

# Les entrées/sorties

**Christina Boura, Stéphane Lopes**

`christina.boura@uvsq.fr,`  
`stephane.lopes@uvsq.fr`

29 mars 2016



# Agenda du jour

## Les flux

Les flux binaires et les flux de caractères

Entrées et sorties standards

Les nouveaux packages pour les entrées/sorties

# Les flux

- Besoin d'échanger des informations (recevoir des données d'une source ou envoyer des données vers un destinataire).
- Source ou destination : La mémoire, un fichier, un disque, le réseau, le programme lui-même, ...
- Réaliser cela à l'aide d'un *flux* (*stream* en anglais).

Un flux est le **canal** entre la source des données et sa destination.

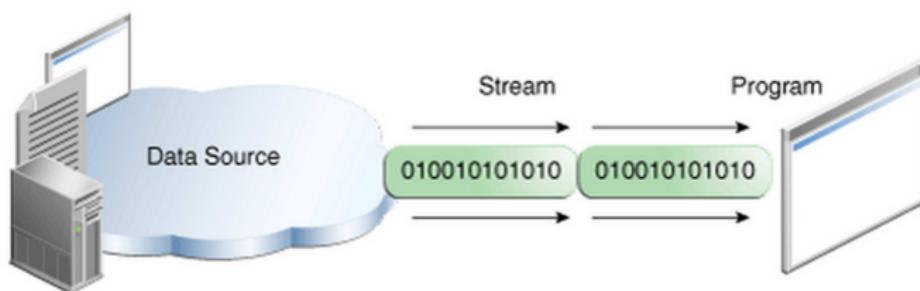
# Opérations entrées/sorties

Toute opération d'entrée/sortie en Java est composée des opérations suivantes :

- Ouverture d'un flux.
- Lecture ou écriture des données.
- Fermeture du flux.
  - Utilisation de la méthode `close()` ou passage du ramasse-miettes (Garbage Collector)

# Les flux d'entrée

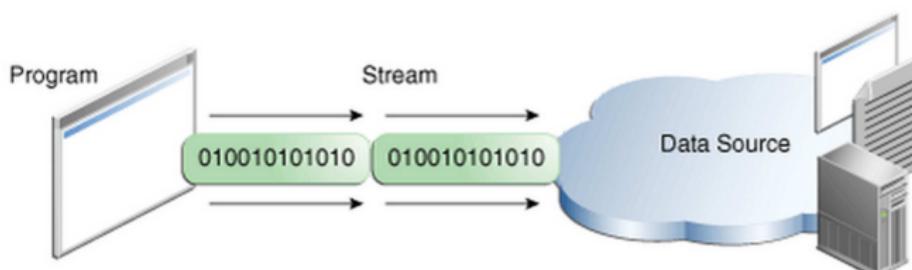
- Lire des données à partir d'une source de façon séquentielle.



Source : [www.docs.oracle.com](http://www.docs.oracle.com)

# Les flux de sortie

- Envoyer des données vers une destination de façon séquentielle.



Source : [www.docs.oracle.com](http://www.docs.oracle.com)

# Catégorisation des flux

En Java, les flux sont divisés de la façon suivante :

- Les flux d'entrées (**input streams**) et les flux de sortie (**output streams**)
- Les flux d'octets (**byte streams**) et les flux de caractères (**character streams**)

**Question :**

Pourquoi a-t-on besoin d'un traitement différent pour les caractères, alors que ce sont des données binaires ?

Java utilise le système **Unicode**. Les caractères sont codés sur **plus de 8 bits**.

# Unicode

Unicode est un standard informatique pour l'encodage de caractères.

- Plus d'un million de caractères peuvent être encodés. Actuellement, 245 000 codes différents sont assignés.
- Unicode attribut à chaque caractère un identifiant nommé **code point** :
- Un **code point** est représenté en hexadimal préfixé par **U+**.
  - **Exemple** : A ↔ U+0041.
- Les 127 premiers caractères d'Unicode correspondent à l'ensemble de caractères **ASCII**.

# L'encodage des caractères Unicode

Plusieurs encodages de la norme **UTF** (Unicode Transformation Format).

- **UTF-32** : Chaque caractère est codé sur 4 octets.
  - **Avantage** : Simple à l'emploi.
  - **Inconvénient** : Très cher en mémoire.
  - **Devenu obsolète**.
- **UTF-16** : Les caractères sont encodées soit sur 2 soit sur 4 octets.
  - **Avantage** : La plupart de caractères courants peuvent être codés sur 2 octets.
  - En train de se faire définitivement remplacer par **UTF-8**.
- **UTF-8** : Successeur de **UTF-16**.

