

# Programmation et Algorithmique II

## Ch.8 – Fichiers et flux d'E/S

**Bruno Quoitin**  
([bruno.quoitin@umons.ac.be](mailto:bruno.quoitin@umons.ac.be))

# Table des Matières

## 1. Introduction

## 2. La classe `File`

## 3. Flux d'entrées/sorties

### 1. Flux d'octets

### 2. Flux d'octets fichiers

### 3. Performances et buffer

### 4. Flux de caractères

## 4. Accès aléatoire à des fichiers

## 5. Flux d'objets

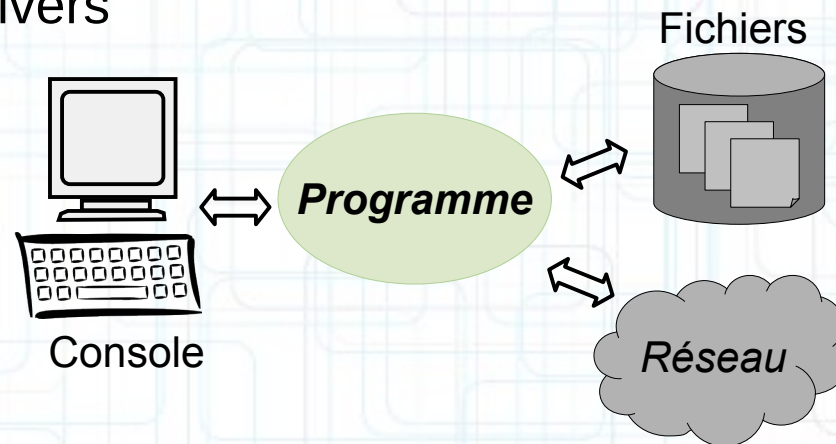


# Fichiers

- **Introduction**

- Les **sources** et **destinations** de données d'un programme peuvent être de types très divers

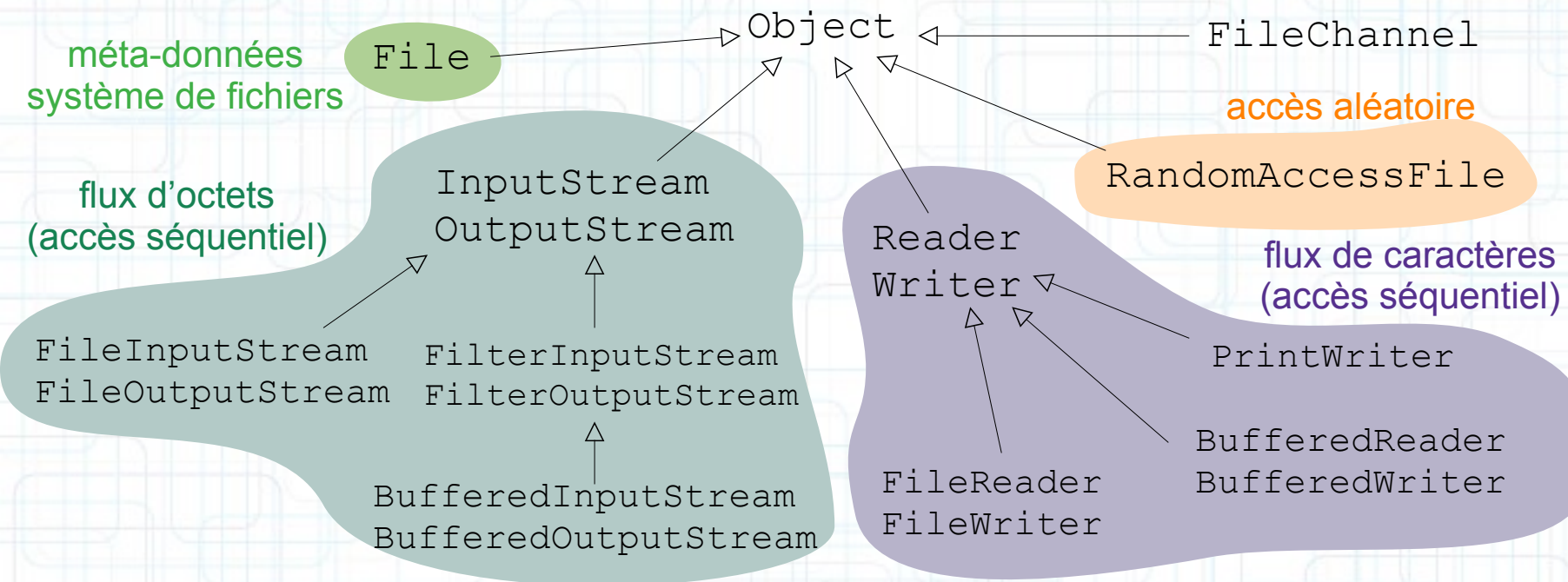
- fichiers
- console
- autres programmes
- connexions réseaux
- ...



- L'**accès** à ces sources et destinations de données peut être réalisé de différentes façons
  - accès séquentiel, accès aléatoire, accès avec tampon mémoire (*buffer*), ...
  - accès binaire, caractère, par ligne, par mot, par enregistrement, ...

# Fichiers

- **I just want to read a file !!...**
  - La bibliothèque Java contient de nombreuses classes qui permettent d'accéder à ces données de différentes manières. Il est difficile de s'y retrouver de prime abord ...



- L'objectif de ce chapitre est de comprendre quelles sont les classes appropriées à chaque cas d'utilisation.



# Fichiers

- Pour les étudiants pressés...

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class SimplestFileAccess {

    public static void main(String [] args) {
        try (BufferedReader r =
            new BufferedReader(new FileReader("toto.txt")))
        {
            String s;
            while ((s= r.readLine()) != null)
                System.out.println(s);
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Afin d'améliorer les performance, permet la lecture au travers d'un tampon mémoire.

Ouvre un fichier et le considère comme un flux de caractères en lecture.

Ce fichier sera fermé automatiquement par la directive **try-with-resource**.

Lit une ligne à partir du fichier et la retourne sous forme d'une chaîne de caractères.

Si la fin du fichier est atteinte, retourne **null**.

# Table des Matières

1. Introduction
- 2. La classe File**
- 3. Flux d'entrées/sorties**
  1. Flux d'octets
  2. Flux d'octets fichiers
  3. Performances et buffer
  4. Flux de caractères
- 4. Accès aléatoire à des fichiers**
- 5. Flux d'objets**



# La classe `File`

- **Introduction**

- La classe `java.io.File` permet d'obtenir et de manipuler l'information connue par le **système de fichiers** à propos d'un fichier ou d'un répertoire particulier. Il s'agit de *méta-données*.
  - Chemin (*path*) : localisation d'un fichier ou d'un répertoire
  - Nom
  - Longueur du fichier
  - Type : fichier, répertoire, fichier caché, ...
  - Droits d'accès (*access rights*) : lecture, écriture, exécution
  - Création d'un répertoire
  - Suppression d'un fichier ou répertoire
  - Renommer un fichier ou répertoire
  - Lister les fichiers d'un répertoire
  - Obtenir l'espace disponible

# La classe `File`

- **Constructeur**

- Pour obtenir et manipuler les méta-données du système de fichiers, il faut commencer par obtenir ou créer une instance de la classe `File`.
- Cela peut être réalisé en spécifiant au constructeur de `File` le chemin du fichier ou du répertoire à considérer.

```
File cheminAbsolu= new File("/home/toto/data/liste.txt");  
File cheminRelatif= new File("data/liste.txt");  
File cheminRepertoireAbsolu= new File("/home/toto");
```

- Il est également possible d'obtenir une instance de `File` à partir d'une instance « parent » et d'une instance « enfant »

```
File cheminRelatif= new File("data/liste.txt");  
File cheminRepertoire= new File("/home/toto");  
File chemin= new File(cheminRepertoire, cheminRelatif);
```



# La classe `File`

- **Constructeur**

- **Attention !** la façon dont un chemin est spécifié dépend du système d'exploitation.

	UNIX	Windows
Séparateur	/	\
Séparateur de chemins	:	;

- La classe `File` fournit ces séparateurs sous la forme de constantes (**`public static final`**)
  - `separatorChar`
  - `pathSeparatorChar`
- Exemple : un chemin absolu aura la forme
  - `/home/toto/liste.txt` (sous UNIX)
  - `C:\home\toto\liste.txt` (sous Windows)

# La classe `File`

- **Attributs d'un fichier**

- La classe `File` permet de déterminer et changer certains attributs d'un fichier
- Type
  - méthodes `isDirectory()`, `isFile()`, `isHidden()`
- Longueur (en octets)
  - méthode `length()`
- Droits d'accès<sup>(1)</sup>
  - **consultation** : méthodes `canRead()`, `canWrite()`, `canExecute()`.
  - **modification** : `setReadable(boolean)`, `setReadOnly()`, `setWritable(boolean)`, `setExecutable(boolean)`

<sup>(1)</sup> Depuis java 7, les droits d'accès UNIX et Windows sont supportés au travers d'API spécifiques (voir `PosixFileAttributeView` et `AclFileAttributeView`)



# La classe `File`

- **Opérations utiles**

- La classe `File` fournit également un arsenal de méthodes permettant de réaliser des manipulations utiles du système de fichiers
- Renommer un fichier
  - méthode `renameTo (File)`
- Supprimer un fichier ou répertoire
  - méthode `delete ()` → supprime le fichier / répertoire désigné par cette instance de `File`
- Créer un répertoire
  - méthode `mkdir ()` → crée le répertoire désigné par cette instance de `File`
- Consultez la documentation pour obtenir la liste complète des opérations supportées !...

