

Input/Output

Rémi Forax
forax@univ-mlv.fr

java.io.File

- java.io.File représente un chemin textuelle vers un fichier ou un répertoire
- Beurk[®] car mix :
 - Gestion du chemin
 - Gestion différents types de fichiers
 - Propriété du fichier associé
 - Méthode pour créer, renommer, etc.en une seule classe

API d'I/O au cours du temps

- IO: Java 1.0
 - File, InputStream/OutputStream
- IO: Java 1.1
 - Reader/Writer
- NIO: Java 1.4 (non-blocking)
 - Charset, ByteBuffer, Channel, Selector
- NIO2: Java 1.7 (async)
 - Path, FileAttribute, WatchService, Asynchronous*Channel

Chemin vers un fichier

- Path (ex File) est un chemin dans l'arborescence
 - Il existe 3 sortes de chemins :
 - relatif `../toto/titi.txt`
 - absolue `/tmp/titi.txt`
 - canonique/realPath `c:\Program Files\Java\`
(ils sont uniques)
- Un Path est une liste chaînée des éléments du plus éloigné vers la racine

`home <- forax <- ens <- io.odp`

un Path est un `Iterable<Path>`

Path et WORA

- Constante `java.io.separatorChar`

/ (Unix), \ (Windows) et : ou / (Mac)

- Création d'un chemin :
 - `Paths.get(filename)`
 - `Paths.get(directory, filename)`

Le séparateur est ajouté automatiquement

Il n'y a rien pour gérer les extensions !

Gestion des chemins

- Manipulation de Path :
 - nom du dernier élément :
getFileName()
 - chemin vers le repertoire père :
getParent()
 - chemin absolu vers le fichier :
toAbsolutePath() / isAbsolute()
 - chemin canonique :
toRealPath(LinkOption)
 - Rétro-compatibilité avec l'ancienne API :
toFile()

Résolution de chemin

- Un Path par rapport à un autre Path/String :
 - Enlève les '..' :
normalize()
 - Résoud un chemin par rapport au chemin courant :
resolve(Path subPath) / resolve(String subPath)
 - Substitue par un frère :
resolveSibling(Path) / resolveSibling(String)
 - Crée un chemin relatif (inverse de resolve) :
relativize(Path path)

Chemin en tant que fichier

- Classe `java.nio.file.Files`
 - Création
 - `createDirectory()`, `createFile`, `createLink`,
`createSymbolicLink`, `createTempDirectory`,
`createTempFile`
 - Prend en paramètre un `FileAttribute`, que l'on peut
créer par exemple à partir de `PosixFilePermissions`
 - Existence
 - `IsDirectory()`, `isRegularFile()`, `isHidden()`
 - Droits
 - `getOwner()`, `getPosixFilePermission()`,
`getFileAttributeView()`,

Chemin en tant que fichier

Classe `java.nio.file.Files`

- Lecture/écriture sur un fichier
 - `newInputStream/newOutputStream`
 - `newBufferedReader/newBufferedWriter`
 - `newByteChannel`
- Existence
 - `isDirectory()`, `isRegularFile()`, `isHidden()`, etc.
- Droits
 - `get/setOwner()`, `get/setPosixFilePermission()`,
`get/setAttribute()`, `getFileAttributeView()`
- Scan de répertoire
 - `newDirectoryStream()`, `walkFileTree()`

FileSystemView

- Interaction avec le bureau en utilisant `javax.swing.filechooser.FileSystemView`
- Permet d'obtenir :
 - `isFileSystemRoot(File)`
 - `isDrive(File)/isFloppyDrive(File)`
 - `File getHomeDirectory()`
 - `String getSystemDisplayName(File)`

Exemple

- Affichage d'info supplémentaire en utilisant le FileSystemView

```
public static void main(String[] args) {  
    FileSystemView view=FileSystemView.getFileSystemView();  
    for(File root:File.listRoots()) {  
        long usable=root.getUsableSpace();  
        if (usable==0)  
            continue;  
        long total=root.getTotalSpace();  
        System.out.printf("%s\n  %,20d: %,20d\n",  
            view.getSystemDisplayName(root),usable,total);  
    }  
}
```

Disque local (C:)

48 452 771 840: 79 933 267 968

forax sur 'IGM (Samba monge) (monge)' (Z:)