# UTF-8

Chaque caractère est encodé en 1, 2, 3 ou 4 octets.

Code point	bits	Octet 1	Octet 2	Octet 3	Octet 4
U+0000 – U+007F	7	0xxxxxx			
U+0080 – U+07FF	11	110xxxx	10xxxxxx		
U+0800 – U+FFFF	16	1110xxx	10xxxxxx	10xxxxxx	
U+10000 – U+1FFFFFF	21	11110xx	10xxxxxx	10xxxxxx	10xxxxxx

Source : <http://defeo.lu/in420/>

- Les codes points dans la plage U+0000 – U+00FF correspondent aux caractères du jeu ASCII 7, ce qui garantit la compatibilité de UTF-8 avec ASCII.

# Les packages pour les entrées-sorties

Les premières librairies de classe pour la gestion des flux se trouvent dans le package `java.io`.

Une extension de cette API a été ajoutée à partir de la version **JDK 1.4**.

- Package `java.nio` (**New Input/Output**)
  - Amélioration des performances sur le traitement des fichiers, du réseau et des buffers.

Une nouvelle API **NIO.2** (package `java.nio.file`) a été rajoutée au **JDK 7**.

- Manipulation facile des **fichiers**.

# Agenda du jour

Les flux

Les flux binaires et les flux de caractères

Entrées et sorties standards

Les nouveaux packages pour les entrées/sorties

# Les flux binaires

- Les flux binaires servent à lire et écrire des octets.
- Ils dérivent de deux classes :
  - la classe `InputStream` pour les flux d'entrée et
  - la classe `OutputStream` pour les flux de sortie.
- Ces deux classes sont abstraites.

# Quelques méthodes de InputStream

- int `available()`
- void `close()`
- void `mark(int readlimit)`
- boolean `markSupported()`
- abstract int `read()`
- int `read(byte[] b)`
- int `read(byte[] b, int off, int len)`
- void `reset()`
- long `skip(long n)`

## La méthode read() de InputStream

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

- Lit un octet du flux d'entrée.
- Renvoie un **int** allant de **0 à 255**.
- Quand il n'y a plus de données à lire, la méthode renvoie **-1**.
- Peut lancer une exception du type **IOException** si une erreur se produit.
- Les sous-classes de la classe **InputStream** doivent **redéfinir** cette méthode.

# Quelques méthodes de OutputStream

- void `close()`
- void `flush()`
- void `write(byte[] b)`
- void `write(byte[] b, int off, int len)`
- abstract void `write(int b)`

## La méthode write() de OutputStream

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

- Écrit un octet sur le flux de sortie.
- Cet octet est les 8 bits de poids faible de l'entier int b.
- Les autres 24 bits sont ignorés.
- Peut lancer une exception du type `IOException` si une erreur se produit (par exemple si le flux a été fermé).
- Les sous-classes de la classe `OutputStream` doivent redéfinir cette méthode.

## Remarques sur l'utilisation de flux binaires

- Les flux binaires sont des entrées/sorties de **bas niveau**.
- Leur utilisation **doit être restreinte** aux entrées/sorties  **primitives**.

**Question :** Pourquoi parle-t-on alors des flux binaires ?

**Réponse :** Parce que tous les autres types de flux sont basés sur les flux binaires.

# Les flux de caractères

- Les flux de caractères permettent de convertir Unicode de/vers le format local.
- Ils dérivent de deux classes :
  - la classe `Reader` pour les flux d'entrée et
  - la classe `Writer` pour les flux de sortie.
- Ces deux classes sont **abstraites**.

## Quelques méthodes de Reader

- void `close()`
- void `mark(int readAheadLimit)`
- boolean `markSupported()`
- abstract int `read()`
- int `read(char[] cbuf)`
- int `read(char[] cbuf, int off, int len)`
- int `read(Charbuffer target)`
- boolean `ready()`
- long `skip(long n)`

## Quelques méthodes de Writer

- Writer `append(char c)`
- Writer `append(CharSequence csq)`
- Writer `append(CharSequence csq, int start, int end)`
- abstract void `close()`
- abstract void `flush()`
- void `write(char[] cbuf)`
- abstract void `write(char[] chuf, int off, int len)`
- void `write(int c)`
- void `write(String str)`
- void `write(String str, int off, int len)`

## Lien entre les deux types de flux

- Les flux de caractères **encapsulent** souvent les flux binaires.
- Les flux de caractères utilisent des flux binaires pour assurer la liaison **physique** des entrées/sorties.
- Ils s'occupent de **faire la traduction** entre les caractères et les octets.
- Par exemple, la classe **FileReader** utilise la classe **FileInputStream** et la classe **FileWriter** utilise la classe **FileOutputStream**.

