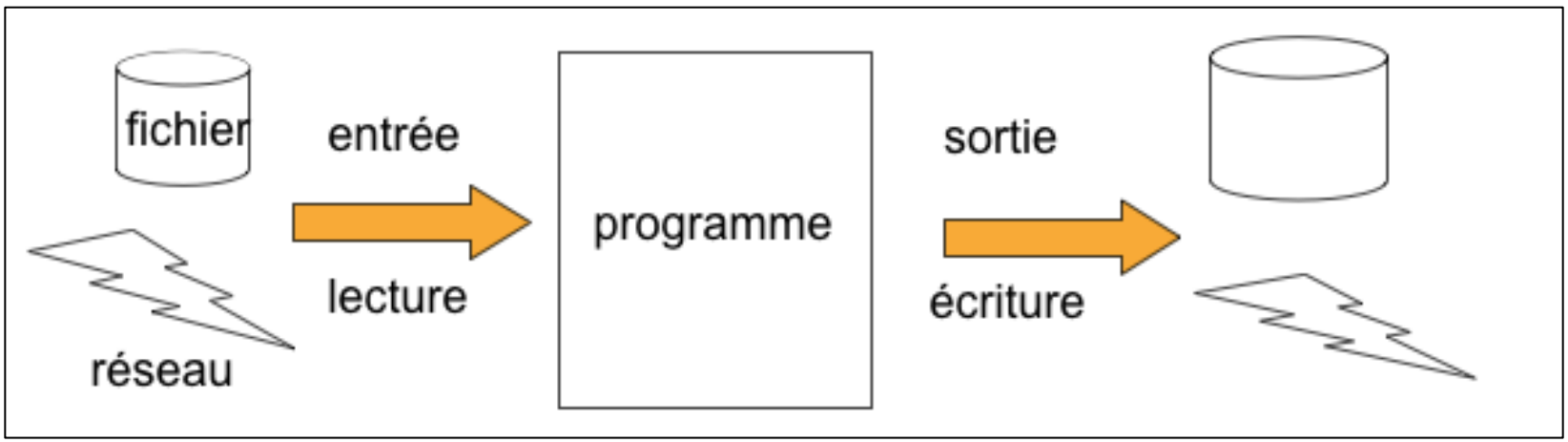
Le système d'Entrée/Sortie de Java

Introduction

- Un programme a souvent besoin de lire des données depuis une source externe ou d'écrire des données vers un destinataire externe.
- La création d'un bon système d'entrée/sortie pour une langage de programmation est l'une des tâches les plus difficiles.
- il existe différentes sources de I / O que nous souhaitons communiquer avec
  - fichiers,
  - la console,
  - les connexions réseau, etc).
- nous avons besoin de parler avec eux dans une grande variété de moyens (
  - séquentiel,
  - accès aléatoire,
  - tamponné,
  - binaire,
  - caractère, par lignes, par mots, etc.).

- Les concepteurs de la bibliothèque Java attaqué à ce problème en créant beaucoup de classes.
- Depuis Java 1.0, il contient les classes *io* pour traiter de flots d'octets et pour traiter de flots de caractères
- En JDK 1.4, les classes *nio* ont été ajoutés pour améliorer les performances et la fonctionnalité.



E/S en Java

- Java.io : Bibliothèque de classes et d'interfaces gérant les E/S
- Les bibliothèques d'E/S utilisent souvent l'abstraction d'un flux [stream]
- Les flux (streams) traitent toujours les données de façon séquentielle
  - à l'ouverture du flux, la position courante est définie (ex: au début d'un fichier ouvert en lecture);
  - dès qu'une donnée est lue ou écrite, la position courante avance; le flux n'a aucune mémoire et donne toujours accès à une seule donnée à un instant donné.

Java.io : les principales

- *Java.io* : deux familles de flux

  - les flux d'octets *(nombres entiers/réels, images, sons…)* en lecture (entrée) et en écriture (sortie)
  - les flux de caractères *(textes, html, xml…)* en lecture (entrée) et en écriture (sortie)

Java.io : les principales

- Package java.io : toutes les classes de gestion des flux
- Les noms des classes se décomposent en un *préfixe* suivi d'un *suffixe*.
- Suffixes (classes **abstraites** de flux):