19 809 697 792: 144 506 355 712

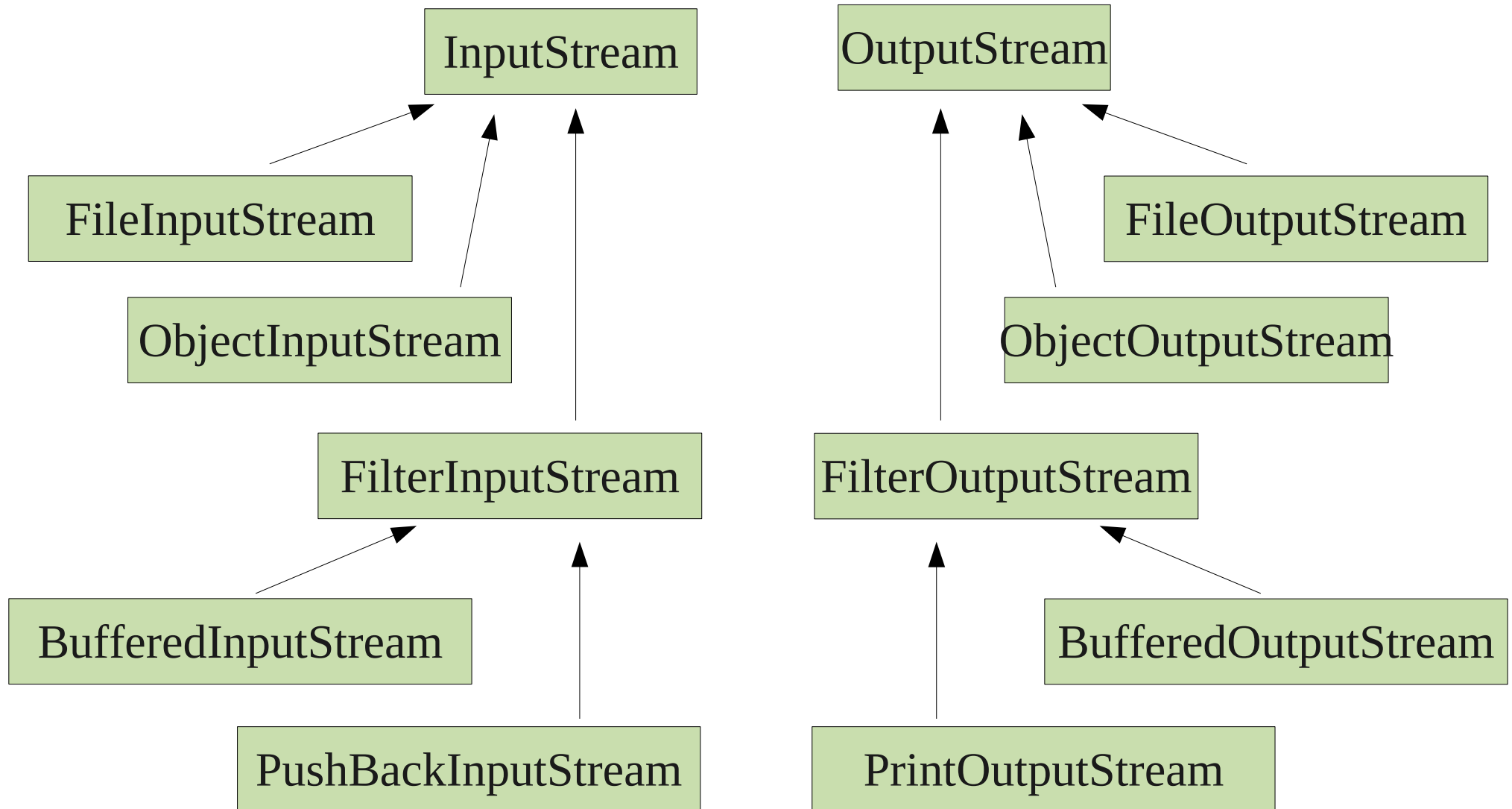
Fichier ou flux

- Il existe deux types d'accès à des données d'entrée/sortie
 - Accès à un fichier direct
 - `RandomAccessFile`, `FileChannel`, `AsynchronousFileChannel`
 - Accès à travers un flux
- Les flux (stream/reader/writer) sont une abstraction qui permet de manipuler des données successive quelles viennent d'un fichier, de la console, d'une interface réseau, etc.