# Préfixes des flux

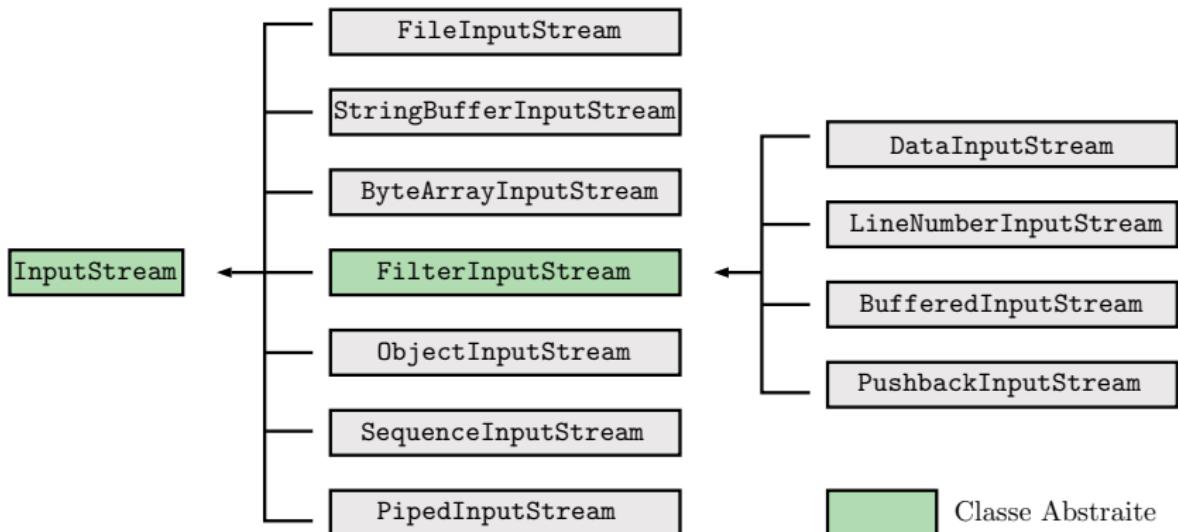
Préfixe du flux	Source ou destination du flux
ByteArray	tableau d'octets en mémoire
CharArray	tableau de caractères en mémoire
File	fichier
Object	objet
Pipe	pipeline entre deux threads
String	chaîne de caractères

## Les filtres

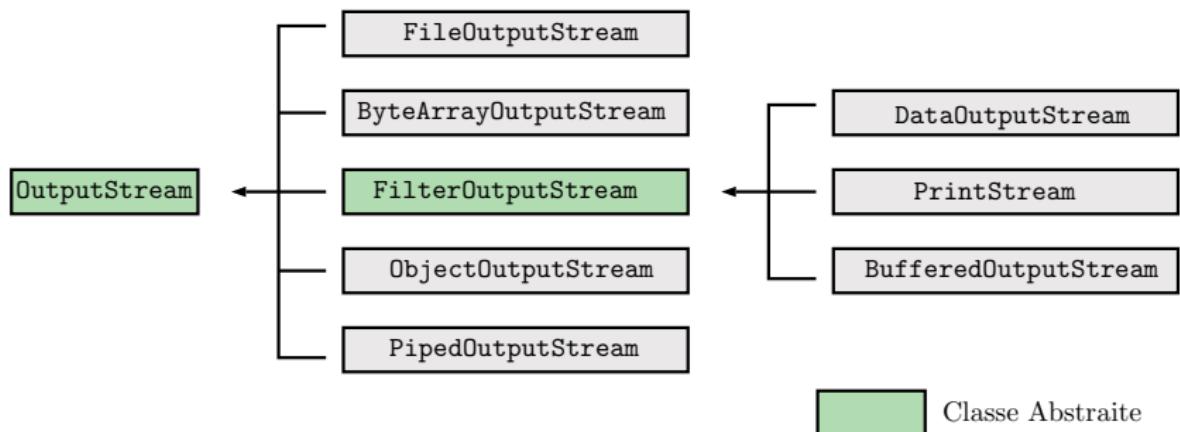
Certaines classes de flux sont destinées à **appliquer un traitement** sur (à filtrer) un flux.

