# Programmation réseau en Java : les flots

Michel Chilowicz


Master 2 TTT
Université Paris-Est Marne-la-Vallée

Version du 14/02/2013

# Plan

# A propos des flots

Objets représentant des séquences d'octets ou de caractères
Fournis par le paquetage `java.io` en bloquant

- Séquence d'octets
  - accessible en lecture : `java.io.InputStream`
  - accessible en écriture : `java.io.OutputStream`
- Séquence de caractères
  - accessible en lecture : `java.io.Reader`
  - accessible en écriture : `java.io.Writer`

## Remarques

Il s'agit de classes abstraites Java ne pouvant être instanciées directement.
On utilise en pratique des classes dérivées pour la manipulation de fichiers ou de flots réseau.
La plupart des méthodes des flots peuvent lever une `java.io.IOException` qui doit être gérée.

# Deux paradigmes : flots bloquants et non-bloquants

- Flots bloquants (`java.io`) : chaque demande d'E/S est bloquante
  Nécessite l'usage de threads (fils d'exécution) pour le traitement de plusieurs flots
  - lecture : retourne après avoir lu au moins un octet
  - écriture : retourne après avoir écrit toutes les données
- Flots non-bloquants (`java.nio`) : chaque demande d'E/S est non-bloquante
  Nécessite de demander au système quels sont les flots sur lesquels les données sont disponibles (mécanisme de sélecteur)

# `InputStream` : flot d'octets en lecture

- Les méthodes de lecture
  - `int read()` : pour lire le prochain octet (sous forme d'int), retourne -1 si fin de flot
  - `int read(byte[] data, int decalage, int longueur)` : lit les prochains octets et les place dans *data[decalage:decalage+longueur]*, retourne le nombre d'octets effectivement lus (entre 0 et *longueur*), -1 si la fin du flot est atteinte
  - `int read(byte[] data)` : idem avec $decalage = 0$ et $longueur = data.length$
- `long skip(long k)` : saut d'au plus $k$ octets dans le flux, retourne le nombre d'octets sautés

# Retour en arrière

- Flots conçus pour être lu séquentiellement : pas de retour en arrière possible
- SAUF si le flot supporte les marques (`is.markSupported()`) :
  1. Pose d'une marque avec indication de $k$ octets à mémoriser : `is.mark(k)`
  2. Lecture (ou saut) jusqu'à $k$ octets
  3. Possibilité de revenir en arrière sur la marque avec `is.reset()`

# Flot en lecture depuis un fichier

## java.io.FileInputStream

Constructeur : `FileInputStream(String cheminDuFichier)`

## java.io.BufferedInputStream

Encapsule un `InputStream` et ajoute un niveau de bufferisation
Supporte l'usage des marques pour un retour arrière
Constructeur : `BufferedInputStream(InputStream encapsule [, int tailleDuBuffer])`

# Fermeture des flots

Pour libérer les ressources système associées, un flux doit être fermé :

- avec la méthode `void close()`
- avec la syntaxe *try-with-resources* disponible depuis Java 1.7

## Exemple : détermination du nombre d'octets d'un fichier

```
public int getSize(String path) throws IOException
{
        byte[] buffer = new byte[BUFFER_LEN];
        try (InputStream is = new FileInputStream(path))
        {
                int size = 0;
                for (int i = is.read(buffer); i >= 0; i = is.read(buffer))
                        size += i;
                return size;
        }
}
```

# OutputStream : flot d'octets en écriture

- Les méthodes d'écriture
  - `void write(int octet)` : écrit un seul octet (obtenu par `octet & 0xff`)
  - `void write(byte[] data, int decalage, int longueur` : écrit dans le flux le contenu de *data[decalage:decalage+longueur]*
  - `void write(byte[] data)` : idem que `write(data, 0, data.length)`
- `void flush()` : écrit effectivement les éventuelles données bufferisées

# Flot en écriture vers un fichier

## java.io.FileOutputStream

Constructeur : `FileOutputStream(String cheminDuFichier, boolean ajout)`
Si `! ajout`, un éventuel fichier existant est écrasé.
Si `ajout`, les données sont ajoutées en fin de fichier si le fichier existe déjà.