2 versions des flux

- **InputStream/OutputStream** manipule des octets (byte en Java) donc dépendant de la plateforme
- **Reader/Writer** manipule des caractères (char en Java) indépendant de la plateforme mais qui nécessite d'indiquer un codage des caractères
- On utilise l'une ou l'autre des versions en fonction de ce que l'on veut faire
 - Lire un fichier de propriété (**Reader**)
 - Ecrire un fichier binaire (**OutputStream**)

Entrées/sorties par byte



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**()

InputStream et appel bloquant

- Les méthodes **read()** sur un flux sont **bloquantes** s'il n'y a pas au moins un byte à lire
- Il existe une méthode **available()** dans InputStream qui est sensée renvoyer le nombre de byte lisible sans que la lecture sur le flux soit bloquée mais mauvais support au niveau des OS (à ne pas utiliser)

InputStream et read d'un buffer

- Attention, la lecture est une demande pour remplir le buffer, le système essaye de remplir le buffer au maximum mais peut ne pas le remplir complètement
 - Lecture dans un tableau de bytes
 - `int read(byte[] b)`
renvoie le nombre de bytes lus
 - `int read(byte[] b, int off, int len)`
renvoie le nombre de bytes lus

InputStream et efficacité

- Contrairement au C (stdio) par défaut en Java, les entrées sortie ne sont pas bufferisés
- Risque de gros problème de performance si l'on lit les données octets par octets
- Solution :
 - Lire utilisant un tableau de bytes
 - Utiliser un BufferedInputStream qui utilise un buffer intermédiaire

InputStream et IOException

Toutes les méthodes de l'input stream peuvent lever une **IOException** pour indiquer que

- Il y a eu une erreur d'entrée/sortie
- Que le stream est fermé après un appel à **close()**
- Que le thread courant a été interrompu en envoyant une **InterruptedException**
(cf cours sur la concurrence)

Il faut penser à faire un **close()** sur le stream dans ce cas

InputStream et la mark

- index indiquant un endroit on l'on aimerait revenir, pratique pour le parsing
 - Indique si le stream supporte cette option (supporté par BufferedInputStream)
 - boolean **markSupported()**
 - Positionne la mark en indiquant le nombre de byte max qui sera lu avant de revenir
 - void **mark(int readlimit)**
 - Revient à la marque
 - void **reset()**

OutputStream

- Flux de byte en sortie : **OutputStream**
 - 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
 - void **flush**()
 - Ferme le flux
 - void **close**()

Copie de flux

- Byte par byte (mal)

```
public static void copy(InputStream in, OutputStream out)
    throws IOException {

    int b;
    while((b=in.read())!=-1)
        out.write(b);
}
```

- Par buffer de bytes

```
public static void copy(InputStream in, OutputStream out)
    throws IOException {

    byte[] buffer=new byte[8192];
    int size;
    while((size=in.read(buffer))!=-1)
        out.write(buffer,0, size);
}
```

Entrée clavier/sortie console

- Les constantes :
 - Entrée standard
System.in est un `InputStream` (donc 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);  
}
```

Entrée standard et appel bloquant

- L'entrée standard en Java n'est pas ouvert en mode canonique
- Donc les appel à read son **bloquant** jusqu'à ce que l'utilisateur appuie sur **entrée**
- La console ne transmet les bytes que ligne par ligne

```
public static void main(String[] args) throws IOException {  
    System.in.read();  
    // bloquant tant que l'utilisateur ,n'a pas appuyé sur return  
}
```

Closable

- Interface générique de toutes les ressources que l'on doit fermer lorsque l'on ne les utilise plus

```
public interface Closable {  
    public void close() throws IOException;  
}
```

- On doit faire les close() car souvent on ne peut pas attendre que le GC le fasse pour nous (on ne contrôle pas son activation)

*Stream et close

- Lorsque l'on ouvre un stream en lecture ou écriture, il **faut** penser à le **fermer**, sinon, il faut attendre le GC pour qu'il recycle le descripteur de fichier
- Comme une exception peut être levée, le **close** doit se faire dans un **finally**

```
public static void main(String[] args) throws IOException {  
    InputStream in=new FileInputStream(args[0]);  
    try {  
        copy(in, System.out);  
    } finally {  
        in.close();  
    }  
}
```

Et avec plusieurs fichiers...

- Il faut imbriquer les **try ... finally**

