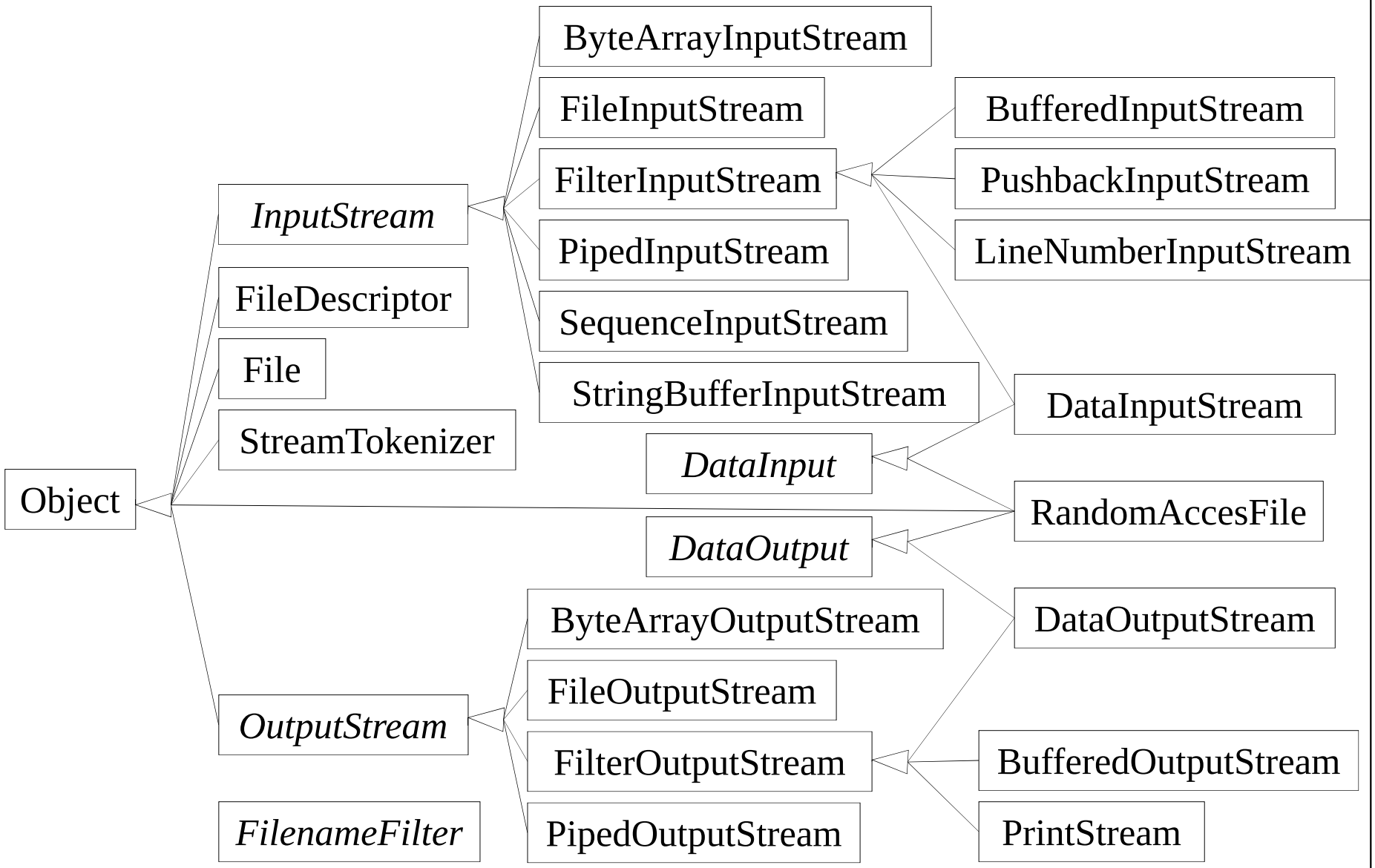
