de passer d'un flux binaire à un flux de caractères (et vice versa) en fournissant un encodage (Charset)

Ressources systèmes !

Les flux de données sont des ressources système allouées pour le processus
par ex: un descripteur de fichier

Contrairement aux objets “classiques” en Java, comme le système limite le nombre de ressources liées à un processus, on ne peut pas s'en remettre au GC pour libérer les ressources

- Le GC réclame la mémoire en grappe et non pas à chaque fois qu'il pourrait réclamer la mémoire

Ressources systèmes !

On fait la libération des ressources à la main

sur un InputStream/OutputStream ou un Reader/Writer il existe une méthode **close()** pour libérer la ressource

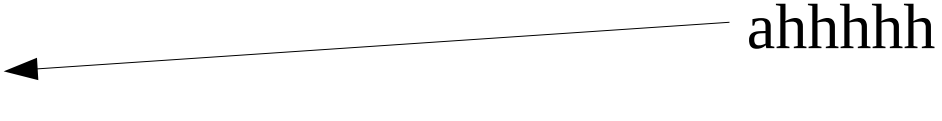
Une ressource système doit toujours être libérée

- donc il incombe au programmeur de coder correctement

Ressources systèmes et exceptions

Le code suivant ne gère pas correctement le close().

```
BufferedReader reader = ...
String line;
while ((line = reader.readLine()) != null) {
  processLine(line);
}
reader.close();
```



processLine() peut lever une exception !

Try/finally

Gestion correcte avec un try/finally:

```
BufferedReader reader = ...
try {
  String line;
  while ((line = reader.readLine()) != null) {
    processLine(line);
  }
} finally {
  reader.close();
}
```

Le try est **APRES** l'assignation

Dans tous les cas ce code est exécuté

Try-with-ressources

En fait on utilise une syntaxe moins verbeuse qui fait appel à close() automatiquement


```
try (BufferedReader reader = ...) {
  String line;
  while ((line = reader.readLine()) != null) {
    processLine(line);
  }
} // appel à close() automatique
```

La ressource doit être de type AutoCloseable

Try-with-ressources

Peut aussi gérer plusieurs ressources

L'appel au close() est dans l'ordre inverse des déclarations


```
try (BufferedReader reader = ...;
      BufferedWriter writer = ...) {
  String line;
  while ((line = reader.readLine()) != null) {
    processLine(writer, line);
  }
} // appel writer.close() puis appel reader.close()
```

La gestion des fichiers

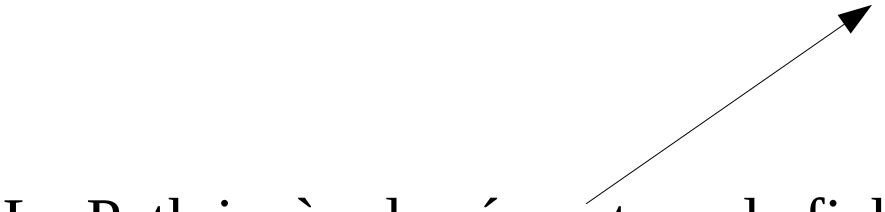
java.nio.file.Path

Un Path représente un chemin dans une arborescence

- Il gère les '/' et '\' tout seul

Un Path est créé à partir de la classe Paths

```
Path current = Paths.get(".");
Path myDirectory = Paths.get("my", "sub", "dir");
```



java.nio.file.Files

La classe Files contient un ensemble de méthodes qui permettent de créer/manipuler/détruire des fichiers en utilisant leur Path

- Création
  - createDirectory(), createFile(), createLink(), createSymbolicLink(), createTempDirectory(), createTempFile()
- Droits
  - getOwner(), getPosixFilePermission(), getFileAttributeView()
- Existence
  - isDirectory(), isRegularFile(), isHidden()

Entre le moment de l'appel et l'accès au fichier correspondant, le file system peut changer!