## java.io.BufferedOutputStream

Encapsule un `OutputStream` et ajoute un buffer en écriture
Permet d'économiser les appels système en cas d'écriture de petites données
Constructeur : `BufferedOutputStream(OutputStream encapsule, int tailleDuBuffer)`
`flush()` doit être appelé pour forcer le vidage du buffer dans le flot encapsulé

# Exemple : une méthode de copie de fichier

```
public static void transfer(InputStream is, OutputStream os) throws IOException
{
        for (int i = is.read(data); i >= 0; i = is.read(data))
                os.write(data, 0, i);
}

public static void copieFichier(String source, String destination) throws IOException
{
        byte[] data = new byte[BUFF_LEN];
        try (
                InputStream is =
                        new BufferedInputStream(new FileInputStream(source));
                OutputStream os =
                        new BufferedOutputStream(new FileOutputStream(destination)) )
        {
                transfer(is, os);
        }
}
```

# Flots vers/depuis un tableau d'octets

- Depuis un tableau d'octets : `new ByteArrayInputStream(byte[] data, int decalage, int longueur)`
- Vers un tableau d'octets : `new ByteArrayOutputStream()` (`toByteArray()` pour récupérer le tableau d'octets)

# Exemple : inversion de l'ordre des octets d'un InputStream

```
public static InputStream invertOrder(InputStream is) throws IOException
{
        byte[] data = new byte[BUFF_LEN];
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        transfer(is, baos);
        byte[] content = baos.toByteArray();
        // Reverse the array
        for (int k = 0; k < content.length/2; k++)
        {
                byte tmp = content[k];
                content[k] = content[content.length - 1 - k]
                content[content.length - 1 - k] = tmp;
        }
        return new ByteArrayInputStream(content);
}
```

# Flots de données binaires

`DataInputStream` et `DataOutputStream` :

- encapsulent respectivement un `InputStream` et un `OutputStream`
- permettent d'y lire/écrire des types primitifs (boolean, byte, char, int, long) avec les méthodes `X readX()` et `void writeX(X x)`
- les données sont exprimées en grand-boutiste (octet de poids fort en tête)

# A propos de `Reader` et `Writer`

- L'utilisation de `Reader` et `Writer` est analogue à celle de `InputStream` et `OutputStream`
- Seule différence : emploi de char à la place de byte
  $\longrightarrow$ char est le type primitif Java représentant un caractère UTF-16 (certains caractères Unicode rares peuvent être représentés sur 2 chars)
- `Writer` offre la possibilité d'écrire directement des `String` avec `void write(String str)`

# Conversion de flot d'octets en flot de caractères

## Convertisseurs

- D'un `InputStream` à un `Reader` : `InputStreamReader`
- D'un `OutputStream` à un `Writer` : `OutputStreamWriter`

## Arguments des constructeurs

1. Flot d'octets à convertir
2. Jeu de caractères (UTF-8, ASCII, iso-8859-15...) à utiliser pour l'encodage/décodage

# Versions bufferisées de `Reader` et `Writer`

## `java.io.BufferedReader`

- Constructeur : `BufferedReader(Reader in [, int tailleDuBuffer])`
- Méthode `String readLine()` pour lire une ligne entière (sans caractère de fin de ligne), retourne null en fin de flot

## `java.io.BufferedWriter`

- Constructeur : `BufferedWriter(Writer out [, int tailleDuBuffer])`

# Exemple : longueur moyenne d'une ligne dans un fichier texte ?

```
public static float getLineAverageLength(String path, String charset) throws IOException
{
        try (
                BufferedReader r = new BufferedReader(
                        new InputStreamReader(new FileInputStream(chemin), charset)) )
        {
                int count = 0, len = 0;
                for (String line = r.readLine(); line != null; line = r.readLine())
                {
                        count++;
                        len += line.length();
                }
                return (float)len / (float)count;
        }
}
```