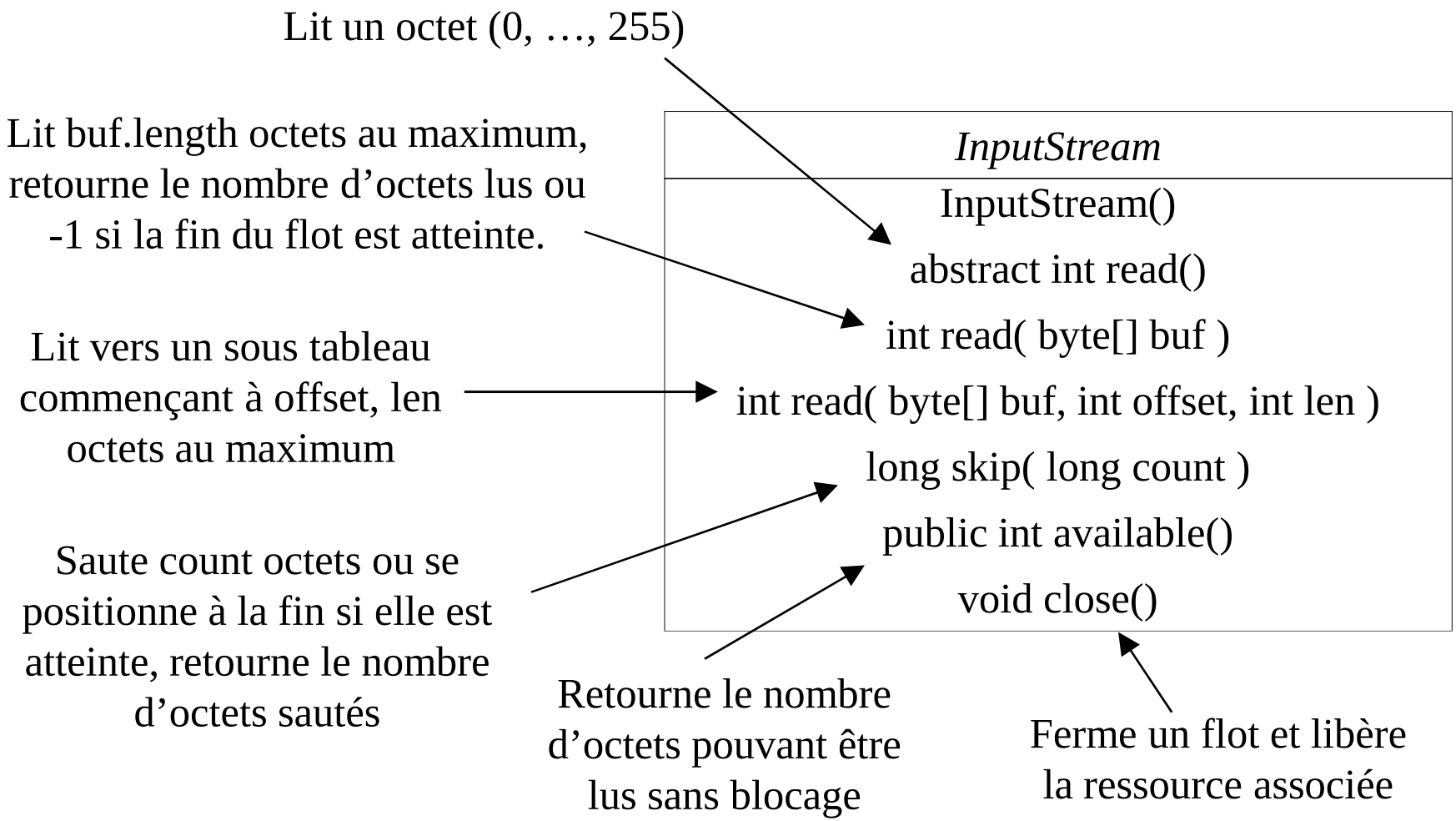
# Les entrées-sorties

