Input/Output

Rémi Forax
forax@univ-mlv.fr

java.io.File

- java.io.File représente un chemin textuel 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

Chemin vers un fichier

- Représente un chemin dans l'arboresence
  - Il existe 3 sortes de chemins :
    - Relatif ../toto/titi.txt
    - Chemin absolu /tmp/titi.txt
    - Chemin canonique (ils sont uniques) c:\Program Files\Java\
- Une représentation en String (path) ainsi qu'une représentation en File
  - Chemin absolu en String : getAbsolutePath()
  - Chemlin absolu en File : getAbsoluteFile()

File et WORA

- Constante java.io.separatorChar

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

- Création d'un chemin :
  - File(File directory,String name)
  - File(String directory,String name)
  - File(pathname)

  Le séparateur est ajouté automatiquement :

  “Write once, run anywhere”

  Il n'y a rien pour gérer les extensions !

Gestion des chemins

- Manipulation des chemins :
  - nom du fichier : **getName**()
  - chemin local vers le fichier : **getPath**()
  - chemin absolu vers le fichier : **getAbsolutePath**()/**getAbsoluteFile**()/**isAbsolute**()
  - chemin canonique : **getCanonicalPath**()/**getCanonicalFile**()
  - chemin vers le repertoire père **getParent**()/**getParentFile**()

Type de fichier

- java.io.File représente un chemin vers :
  - Une racine de l'arborescence **isRoot**()
  - Un répertoire **isDirectory**()
  - Un fichier classique ou temporaire **isFile**()
- C'est la même classe qui gère tout

Les racines

- Méthode statique qui liste les racines :
  File[] **listRoots**()

- Méthodes spécifiques (1.6) :
  - Espace totale long **getTotalSpace**()
  - Espace disponible :
    - long **getFreeSpace**()
    - long **getUsableSpace**() (regarde les quotas, etc.)

Exemple

- Liste des racines et calcul du ratio occupé/libre

```
public static void main(String[] args) {
  for(File root:File.listRoots()) {
    long usable=root.getUsableSpace();
    long total=root.getTotalSpace();
    double ratio=100.0*usable/total;
    System.out.printf("%-8s %,20d: %,20d:%2.1f%%\n",root,usable,total,ratio);
  }
}
```

```
A:\                             0:                    0: NaN%
C:\              48 452 947 968:       79 933 267 968: 60,6%
D:\                             0:                    0: NaN%
Z:\              19 809 697 792:      144 506 355 712: 13,7%
```

Les répertoires

- Savoir si un fichier est un répertoire :
  - **isDirectory**()
- Lister les fichiers présents :
  - String[] **list**() ou File[] **listFiles**()
- Il est possible de filtrer les fichiers :
  - filtre sur le nom : **listFiles**(FilenameFilter filter)
  - filtre sur le fichier : **listFiles**(FileFilter filter)

Les filtres sur les fichiers

- **FilenameFilter**
  permet de filtrer suivant le nom

```
public interface  FilenameFilter() {
  public boolean accept(File dir, String name);
}
```

- **FileFilter**
  permet de filtrer suivant le fichier (le type, la taille)

```
public interface FileFilter() {
  public boolean accept(File file);
}
```

Exemple

- Liste des fichiers avec des filtres

```
public static void main(String[] args) {
  FilenameFilter txtFilter=new FilenameFilter() {
    public boolean accept(File dir, String name) {
      return name.endsWith(".txt");
    }
  };
  File file=new File(".");
  System.out.println(Arrays.toString(
    file.listFiles(txtFilter)));

  FileFilter dirFilter=new FileFilter() {
    public boolean accept(File file) {
      return file.isDirectory();
    }
  };
  System.out.println(Arrays.toString(
      file.listFiles(dirFilter)));
}
```

Les fichiers temporaires

- Création d'un fichier temporaire
  - static File createTempFile(String prefix,String suffix,File directory)
    - suffix peut être null (“.tmp” est alors pris)
    - directory peut être null, pour le répertoire spécifique à l'OS (ex: /tmp)
- Souvent combiné avec deleteOnExit(), supression du fichier lors de la terminaison de la VM