DirectoryStream

Un DirectoryStream est un itérateur qui permet de parcourir les Paths d'un répertoire

```
Path directory = ...
try(DirectoryStream<Path> stream =
            Files.newDirectoryStream(directory)) {
  for(Path path: stream) {
    System.out.println(path);
  }
} // call stream.close() implicitly
```

Parcourir récursivement des répertoires

- walkFileTree()

Fichier en lecture/écriture

Ouvrir un fichier pour lire des caractères

- Files.**newBufferedReader**(path, encoding)

Ouvrir un fichier pour écrire des caractères

- Files.**newBufferedWriter**(path, encoding)

Ouvrir un fichier pour lire des octets

- Files.**newInputStream**(path)

Ouvrir un fichier pour écrire des octets

- Files.**newOutputStream**(path)

Legacy

La classe java.io.File est l'ancienne version des java.nio.file.Path mais

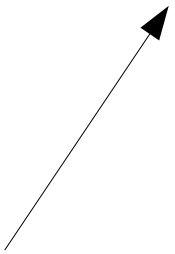
- Elle a de gros problème de perf (parcours de répertoire)
- Elle présuppose que l'encoding du système de fichier est le même entre les points de montage

Lire à partir d'une URL

java.net.URI & java.net.URL

La classe URI (comme son nom ne l'indique pas) permet de décoder une URL

La classe URL permet de décoder une URL et d'ouvrir une connexion distante pour récupérer une ressource via le protocole HTTP



La classe URL est une très grosse classe !

URLConnection

A partir d'une URL, il est possible d'ouvrir une connection et de récupérer

- Une java.util.Map des headers de la réponse
- Un InputStream contenant les données de la réponse

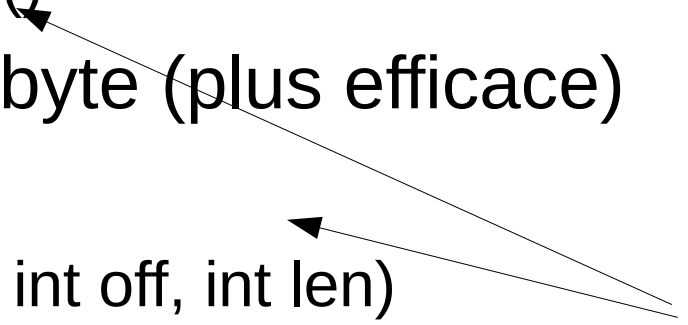
```
URL url = new URL("http://www.playboy.com");
URLConnection con = url.openConnection();
Map<String, List<String>> headers =
        con.getHeaderFields();
try(InputStream input = con.getInputStream()) {
  ...
}
```

InputStream/OutputStream

java.io.InputStream

Flux de **byte** en entrée

- Lit un byte et renvoie ce byte ou -1 si c'est la fin du flux
  - abstract int **read**()
- Lit un tableau de byte (plus efficace)
  - int **read**(byte[] b)
  - int **read**(byte[] b, int off, int len)
- Saute un nombre de bytes
  - long **skip**(long n)
- Ferme le flux
  - void **close**()



Attention aux performances !

On ne lit pas octet par octet (ahhh!)

```
public static void copy(Inputstream in,OutputStream out)
    throws IOException {
  int b;
  while ((b = in.read()) != -1) {
    ...
  }
}
```

Pas de try/catch au milieu
on utilise throws !

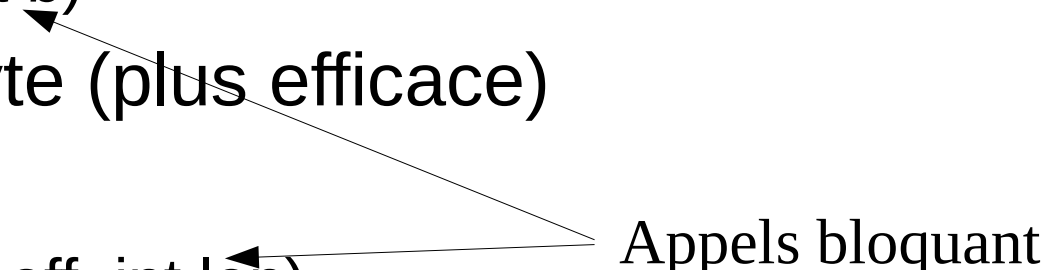
On lit avec un tableau !

```
public static void copy(Inputstream in,OutputStream out)
    throws IOException {
  byte[] buffer = new byte[8192];
  int size;
  while ((size = in.read(buffer)) != -1) {
    ...
  }
}
```