```
public static void main(String[] args) throws IOException {  
    InputStream in = Files.newInputStream(Paths.get(args[0]));  
    try {  
        OutputStream out = Files.newOutputStream(Paths.get(args[1]));  
        try {  
            copy(in, out);  
        } finally {  
            out.close();  
        }  
    }  
    finally {  
        in.close();  
    }  
}
```

Il n'est pas nécessaire de faire un **close()** si le **new** lève une exception

Et avec un try-with-resources

Le mieux est encore d'utiliser le try-with-resource

```
public static void main(String[] args) throws IOException {  
    try(InputStream in=new FileInputStream(args[0]);  
        OutputStream out=new FileOutputStream(arg[1])) {  
        copy(in, out);  
    } // appel out.close() puis in.close()  
}
```

Le try-with-resource attend un AutoClosable, Closable hérite de AutoClosable donc tout les streams peuvent utiliser le try-with-resource

Les requêtes SQL sont aussi AutoClosable

Bufferisation automatique

- BufferedInputStream/BufferedOutputStream agissent comme des proxies en installant un buffer intermédiaire entre le flux et le système
- Constructeurs :
 - **BufferedInputStream**(InputStream input,int bufferSize)
 - **BufferedOutputStream**(OutputStream input,int bufferSize)
- Attention à éviter le créer des buffered de buffered de buffered ...

Flush et close

- flush() vide le buffer
- L'appel à close délègue aux flux sous-jacent

```
public static void copy(InputStream in, OutputStream out)
    throws IOException {

    try(BufferedInputStream input=new BufferedInputStream(in);
        BufferedOutputStream output=new BufferedOutputStream(out)) {

        // caractère par caractère mais comme c'est bufferisé, ok
        int b;
        while((b=input.read())!=-1)
            output.write(b);