# La classe File

- Lister les fichiers d'un répertoire
  - L'exemple ci-dessous illustre comment lister les fichiers du répertoire courant en exploitant la méthode `listFiles` de la classe `File`.

```
import java.io.File;

public class ListeFichiers {

    public static void main(String [] args) {
        File monChemin= new File(".");
        File [] fichiers= monChemin.listFiles();
        for (int i= 0; i < fichiers.length; i++)
            System.out.println(fichiers[i].getName());
    }
}
```

Le chemin « . » désigne le répertoire courant.

```
bash-3.2$ java ListeFichiers
liste-courses.txt
todo.txt
examen-algo2-juillet-2020.pdf
```



# La classe File

- **Lister les fichiers d'un répertoire**
  - Question : comment lister tous les fichiers d'un répertoire, y compris ceux situés dans les sous-répertoires ?
  - Contrainte : les fichiers d'un sous-répertoire doivent être affichés **décalés vers la droite de 2 caractères** par rapport au nom du répertoire parent (cf. illustration ci-dessous).

```
/home/bquoitin
↔cours-prog-algo-2
↔08-fichiers et flux d'E/S.pdf
  examen-algo2-juillet-2020.pdf
  (...)
  cours-fonct-ordis
    4-processeur-monocycle.pdf
    examen-fonct-juillet-2020.pdf
    (...)
```

# La classe File

- Lister les fichiers d'un répertoire

- La méthode illustrée ci-dessous liste les fichiers d'un répertoire donné. Pour chaque répertoire trouvé dans cette liste, elle s'appelle **récurivement** pour explorer ce répertoire.

```
import java.io.File;

public class ListeFichiersRecursive {

    public static void explorer(File base, String prefix) {
        File [] fichiers= base.listFiles();
        for (int i= 0; i < fichiers.length; i++) {
            System.out.println(prefix + fichiers[i].getName());
            if (fichiers[i].isDirectory())
                explorer(fichiers[i], prefix + " ");
        }
    }

    public static void main(String [] args) {
        explorer(new File("."), "");
    }
}
```

permet d'indenter  
le texte écrit à la  
console en  
fonction de la  
profondeur de la  
récursion.



# Table des Matières

1. Introduction
2. La classe `File`
- 3. Flux d'entrées/sorties**
  1. Flux d'octets
  2. Flux d'octets fichiers
  3. Performances et buffer
  4. Flux de caractères
4. Accès aléatoire à des fichiers
5. Flux d'objets

# Flux d'Entrées/Sorties

- **Introduction**

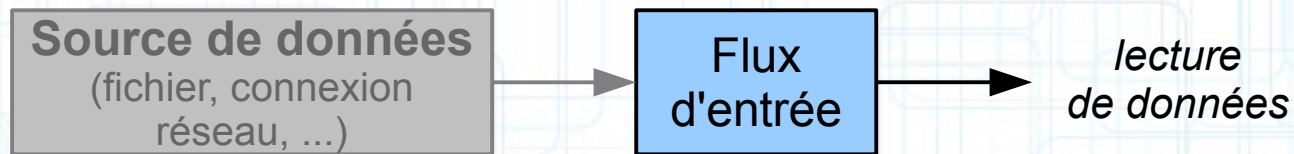
- De nombreux langages de programmation utilisent la notion de **flux** (*stream*) pour abstraire les sources et destinations de données.
- Un **flux** est une abstraction permettant l'interaction avec une entité capable de produire ou de consommer une séquence de données.
  - Un flux possède une direction : il peut être **lu** ou **écrit**
  - Un flux a une granularité : la séquence de données est une séquence d'octets, une séquence de caractères, ... voire même une séquence d'objets (instances)
  - L'accès au flux est indépendant du type de la source/destination : lire ou écrire des données se fait de la même façon qu'il s'agisse d'un flux associé à un fichier, à la console ou à une connexion réseau !



# Flux d'Entrées/Sorties

- **Introduction**

- Les flux sont distingués selon la **direction** des données.
- **Flux d'entrée** (ou *input stream*)
  - Un objet à partir duquel il est possible de lire une séquence de données



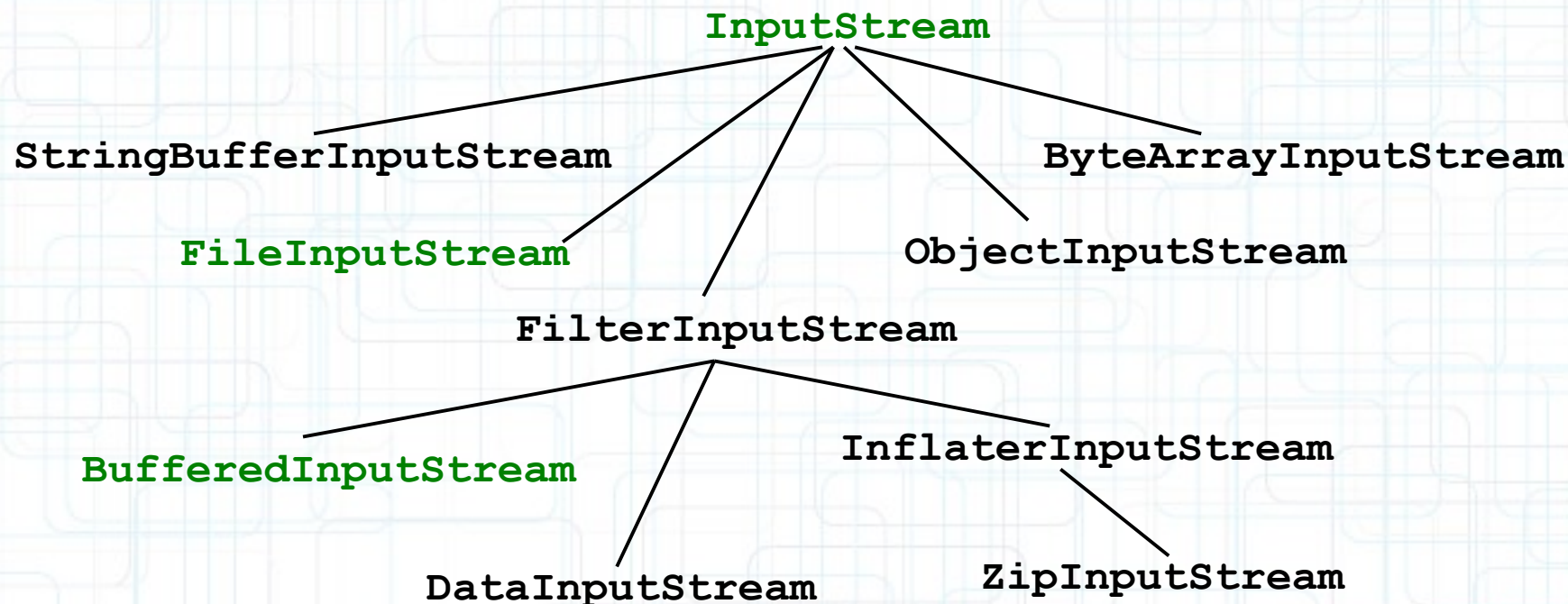
- **Flux de sortie** (ou *output stream*)
  - Un objet vers lequel il est possible d'écrire une séquence de données



# Flux d'Entrées/Sorties

- Zoo des flux

- La bibliothèque Java fournit des **10<sup>aines</sup> de classes de flux différentes**, chacune adaptée à un usage particulier.
- Exemple pour les flux entrants de type `InputStream`





# Flux d'Entrées/Sorties

- **Introduction**

- **Flux d'octets**

- Classes descendant de `InputStream` et `OutputStream`
    - Dédiées à la manipulation des flux d'octets.

- **Flux de caractères**

- Classes descendant de `Reader` et `Writer`
    - Dédiées à la manipulation des flux de caractères.
    - Gèrent l'encodage de caractères en séquences d'octets.

# Table des Matières

1. Introduction
2. La classe `File`
3. Flux d'entrées/sorties
  1. **Flux d'octets**
  2. Flux d'octets fichiers
  3. Performances et buffer
  4. Flux de caractères
4. Accès aléatoire à des fichiers
5. Flux d'objets



# Flux d'octets

- **Introduction**

- Nous allons commencer par nous intéresser aux classes dédiées aux flux d'octets
  - `java.io.InputStream`
  - `java.io.OutputStream`
- Ensuite, nous discuterons des sous-classes qui en sont dérivées.
  - Un exemple de flux d'entrée basé sur `InputStream` est l'instance fournie dans `System.in`

# Flux d'octets

- Introduction

- Les classes `InputStream` et `OutputStream` sont des classes *abstraites*. Elles fournissent chacune une seule méthode importante : respectivement `read` et `write`.

```
public abstract class InputStream {  
    public int available() { ... }  
    public void close() { ... }  
    public abstract int read();  
    public int read(byte [] b) { ... }  
    public int read(byte [] b, int off, int len) { ... }  
    public long skip(long n) { ... }  
}
```

```
public abstract class OutputStream {  
    public void close() { ... }  
    public int flush() { ... }  
    public abstract void write(int b);  
    public void write(byte [] b) { ... }  
    public void write(byte [] b, int off, int len) { ... }  
}
```

Note : d'autres méthodes sont supportées. Voir documentation...