- **Buffered** : Mettre les données du flux (en entrée et en sortie) dans un tampon.
- **Sequence** : Fusionner plusieurs flux.
- **Data** : Traiter les octets sous forme de type de données
- **LineNumber** : Permet de numérotter les lignes contenues dans le flux
- **PushBack** : Permet de remettre des données lues dans le flux
- **Print** : Permet de réaliser des impressions formatées
- **Object** : Utilisé par la sérialisation
- **InputStream / OutputStream** : Convertir des octets en caractères

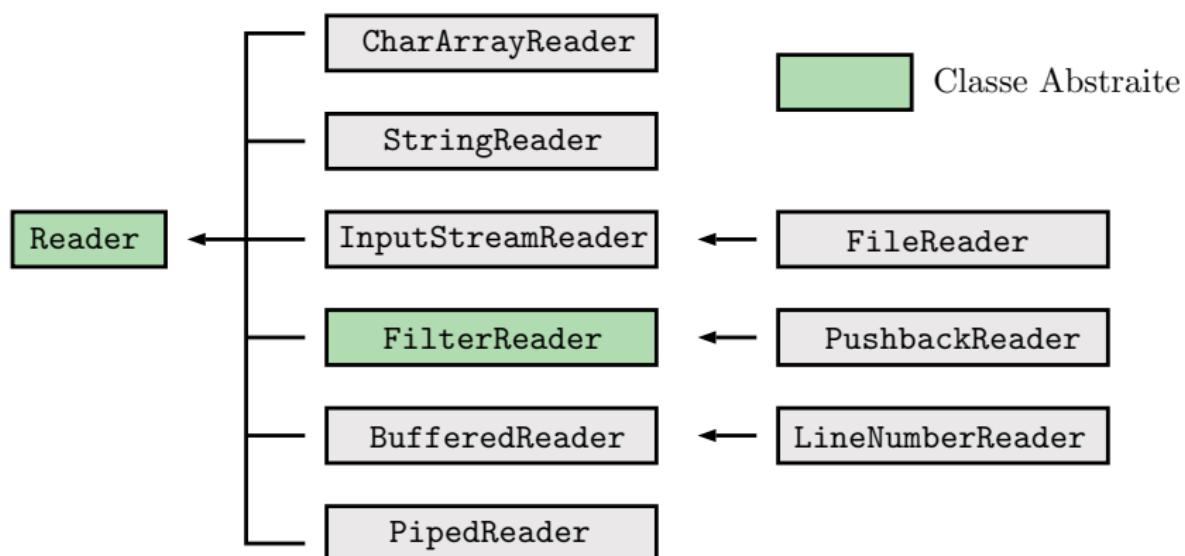
# Hiérarchie InputStream



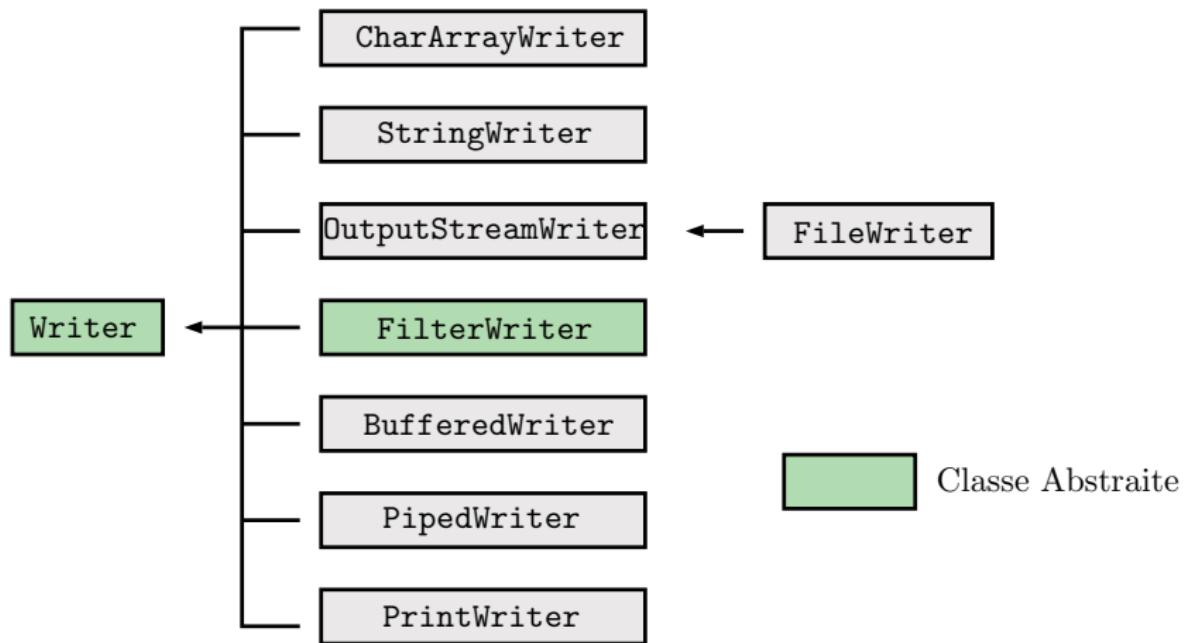
# Hiérarchie OutputStream



## Hiérarchie Reader



# Hiérarchie Writer



# Utilisation des flux (I)

Type d'E/S	Flux de caractères	Flux d'octets
Mémoire	CharArrayReader	ByteArrayInputStream
	CharArrayWriter	ByteArrayOutputStream
Fichier	StringReader	
	StringWriter	
Fichier	FileReader FileWriter	FileInputStream FileOutputStream
Avec buffer	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Affichage	PrintWriter	PrintStream
Sérialisation		ObjectInputStream ObjectOutputStream