Propriété sur les fichier

- Tous type de fichier possède les propriétés suivante :
  - Type de fichier : **isFile**()/**isDirectory**()
  - Fichier caché : **isHidden**()
  - Date de dernière modification : **lastModified**(), **setLastModified**()
  - Longueur en octet : **length**()

FileSystemView

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

Exemple

- Affichage d'info supplémentatire 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
```

Droit sur les fichiers

- Droit aux fichiers :
  - En exécution, canExecute()/setExecutable(boolean)
  - En lecture, canRead()/setReadable(boolean)
  - En écriture, canWrite()/setWritable(boolean);
- Les méthodes set ont une version + complète : **setExecutable**(boolean executable, boolean ownerOnly) qui permet de changer pour tout le monde ou le possesseur

Opérations sur les fichiers

- Opérations sur les fichiers :
  - Création d'un fichier vide : **createNewFile**()
  - Supression d'un fichier : **delete**()
  - Supression à la terminaison de la VM: **deleteOnExit**()
  - Création de répertoires : **mkdir**()/**mkdirs**()
  - Rénomage d'un fichier : **renameTo**(File)
- ToURI() : Uniform Resource Identifiers

Accès aux fichiers

- On peut lire et écrire dans un fichier avec la classe 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**()
- 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 UTF modifié (*surrogate* gérées différemment du UTF-8)

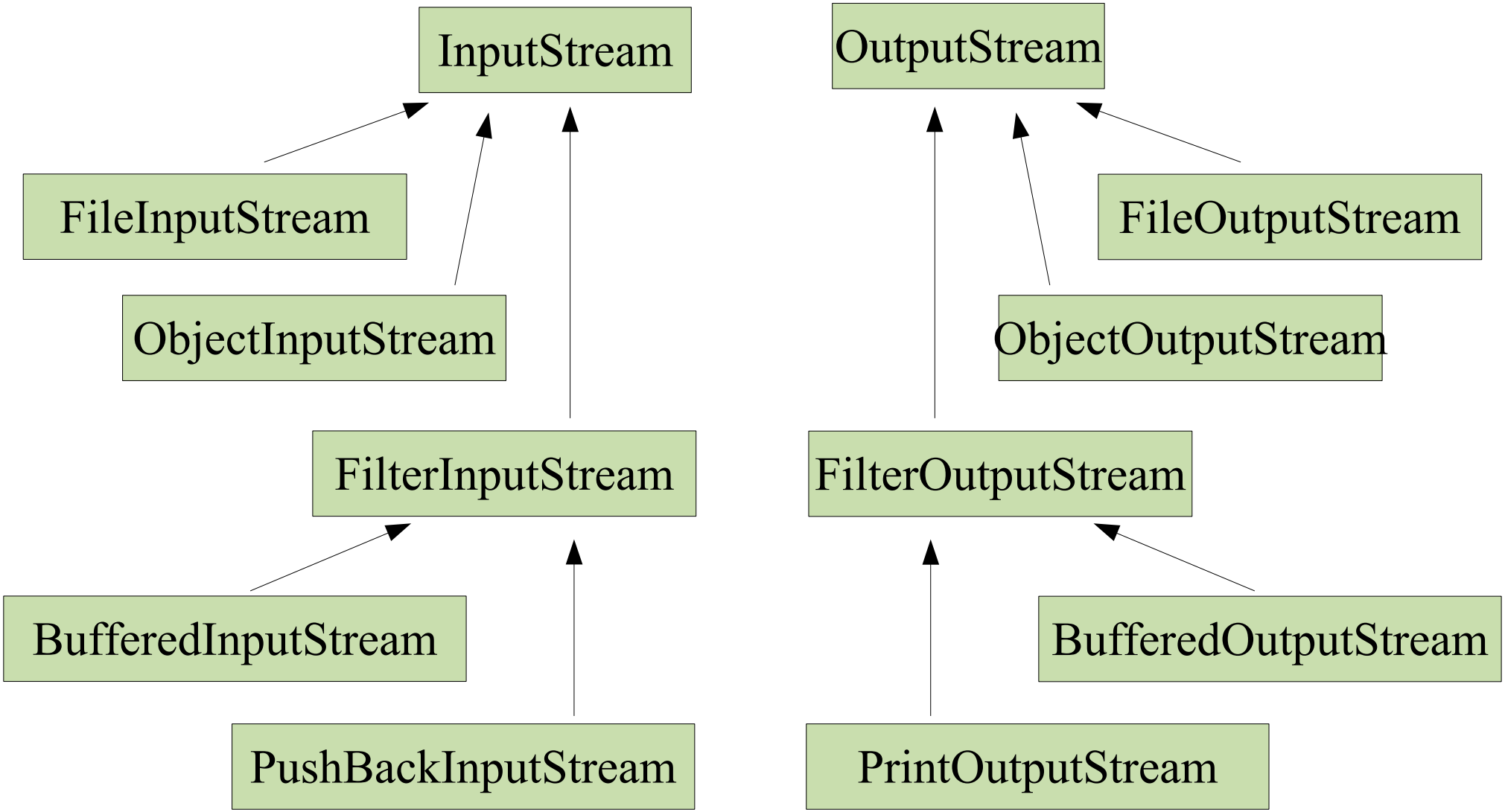
Entrée sortie et Unicode

- 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 !!)

2 versions des Entrées/sorties

- **InputStream**/**OutputStream** manipule des octets (byte en Java) donc dépendant de la plateforme
- **Reader**/**Writer** manipule des caractère (char en Java) indépendant de la plateforme mais qui nécessite d'indiquer un codages 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 censée renvoyer le nombre de byte lisibles sans que la lecture sur le flux soit bloquée
  mais mal supportée 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 problèmes de performances si on lit les données octet par octet
- Solution :
  - Lire en utilisant un buffer
  - Utiliser un BufferedInputStream qui utilise un buffer intermédiare

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 **InterruptedIOException** (cf cours sur la concurrence)
- Il faut penser un faire un **close**() sur le stream dans ce cas

InputStream et la mark

- mark : index indiquant un endroit où 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 seront lus 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**

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

- La console ne transmet les bytes que ligne par ligne

FileInputStream/FileOutputStream

- Permet de créer un InputStream sur un fichier
  - **FileInputStream**(String name)
  - **FileInputStream**(File file)

  Renvoie l'exception **FileNotFoundException** qui hérite de **IOException** si le fichier n'existe pas
- Permet de créer un OutputStream sur un fichier
  - **FileOutputStream**(String name, boolean append)
  - **FileOutputStream**(File file, boolean append)

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

Closeable

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