# Flux d'octets

- **Classe InputStream**

- La méthode `read()` est chargée de **lire un octet à partir du flux** et de le retourner comme valeur de retour.

```
public abstract int read() throws IOException;
```

- Seuls les 8 bits de poids faible de la valeur retournée sont à utiliser.
  - Si la fin du flux est atteinte, la valeur -1 est retournée. En cas d'erreur, une exception est générée.
  - Les appels à `read` sont bloquants. Cela signifie que s'il n'y a pas au moins un octet prêt à être lu, le programme sera suspendu.
- La méthode `available()` permet de déterminer combien d'octets sont directement disponibles pour une lecture non bloquante dans le flux.

```
public int available() throws IOException;
```

- Attention : l'implémentation de cette méthode dans la classe abstraite `InputStream` retourne toujours 0.

# Flux d'octets

- **Classe InputStream**

- Exemple : lecture des octets disponibles sur le flux d'entrée.
  - Les caractères entrés au clavier sont disponibles après pression de la touche « Enter ».
  - Pour terminer le flux, entrer « Ctrl-D » (sous Windows « Ctrl-Z »)

```
import java.io.InputStream;
import java.io.IOException;

public class TestInputStream {

    public static void main(String [] args)
        throws IOException {
        InputStream in= System.in;
        int b;
        do {
            b= in.read();
            System.out.println(b);
        } while (b >= 0);
    }
}
```



# Flux d'octets

- **Classe InputStream**

- Exemple : lecture des octets disponibles sur le flux d'entrée.

```
bash-3.2$ java TestInputStream
Hello <Enter>
72
101
108
108
111
10
World <Enter>
87
111
114
108
100
10
<Ctrl-D>
-1
bash-3.2$
```

Les octets lus à partir du flux correspondent aux codes des caractères entrés à la console.

Par exemple  
72 est le code de 'H'  
10 marque la fin de ligne

Note : sous Windows, la fin de ligne est marquée par deux caractères : 13 (\r) retour en début de ligne et 10 (\n) nouvelle ligne

Note : Dans l'exemple ci-dessus, chaque caractère entré correspond à un code sur un octet. Il s'agit d'un cas particulier. Que se passe-t-il si un caractère spécial tel que € est entré ?

# Flux d'octets

- **Classe OutputStream**

- La méthode `write()` permet l'**écriture d'un octet sur le flux**.

```
public abstract void write(int b) throws IOException;
```

- Bien que la méthode prenne un argument de type `int`, seuls les 8 bits de poids faible de celui-ci sont pris en compte.
      - Les 24 bits de poids fort sont ignorés.
    - Les appels à `write` sont **bloquants**.
      - Le programme peut être suspendu s'il n'est pas possible d'écrire immédiatement un octet sur le flux
      - Il n'est pas possible de déterminer a priori si une écriture sur un flux sortant sera bloquante ou non.



# Flux d'octets

- **Classes {In|Out}putStream**

- Exemple : transformation d'un flux d'octets lus sur l'entrée standard (System.in), écriture des octets transformés sur la sortie standard (System.out).

```
import java.io.*;

public class TranslateStream {

    public static void main(String [] args)
        throws IOException
    {
        InputStream in= System.in;
        int b;
        do {
            b= in.read();
            if ((b >= 'a') && (b <= 'z'))
                b+= ('A' - 'a');
            System.out.write(b);
        } while (b >= 0);
    }
}
```

Question subsidiaire : quelle transformation est effectuée ?

Note : la transformation est implémentée en faisant l'hypothèse que les caractères sont représentés sur un octet.

# Flux d'octets

- **Classes {In|Out}putStream**

- Exemple : lecture des octets disponibles sur le flux d'entrée.

```
bash-3.2$ cat texte.txt  
Ceci n'est pas  
un petit texte.
```

Signé: R. Magritte

```
bash-3.2$ cat texte.txt | java TranslateStream  
CECI N'EST PAS  
UN PETIT TEXTE.
```

SIGNÉ: R. MAGRITTE

```
bash-3.2$
```

L'utilitaire `cat` permet d'écrire le contenu du fichier passé en paramètre sur la sortie standard.

Cette commande exécute 2 programmes (`cat` et notre classe `java`) de sorte que la sortie standard de `cat` est reliée à l'entrée standard du programme `java`.



# Flux d'octets

- **Classe abstraite ou interface ?**
  - Les classes `InputStream` et `OutputStream` sont fournies sous la forme de classes abstraites et non sous la forme d'interfaces
  - Raisons
    - Une seule méthode abstraite est déclarée
      - `read` pour `InputStream`
      - `write` pour `OutputStream`
    - D'autres méthodes (concrètes) se basant sur `read` ou `write` sont également fournies par la classe. Celles-ci permettent notamment d'effectuer des écritures / lectures de multiples octets.

# Table des Matières

1. Introduction
2. La classe `File`
3. Flux d'entrées/sorties
  1. Flux d'octets
  2. **Flux d'octets fichiers**
  3. Performances et buffer
  4. Flux de caractères
4. Accès aléatoire à des fichiers
5. Flux d'objets



# Flux d'E/S fichiers

- **Classes `File{In|Out}putStream`**

- Les classes **`FileInputStream`** et **`FileOutputStream`** permettent respectivement de lire et d'écrire des fichiers séquentiellement en utilisant les flux.
- Les classes `FileInputStream` et `FileOutputStream` sont des sous-classes de `InputStream` et `OutputStream` respectivement.
- Plusieurs constructeurs sont disponibles pour chacune des classes (voir documentation).

```
public FileInputStream(String name)
public FileInputStream(File file)
```

```
public FileOutputStream(String name)
public FileOutputStream(String name, boolean append)
public FileOutputStream(File file)
public FileOutputStream(File file, boolean append)
```

Chaque constructeur est susceptible de générer une exception `FileNotFoundException`.

Les versions avec 'append' permettent d'écrire à la fin d'un fichier déjà existant.

# Flux d'E/S fichiers

- **Classe FileInputStream**

- Exemple : lecture et comptage de tous les octets d'un fichier.

```
try {  
    FileInputStream fis= new FileInputStream("/tmp/data.bin");  
    long count= 0;  
    while (fis.read() >= 0)  
        count++;  
    fis.close();  
} catch (IOException e) {  
    System.err.println("Erreur de lecture" + e.getMessage());  
}
```



# Flux d'E/S fichiers

- **Fermeture du flux**

- Une fois qu'un flux n'est plus utilisé (en lecture comme en écriture), il est recommandé d'appeler la méthode `close`. Cette méthode va permettre la libération des ressources système liées au flux.

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

- Pourquoi ?



- Il n'est **pas garanti** que les octets écrits dans un fichier au travers d'un flux de sortie **soient effectivement écrits** dans le fichier tant que le flux n'est pas fermé.
- Le système d'exploitation n'autorise généralement pas une application à ouvrir simultanément plus d'une 10<sup>aine</sup> de fichiers. Cette limite dépend du système d'exploitation. Si l'application omet de fermer les fichiers qu'elle n'utilise plus, elle pourrait être empêchée d'en ouvrir de nouveaux !

# Flux d'E/S fichiers

- Fermeture du flux en cas d'exception

- Problème de l'exemple précédent : si une exception est générée dans le bloc **try**, la méthode `close` ne sera pas appelée et le fichier ne sera donc pas fermé.

```
try {  
    FileInputStream fis= new FileInputStream("/tmp/data.bin");  
    long count= 0;  
    while (fis.read() >= 0) {  
        count++;  
        fis.close();  
    }  
} catch (IOException e) {  
    System.err.println("Erreur de lecture" + e.getMessage());  
}
```

En cas d'exception  
la méthode `close` n'est  
pas appelée....



# Flux d'E/S fichiers

- Fermeture du flux en cas d'exception

- Solution : fermer le fichier dans le **try** et dans le **catch**.

```
FileInputStream fis;  
try {  
    fis= new FileInputStream("/tmp/data.bin");  
    long count= 0;  
    while (fis.read() >= 0)  
        count++;  
    fis.close();  
} catch (IOException e) {  
    System.err.println("Erreur de lecture" + e.getMessage());  
    fis.close();  
}
```

- Problème : **duplication de code !**

# Flux d'E/S fichiers

- **Clause `try... finally...`**

- La clause `try finally` permet de s'assurer qu'une opération est effectuée même si une exception est déclenchée.
- Cette clause est particulièrement intéressante lorsque l'on travaille avec des ressources qui doivent être libérées après utilisation (comme des fichiers) afin de s'assurer que les opérations de libération seront bien exécutées même en cas d'exception.
- Syntaxe

```
try {  
    ... code qui peut déclencher une exception ...  
} finally {  
    ... code toujours exécuté ...  
}
```



# Flux d'E/S fichiers

- **Clause try... finally...**

- Exemple

```
try {  
    FileInputStream fis= new FileInputStream("/tmp/data.bin");  
    try {  
        long count= 0;  
        while (fis.read() >= 0)  
            count++;  
    } finally {  
        fis.close();  
    }  
} catch (IOException e) {  
    System.err.println("Erreur de lecture" + e.getMessage());  
}
```

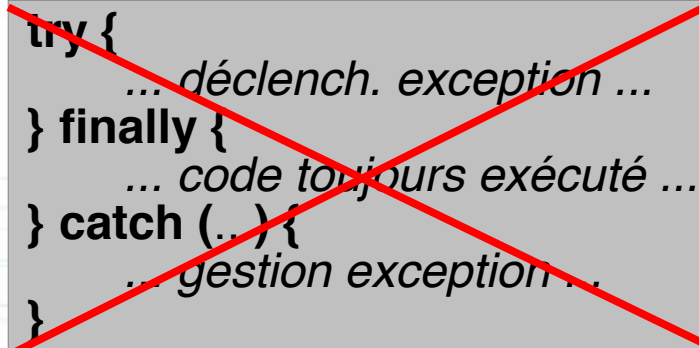
Les opérations de libération de ressources  
(fermeture de fichier) sont écrites une seule fois  
dans le programme.