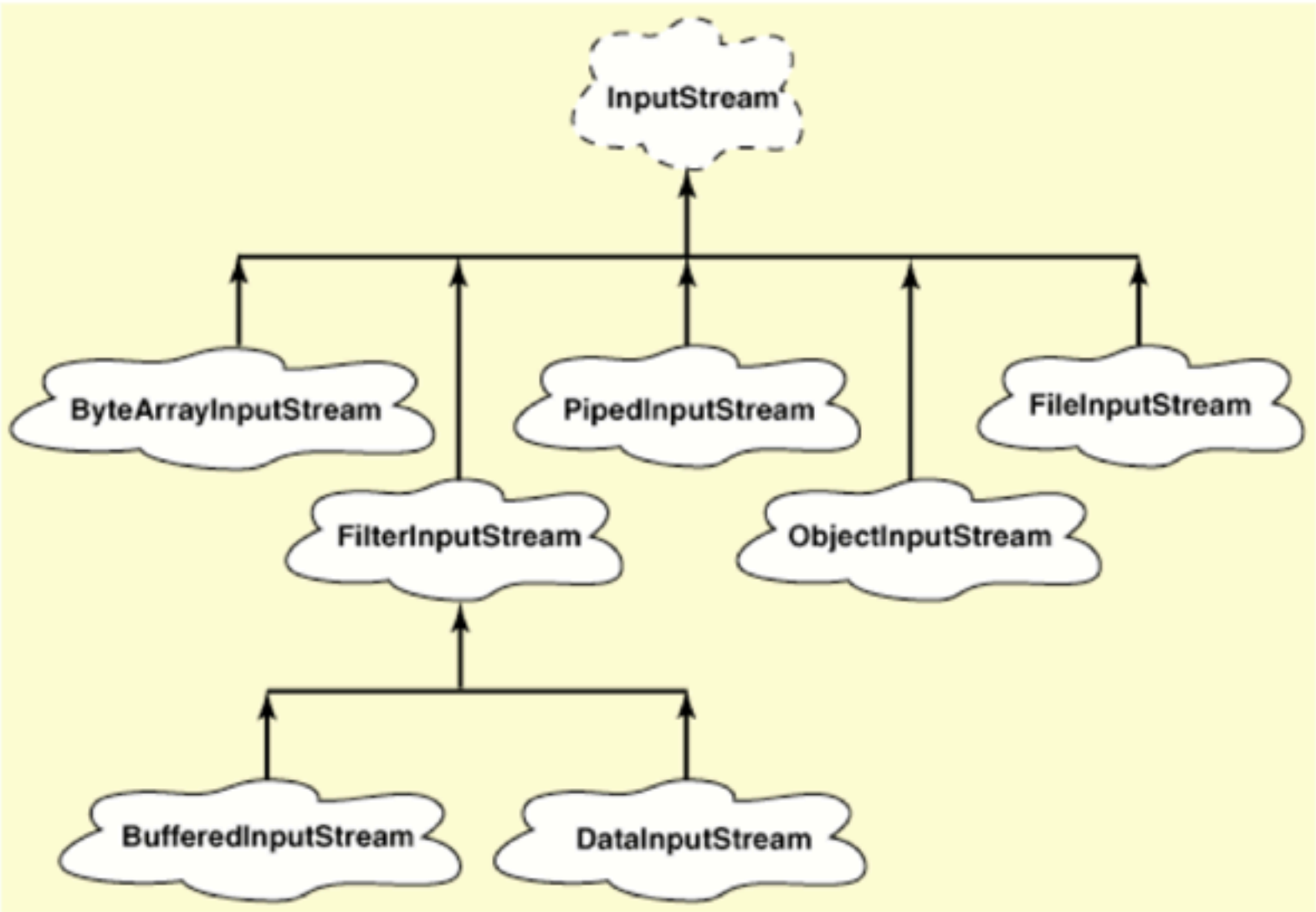
	Flux de caractères	*Flux d'octets*
Flux entrée	***Reader***	***InputStream***
Flux sortie	***Writer***	***OutputStream***

- Préfixes: indiquent **avec quoi le flux communique** (***File, Object, String***) ou le type de traitements (**filtre**) associé (***Buffered, Data***…).

Les différents filtres

Type de traitement	*description*	*préfixe*	*en entrée*	*en sortie*
mise en tampon	les données du flux transitent par un **tampon** (buffer) pour optimiser les lectures/écritures	**Buffered**	oui	oui
conversion de données	traitement des octets sous forme de **données typées (double, int, …)**	**Data**	oui pour les flux d'octets	oui pour les flux d'octets
impression	impression formatée	**Print**	non	oui
sérialisation	traitement des octets sous forme d'**objets**	**Object**	oui pour les flux d'octets	oui pour les flux d'octets