        output.flush(); // penser à vider le buffer
    }
}
```

Flux en Unicode

- En Java, les caractères sont manipulés en unicode (char est sur 16 bits)
- Le problème est que le système de fichier code souvent ses fichiers dans un autre format
 - En ASCII (8bits sans accent)
 - En ISOLatin 1 (8 bits avec accent)
 - En Windows Latin 1 (microsoft pas à la norme ISO)
 - En UTF8 (8/16/24 bits, compliqué)
 - En Unicode 16LE/16BE (Little or Big Endian !!)

Les Charsets

- La classe **java.nio.Charset** définit la table de conversions de **byte** vers **char** et vice versa.
- Possède deux méthodes principales :
 - CharsetDecoder **newDecoder()**
 - CharsetEncoder **newEncoder()**
- **Charset.forName(String name)** permet d'obtenir un charset à partir de son nom

StandartCharsets

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

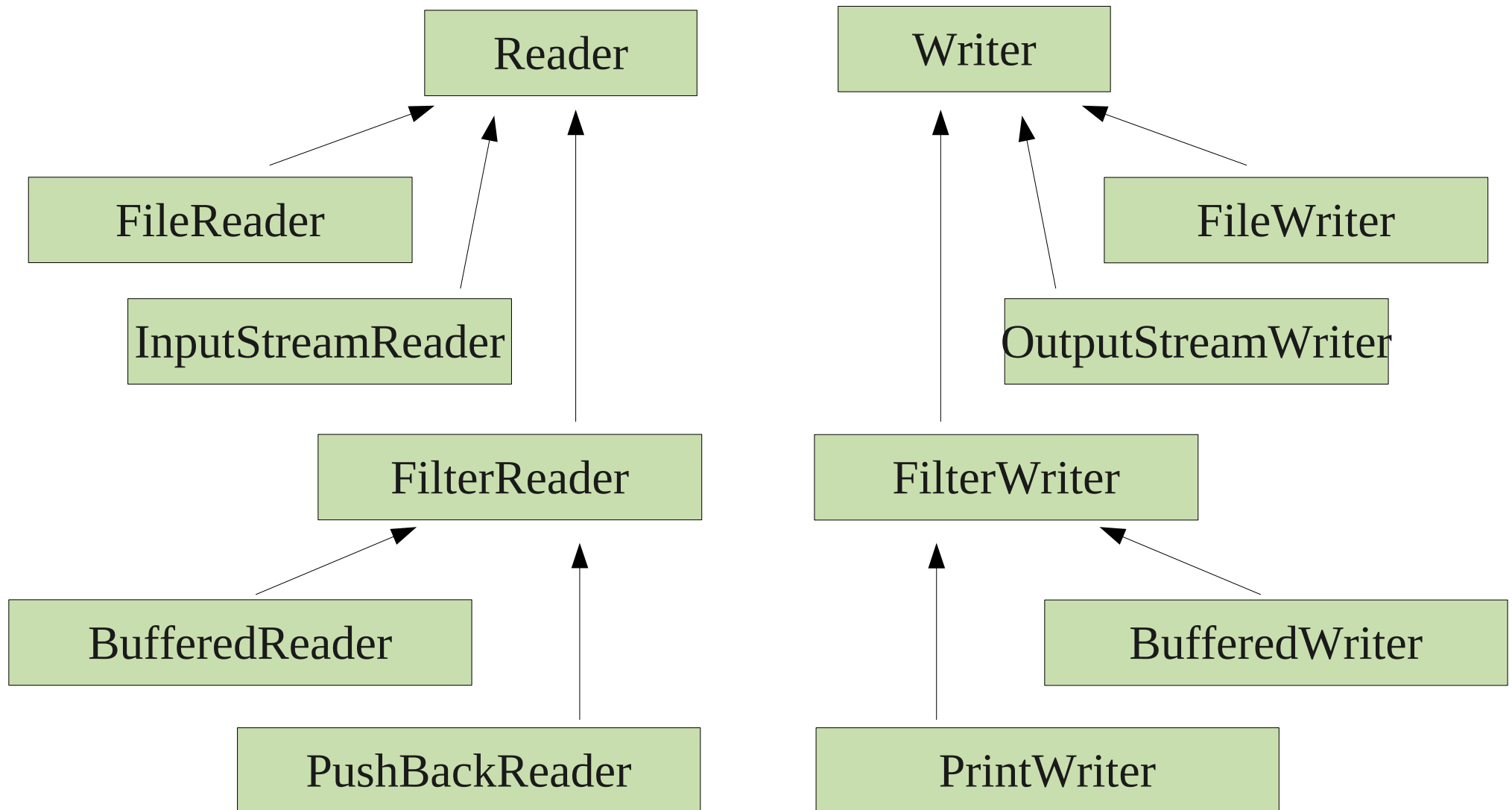
UTF-16 Sixteen-bit UCS Transformation Format,
byte order identified by an optional byte-order mark

La classe `java.nio.charset.`**StandartCharsets** contient
ces constantes

Utilisation classique des Charsets

- Convertir en mémoire un tableau de `byte[]` en `String` suivant un certain codage
 - **String**(`byte[] bytes`, **Charset** charset)
 - **String**(`byte[] bytes`, `int offset`, `int length`, **Charset** charset)
 - **string.getBytes(Charset** charset)
- Crée un `Writer/Reader` à partir d'un `InputStream/OutputStream`
 - new **InputStreamReader**(`InputStream inputstream`)
 - new **OutputStreamWriter**(`OutputStream outputstream`)

Entrées/sorties par char



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()**

Reader, appel bloquant et mark

- Comme les `InputStream`, les méthodes peuvent lever **`IOException`**
- Il existe une méthode **`ready()`** qui renvoie vrai si au moins 1 caractère est lisible (à ne pas utiliser)
- les méthodes **`mark`** et **`reset`** permettent de marquer le flux à la position courante pour y retourner, `skip` de sauter des caractères ;
- la méthode **`markSupported`** indique si ce mécanisme est implémenté.

- 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 caractère (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()`

Writer et chaîne de caractère

- Un Writer possède des méthodes spéciales pour l'écriture de chaîne de caractères
 - Ecrire une String,
 - void **write**(String s)
 - Void **write**(String str, int off, int len)
- Un writer est un Appendable donc
 - Ecrire un CharSequence
 - Writer **append**(CharSequence csq)
 - Writer **append**(CharSequence csq, int start, int end)

Copie de flux

- Caractère par caractère (mal)

```
public static void copy(Reader in,Writer out)
    throws IOException {

    int c;
    while((c=in.read())!=-1)
        out.write(c);
}
```

- Par buffer de caractère