# Flux d'E/S fichiers

- **Clause `try... catch... finally...`**

- Les clauses `try/catch` et `try/finally` peuvent être fusionnées en une clause `try/catch/finally`.
- La syntaxe est la suivante

```
try {  
    ... déclench. exception ...  
} catch (...) {  
    ... gestion exception ...  
} finally {  
    ... code toujours exécuté ...  
}
```



```
try {  
    ... déclench. exception ...  
} finally {  
    ... code toujours exécuté ...  
} catch (...) {  
    ... gestion exception ...  
}
```

Recommandation : garder les deux clauses séparées pour des raisons de lisibilité et de correction.

Avec des clauses séparées, une erreur dans `finally` pourra être capturée par `catch`.



# Flux d'E/S fichiers

- **Clause try... finally... et return**

- **Attention !** Le fonctionnement de la clause **finally** peut prêter à confusion lorsque le bloc associé à **finally** contient un **return**.

- Exemple

```
public static int f(int n)
{
    try {
        int r= n*n;
        return r;
    } finally {
        if (n == 2) return 0;
    }
}
```

Supposons que n vaut 2

retourne 4

mais **finally** toujours exécuté  
→ retourne 0 !!!

# Flux d'E/S fichiers

- **Clause *try-with-resources***

- Depuis la version 7 de java, une nouvelle forme de directive **try** est introduite, adaptée à la gestion d'erreurs en présence de ressources qui nécessitent une libération après usage.
- La nouvelle directive, nommée **try-with-resources**, garantit la fermeture de ces ressources.
- Pour qu'une ressource puisse être utilisée dans une directive **try-with-resources**, il faut qu'elle implémente la nouvelle interface `java.lang.AutoCloseable` (définissant une méthode unique `close()`)

```
public static void main(String [] args) throws Exception
{
    try (InputStream is= new FileInputStream("/tmp/data.bin")) {
        long count= 0;
        while (is.read() >= 0)
            count++;
    }
}
```



# Flux d'E/S fichiers

- **Clause *try-with-resources***

- Le même type de construction est possible en Python (depuis la version 2.5) sous la forme **with...as** comme illustré dans l'exemple ci-dessous

```
with open("/var/log/messages", "r") as myFile :  
    for line in myFile :  
        print line
```

- Les objets fichier en Python sont équipés de méthodes `__enter__` et `__exit__` qui sont appelées respectivement lors de l'entrée et de la sortie dans le bloc **with...as**. La méthode `__exit__` peut ainsi s'occuper de fermer le fichier.

# Table des Matières

1. Introduction
2. La classe `File`
3. Flux d'entrées/sorties
  1. Flux d'octets
  2. Flux d'octets fichiers
  3. **Performances et buffer**
  4. Flux de caractères
4. Accès aléatoire à des fichiers
5. Flux d'objets



# Flux d'E/S fichiers

- **Performances de la lecture**

- La lecture et l'écriture d'un fichier *octet par octet* **doivent être évitées autant que possible**. Un accès par blocs de plusieurs octets améliore significativement les performances !!!
- Les classes `InputStream` et `OutputStream` fournissent des méthodes permettant de lire / écrire plusieurs octets en un seul appel.
- Exemple pour `InputStream`

```
public int read(byte[] b) throws IOException;  
public int read(byte[] b, int off, int len)  
    throws IOException;
```

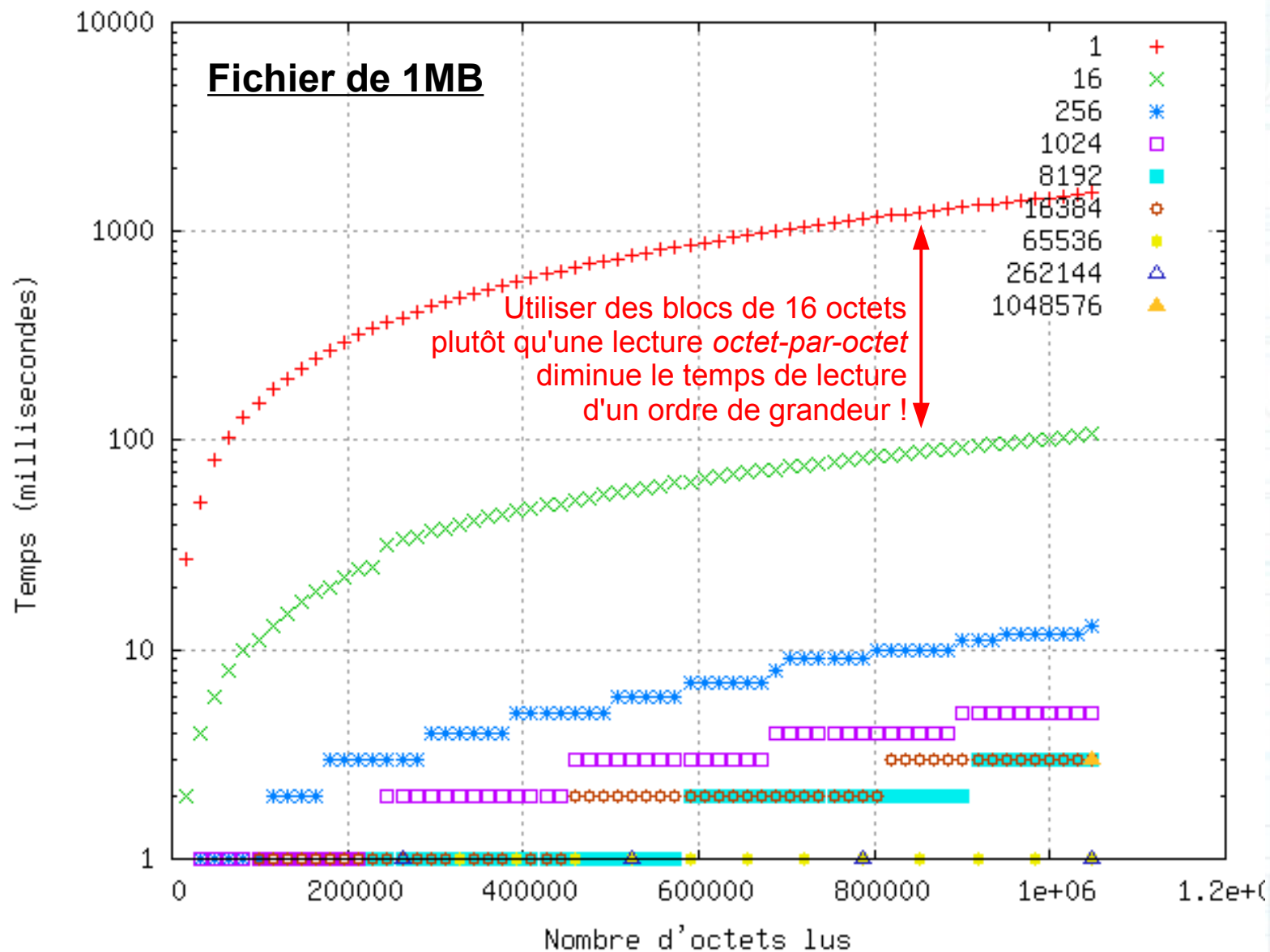
- La 1<sup>ère</sup> méthode lit au maximum `b.length` octets placés à partir de l'index 0. La seconde méthode lit au maximum `len` octets placés à partir de l'index `off`.
- Les deux méthodes retournent le nombre d'octets effectivement lus ou -1 si la fin du flux est atteinte avant qu'un seul octet soit lu.