# Les pricipales classes du paquetage java.io



# La classe abstraite InputStream

- les méthodes read se bloquent tant que l'entrée n'est pas disponible ;
- toutes les méthodes lancent IOExeption.



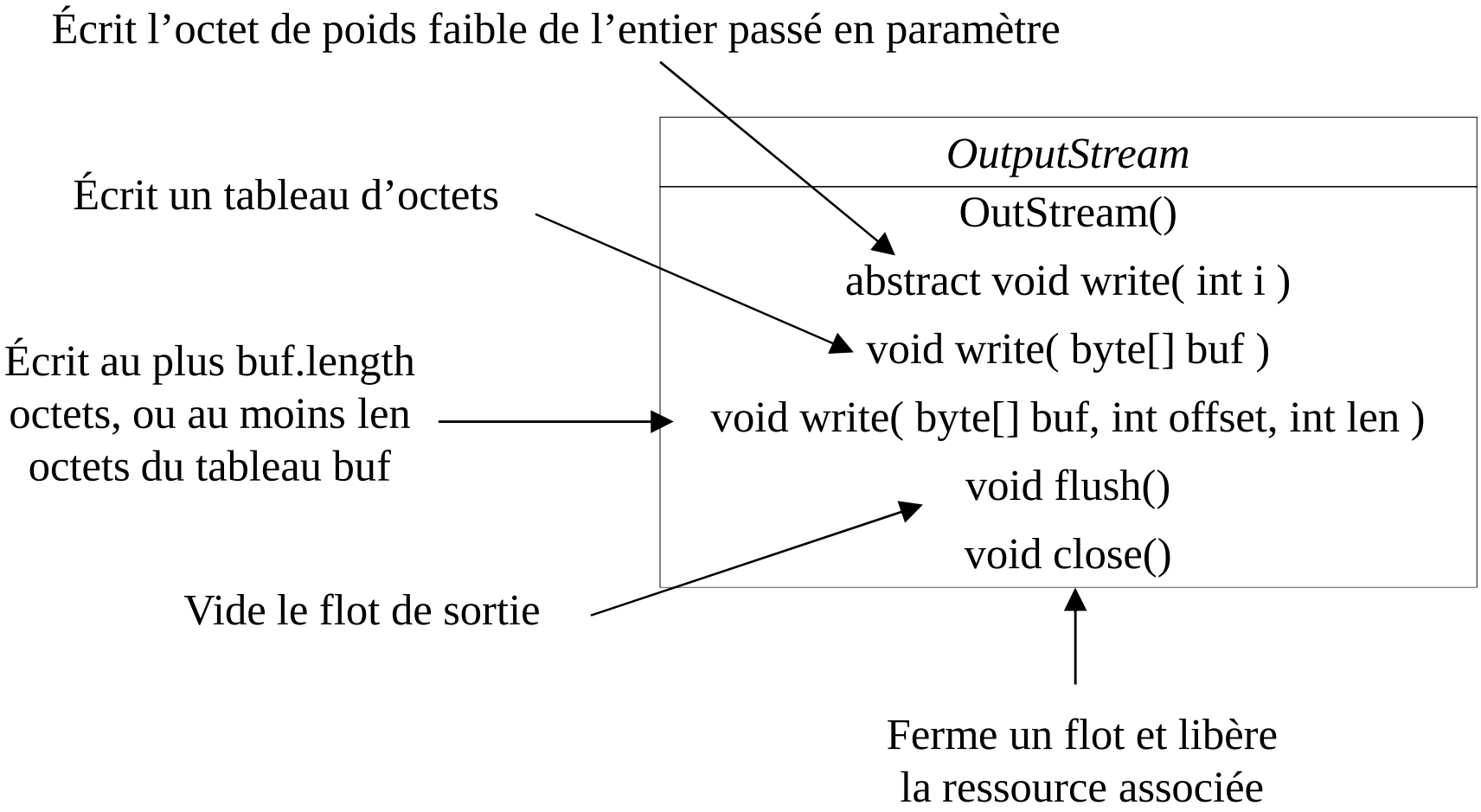
# La classe abstraite InputStream

- La classe java.lang.System offre un accès à :
  - l'entrée standard System.in ;
  - la sortie standard System.out ;
  - la sortie d'erreur standard System.err.

```
try{
        int i = Systen.in.read() ;
        byte b = (byte)i ;
        int nbOk = System.in.available() ;          // nombre d'octets disponibles
        if( nbOk > 0 ){
                byte [] données = new byte[ nbOk ] ;
                System.in.read( données ) ;
} catch( IOException e ) {
        System.out.println( "Exception: " + e ) ;
}
```