```
public static void copy(Reader in,Writer out)
    throws IOException {

    char[] buffer=new char[8192];
    int size;
    while((size=in.read(buffer))!=-1)
        out.write(buffer,0,size);
}
```

java.lang.Appendable

- Interface générique de tout les objets qui savent écrire des chaines de caractères
 - Ajoute un caractère ou une chaine de caractère
Appendable append(char c)
Appendable append(CharSequence csq)
Appendable append(CharSequence csq, int start, int end)
- Est utilisé par le **Formatter** comme flux de caractères

java.util.Formatter

- Classe permettant de sortir vers un Appendable des chaînes de caractères en utilisant la syntaxe de printf

-

Entrée clavier/sortie console

- `System.console()` permet d'obtenir un objet `java.io.Console`
- Lecture :
 - **`readLine(String fmt, Object... args)`**
 - **`readPassword(String fmt, Object... args)`**
- Ecriture :
 - **`printf(String format, Object... args)`**
 - **`flush()`**

Exemple

- Copier l'entrée standard ou un fichier sur la sortie (comme cat)

```
public static void main(String[] args)
    throws IOException {

    Console console=System.console();
    Reader in;
    if (args.length==0)
        in=console.reader();
    else
        in=new FileReader(args[0]);

    Writer out=console.writer();
    copy(in,out);
}
```

FileReader/FileWriter

- Permet de créer un Reader sur un fichier

- **FileReader**(File file)

Renvoie l'exception **FileNotFoundException** qui hérite de **IOException** si le fichier n'existe pas

L'encodage utilisé est celui de la plateforme

- Permet de créer un Writer sur un fichier

- **FileWriter**(File name, boolean append)

append à true si l'on veut écrire à la fin

File{Reader/Writer} et encodage

- Un File{Reader/Writer} utilise toujours l'encodage de la plateforme si l'on veut utiliser un autre encodage, il faut utiliser un InputStreamReader ou OutputStreamWriter

```
public static void main(String[] args) throws ... {  
    File fileIn=new File(args[0]);  
    File fileOut=new File(args[1]);  
    Charset charsetIn=Charset.forName(args[2]);  
    Charset charsetOut=Charset.forName(args[3]);  
  
    InputStreamReader reader=new InputStreamReader(  
        new FileInputStream(fileIn),charsetIn);  
    OutputStreamWriter writer=new OutputStreamWriter(  
        new FileOutputStream(fileOut),charsetOut);  
    copy(reader,writer);  
}
```

Flots particuliers

- Il existe des flots qui ne correspondent pas à des descripteurs mais qui héritent des classes de flots
 - Les flots de chaîne de caractères
 - Les tubes (en mémoire)
 - Les flots “pushback”
 - Les flots d'écritures binaires
 - Plus quelques flots exotiques

Flots et chaîne de caractère

- Les classes `CharArrayReader`, `StringReader` et `ByteArrayInputStream` permettent de voir un tableau ou une chaîne comme un flot(supporte la mark)
- Les classes `CharArrayWriter`, `StringWriter` et `ByteArrayOutputStream` permettent d'écrire dans un tableau avec un mécanisme de flux, la taille du tableau est augmentée quand nécessaire

Flots d'une chaîne de caractères

- On récupère le tableau avec `toCharArray` ou `toByteArray`.
- Pour `CharArrayWriter` et `StringWriter`, la méthode `toString` renvoie la chaîne correspondante à ce qui a été écrit ;
- pour `StringWriter`, la méthode `getBuffer` retourne l'objet `StringBuffer` associé.

Tubes

- Les classes `Piped{InputStream, Reader, Writer}` permettent d'avoir des flux « tubes » :
 - un tube en entrée est connecté à un tube en sortie de même type (octets ou caractères) ;
 - les données écrites dans l'entrée du tube (`Writer` ou `OutputStream`) peuvent être lues dans la sortie du tube (`Reader` ou `InputStream`).
- **Utiliser une `BlockingQueue` à la place !!**

Flots « Pushback »

- Les flots `PushbackInputStream` et `PushbackReader` permettent de remettre des caractères ou octets, lus ou non, dans le flot pour les lire à nouveau ;
- on remet des caractères avec `unread` ;
- ceci est réalisé avec un buffer fini, dont la méthode `unread` lance un `IOException` quand le buffer est plein ;
- Sert pour écrire un lexer mais la doc est pas top

Flots de données

- Les flots `Data{Input/Output}Stream` servent à écrire ou lire des données binaire
- les spécifications des méthodes sont données par les interfaces `DataInput` et `DataOutput`
- l'exception `EOFException` est lancée à la fin du flot
- la méthode `readFully` permet de bloquer jusqu'à ce que soient lues le nombre d'octets demandés

Exemple

- Lire et écrire un objet particulier