# Flux d'E/S fichiers

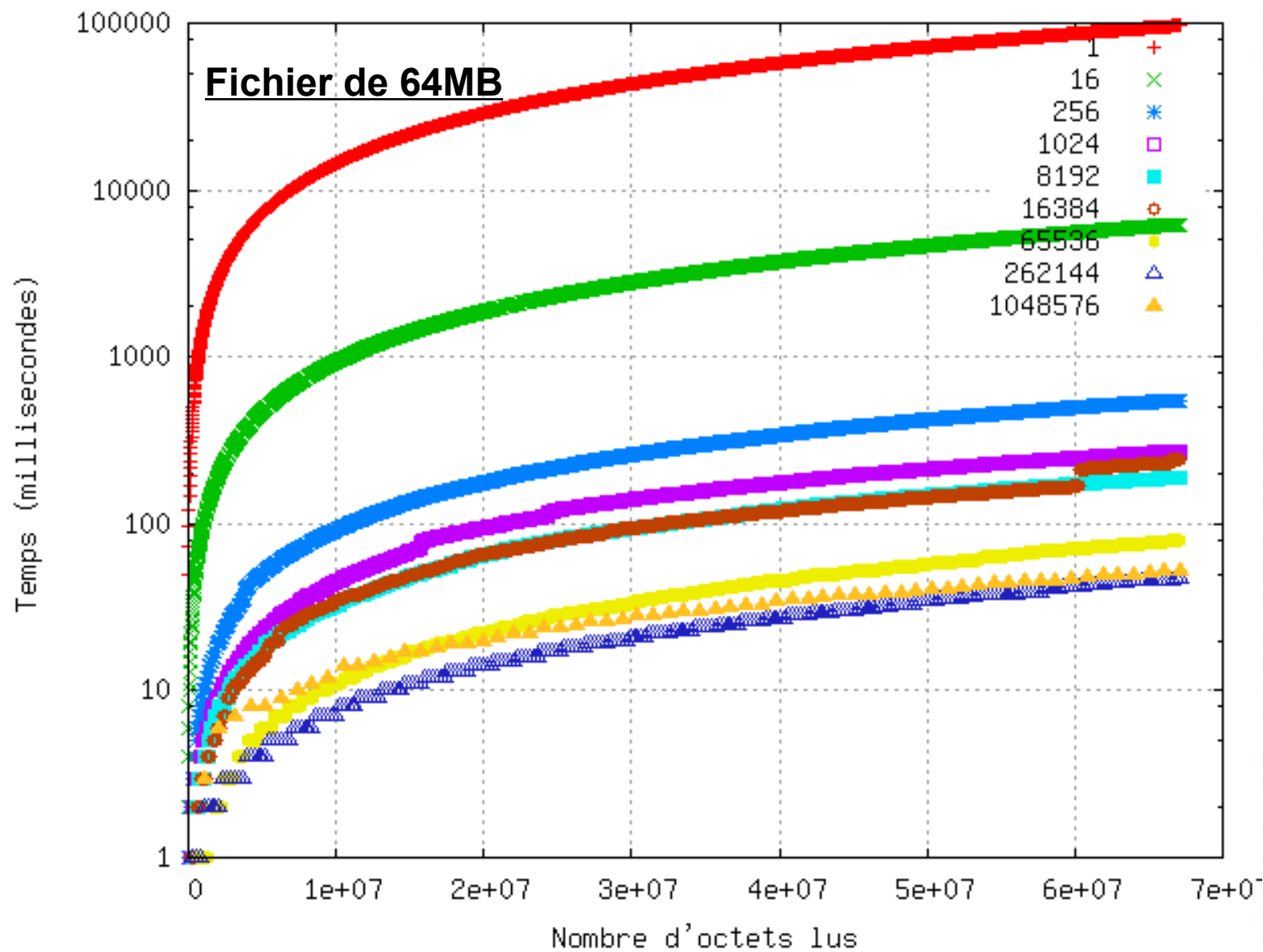
- **Performances de la lecture**
  - Afin de montrer l'impact des lectures octet par octet et par blocs de plusieurs octets, effectuons une expérience de lecture de fichiers relativement volumineux en utilisant des blocs de tailles différentes et comparons les temps de lecture
    - tailles des fichiers: 1MB et 64MB
    - tailles de blocs: 1, 16, 256, 1024, 8192, 16384, 65536, 262144, 1048576 octets



# Flux d'E/S fichiers



# Flux d'E/S fichiers





# Flux d'E/S fichiers

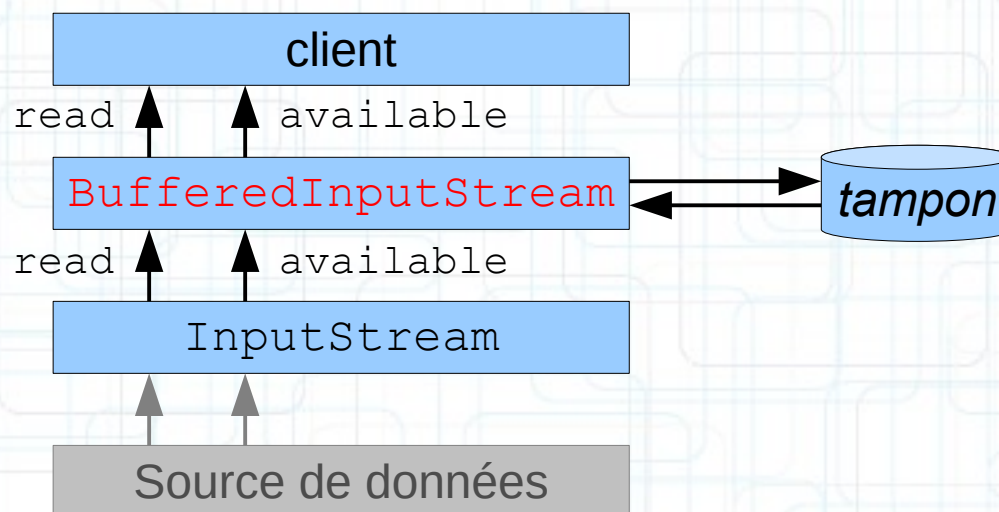
- **Classes tampon**

- Afin d'aider le programmeur à effectuer des lectures / écritures de flux plus efficaces, la bibliothèque java fournit des classes dédiées permettant d'adjoindre une lecture/écriture « avec tampon » à un flux existant.
- Un **tampon** (*buffer*) est un espace mémoire intermédiaire entre le consommateur / producteur de caractères et le flux d'entrée ou de sortie. Le programme peut faire des accès octet-par-octet au tampon, mais les accès aux flux sont effectués par blocs.
- Les classes « tampon » ont un nom préfixé par « `Buffered` ». Par exemple,
  - la classe `BufferedInputStream` permet d'ajouter un tampon à un flux en entrée `InputStream`.
  - la classe `BufferedOutputStream` permet d'ajouter un tampon à un flux en sortie `BufferedOutputStream`.

# Flux d'E/S fichiers

- **Design pattern « Decorator »**

- Les classes tampon sont « mises en série » avec les classes flux existantes. Cela permet d'ajouter la fonctionnalité de tampon à une classe flux existante sans modifier cette dernière ! Il s'agit d'un *design pattern* appelé « **Decorator** ».



- **Exemple**

```
FileInputStream fis= new FileInputStream("/tmp/data.bin");  
BufferedInputStream bis= new BufferedInputStream(fis);
```

Note : un autre constructeur permet de spécifier la taille du tampon.



# Flux d'E/S fichiers

- **Classe BufferedInputStream**

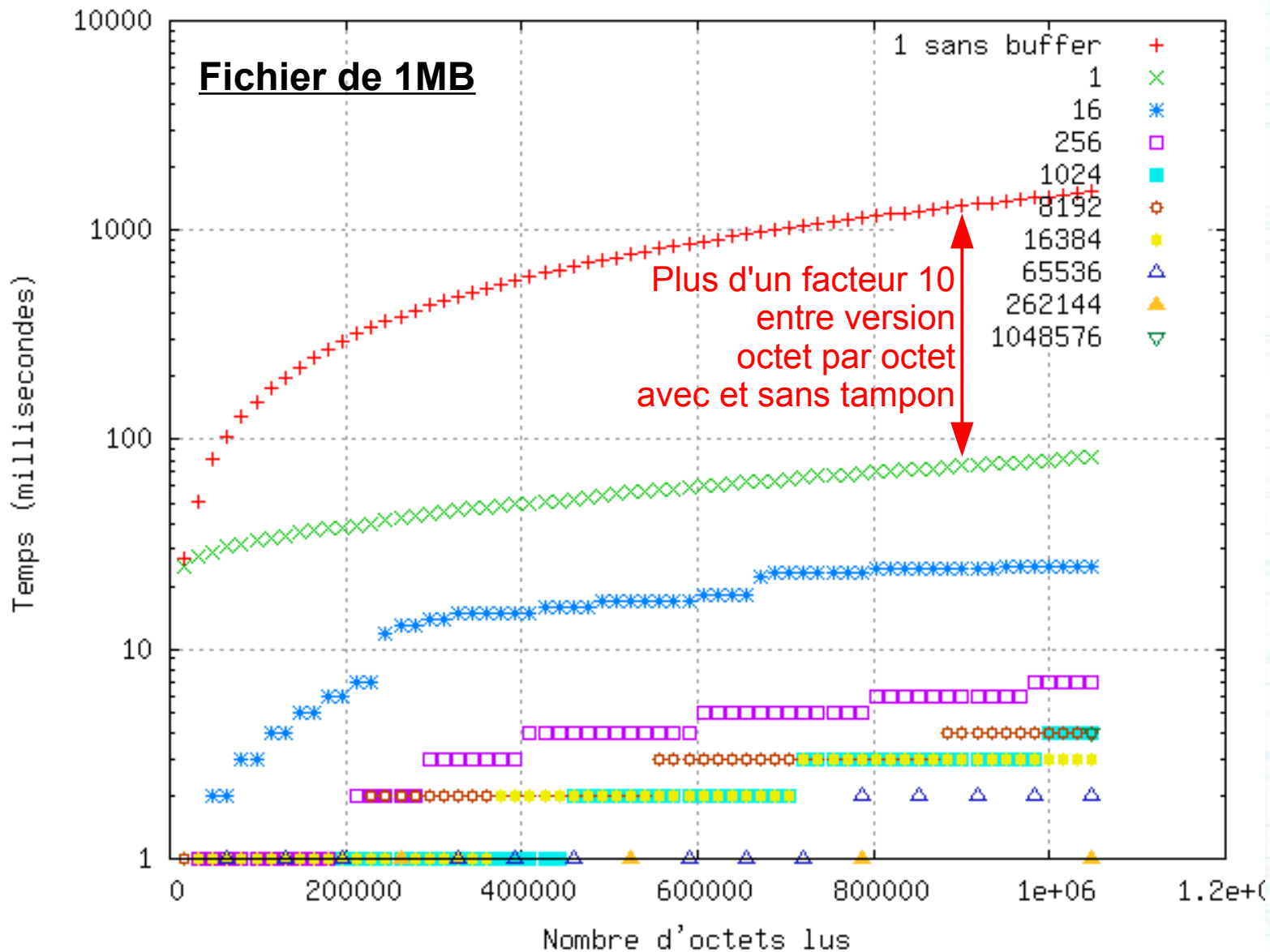
- Exemple

```
FileInputStream fis= new FileInputStream("/tmp/data.bin");
BufferedInputStream bis= new BufferedInputStream(fis);
try {
    int b;
    while ((b= bis.read()) >= 0) {
        // ... fait qqechose avec ce qui est lu ...
    }
} finally {
    bis.close();
}
```