Flux d'octets : InputStream



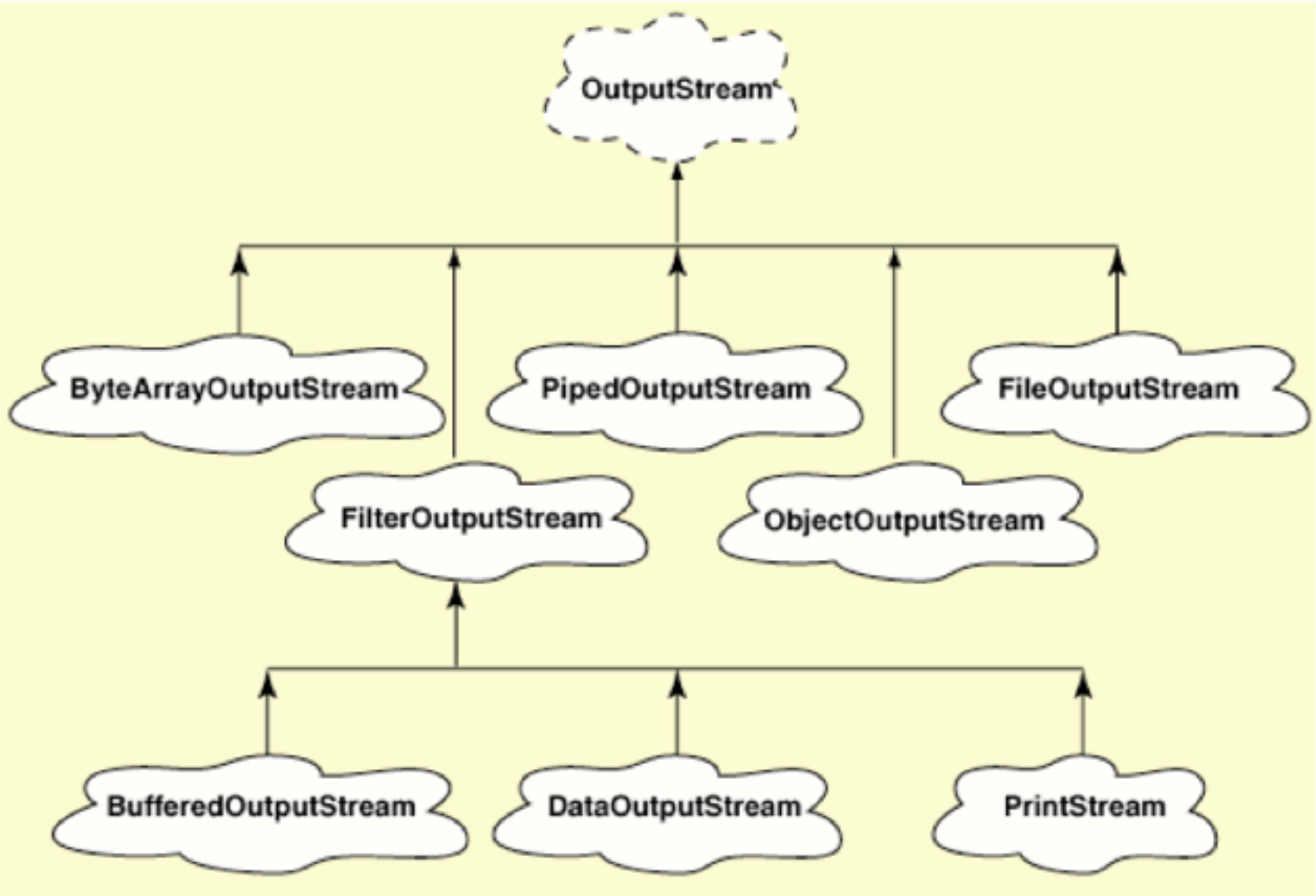
Flux d'octets : InputStream

- *Classe abstraite*
- *Transportent des données binaires en lecture sous forme d'octets. Ils peuvent traiter toutes les données.*
- *Les classes qui gèrent les flux d'octets héritent d'une des classes abstraites InputStream.*

Méthodes	Description
`public abstract int read()`	retourne l'octet lu ou -1 si la fin de la source de données est atteinte. Méthode à définir dans les sous-classes concrètes
`int read(byte[] b)`	remplit un tableau d'octets et retourne le nombre d'octets lu
`int read (byte [] b, int off, int len)`	remplit un tableau d'octets à partir d'une position donnée et sur une longueur donnée
`void close()`	ferme le flux
`int available()`	retourne le nombre d'octets prêts à être lus dans le flux
`long skip(long n)`	ignore un certain nombre d'octets en provenance du flot et renvoie le nombre d'octets effectivement ignorés.

Ces méthodes peuvent lever des IOException.

Flux d'octets : OutputStream



Flux d'octets : OutputStream

- *Classe abstraite*
- *Transportent des données binaires en ecriture sous forme d'octets. Ils peuvent traiter toutes les données.*
- *Les classes qui gèrent les flux d'octets héritent d'une des classes abstraites outputStream.*

Méthodes	Description
`public abstract void write(int)`	écrit l'octet passé en paramètre
`void write(byte[] b)`	écrit les octets lus depuis le tableau d'octets `b`
`voidread (byte [] b, int off, int len)`	écrit les octets lus depuis le tableau d'octets `b` à partir d'une position donnée `off` et sur une longueur donnée `len`
`void close()`	ferme le flux
`void flush()`	permet de purger le tampon pour les écritures bufferisées

Ces méthodes peuvent lever des IOException.

Flux d'octet

- **La classe concrète *FileInputStream***
  - *flux d'octets en lecture sur un fichier*
  - *Plusieurs constructeurs qui peuvent lever une FileNotFound Exception.*

```
FileInputStream fich = new FileInputStream ("monfichier.dat");
```

- **La classe concrète FileOutputStream**
  - *en écriture sur un fichier*
  - *si le nom du fichier n'existe pas alors le fichier est créé; sinon, le flux est ouvert en écriture vers ce fichier (dont les données sont écrasées).*

```
FileOutputStream fich = new FileOutputStream("monfichier.dat");
```

Exemple

★ **On désire copier d'un fichier quelconque octet par octet.**

La première chose à faire est de se mettre dans une structure `try-catch` car presque toutes les opérations d'entrée-sortie peuvent lever des exceptions :

```
try {
   ...
} catch (Exception e) {
   e.printStackTrace() ;
}
```

Exemple

```
import java.io.*;
public class CopieFichier {
    private String source, destination;
    public CopieFichier(String src, String dest) {
        this.source = src; this.destination = dest;
    }
    public static void main(String[ ] args) {
        new CopieFichier("source","copie").copie( );
    }
    public void copie( ) {
        try {
            FileInputStream fis = new FileInputStream(source);
            FileOutputStream fos = new FileOutputStream(destination);
            int oc=fis.read( );
            while(oc != -1) {
                fos.write(oc);
                oc=fis.read( );
            }
            fis.close( ); fos.close( );
         } catch (IOException e) {
             e.printStackTrace( );
} } }
```

Bufferisation en entrée

- *Imaginons un programme qui lit ou écrit des octets dans un fichier sur un support matériel (disque dur, clé USB, etc.).*
- *Si chaque instruction de lecture ou d'écriture provoque la mise en marche du matériel, le programme sera lent.*
- *On règle ce problème par l'utilisation d'une zone de mémoire, intermédiaire entre le programme et le flux, appelée tampon, buffer en anglais.*
  - Quand un byte doit être lu, il est pris dans le buffer.
  - Si le buffer est vide, on le remplit à partir du disque dur en une seule opération de lecture.
  - Cela réduit le nombre d'opérations physiques de lecture.
  - La taille du buffer est un facteur. Java fournit des tailles *par défaut*.

Bufferisation en entrée

★ La classe concrète *BufferedInputStream*

- *Pour bufferiser un flux d'entrée d'octets en sortie, on encapsule ce flux dans un BufferedInputSream :*

```
FileInputStream     fos = new FileInputStream("datafile") ;
BufferedInputStream bos = new BufferedInputStream(fos) ;
```

- *Puis le BufferedInputSream s'utilise comme le FileInputStream mais il réalise l'opération de bufferisation.*

- Dans cette exemple, on a utilisé ***de flux filtrés***

Bufferisation en sortie

★ **La classe concrète *BufferedOutputSream***

- *Pour bufferiser un flux de sortie d'octets, on encapsule ce flux dans un BufferedOutputSream :*

```
FileOutputStream     fos = new FileOutputStream("datafile") ;
BufferedOutputStream bos = new BufferedOutputStream(fos) ;
```

- *Puis le BufferedOutputSream s'utilise comme le FileOutputStream mais il réalise l'opération de bufferisation.*

- Dans cette exemple, on a utilisé ***de flux filtrés***

Empilement de flux filtrés

- **En Java, chaque type de flux est destiné à réaliser une tâche.**
- **Lorsque le programmeur souhaite un flux qui ait un comportement plus complexe**
  - "empile", à la façon des poupées russes, plusieurs flux ayant des comportements plus élémentaires.
    - On parle de flux filtrés.
  - Concrètement, il s'agit de passer, dans le constructeur d'un flux, un autre flux déjà existant pour combiner leurs caractéristiques.