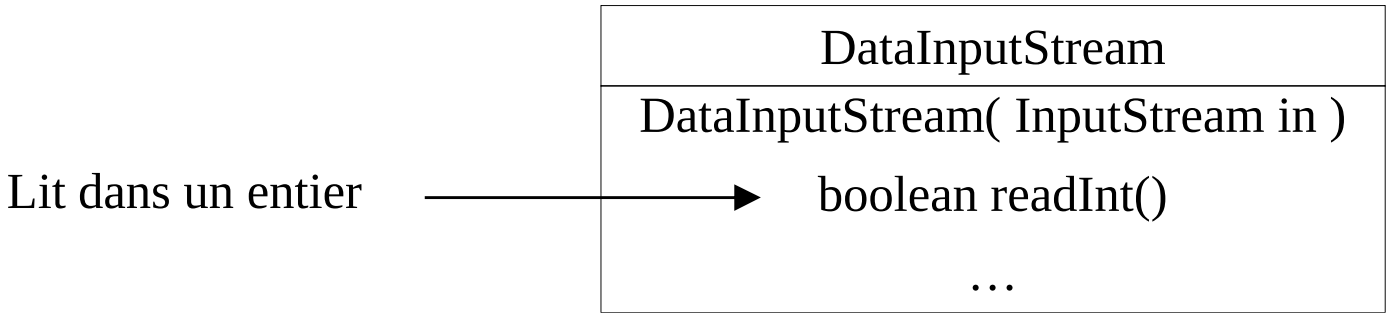
# La classe abstraite OutputStream

- les méthodes write se bloquent tant que l'écriture n'a pas été réalisée ;
- toutes les méthodes lancent IOExeption.