Construction d'un  
BufferedInputStream  
à partir d'un autre  
InputStream

La fermeture (close) de  
BufferedInputStream  
entraîne la fermeture de  
InputStream

# Flux d'E/S fichiers





# Flux d'E/S fichiers

- **Exercice**

- Implémenter votre propre classe `InputStream` avec *buffer*. Cette classe sera nommée `MyBufferedInputStream`.
- Les contraintes sont les suivantes
  - La classe doit pouvoir être utilisée avec `InputStream` et ses descendants.
  - Utiliser le *design pattern* « decorator ».
  - Mesurer le gain de performance obtenu en effectuant des lectures de fichier octet par octet avec et sans `MyBufferedInputStream`.
  - Tracer les performances de l'accès dans les deux cas dans un même graphique (en utilisant `gnuplot`).

# Table des Matières

1. Introduction
2. La classe `File`
3. Flux d'entrées/sorties
  1. Flux d'octets
  2. Flux d'octets fichiers
  3. Performances et buffer
  4. **Flux de caractères**
4. Accès aléatoire à des fichiers
5. Flux d'objets



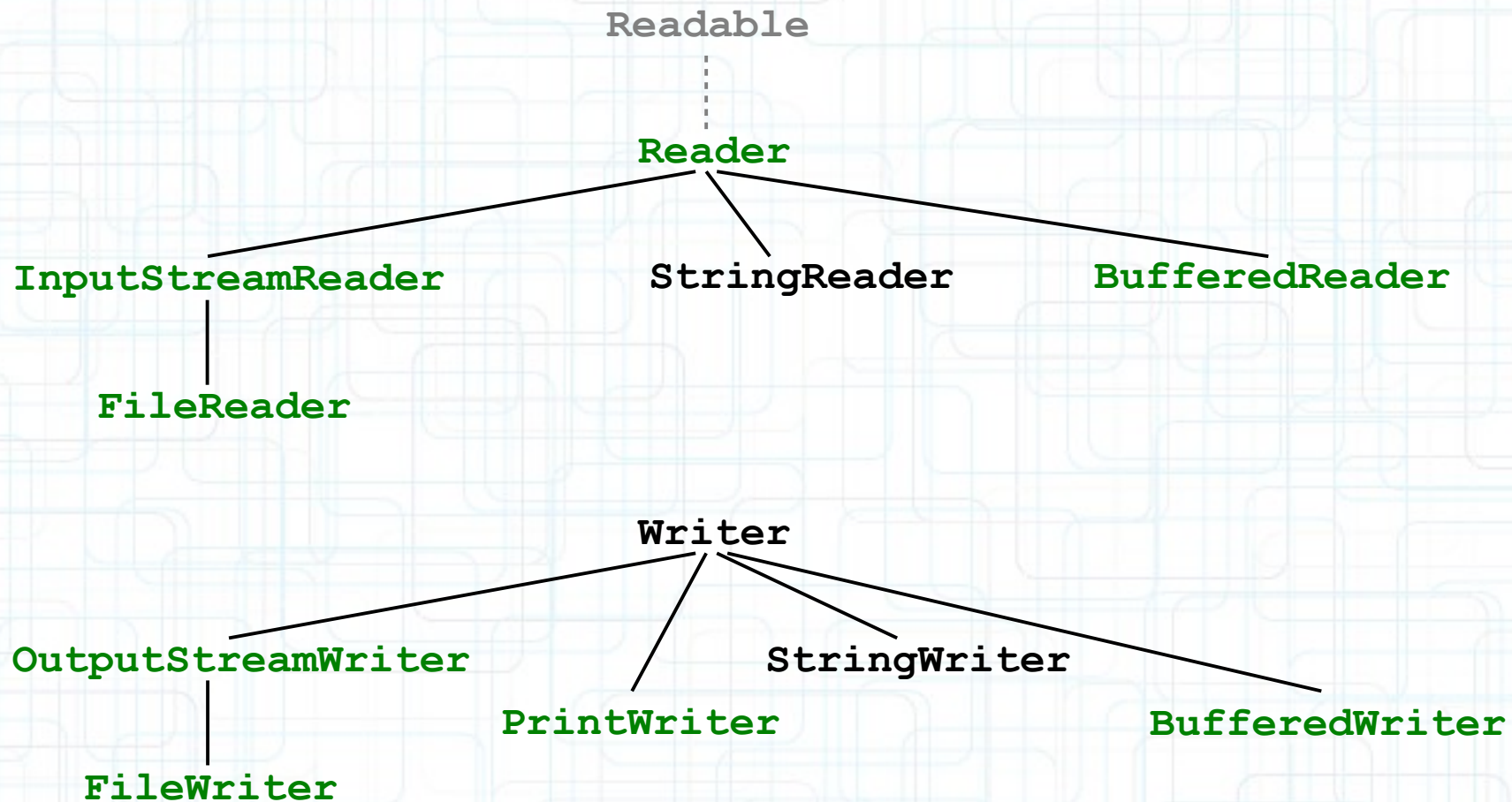
# Flux de caractères

- **Introduction**

- Les flux d'entrées/sorties basés sur `InputStream` et `OutputStream` permettent de lire et écrire des séquences d'octets ou d'objets. A priori ils pourraient aussi servir à lire des fichiers texte sous la forme de flux de caractères.
- Cependant, en toute généralité, **un caractère ne correspond pas à un octet** ! Pour cette raison, la librairie Java fournit des **familles de classes spécifiques à la manipulation de flux de texte**.
  - `java.io.Reader` et `java.io.Writer` : dédiées à l'écriture et la lecture de caractères.
  - `PrintWriter` : destinée à écrire des objets / types primitifs divers sous la forme de suite de caractères.
  - `InputStreamReader` et `OutputStreamWriter` : permettent d'adapter le comportement des flux d'octets

# Flux de caractères

- Hiérarchie de classes





# Flux de caractères

- **Introduction**

- Les classes `Reader` et `Writer` sont des **classes abstraites**. Elles définissent chacune une seule méthode importante : respectivement `read` et `write`.

```
public abstract class Reader {  
    public abstract void close();  
    public int read();  
    public abstract int read(char [] cbuf, int off, int len);  
    public boolean ready() { ... }  
    public long skip(long n) { ... }  
}
```

```
public abstract class Writer {  
    public Writer append(char c) { ... }  
    public abstract void close();  
    public abstract void flush();  
    public void write(int c) { ... }  
    public abstract void write(char [] cbuf, int off, int len);  
    public void write(String str) { ... }  
}
```

Note : d'autres méthodes sont supportées. Voir documentation...

# Flux de caractères

- **Lecture / écriture de caractères**

- La méthode `read()` de la classe `Reader` permet de lire un certain nombre de caractères à partir du flux.

```
public abstract int read(char[] cbuf, int off, int len)
    throws IOException;
```

- Retourne le nombre de caractères lus, ou -1 si la fin du flux est atteinte.
  - Cette méthode est bloquante.
- La méthode `write()` de la classe `Writer` permet d'écrire un certain nombre de caractères sur le flux.

```
public abstract void write(char[] cbuf, int off, int len)
    throws IOException;
```



# Flux de caractères

- **Ecriture formatée**

- La classe `PrintWriter` permet l'écriture de types primitifs et objets sous forme de suites de caractères

```
public void print(int i);  
public void print(double d);  
public void print(String s);  
public void print(Object obj);  
public void println(int i);  
public void println(double d);  
public void println(String s);  
public void println(Object obj);  
...
```

- ainsi que l'écriture de chaînes formatées

```
public void printf(String format, Object... args);
```

par exemple

```
pw.printf("Le caractère %c a le code %x\n",  
        'ö', Character.codePointAt("ö", 0));
```

Spécificateurs de format. Il doit y en avoir autant que de paramètres `args`.