Les flux de données binaires

- Lire un flux d'octets octet par octet n'est pas toujours pratique. Les classes DataInputStream et DataOutputStream permettent de lire/écrire directement des données typées entières, réelles, booléennes...

★ La classe concrète *DataInputStream*

- *flux de données typées en lecture*
- *Constructeur:* crée un flux de données typées en lecture à partir du flux d'octets en paramètre.

```
DataInputStream fich = new DataInputStream(new FileInputStream("monfichier.dat"));
```

- *Méthodes:*

boolean readBoolean() : lit un booléen (un octet).

int readInt() : lit un entier (4 octets).

double readDouble() : lit un réel (8 octets).

*Ces méthodes peuvent lever des **EOFException si la fin de flux a été atteinte** ou des IOException si une erreur d'E/S s'est produite.*

Les flux de données binaires

★ La classe concrète *DataOutputStream*

- *flux de données typées en écriture :*

- *Constructeur:* crée un flux de données typées en écriture à partir du flux d'octets en paramètre.

```
DataOutputStream fich = new DataOutputStream(new
                                    FileOutputStream("monfichier.out"));
```

- *Méthodes:*

void writeBoolean(boolean) : écrit le booléen en paramètre.

void writeInt(int) : écrit l'entier en paramètre.

void writeDouble(double) : écrit le réel en paramètre.

Ces méthodes peuvent lever des IOException.

Les flux d'objets – la sérialisation

- *La sérialisation permet d'écrire (puis lire) des objets dans des flux d'octets. Java transforme l'objet et tous les objets qui lui sont liés en une suite d'octets qui peut à son tour être réinterprétée comme l'objet initial.*
- *C'est au programmeur de vérifier que les objets qu'il manipule sont sérialisables (c.à.d. que la transformation peut leur être appliquée). Le programmeur l'indique en précisant que la classe implémente l'interface prédéfini Serializable.*
- *Un objet est sérialisable si tous ses attributs d'instance sont :*
  - *soit d'un type de base de Java (int, double…).*
  - *soit d'un type objet sérialisable.*

Les flux d'objets – la sérialisation

Exemple

```
public class A {
    private int x;     // A est sérialisable
}
public class B {
    private double v;
    private A refA;       // B est sérialisable (car A l'est)
}
```

Pour permettre à Java de sérialiser (transformer en octets) les objets de A et B il faut écrire :

```
public class A implements Serializable {
    private int x;
}
public class B implements Serializable {
    private double v;
    private A refA;
}
```

Les flux d'objets – la sérialisation

- On peut lire et écrire des objets sérialisables dans des flux d'octets grâce aux classes ObjectInputStream et ObjectOutputStream.

★ La classe concrète *ObjectInputStream*

- *flux d'objet en lecture*
- *Constructeur:* crée un flux d'objet en lecture à partir du flux d'octets en paramètre.

ObjectInputStream (InputStream)

- *Méthodes:*

Object readObject() : lit un objet (taille quelconque).

Le type retourné est *Object.* Il faut en général "*caster*" pour utiliser l'objet dans son type d'origine.

Peut lever des exceptions liées aux E/S (IOException) ou à la sérialisation (attrapées par Exception).

Les flux d'objets – la sérialisation

★ La classe concrète *ObjectOutputStream*

- *flux d'objet en écriture*
- *Constructeur:* crée un flux d'objet en écriture à partir du flux d'octets en paramètre.

ObjectOutputStream (OutputStream)

- *Méthodes:*

  Object writeObject() : écrit un objet via la sérialisation (taille quelconque).

Il faut noter que le type du paramètre est *Object,* ce qui permet d'écrire (par sous-typage) n'importe quel type d'objet.

Peut lever des exceptions liées aux E/S (IOException) ou à la sérialisation

Les flux d'objets – la sérialisation

Exemple : écriture d'un point et relecture

```
import java.io.*;
class UnPoint implements Serializable {
   private int abscisse,ordonnee;
   public UnPoint(int x,int y) {
       abscisse=x;
       ordonnee=y;
   }
}
public class TestObjectStreams {
   private UnPoint point;
   private String nomf;                                  // nom fichier
   public TestObjectStreams(UnPoint pt,String nf) {
      this.point = pt;
      this.nomf = nf;
   }
   public static void main(String[] args) {
      TestObjectStreams tos=new TestObjectStreams(new UnPoint(20,30), "ficobj");
       tos.sauve();
       tos.charge();
   }
```

Les flux d'objets – la sérialisation

```
public void sauve() {
   try {
      ObjectOutputStream oos=new ObjectOutputStream(new
                                                FileOutputStream(nomf));
      oos.writeObject(point);
      oos.close();
   } catch (IOException e) {
      System.out.println("erreur d'E/S");
   } catch (Exception e) {
      System.out.println("erreur hors E/S");
   }
}
public void charge() {
   try {
      ObjectInputStream ois=new ObjectInputStream(new
                                                FileInputStream(nomf));
      point=(UnPoint)(ois.readObject());
      ois.close();
    } catch (IOException e) {
      System.out.println("erreur d'E/S");
    } catch (Exception e) {
      System.out.println("erreur hors E/S");
    }
} }
```