## Utilisation des flux (II)

Type d'E/S	Flux de caractères	Flux d'octets
Conv. de données		DataInputStream DataOutputStream
Filtrage	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream
Concaténation		SequenceInputStream
Comptage	LineNumberReader	
Lecture en avant	PushbackReader	PushbackInputStream
Conv. oct./car.	InputStreamReader OutputStreamWriter	

## La classe File

Les classes des flux de fichiers sont

- `FileReader/FileWriter` pour l'accès aux fichiers textes.
- `FileInputStream/FileOutputStream` pour les fichiers binaires.
- Un flux de fichier peut être créé à partir d'un nom de fichier sous la forme d'une chaîne de caractères.

Constructeur le plus utilisé :

`File(String chemin)`

# Chemin d'accès

Le **chemin d'accès** d'un fichier ou d'un répertoire est une chaîne de caractères décrivant la **position** de ce fichier ou répertoire dans le **système de fichiers**.

## Exemples :

- **Linux** : home/christina/Documents/fichier.txt
- **Windows** : C:\Mes Documents\fichier.txt

Un chemin peut être **relatif** ou **absolu**.

- **Chemin absolu** : Contient la racine.
- **Chemin relatif** : Doit être combiné avec un autre chemin afin d'accéder au fichier.

# Quelques méthodes de la classe File (I)

- boolean `canRead()`
- boolean `canWrite()`
- boolean `exists()`
- boolean `isFile()`
- boolean `isDirectory()`
- boolean `isAbsolute()`

## Quelques méthodes de la classe File (II)

- `String getAbsolutePath()`
- `String getName()`
- `String getPath()`
- `String getParent()`
- `long length()`
- `long lastModified()`
- `String[] list()`

# Afficher des informations sur un fichier

```
import java.io.File;

public class TestFichier {

    public static void analyserChemin(String chemin) {
        File nom = new File(chemin);

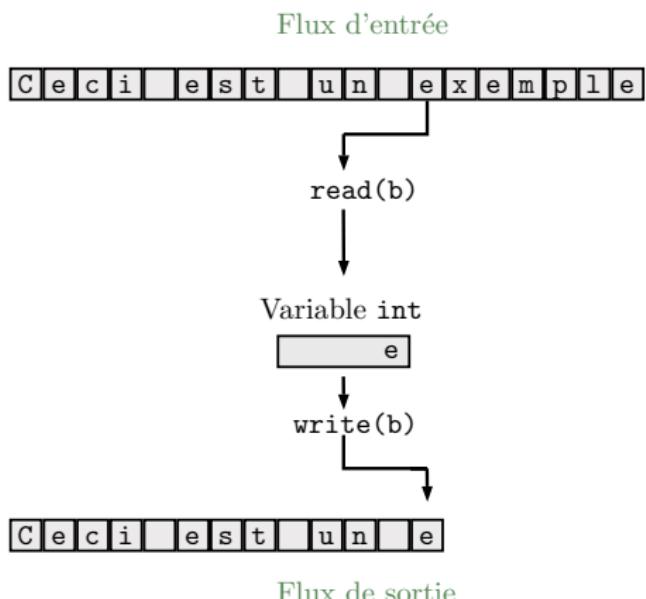
        if(nom.exists()) {
            System.out.println("=====");
            System.out.println(" nom:" + nom.getName());
            System.out.println(" est un fichier:" + nom.isFile());
            System.out.println(" est un répertoire:" + nom.isDirectory());
            System.out.println("est un chemin absolu:" + nom.isAbsolute());
            System.out.println(" quand modifié:" + nom.lastModified());
            System.out.println(" longeur en octets:" + nom.length());
            System.out.println(" chemin:" + nom.getPath());
            System.out.println(" chemin absolu:" + nom.getAbsolutePath());
            System.out.println(" répertoire parent:" + nom.getParent());
        }
        else {
            System.out.println("Le fichier " + nom + " n'existe pas.");
        }
    }

    public static void main(String[] args) {
        analyserChemin("fichierEntree.txt");
    }
}
```