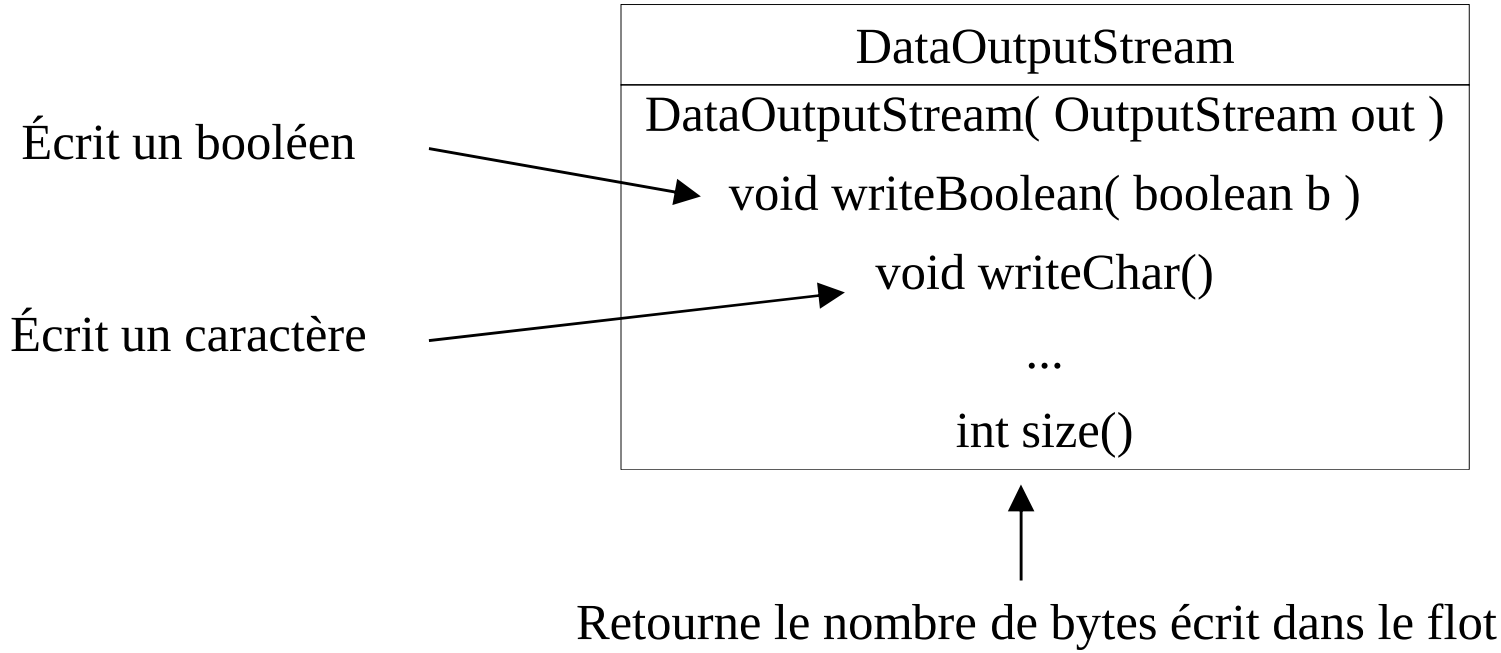
# La classe DataInputStream

- Comment lire des chaînes de caractères et obtenir des types de bases de java ?



# La classe DataOutputStream

- Comment écrire des données d'un type de base dans un flot ?




- Toutes les méthodes lancent IOExeption.

# La gestion des fichiers

- Les classes FileInputStream et FileOutputStream permettent de lire et d'écrire dans des fichiers :

```
FileInputStream fichier = new FileInputStream( "bidon" ) ;
tailleFichier = fichier.available() ;                    // récupère la taille du fichier
byte []contenuFichier = new byte[ tailleFichier ] ;      // alloue à la bonne taille
for( int i=0; i<tailleFichier; i++ ){
        contenuFichier[ i ] = (byte)fichier.read() ;
}
fichier.close() ;
```

# Les entrées / sorties bufferisées (1/2)

- Ces classes utilisent un buffer pour réduire le nombre de lecture et d'écriture réellement effectuées dans le flux ;

- ces classes sont souvent utilisées avec des fichiers :

```
OutputStream FichierBufferisée( String nom ) throws IOException{
        OutputStream out = new FileOutputStream( nom ) ;
        return new BufferedOutputStream( out ) ;              // buffer de 512 bytes
}
```

- On peut préciser la taille du buffer dans le constructeur :

```
BufferedInputStream( InputStream in, int size)
```

# Les entrées / sorties bufferisées (2/2)

- Exemple de la lecture d'un entier au clavier :

```
BufferedReader buf = new BufferedReader( new InputStreamReader(System.in) ) ;
try{
        int i = Integer.parseInt( buf.readLine() ) ;
} catch( NumberFormatException e ){
}
```

- Exemple de la lecture ligne par ligne d'un fichier :

```
BufferedReader in = new BufferedReader( new FileReader( "bidon.txt" ) );
String line ;
while( (line=in.readLine()) != null ){
        System.out.println( line ) ;
}
in.close() ;
```

# L’accès au système de fichier

# La classe File

- La classe File est une représentation abstraite du système de gestion de fichiers :

```
BufferedReader buf = new BufferedReader( new InputStreamReader(System.in) ) ;
try{
        int i = Integer.parseInt( buf.readLine() ) ;
} catch( NumberFormatException e ){
}
```

- Exemple de la lecture ligne par ligne d'un fichier :

```
BufferedReader in = new BufferedReader( new FileReader( "bidon.txt" ) );
String line ;
while( (line=in.readLine()) != null ){
        System.out.println( line ) ;
}
in.close() ;
```