%c → caractère  
%x → entier affiché en hexa  
%d → entier décimal  
%s → chaîne de caractères

(... consulter la doc pour les autres spécificateurs...)

# Flux de caractères

- ***Code points***

- Les caractères manipulés par java sont représentés dans la machine virtuelle par des **codes** (*code points*) en suivant le standard **Unicode**. Chaque code identifie de façon unique un caractère.
- Exemples
  - 'A' est identifié par le *code point* 0041 (en hexadécimal)
  - 'ö' est identifié par le *code point* 00F6
  - '€' est identifié par le *code point* 20AC
- Détail
  - Pour des raisons historiques, les codes utilisés par java sont représentés sur 16 bits mais Unicode a évolué et utilise des codes compris entre 0x000000 et 0x10FFFF. Pour cette raison, certains caractères en java sont représentés par une paire de codes sur 16 bits !



# Flux de caractères

- **Encodage de caractères**

- Lorsque des caractères sont écrits dans un fichier ou lus à partir d'un fichier, un **encodage de caractères** (*character encoding*) est utilisé.
- L'encodage définit la correspondance entre les *code points* et les suite d'octets qui les représentent. Il existe un grand nombre d'encodages (p.ex. : [ASCII](#), [UTF-8](#), [UTF-16](#), [ISO-8859-15](#), ...)
- Il est possible de déterminer l'encodage par défaut de la JVM.

```
OutputStreamWriter out=  
    new OutputStreamWriter(new ByteArrayOutputStream()) ;  
System.out.println(out.getEncoding());
```

- L'encodage par défaut varie d'une plateforme à l'autre.
- Cela signifie que des fichiers texte écrits sur des plateformes différentes peuvent ne pas être compatibles !
- Exemples : Linux (Ubuntu 12.04, JDK1.7u17) → UTF8  
Mac (OS X 10.6.8, JDK1.6.0\_43) → MacRoman

# Flux de caractères

- Encodage de caractères

- Petite manipulation : exécution d'un même programme java sur deux plateformes utilisant des encodages différents.

```
public static void main(String [] args) throws Exception
{
    OutputStreamWriter out= new OutputStreamWriter(System.out);
    System.out.println(out.getEncoding());
    PrintWriter pw= new PrintWriter(out);
    pw.println("\u0041");
    pw.println("\u00F6");
    pw.println("\u20AC");
    pw.close();
}
```

```
bash-3.2$ hexdump -C encoding-mac.txt
00000000  4d 61 63 52 6f 6d 61 6e 0a 41 0a 9a 0a db 0a  |MacRoman.A....|
0000000f
```

```
bash-3.2$ hexdump -C encoding-linux.txt
00000000  55 54 46 38 0a 41 0a c3 b6 0a e2 82 ac 0a  |UTF8.A.....|
0000000e
```



# Flux de caractères

- **Encodage de caractères**

- Il existe 2 moyens de contrôler l'encodage des caractères en Java
  - **Argument du constructeur d'`OutputStreamWriter`** : permet de spécifier l'encodage de caractères à utiliser. Celui-ci est spécifié sous la forme d'une chaîne de caractères. Par exemple : « `utf8` » ou « `iso-8859-15` ».
  - **Option de la machine virtuelle** : il est possible de spécifier l'encodage de caractères par défaut. Avec la JVM d'Oracle, l'option `-Dfile.encoding` peut être utilisée.

```
bash-3.2$ java -Dfile.encoding=utf8 TestWriter
...
```

# Flux de caractères

- **Classes `File{Reader|Writer}`**

- Les classes `FileReader` et `FileWriter` permettent respectivement la lecture et l'écriture d'un fichier sous la forme d'un flux de caractères. Ces classes utilisent l'encodage de caractères par défaut de la JVM.

- Exemple

```
FileReader reader= new FileReader("/tmp/data.txt");
int c;
while ((c= reader.read()) >= 0) {
    char chr= (char) c;
    // ... traite le caractère lu...
}
reader.close();
```



# Flux de caractères

- **Classes Buffered{Reader|Writer}**
  - Pour les mêmes raisons de performances qu'évoquées avec les flux d'octets, il est important d'effectuer des lectures/écritures de caractères par blocs.
  - La bibliothèque Java fournit les classes `BufferedReader` et `BufferedWriter`. Celles-ci utilisent également le *design pattern* « *decorator* ».
  - Exemple

```
try (BufferedReader br=  
    new BufferedReader(new FileReader("/tmp/data.txt"))) {  
    int c;  
    while ((c= br.read()) >= 0) {  
        char chr= (char) c;  
        //... traite le caractère lu...  
    }  
}
```

# Flux de caractères

- **Classes `BufferedReader`**

- La classe `BufferedReader` offre une méthode supplémentaire très utile : la lecture d'une ligne complète de texte.

```
public String readLine();
```

- retourne `null` si la fin du flux est atteinte
- retourne une ligne de caractères : la délimitation d'une ligne est basée sur un caractère *carriage-return* (`\r`), un *line-feed* (`\n`), la séquence `\r\n` ou la fin de ligne
- les caractères délimitant les lignes ne sont pas retournés dans la chaîne



# Flux de caractères

- **Classe Scanner**

- La classe **Scanner** du package (`java.util`) n'est pas une classe flux. Cependant, elle permet de faciliter la lecture de données à partir d'un flux de caractères.
- Elle permet d'extraire une ligne, un entier, un double, ... ou encore des suites de caractères correspondant à un « *template* » bien défini.
  - Source de données : un flux de caractères, sous la forme d'une classe implémentant l'interface **Readable**. L'interface `Readable` définit une méthode unique `read()`. La classe `Reader` et ses sous-classes implémentent cette interface.

```
public int read(CharBuffer cb) throws IOException;
```

- Délimitation des données : la classe `Scanner` découpe les caractères de la source en groupes appelés « **tokens** », en se basant sur un délimiteur. Un délimiteur est typiquement une suite de caractères bien spécifique ou une *expression régulière*.

# Flux de caractères

- **Classe Scanner**

- Les méthodes de `Scanner` peuvent être séparées en 3 groupes
- Tester si un *token* est disponible

```
public boolean hasNext();  
public boolean hasNextInt();  
public boolean hasNextDouble();  
...
```

- Obtenir le *token* suivant

```
public String next();  
public int nextInt();  
public double nextDouble();  
public String nextLine();  
...
```

- Configurer le découpage en *tokens* ; le délimiteur par défaut = un blanc (espace, tabulation, retour à la ligne, ...)

```
public void useDelimiter(String pattern);
```



# Flux de caractères

- **Classe Scanner**

- Exemple : supposons que l'on veuille lire le contenu du fichier `/etc/passwd` sur un système UNIX. Il s'agit d'un fichier texte dans lequel chaque utilisateur est décrit par une ligne. Chaque ligne concerne un utilisateur et contient plusieurs informations

- *nom de login*
- *mot de passe chiffré* (en réalité stocké dans autre fichier)
- *identifiant d'utilisateur* (user ID - UID)
- *identifiant de groupe* (group ID - GID)
- *description textuelle*
- *répertoire « home »*
- *programme shell*

```
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
jsmith:*:100:100:John Smith:/home/jsmith:/bin/bash
```

# Flux de caractères

- **Classe Scanner**

```
import java.io.*;
import java.util.Scanner;

public class PasswdScanner {

    public static void main(String[] args) throws IOException {
        FileReader fr= new FileReader("/etc/passwd");
        Scanner s= new Scanner(fr);
        s.useDelimiter(":");
        while (s.hasNext()) {
            System.out.print("user=" + s.next());
            System.out.print(", pwd=" + s.next());
            System.out.print(", uid=" + s.nextInt());
            System.out.print(", gid=" + s.nextInt());
            System.out.print(", desc=" + s.next());
            System.out.print(", home=" + s.next());
            System.out.println(", sh=" + s.nextLine());
        }
        fr.close();
    }
}
```



# Flux de caractères

- **Classe Scanner**

```
bash-3.2$ java PasswdScanner
user=nobody, pwd=*, uid=-2, gid=-2, desc=Unprivileged User, home=/var/empty,
sh=:/usr/bin/false
user=root, pwd=*, uid=0, gid=0, desc=System Administrator, home=/var/root,
sh=:/bin/sh
user=daemon, pwd=*, uid=1, gid=1, desc=System Services, home=/var/root,
sh=:/usr/bin/false
user=jsmith, pwd=*, uid=100, gid=100, desc=John Smith, home=/home/jsmith,
sh=:/bin/bash
```

# Table des Matières

1. Introduction
2. La classe `File`
3. Flux d'entrées/sorties
  1. Flux d'octets
  2. Flux d'octets fichiers
  3. Performances et buffer
  4. Flux de caractères
- 4. Accès aléatoire à des fichiers**
5. Flux d'objets



# Accès Fichier Aléatoire

- **Introduction**

- L'abstraction fournie par les flux est utile lorsqu'un programme doit lire de façon séquentielle la totalité d'un fichier.
- Cependant, certaines applications nécessitent de pouvoir aller lire à n'importe quel endroit dans un fichier ou nécessitent de pouvoir retourner en arrière. On parle d'**accès aléatoire** à un fichier (*random access*). L'utilisation de flux n'est pas appropriée pour ces applications.
- Note
  - Certaines sources/destinations de données ne permettent pas l'accès aléatoire. Ainsi, l'accès aléatoire est possible pour le contenu d'un fichier mais pas pour le contenu d'une connexion réseau ou de l'entrée standard (`System.in`).