```
public void writeNamedPoint(DataOutputStream out, NamedPoint p)
    throws IOException {
    out.writeUTF(p.name);
    out.writeInt(p.x);
    out.writeInt(p.y);
}

public NamedPoint readNamedPoint (DataOutputStream in)
    throws IOException {
    String name = in.readUTF();
    int x = in.readInt();
    int y = in.readInt();
    return new NamedPoint(name,x,y);
}
```

Numéro de ligne

- La classe `LineNumberReader` maintient en plus un numéro de ligne (la méthode `setLineNumber` ne change pas la position dans le flot) ;
- la classe `LineNumberInputStream` est dépréciée car n'a pas de gestion correcte des codages de caractères.

Flots d'objets

- Les flots `ObjectInputStream` et `ObjectOutputStream` permettent de stocker et restaurer des objets en binaire
- Les objets écrits dans les flots doivent implémenter l'interface marqueur `Serializable`
- Le système inclus un système de version de classe (`serialVersionUID`)
- `EOFException` est lancée à la fin du flot
- `java.lang.Object` n'est pas sérialisable.

Serialization sur un flots d'objets

- Lors du stockage d'un objet, la machine virtuelle recherche tous les objets dépendants et les stocke aussi (en repérant les boucles)
- Seuls les attributs de la classe et des superclasses sérialisables sont stockés dans le flot, sauf s'ils sont marqués **transient**
- Ces attributs doivent être eux-mêmes sérialisable ou l'exception `NotSerializableException` est levée.

Deserialization

- Le constructeur sans paramètres de la première superclasse non sérialisable est appelé (souvent **Object()** mais pas toujours), et leurs champs sont initialisés par ce dernier. Ils ne reprennent donc pas leur ancienne valeur
- `InvalidClassException` est levée si ce constructeur n'existe pas
- Puis les champs des super-classes `Serializable` sont initialisés

Deserialization (suite)

- Pour toutes les super-classes **Serializable** (la classe courante incluse)
- les champs **non transient** sont restaurés, sans appel à un constructeur,
- les champs **transient** initialisés à leur valeur par défaut (0, null, false, etc.)
- Même les champs **final** sont initialisés

Example

```
public class A {  
    int x,y;  
    public A() { this(3,3); }  
}  
public class B extends A implements Serializable {  
    transient int a = 8;  
    int z;  
}
```

```
B b=new B();  
b.x=10;b.y=11;b.a=12;b.z=13;  
out.writeObject(b);  
out.close(); // auto-flush
```