# À l'execution

```
=====
    nom:fichierEntree.txt
    est un fichier:true
    est un répertoire:false
est un chemin absolu:false
    quand modifié:1426431093000
longeur en octets:34
    chemin:fichierEntree.txt
    chemin absolu:/Users/christina/Documents/workspace/TestProject/fichierEntree.txt
répertoire parent:null
```

# Copie d'un fichier



## Quelques Remarques

- **TOUJOURS** fermer un flux quand on n'en a plus besoin.
- Utiliser un bloc **finally** pour garantir que les flux se ferment **même** si une erreur se produise.
- Avant d'invoquer la méthode **close**, vérifier que les variables correspondant aux flux, ne sont pas **null**.

# Copie d'un fichier binaire

```
import java.io.*;

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

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("fichierEntree.txt");
            out = new FileOutputStream("fichierSortie.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

# Copie d'un fichier texte

```
import java.io.*;

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

        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("fichierEntree.txt");
            out = new FileWriter("fichierSortie.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

# Copier un fichier ligne par ligne

```
import java.io.*;

public class CopierLigneParLigne {

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

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("fichierEntree.txt"));
            outputStream = new PrintWriter(new FileWriter("fichierSortie.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

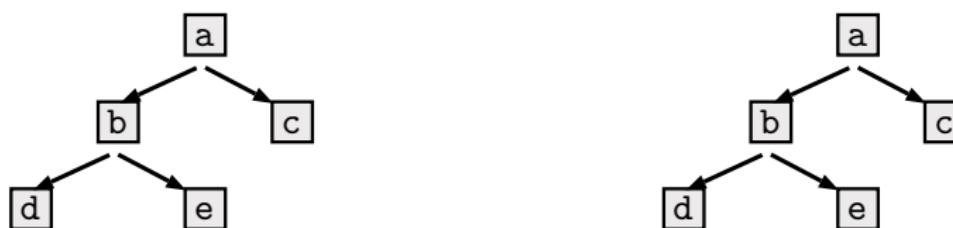
# La sérialisation

La **sérialisation** est un processus permettant de lire et d'écrire des objets, à l'aide des **Object Streams**.

- La sérialisation est assurée par les classes **ObjectInputStream** et **ObjectOutputStream**.
- **ObjectOutputStream** implemente les interfaces **DataOutput** et **ObjectOutput**.
- **ObjectInputStream** implemente les interfaces **DataInput** et **ObjectInput**.

## Copier des objets complexes

Certains objets peuvent contenir des références vers d'autres objets.



# Agenda du jour

Les flux

Les flux binaires et les flux de caractères

Entrées et sorties standards

Les nouveaux packages pour les entrées/sorties

# Entrées et sorties standards en Java

La classe **System** fournit des flux pour :

- l'entrée standard (attribut `in` de type `InputStream`))
- la sortie standard (attribut `out` de type `PrintStream`)
- la sortie d'erreurs (attribut `err` de type `PrintStream`)

## Affichage sur la sortie standard

La classe `PrintStream` fournit les fonctionnalités nécessaires pour l'affichage :

- `PrintStream append(...)` : Ajoute des caractères au flux ( $> \text{JDK 1.5}$ ).
- `void flush(...)` : Force la sortie des caractères.
- `void print(...)` : Affiche différents types de données sur le flux.
- `void println(...)` : Idem mais suivi d'un retour à la ligne.
- `PrintStream printf(...)` : affiche une chaîne selon un format ( $> \text{JDK 1.5}$ ).

## La méthode System.out.printf(...)

Fonctionnement et syntaxe similaires à la fonction `printf(...)` du langage **C**.

```
System.out.printf("Chaîne de caractères formatée",  
                  arg1, arg2, ...);
```

% [flags] [largeur] [.precision] **code de conversion**

Seul le paramètre en rouge **est obligatoire**.

- Largeur : Nombre minimal de caractères à écrire.
- Précision : Nombre de caractères à écrire après la virgule pour les nombres réels.

# Utiliser la méthode printf()

## Code de conversion :

- **d** : Variables entiers `byte`, `short`, `int`, `long`
- **f** : Variables réelles `float`, `double`
- **c** : `character`
- **s** : `String`

## Exemples :

- `System.out.printf("La valeur de i est : %d", i);`

La méthode `System.out.format()` permet d'atteindre le même résultat.

## Lire à partir de l'entrée standard

Les possibilités de la classe `InputStream` étant très limitées, il est nécessaire de décorer l'entrée standard avec des flux plus puissants.

A partir du `JDK 1.5`, un certain nombre d'améliorations a été apporté :

- La classe `java.util.Scanner` simplifie le processus de saisie.
- La classe `java.util.Formatter` permet le traitement d'entrées plus sophistiquées.

## La classe Scanner

La classe **Scanner** fait partie du package **java.util**. Elle permet de lire des valeurs de divers types.

Cette classe possède plusieurs **constructeurs**, le plus intéressant pour nous est le suivant :

**Scanner(InputStream source)**

**Exemple :** `Scanner sc = new Scanner(System.in);`

## Quelques méthodes de la classe Scanner

- int `nextInt()`
- long `nextLong()`
- float `nextFloat()`
- double `nextDouble()`
- String `next()`
- String `nextLine()`
- void `close()`

### Exemples :

- int nombreEntier = sc.nextInt();
- int nombreReel = sc.nextDouble();

# Exceptions qui peuvent être levées

## L'instruction