java.io.OutputStream

Flux de byte en sortie

- Ecrit un byte, en fait un int pour qu'il marche avec le read
  - abstract void **write**(int b)
- Ecrit un tableau de byte (plus efficace)
  - void **write**(byte[] b)
  - void **write**(byte[] b, int off, int len)
- Demande d'écrire ce qu'il y a dans le buffer (s'il y en a un !)
  - void **flush**()
- Ferme le flux
  - void **close**()



write(byte[], offset, count)

On utilise rarement l'appel write(byte[]) car souvent le byte[] a été rempli par read(byte[]) or read(byte[]) ne remplit pas nécessairement le tableau

Exemple de copie:

```
public static void copy(Inputstream in,OutputStream
out)
    throws IOException {
  byte[] buffer = new byte[8192];
  int size;
  while ((size = in.read(buffer)) != -1) {
    output.write(buffer, 0, size);
  }
}
```

On écrit que ce que l'on a lu !

Clavier/console

Les constantes :

- Entrée standard
  System.in est un InputStream (pas bufferisé !)
- Sortie standard
  System.out est un PrintStream
  (un OutputStream avec print()/println() en plus)
- Sortie d'erreur standard
  System.err est un PrintStream

```
public static void main(String[] args) throws IOException
   copy(System.in,System.out);
}
```

L'encodage

java.nio.charset.Charset

La classe Charset représente une conversion entre des octets et des caractères et vice-versa

Convertir en mémoire un tableau de byte[] en String suivant un certain codage

- **String**(byte[] bytes, **Charset** charset)
- **string**.getBytes(**Charset** charset)

Créer un Writer/Reader à partir d'un InputStream/OutputStream

- new **InputStreamReader**(InputStream input, Charset charset)
- new **OutputStreamWriter**(OutputStream output, Charset charset)

java.nio.charset.StandartCharsets

La classe StandartCharsets contient les constantes

US-ASCII Seven-bit ASCII, a.k.a. ISO646-US

ISO-8859-1 ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1

UTF-8 Eight-bit UCS Transformation Format

UTF-16BE Sixteen-bit UCS Transformation Format, big-endian byte order

UTF-16LE Sixteen-bit UCS Transformation Format, little-endian byte order

UTF-16 Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

Reader/Writer

java.io.Reader

Flux de **char** en entrée (méthode bloquante)

- Lit un char et renvoie celui-ci ou -1 si c'est la fin du flux
  - abstract int **read**()
- Lit un tableau de char (plus efficace)
  - int **read**(char[] b)
  - int **read**(char[] b, int off, int len)
- Saute un nombre de caractères
  - long **skip**(long n)
- Ferme le flux
  - void **close**()

java.io.BufferedReader

On utilise souvent un BufferedReader plutôt qu'un Reader par un BufferedReader possède une méthode readLine() qui permet de lire ligne à ligne

String BufferedReader.readLine()

renvoie une ligne sans le caractère de fin de ligne

- \n sous Unix, \r\n sous Windows

ou null si il n'y a plus de ligne à lire

java.io.Writer

Flux de caractère en sortie

- Ecrit un caractère, un int pour qu'il marche avec le read
  - abstract void write(int c)
- Ecrit un tableau de caractères (plus efficace)
  - void **write**(char[] b)
  - void **write**(char[] b, int off, int len)
- Demande d'écrire ce qu'il y a dans le buffer
  - void **flush**()
- Ferme le flux
  - void **close**()

Codes idiomatiques

La lecture ligne à ligne à partir d'un fichier

```
Path path = ...
try (BufferedReader reader =
                    Files.newBufferedReader(path)) {
  String line;
  while ((line = reader.readLine()) != null) {
    ...
  }
}
```

La gestion des IOException est faite par le code appelant !