```
B b=(B)in.readObject();  
System.out.printf("x=%d y=%d a=%d  
    z=%d\n",b.x,b.y,b.z,b.a);  
in.close();
```

x=3 y=3 a=0 z=13

Flots d'objets

- La sérialisation permet aussi de transmettre des objets via le réseau, ou à une autre application Java (drag and drop, copier/coller)
- On peut ajouter des méthodes éventuellement privées à une classe, `readObject(ObjectInputStream)` et `writeObject(ObjectOutputStream)` pour mieux contrôler la sauvegarde et la restauration
- Elle seront appelées à la place du mécanisme par défaut

Créations de nouveaux flots

- Les classes abstraites `InputStream`, `OutputStream`, `Reader` et `Writer` permettent de créer facilement de nouveaux flots ;
- les classes `FilterInputStream`, `FilterOutputStream` et les classes abstraites `FilterReader` et `FilterWriter` permettent de créer de nouveaux filtres.

Accès direct à un fichier

- 3 classes
 - RandomAccessFile (bloquant)
 - Ancienne version, on doit utiliser des tableaux de bytes
 - FileChannel (bloquant)
 - Version plus récente, on utilise des ByteBuffers
 - AsynchronousFileChannel (asynchrone)
 - Utilise des ByteBuffer + les appels sont asynchrone
 - Future
 - CompletionHandler

RandomAccessFile

On peut spécifier différents modes :

- "r" lecture seule ;
- "rw" lecture et écriture, avec création du fichier ;
- "rwd" comme "rw", mais synchronisé avec le système de fichiers (sauf réseau) ;
- "rws" comme "rwd", mais les statistiques du fichier sont aussi synchronisées

Accès aux fichiers

- On se déplace dans le fichier avec la méthode **seek()**
- on accède à la position courante avec **getFilePointer()**
 - C'est la même position pour les lectures/écriture
- La longueur est gérée avec **length()/setLength()**
- on le ferme avec **close()**
- On peut écrire et lire des données binaire comme des `DataInput` et `DataOutput`

Lire/Ecrire des données binaires

- Les interfaces `DataInput/DataOutput` définissent la lecture et l'écriture de données binaires
 - `readByte()`, `readShort()`, `readInt()` etc. permettent de lire des types primitifs
 - `writeByte()`, `writeShort()`, `writeInt()` etc. permettent de lire des types primitifs
 - `readUTF()/writeUTF()` permette de lire et d'écrire dans un format UTF8 modifié (*surrogate* gérées différemment)

Création d'un FileChannel

- Creation
 - FileChannel.open(Path, OpenOption...)• CREATE (crée le fichier si nécessaire)
 - CREATE_NEW (marche pas si un fichier existe déjà)
 - APPEND (écrit à la fin)
 - TRUNCATE_EXISTING (on commence à zéro)
 - DELETE_ON_CLOSE (le fichier disparaît après close)
 - SPARSE (ne doit pas allouer la place si possible)
 - SYNC et DSYNC (syncho à chaque read/write)
- ou alors FileInputStream.getChannel() et
FileOutputStream.getChannel()

Utilisation d'un FileChannel

- Lecture bloquante
 - `read(ByteBuffer)` ou `read(ByteBuffer, position)`
- Ecriture bloquante
 - `write(ByteBuffer)` ou `write(ByteBuffer, position)`
- Demander la synchro sur le disque
 - `force(boolean metadata)`
- Un verrou sur le fichier
 - `lock()`, `lock(position, size, shared)`, `tryLock()`
- Fermer le fichier
 - `close()`

AsynchronousFileChannel

- Même operation qu'un FileChannel
(open, close, force, lock)
- Mais les read/write sont différents,
les appels ne sont pas bloquant et se feront dans le future
 - `Future<Integer> read/write(ByteBuffer, position)`
 - Le future permet d'obtenir le résultat ultérieurement
 - `void read/write(ByteBuffer, position, attachment, completionHandler)`
 - Le CompletionHandler est appelé lorsque le système a effectué l'appel

CompletionHandler

Paramétré

- par le type de la valeur de retour
(dans notre cas toujours un Integer) et
- par le type de l'attachement

```
interface CompletionHandler<V, A> {  
    void completed(V result, A attachment);  
    void failed(Throwable t, A attachment);  
}
```

completed est appelé si tout c'est bien passé,
sinon failed est appelé avec l'exception

Example

```
public static void copy(final AsynchronousFileChannel in,
                        final AsynchronousFileChannel out) {
    final ByteBuffer buffer = ByteBuffer.allocate(8192);
    class ReadCompletionHandler extends
        DefaultCompletionHandler<Integer, Integer> {
        @Override
        public void completed(Integer result, Integer position) {
            if (result == -1) {
                LATCH.countDown(); // signale que l'on a fini
                return;
            }
            buffer.flip();
            out.write(buffer, position, position + result,
                new DefaultCompletionHandler<Integer, Integer>() {
                    @Override
                    public void completed(Integer result, Integer newPosition) {
                        buffer.compact(); // on peut écrire moins que ce que l'on a lu
                        in.read(buffer, newPosition, newPosition, ReadCompletionHandler.this);
                    }
                });
        }
    }
    in.read(buffer, 0, 0, new ReadCompletionHandler());
}
```

Exemple (suite)

```
static abstract class DefaultCompletionHandler<V, A>
    implements CompletionHandler<V, A> {
    @Override
    public void failed(Throwable exc, A attachment) {
        LATCH.countDown();
        throw new IOError(exc);
    }
}
static final CountDownLatch LATCH = new CountDownLatch(1);

public static void main(String[] args)
    throws IOException, InterruptedException {
    copy(AsynchronousFileChannel.open(Paths.get("async.txt"),
        READ),
        AsynchronousFileChannel.open(Paths.get("async-out.txt"),
        WRITE, TRUNCATE_EXISTING, CREATE));
    LATCH.await();
}
```

Comme la copie est asynchrone le programme peut faire autre chose pendant ce temps