```
int nombreEntier = sc.nextInt();
```

et les instructions équivalentes peuvent lever des **exceptions** suivantes :

- **InputMismatchException** : L'entrée suivante ne correspond pas au bon type ou sa taille est hors la plage des valeurs acceptées.
- **NoSuchElementException** : L'entrée suivante n'existe pas.
- **IllegalStateException** : L'objet scanner est fermé.

# Accéder à l'entrée et la sortie standard (JDK >= 1.5)

```
System.out.println("Votre nom et votre age ?");

Scanner sc = new Scanner(System.in);
String nom = sc.next();
int age = sc.nextInt();

System.out.printf("Votre nom est %s et vous avez %d ans.", nom, age);

sc.close();
```

# Accéder à l'entrée et la sortie standard (JDK < 1.5)

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

String nom = in.readLine();
int age = Integer.parseInt(in.readLine());

System.out.println("Votre nom est " + nom + " et vous avez " + age + " ans.");

in.close();
```

# Agenda du jour

Les flux

Les flux binaires et les flux de caractères

Entrées et sorties standards

Les nouveaux packages pour les entrées/sorties

# Le package `java.nio`

Le `java.nio` package ("New I/O") a été introduit avec **JDK 1.4**.

- Améliore les performances sur le traitement des **fichiers**, du **réseau** et des **buffers**.
- Permet de lire les données d'une façon différente.
- Contrairement aux objets du package `java.io` qui traitent les données par octets, les objets du package `java.nio` traitent les données par **blocs de données**.
- La **lecture** est donc **accélérée**.

# Quelques fonctionnalités intéressantes de java.nio

- Le package `java.util.zip` permet la manipulation des fichiers compressés aux format `ZIP` et `GZIP`.
- Le package `java.util.jar` permet de manipuler des fichiers `JAR`.
- La package `javax.imageio` supporte différents types de format d'`images`.

# Introduction à l'API NIO.2

- Introduite par le **JDK 7** et contenue dans le package `java.nio.file`.
- Elle remplace en particulier la classe `java.io.File`.

# Nouveautés

- Meilleure **gestion des exceptions** : la plupart des méthodes de la classe `File` renvoient juste `null` en cas de problème. Avec ce nouveau package, des exceptions seront levées permettant de mieux cibler la cause du (ou des) problème(s).
- Accès complet au système de fichiers (support des liens/liens symboliques, ...).
- Ajout de méthodes utilitaires tels que le déplacement/la copie de fichier, la lecture/écriture binaire ou texte ...
- Remplacement de la classe `java.io.File` par l'interface `java.nio.file.Path`.

## La classe Path

Créer un objet **Path** avec la méthode **Paths.get(...)** :

```
Path chemin = Paths.get("/home/fichier.txt");
```

- Il existe plusieurs autres **constructeurs**.

# Informations sur un chemin

```
import java.nio.file.*;

public class TestChemin {

    public static void main(String[] args) {
        Path chemin = Paths.get("/Users/christina/Documents/workspace/TestProject/fichierEntree.txt");

        System.out.printf("toString: %s%n", chemin.toString());
        System.out.printf("getFileName: %s%n", chemin.getFileName());
        System.out.printf("getName(0): %s%n", chemin.getName(0));
        System.out.printf("getNameCount: %d%n", chemin.getNameCount());
        System.out.printf("subpath(0,2): %s%n", chemin.subpath(0,2));
        System.out.printf("getParent: %s%n", chemin.getParent());
        System.out.printf("getRoot: %s%n", chemin.getRoot());
    }
}
```

# À l'execution

```
toString: /Users/christina/Documents/workspace/TestProject/fichierEntree.txt
getFileName: fichierEntree.txt
getName(0): Users
getNameCount: 6
subpath(0,2): Users/christina
getParent: /Users/christina/Documents/workspace/TestProject
getRoot: /
```

## La classe Files

Cette classe offre un ensemble complet des méthodes **statiques** pour la **lecture**, **écriture** et la **manipulation** des **fichiers** et des **répertoires**.