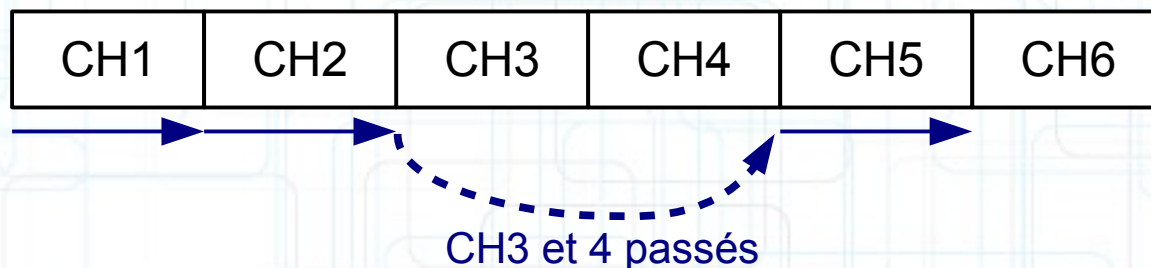
# Accès Fichier Aléatoire

- **Exemple – lecture audio / vidéo**

- Un programme de lecture de vidéo / musique nécessite de pouvoir accéder de façon aléatoire au fichier contenant la vidéo/musique.
- la fonction « play » effectue une lecture séquentielle.



- les fonctions d'avance/recul rapides ou de sélection de chapitre nécessitent d'aller se positionner à un endroit particulier du fichier vidéo.





# Accès Fichier Aléatoire

- **Exemple – base de données**

- Une base de données simple contient un ensemble d'enregistrements de taille fixe.
- Chaque enregistrement a une structure et une taille identiques. Par exemple
  - Numéro compte bancaire : 16 octets
  - Nom propriétaire : 20 octets
  - Solde : 4 octets
- Dans un fichier contenant 10000 enregistrements, il est facile d'accéder à n'importe quel enregistrement individuel (avec `seek()`)

1	2		1233	1234	1235		10000
---	---	--	------	------	------	--	-------

$\text{seek}(1234 * (16+20+4))$

# Accès Fichier Aléatoire

- **Class RandomAccessFile**

- La classe **RandomAccessFile** permet d'accéder de façon aléatoire au contenu d'un fichier.
- Les constructeurs de la classe `RandomAccessFile`, prennent en argument un chemin d'accès (instance de `File`) ou un nom de fichier ainsi qu'un mode d'accès.

```
public RandomAccessFile(String name, String mode);  
public RandomAccessFile(File file, String mode);
```

- Le mode d'accès est soit la lecture uniquement (« `r` ») soit la lecture et l'écriture (« `rw` »).



# Accès Fichier Aléatoire

- **Class RandomAccessFile**

- La classe `RandomAccessFile` fournit des méthodes permettant la lecture et l'écriture.

```
public int read(int b);  
public int read(byte[] bytes);  
public int read(byte[], int off, int len);  
public void write(int b);  
public void write(byte[] bytes);  
public void write(byte[], int off, int len);
```

- Cependant, contrairement aux flux, la position de lecture peut être manipulée
  - Pointeur de fichier : `RandomAccessFile` maintient un pointeur de fichier (variable d'instance interne) indiquant la position actuelle de lecture ou d'écriture.
  - Lecture ou écriture : ont lieu à la position donnée par le pointeur de fichier. A chaque lecture ou écriture, le pointeur de fichier est avancé du nombre d'octets lus/écrits.

# Accès Fichier Aléatoire

- **Class RandomAccessFile**

- La classe `RandomAccessFile` fournit également des méthodes pour manipuler le pointeur de fichier.

- Position du pointeur de fichier

```
public long getFilePointer();
```

- Déplacer le pointeur de fichier

```
public void seek(long pos);
```

- Passer un certain nombre d'octets

```
public void skipBytes(int n);
```



# Accès Fichier Aléatoire

- **Exercice**

- Implémenter un **système d'annuaire téléphonique** avec des fichiers à accès aléatoire.
- Le fichier contient des enregistrements de taille fixe reprenant les coordonnées de vos contacts : nom, prénom, localité, numéro de téléphone fixe, numéro de téléphone mobile et adresse e-mail.
- Les opérations suivantes doivent être possibles
  - récupérer les données du  $i^{\text{ème}}$  enregistrement, sans lire l'ensemble du fichier.
  - en supposant le fichier trié, implémenter une recherche dichotomique permettant de rechercher l'enregistrement correspondant à une personne particulière, sur base de son nom. Cette recherche doit également pouvoir être effectuée sans lire complètement le fichier.

# Table des Matières

1. Introduction
2. La classe `File`
3. Flux d'entrées/sorties
  1. Flux d'octets
  2. Flux d'octets fichiers
  3. Performances et buffer
  4. Flux de caractères
4. Accès aléatoire à des fichiers
- 5. Flux d'objets**



# Flux d'objets (sérialisation)

- **Sauver / charger une liste**
  - Note : ce problème a été posé par un étudiant en fin de cours. Il n'a pas été couvert durant le cours.
  - Question : En python, avec le module `pickle`, il est facile de sauvegarder une liste dans un fichier et de récupérer cette liste par la suite. Comment faire la même chose en Java ?

```
import pickle

# Create list and save to file
l= [17, 3.56, -3]
with open("listepy.dat", "w") as f :
    pickle.dump(l, f)

# Load list from file
with open("listepy.dat", "r") as f :
    l= pickle.load(f)
```

# Flux d'objets (sérialisation)

- **Sauver / charger une liste**
  - En java, il est possible d'utiliser les flux d'objets. Les objets supportant l'interface `java.io.Serializable` peuvent être sérialisés et désérialisés (convertis de et vers une suite d'octets).
  - Les classes `ObjectOutputStream` et `ObjectInputStream` permettent d'effectuer la (dé-)sérialisation.



# Flux d'objets (sérialisation)

- Sauver / charger une liste

```
import java.io.*;
import java.util.*;

public class ListSerialization {

    public static void main(String [] args)
        throws IOException, ClassNotFoundException
    {
        List<Number> l= new ArrayList<Number>();
        // Add content to list here ...
```

```
        OutputStream fos= new FileOutputStream("listejava.dat");
        ObjectOutputStream oos= new ObjectOutputStream(fos);
        oos.writeObject(l);
        oos.close();
```

Sérialisation

```
        InputStream fis= new FileInputStream("listejava.dat");
        ObjectInputStream ois= new ObjectInputStream(fis);
        l= (List<Number>) ois.readObject();
        ois.close();
```

Désérialisation

```
        System.out.println(l);
    }
}
```

# Flux d'objets (sérialisation)

- Sauver / charger une liste
  - Exemple de fichier contenant une `ArrayList` avec les instances suivantes: `Integer(5)`, `Double(5.36)` et `Byte(-3)`

```
$ hexdump -C file.dat
00000000 ac ed 00 05 73 72 00 13 6a 61 76 61 2e 75 74 69 |....sr..java.util|
00000010 6c 2e 41 72 72 61 79 4c 69 73 74 78 81 d2 1d 99 |l.ArrayListx....|
00000020 c7 61 9d 03 00 01 49 00 04 73 69 7a 65 78 70 00 |.a....I..sizexp.|
00000030 00 00 03 77 04 00 00 00 03 73 72 00 11 6a 61 76 |...w.....sr..jav|
00000040 61 2e 6c 61 6e 67 2e 49 6e 74 65 67 65 72 12 e2 |a.lang.Integer..|
00000050 a0 a4 f7 81 87 38 02 00 01 49 00 05 76 61 6c 75 |.....8...I..valu|
00000060 65 78 72 00 10 6a 61 76 61 2e 6c 61 6e 67 2e 4e |exr..java.lang.N|
00000070 75 6d 62 65 72 86 ac 95 1d 0b 94 e0 8b 02 00 00 |umber.....|
00000080 78 70 00 00 00 05 73 72 00 10 6a 61 76 61 2e 6c |xp....sr..java.l|
00000090 61 6e 67 2e 44 6f 75 62 6c 65 80 b3 c2 4a 29 6b |ang.Double...J)k|
000000a0 fb 04 02 00 01 44 00 05 76 61 6c 75 65 78 71 00 |.....D..valuexq.|
000000b0 7e 00 03 40 15 70 a3 d7 0a 3d 71 73 72 00 0e 6a |~..@.p...=qsr..j|
000000c0 61 76 61 2e 6c 61 6e 67 2e 42 79 74 65 9c 4e 60 |ava.lang.Byte.N`|
000000d0 84 ee 50 f5 1c 02 00 01 42 00 05 76 61 6c 75 65 |..P.....B..value|
000000e0 78 71 00 7e 00 03 fd 78 |xq.~...x|
000000e8
$
```



# Annexes

# Flux d'E/S fichiers

- **Classes tampon - implémentation**
  - En interne, l'instance de `BufferedInputStream` possède
    - un tampon « circulaire » (`buf`, tableau d'octets),
    - un index du prochain octet à lire par le client (`pos`),
    - un compteur indiquant le nombre d'octets valides dans le tampon (`count`)