Les flux de caractères

- *Lorsque l'on parle de caractères, on est obligé de parler de jeux de caractères (charset) et de codage de caractères (encoding).*
- *Le jeu de caractères est la liste des symboles que l'on considère comme étant des caractères.*
- *Le codage est la manière dont les caractères sont représentés sous forme d'octets.*
- *Un jeu de caractères contient des symboles visibles mais aussi invisibles comme par exemple le caractère « saut de ligne ».*
- *En mémoire, Java utilise le jeu de caractères UNICODE qui vise à donner à tout caractère de n'importe quel système d'écriture un nom. L'encodage sous forme d'octets (le plus courant) des numéros de caractères UNICODE est défini dans la norme UTF-8 (Universal Transformation Format).*

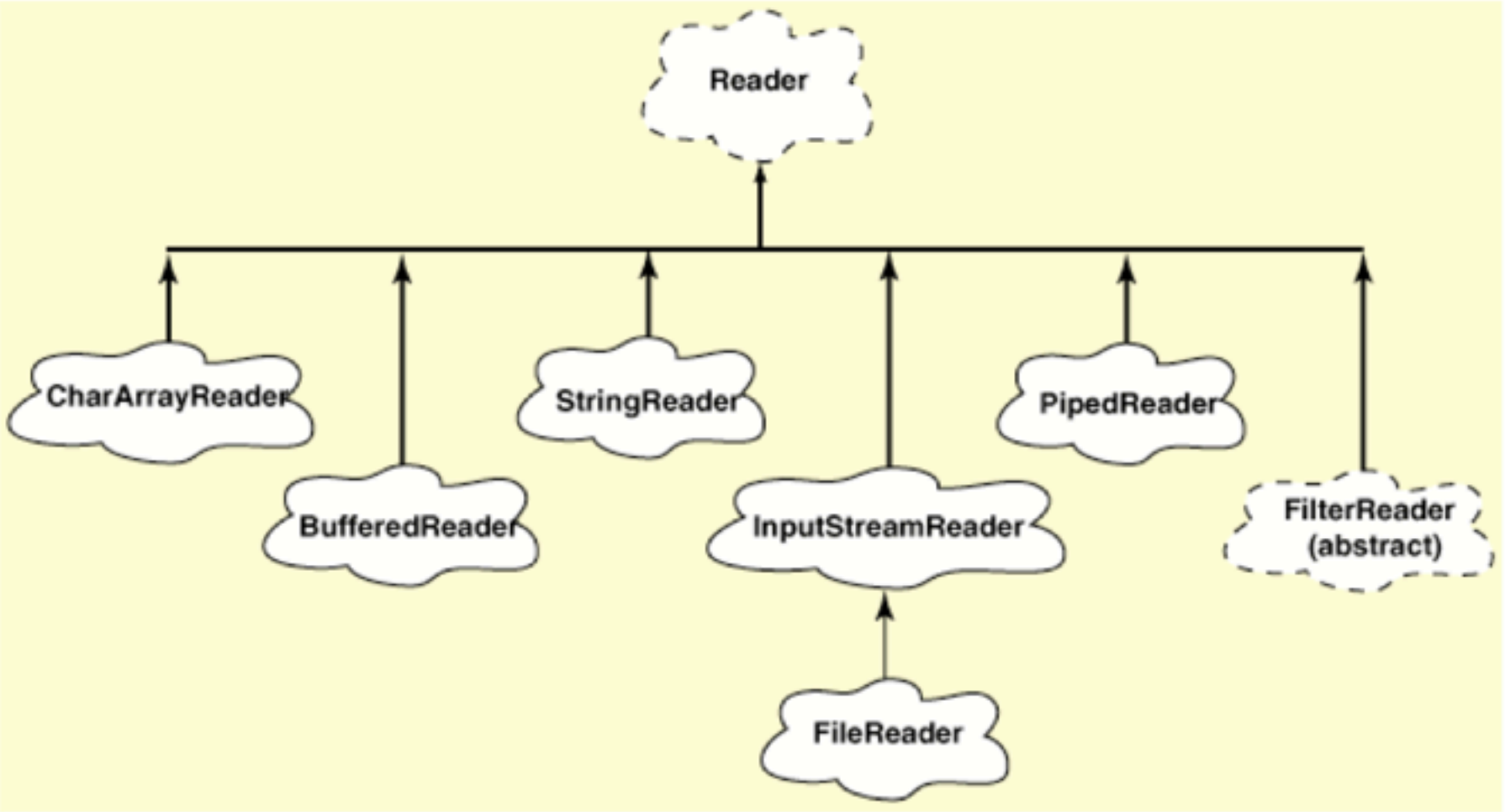
Les flux de caractères: Reader

- *Classe abstraite*
- *Transportent des données sous forme de caractères (unicode) sur 2 octets.*
- *Les classes qui gèrent les flux de caractères en lécture héritent d'une des classes abstraites Reader.*

Méthodes	description
int read( )	renvoie le caractère lu (entier dont la valeur est le code unicode du caractère lu) ou -1 si la fin du flux est atteinte.
int read(char[ ])	lit plusieurs caractères et les place dans un tableau de caractères, retourne le nombre de caractères lus ou -1 si la fin du flux est atteinte.
void close( )	ferme le flux et libère les ressources associées.

Ces méthodes peuvent lever des IOException.