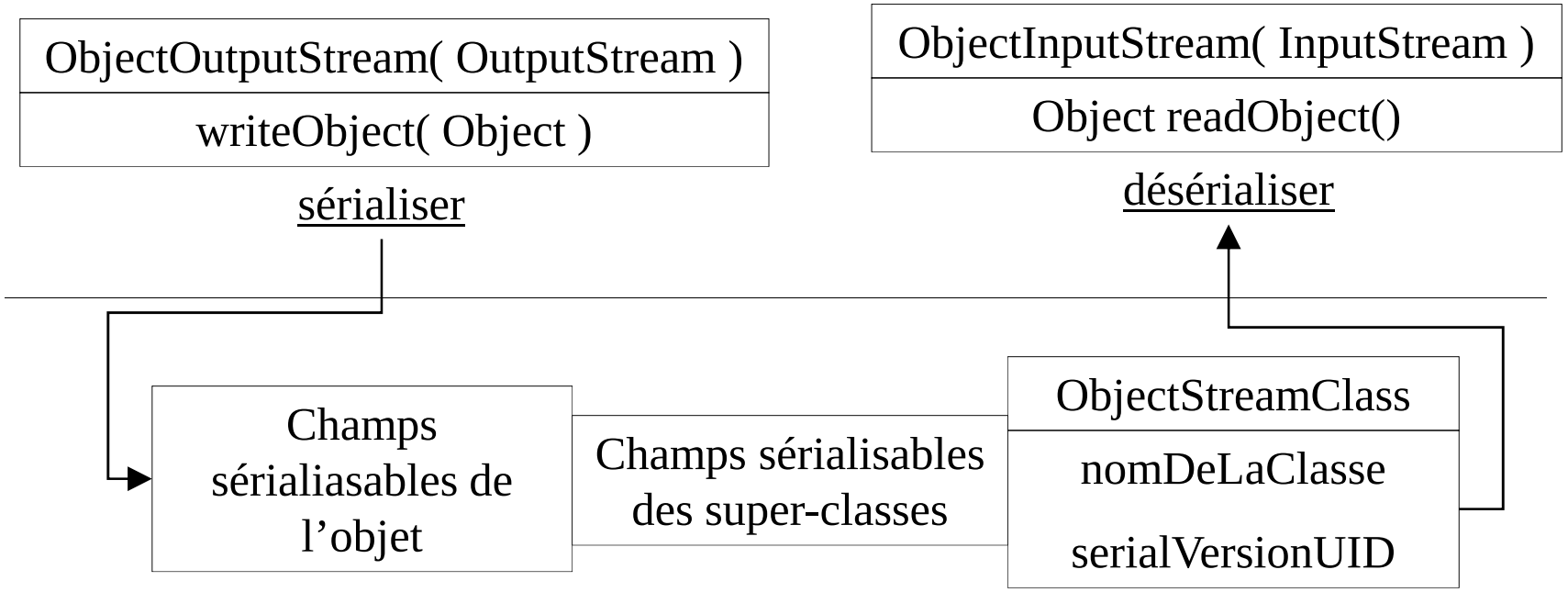
# La sérialisation d’objets

# La sérialisation d'objets

Intérêt de la sérialisation :

- les applications réparties sur plusieurs machines ont besoin de s'échanger des objets ;
- l'état des objets d'une application (valeur instantanée des attributs des objets) doit pouvoir être sauvegardé (stockage sur disque par exemple) et restitué.

L'API de sérialisation :



# La sérialisation d'objets

- tous les types de bases ainsi que la plupart des objets des librairies peuvent être sérialisés ;
- un champs statique ne sera pas sérialisé ;
- pour empêcher qu'un champs soit sérialisé on peut le déclarer avec le mot clef **transient** ;
- pour indiquer qu'une classe peut être sérialisée, elle doit implémenter l'interface java.io.Serializable :

```
class Bidon implements java.io.Serializable {
    // …
}
```

- pour sérialiser un objet pointant sur des ressources système (un descripteur de fichier par exemple), il est possible de surcharger les méthodes writeObject et readObject (pour fermer et ouvrir correctement les fichiers) ;
- pour désérialiser un objet (readObject), il faut que la classe correspondante puisse être chargée.

# Exemple de sérialisation dans un fichier

```
FileOutputStream monFichier = new FileOutputStream( "fichier.tmp" ) ;
// diriger le flux de sérialisation vers un fichier
ObjectOutputStream fluxObjet = new ObjectOutputStream( monFichier ) ;
monFlux.writeObject(  "aujourd'hui" ) ;          // sérialise une chaîne
monFlux.writeObject( new Date() ) ;              // sérialise une date
monFlux.flush() ;                                // vide le flux
monFlux.close() ;
monFichier.close() ;
```

# La désérialisation à partir d'un fichier

```
FileInputStream monFichier = new FileInputStream( "fichier.tmp" ) ;
// diriger le fichier vers le flux de désérialisation
ObjectInputStream fluxObjet = new ObjectInputStream( monFichier ) ;
String chaine = (String)monFlux.readObject() ;          // désérialise la chaîne
Date date = (Date)monFlux.readObject() ;                // désérialise une date
```