```
public interface Closeable {
  public void close();
}
```

- 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=new FileInputStream(args[0]);
  try {
    OutputStream out=new FileOutputStream(arg[1]);
    try {
      copy(in,System.out);
    } finally {
      out.close();
    }
  }
  finally {
    in.close();
  }
}
```

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

Bufferisation automatique

- BufferedInputStream/BufferedOuputStream 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 de créer des buffered de buffered de buffered ...

Flush et close

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

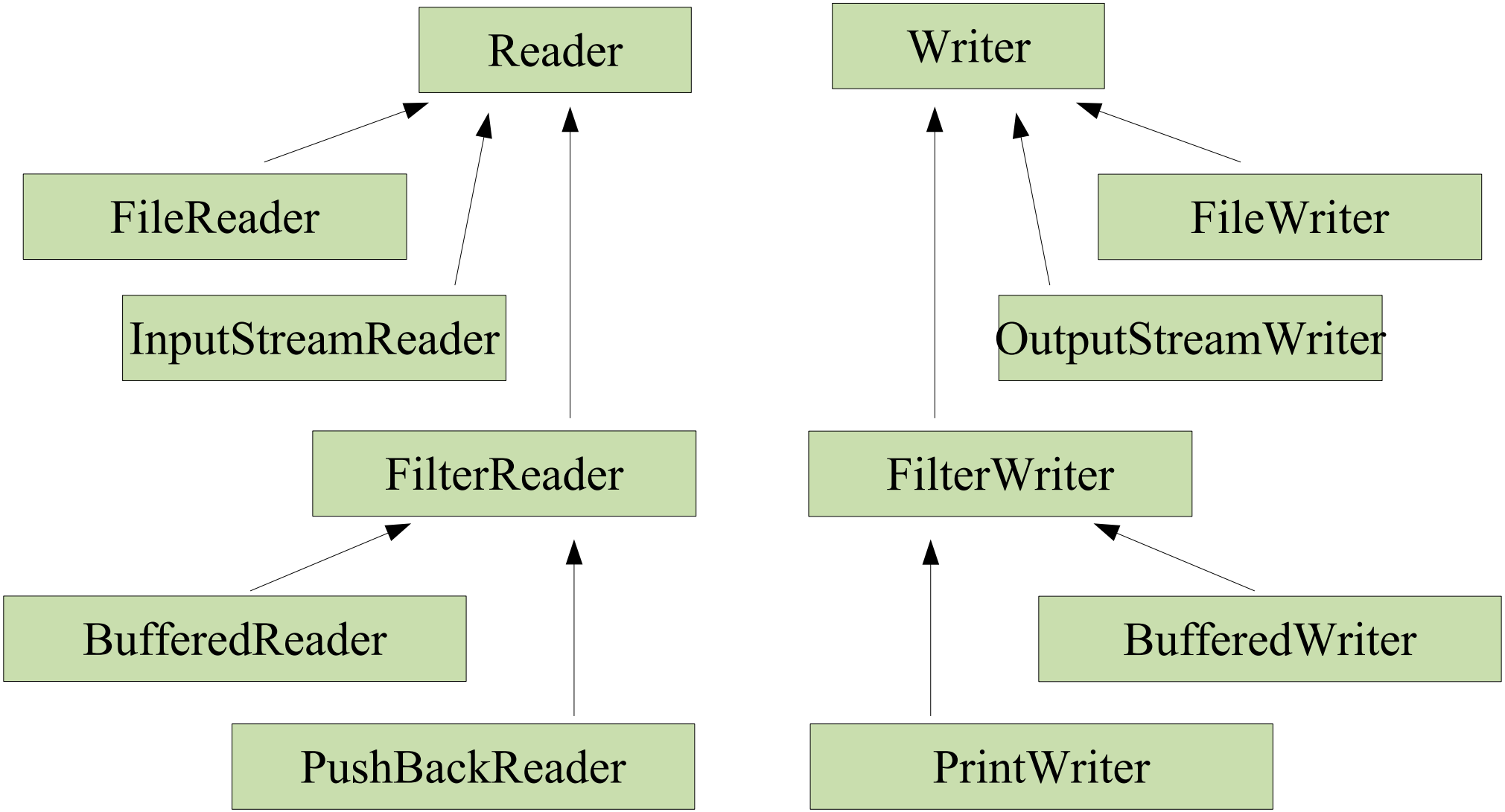
```
public static void copy(Inputstream in,OutputStream out)
  throws IOException {

  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
  output.close(); // ferme aussi in
  input.close();  // ferme aussi out
}
```

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.

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 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 chaine de caractère

- Un Writer possède des méthodes spécials 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 caractères

```
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)
- On peut utiliser un java.util.**Formatter** pour construire un Appendable à partir de données avec une syntaxe similaire à 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 veux utiliser un autre encodage, il faut utiliser un InputStreamReader ou OutputStreamWriter (*cf.* Charset dans les nio)

```
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 corresponde pas à des descripteurs mais qui hérite 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 soit lu le nombre d'octets demandés

Scanner

- La classe **Scanner** permet de parser un document en récupérant les valeurs de données
  - **Scanner(InputStream source)**
  - **Scanner(File file)**
- Un scanner est un itérateur de String qui possède aussi des méthodes **next{Int,Float,Double,...}()** et **hasNext{Boolean,Byte,...}()**
- Le scanner intercepte les **IOException** : hasNext() retourne alors faux et l'exception est récupérable par **ioException()**
- Un scanner doit être fermé par **close()**

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.

Flot de flots

- La classe SequenceInputStream prend un ensemble d'InputStream, et émule la concaténation de ces flots :
  - il retourne d'abord les octets du premier jusqu'à la fin ;
  - puis les octets des flots suivants, jusqu'à arriver à la fin du dernier flot.

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 sérialisables ou l'exception NotSerializableException est levée.

Deserialization

- Le constructeurs 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 inclue)
- 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

Exemple

```
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.