Les flux de caractères: Reader



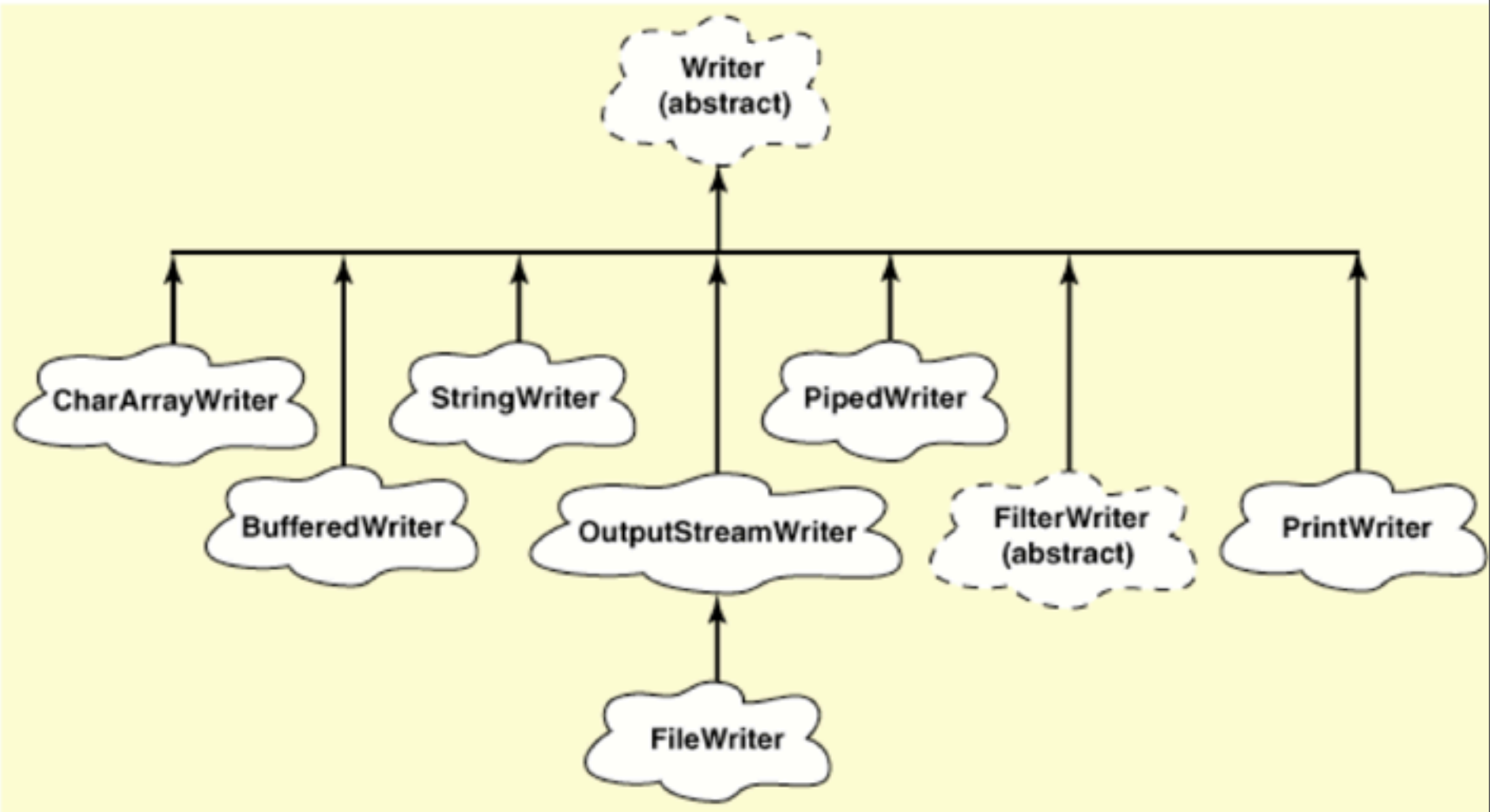
Les flux de caractères: Writer

- *Classe abstraite*
- *Transportent des données sous forme de caractères (unicode) sur 2 octets.*
- *Les classes qui gèrent les flux de caractères en écriture héritent d'une des classes abstraites Writer.*

Méthodes	description
void write(int)	écrit le caractère en paramètre (entier interprété en unicode) dans le flux.
void write(String)	écrit la chaîne de caractères en paramètre dans le flux.
void close( )	ferme le flux et libère les ressources associées.

Ces méthodes peuvent lever des IOException.

Les flux de caractères: Writer



Flux de caractère

★ La classe concrète *FileReader*

- *flux de caractères en lecture sur un fichier.*
- *Plusieurs constructeurs qui peuvent lever une FileNotFound Exception.*

```
FileReader fich = new FileReader("monfichier.txt");
```

★ La classe concrète *FileWriter*

- *flux de caractères en écriture sur un fichier.*
- *si le nom du fichier n'existe pas alors le fichier est créé; sinon, le flux est ouvert en écriture vers ce fichier (dont les données sont écrasées).*

```
FileWriter fich = new FileWriter ("monfichier.res");
```

- *FileWriter(String, boolean) : le booléen permet de préciser si les données sont ajoutées en fin de fichier (valeur true) ou écrasent les données existantes (valeur false).*

Les flux de caractères « bufferisés »

- Mise en tampon des données pour traiter un ensemble de caractères représentant une ligne plutôt que lire/écrire caractère par caractère (amélioration des performances).

★ La classe concrète *BufferedReader*

- *flux de caractères bufferisé en lecture :*
- *Constructeur:* le paramètre fourni doit correspondre au flux à lire.

```
BufferedReader fich = new BufferedReader(new FileReader ("monfichier.txt"));
```

- *Méthodes:*

  String readLine( ) : lit la prochaine ligne de caractères dans le flux. Une ligne est une suite de caractères qui se termine par un retour chariot '\r' ou un saut de ligne '\n' ou les deux. Rend **null** en fin de fichier. Sinon, rend la ligne sans le caractère de fin.

Les flux de caractères « bufferisés »

★ **La classe concrète *BufferedWriter***

- *flux de caractères bufferisé en écriture :*
- *Constructeur:* le paramètre fourni doit correspondre au flux à écrire.

```
BufferedWriter fich = new BufferedWriter(new FileWriter ("monfichier.txt"));
```

- *Méthodes:*

  void newLine( ) : écrit un séparateur de ligne

Les flux de caractères formatées

★ La classe concrète *PrintWriter*

- *flux de caractères pour écrire des données formatées (entiers, réels, objets...)*
- *Constructeur*

```
PrintWriter fichier = new PrintWriter(new FileWriter ("monfichier"));
```

- *Méthodes:*

void print(double ou int ou boolean) : écrit un réel ou une entier ou un booléen

void print(Object) : écrit un objet (via un appel à la méthode *toString()*

void print(String) : écrit une chaîne de caractères.

void println() : écrit un séparateur de lignes.

void println(boolean ou double ou int) : écrit un booléen ou un réel ou un entier puis un séparateur de lignes.

void println(Object) : écrit un objet puis un séparateur de lignes.

void printf(String format, Object... args) : écrit une chaîne de caractères formatée à partir d'un nombre variable d'objets

Les flux de caractères formatées

Exemples :

```
double a = 2565.634;
double b = 20.0;
int c = 123;
PrintWriter fichier = new PrintWriter(new FileWriter("fich"));

fichier.println(a + " * " + b + " + " + c + " = " + (a*b+c));

fichier.printf("%f * %f + %d = %f %n", a, b, c, a*b+c);
```

format ⟷ objets

%f : réel %d : entier %n : séparateur de ligne + et = tels quels

```
fichier.printf("%3.2f * %3.2f + %x = %3.2f %n", a, b, c, a*b+c);
```

%3.2f : réel avec 2 chiffres après la virgule et au minimum 3 chiffres

%x : entier en héxadécimal

Ecrit dans le fichier fich :

2565.634 * 20.0 + 123 = 51435.68

2565.634000 * 20.000000 + 123 = 51435.680000

2565.63 * 20.00 + 7b = 51435.68

Entrées/Sorties standards

- Flux d' E/S standards (flux prédéfinis)
  - Il existe 3 flux prédéfinis dans la classe `System`
    - La sortie standard : out (Flux de type PrintStream)
    - L'entrée standard : in (Flux de type InputStream)
    - La sortie d'erreur : err (Flux de type PrintStream)
- Exemples d'utilisation
    - InputStream ent = System.in ;
    - OutputStream sor = System.out ;
    - OutputStream erre= System.err ;

Entrées/Sorties standards

- On peut lire un octet à la fois sur l'entrée standard avec la méthode read() de la classe InputStream :

```
try {
    int c;
    while((c = System.in.read()) != -1) {
        System.out.print(c);
    }
} catch(IOException e) {
    System.out.print(e);
}
```

La classe Scanner

- Depuis le JDK 5 la classe Scanner permet de récupérer des données au milieu d'un flot d'entrée, comme scanf du langage C

- Utilisation
  - *On crée une instance de Scanner avec un de ses nombreux constructeurs*
  - *On lui passe en paramètre le flot d'entrée dans lequel on va extraire les données ; par exemple, pour lire l'entrée standard*
  - *On extrait ensuite les données une à une avec une des méthodes nextXXX :*

```
Scanner sc = new Scanner (System.in);
int i = sc.nextInt()
```

  - *Ils peuvent prendre en paramètre une instance de java.lang.Readable (source de données de type caractère).*

La classe Scanner

- Délimiteurs
  - Par défaut les données sont séparées par des espaces, tabulations, passages à la ligne,...
  - On peut en indiquer d'autres avec la méthode *useDelimiter* qui prend en paramètre une expression régulière :

```
sc.useDelimiter(";|\n");
```

- Localisation
  - *Une instance de Scanner utilise la locale par défaut*
  - *On peut modifier la locale avec la méthode useLocale*

```
sc.useLocale(Locale.US); // Pour les floats
```

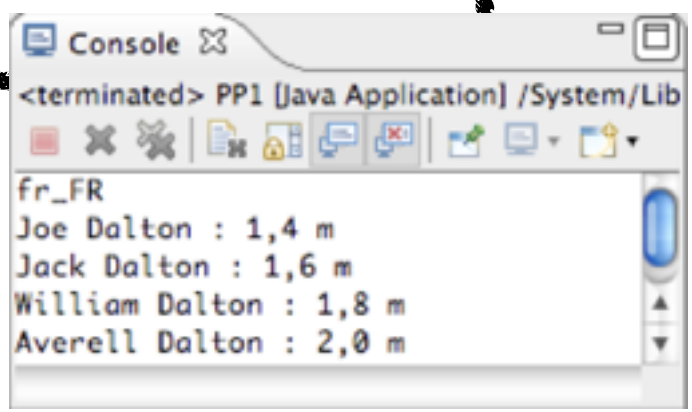
La classe Scanner

- Méthodes nextXXX
  - On trouve une méthode next par type primitif : nextInt, nextDouble,...
  - next() retourne une String qui contient la prochaine donnée placée entre 2 délimiteurs
  - nextLine() avance jusqu'à la prochaine ligne et retourne ce qui a été sauté
- Méthodes hasNextXXX
  - *Une méthode hasNextXXX par méthode nextXXX : hasNextInt, hasNextDouble (sauf pour nextLine)*
  - *Retourne true si la prochaine donnée correspond au type que l'on cherche*
  - *Ces méthodes nextXXX et hasNextXXX peuvent bloquer en attente du flot d'entrée*

La classe Scanner

- Exemple

```
String s =
"Dalton;Joe;1.4\n" +
"Dalton;Jack;1.6\n" +
"Dalton;William;1.8\n" +
"Dalton;Averell;2.0";
Scanner scan = new Scanner(s);
scan.useDelimiter(";|\n");
scan.useLocale(Locale.US); // Pour les floats
 while(scan.hasNextLine()) {
   System.out.printf("%2$s %1$s : %3$.1f m %n",
   scan.next(), scan.next(), scan.nextFloat());
}
```



La classe File

- Gestion de fichiers au sens large
  - Méthodes permettant d'interroger ou d'agir sur le système de gestion de fichiers de l'OS
  - Fichier ou répertoire
- Et encore plus !!!
  - Utilisation de File pour
    - créer un nouveau répertoire ou un chemin complet de répertoire s'ils n'existent pas
    - regarder les caractéristiques des fichiers (taille, dernière modification, date, lecture/écriture)
    - voir si un objet File représente un fichier ou un répertoire,
    - supprimer un fichier
    - ...

La classe File

- Constructeur
  - Les chemins passés en premier paramètre peuvent être des noms relatifs ou absolus
    - File(String chemin)
    - File(String chemin, String nom)
    - File(File parent, String nom)
- Quelques methodes
  - *boolean isFile( ) / boolean isDirectory( )*
  - *boolean mkdir( )*
  - *boolean exists( )*
  - *boolean delete( )*
  - *boolean canWrite( ) / boolean canRead( )*
  - *File getParentFile( )*
  - *long lastModified( )*

La classe File

```
import java.io.*;
public class Listeur
{
 public static void main(String[] args)
  {
   litrep(new File("."));
  }
 public static void litrep(File rep)
 {
   if (rep.isDirectory())
    { //liste les fichier du répertoire
      String t[]=rep.list();
      for (int i=0;i<t.length;i++)
        System.out.println(t[i]);
    }
  }
}
```

La classe File

```
import java.io.*;
public class Listeur
{
 public static void main(String[] args)
  {
   litrep(new File("."));
  }
 public static void litrep(File rep)
 {
   if (rep.isDirectory())
    { //liste les fichier du répertoire
      String t[]=rep.list();
      for (int i=0;i<t.length;i++)
        System.out.println(t[i]);
    }
  }
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

La classe File

```
import java.io.*;
public class Listeur
{
 public static void main(String[] args)
  {
   litrep(new File("."));
  }
 public static void litrep(File rep)
 {
   if (rep.isDirectory())
    { //liste les fichier du répertoire
      String t[]=rep.list();
      for (int i=0;i<t.length;i++)
        System.out.println(t[i]);
    }
  }
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet File : ici on va lister le répertoire courant (« . »)

La classe File

```
import java.io.*;
public class Listeur
{
 public static void main(String[] args)
  {
   litrep(new File("."));
  }
 public static void litrep(File rep)
 {
   if (rep.isDirectory())
    { //liste les fichier du répertoire
      String t[]=rep.list();
      for (int i=0;i<t.length;i++)
        System.out.println(t[i]);
    }
  }
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet File : ici on va lister le répertoire courant (« . »)

Les méthodes isFile() et isDirectory() permettent de déterminer si mon objet File est une fichier ou un répertoire

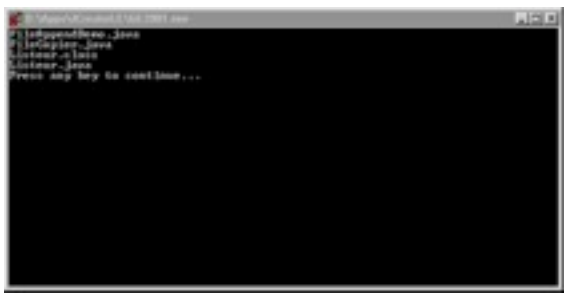
La classe File

```
import java.io.*;
public class Listeur
{
 public static void main(String[] args)
  {
   litrep(new File("."));
  }
 public static void litrep(File rep)
 {
   if (rep.isDirectory())
    { //liste les fichier du répertoire
      String t[]=rep.list();
      for (int i=0;i<t.length;i++)
        System.out.println(t[i]);
    }
  }
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet File : ici on va lister le répertoire courant (« . »)

Les méthodes isFile() et isDirectory() permettent de déterminer si mon objet File est une fichier ou un répertoire



La classe File

- Limitation de cette classe
  - L'absence d'exception utile

    En effet mis à part quelques exceptions basiques (NullPointerException, SecuityException), les exceptions sont très peu utilisées

  - Un accès limité aux propriétés des fichiers.

    On ne dispose en effet que du "minimum syndical" sans véritable accès à bon nombre de propriétés du système de fichier (propriétaire du fichier, groupe, permissions, ACL, attributs archive ou système, …)

  - *Une API «Application Programming Interface» un peu incomplète.*

    Certaines fonctionnalités pourtant basique sont absente (copie/déplacement de fichier, gestion des liens, résolution de chemin relatif)

Java.nio 2 (Java 7)

La classe `File` ***est donc destinée à être remplacée par une nouvelle classe (***`Path`***) ou plus précisément par tout un package : java.nio.***

- New I/O
  - Support au parcours d'un répertoire : lister son contenu avec possibilité de filtre sur les fichiers à lister
  - Ajout de fonctionnalités permettant de copier, déplacer les fichiers
  - Ajout de fonctionnalités pour un accès complet au système de fichiers : liens physiques / symboliques (création, détection, ...)
  - Possibilité de gérer des fichiers zip (via un objet FileSystem)
  - Meilleure gestion des exceptions
  - ...

Java.nio 2 (Java 7)

```
// Ancien code :
File file = new File("file.txt");

// Nouveau code :
Path path = Paths.get("file.txt");
```

Java.nio 2 (Java 7)

Création

```
Path source = Paths.get("test.txt");
System.out.println(" fichier existant ? " +
                   Files.exists(source));
Path file = Files.createFile(source);
System.out.println("Nom du fichier : " +
path.getFileName());
```

Copie

```
Path source = Paths.get("test.txt");
Path cible = Paths.get("test1.txt");
try{
    Files.copy(source, cible,
    StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) { e.printStackTrace();  }
```

Java.nio 2 (Java 7)

Lecture d'un fichier

```
Path file = ...;
try (InputStream in = Files.newInputStream(file);
    BufferedReader reader =
      new BufferedReader(new InputStreamReader(in))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.println(x);
}
```

Lister les répertoires racines du système de fichiers

```
Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
for (Path name: dirs) {
    System.err.println(name);
}
```

Suppression

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```