- Elles permettent de réaliser des opérations sur des fichiers ou des répertoires dont le chemin est encapsulé dans un objet de type **Path**.

# Les méthodes de la classe Files

Les méthodes de la classe Files permettent :

- Créer des éléments : `createDirectory()`, `createFile()`,  
`createLink()`, `createSymbolicLink()`,  
`createTempFile()`, `createTempDirectory()`, ...
- Manipuler des éléments : `delete()`, `move()`, `copy()`, ...
- Obtenir le type d'un élément : `isRegularFile()`,  
`isDirectory()`, ...
- Obtenir des métadonnées et gérer les permissions :  
`getAttributes()`, `getPosixFilePermissions()`,  
`isReadable()`, `isWriteable()`, `size()`, ...

# Tester si un fichier est lisible

```
import java.nio.file.*;  
  
public class TestFichier2 {  
  
    public static void main(String[] args) {  
        Path fichier = Paths.get("/Users/christina/fichierEntree.txt");  
        boolean estLisible = Files.isRegularFile(fichier) & Files.isReadable(fichier);  
  
        System.out.println("Le fichier " + " est lisible : " + estLisible);  
  
    }  
}
```

Le fichier est lisible : true

## Créer un fichier ou un répertoire

- Path `createFile(Path path, FileAttribute<?>... attrs)` : Créer un fichier dont le chemin est encapsulé par l'instance de type Path fournie en paramètre
- Path `createDirectory(Path dir, FileAttribute<?>... attrs)` : Idem, mais pour un répertoire.
- Path `createDirectories(Path dir, FileAttribute<?>... attrs)` : Créer dans le répertoire dont le chemin est fourni en paramètre un sous-répertoire avec les attributs fournis.
- Path `createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)` : Créer dans le répertoire dont le chemin est fourni en paramètre un sous-répertoire temporaire dont le nom utilisera le préfixe fourni.

## Copier un fichier vers un autre

```
Path fichier = Paths.get("/Users/christina/fichier.txt");
Path fichierCopie = Paths.get("/Users/christina/fichierCopie.txt");

Path file = Files.copy(fichier, fichierCopie);
```

## Lire l'intégralité d'un fichier

La classe **Files** possède les méthodes

- `readAllLines()` qui renvoie une collection de type `List<String>` et
- `readAllBytes()` qui renvoie un tableau d'octets.

contenant l'intégralité d'un fichier texte ou binaire.

**Utilisation** : Réservée pour des fichiers de petite taille.

## Méthode `readAllLines()`

```
public static List<String> readAllLines(Path  
    path, Charset cs) throws IOException
```

- `path` : le chemin d'accès
- `cs` : système d'encodage

```
List<String> lignes = Files.readAllLines(Paths.get("/Users/christina/fichier.txt")  
    StandardCharsets.UTF_8);  
  
for (String ligne : lignes)  
    System.out.println(ligne);
```

## Méthode `readAllBytes()`

```
public static byte[] readAllBytes(Path path)
        throws IOException
```

- `path` : le chemin d'accès

```
Path file = Paths.get("/Users/christina/fichier.bin");
byte[] contenu = Files.readAllBytes(file);
```

# Écrire l'intégralité d'un fichier (I)

La méthode `write()` permet d'écrire le contenu d'un fichier.

```
public static Path write(Path path, Iterable<?  
    extends CharSequence> lines, Charset cs,  
    OpenOption... options) throws IOException
```

- `path` : le chemin d'accès
- `lines` : un objet pour l'itération
- `cs` : l'encodage utilisé
- `options` : options sur comment effectuer l'ouverture du fichier

# Écrire l'intégralité d'un fichier (I)

```
final Path pathSource = Paths.get("/Users/christina/source.txt");
final Path pathCible = Paths.get("/Users/christina/cible.txt");

final List<String> lignes = Files.readAllLines(pathSource, Charset.defaultCharset());
Files.write(pathCible, lignes, Charset.defaultCharset());

final Path pathSource = Paths.get("/Users/christina/source.bin");
final Path pathCible = Paths.get("/Users/christina/cible.bin");

final byte[] bytes = Files.readAllBytes(pathSource);
Files.write(pathCible, bytes);
```

# Lecture bufférisée d'un fichier

## Avant JDK 7 :

- Pour lire un fichier avec un tampon, il fallait invoquer le constructeur de la classe `BufferedReader` en lui passant en paramètre un objet de type `Reader` :

```
BufferedReader in = new BufferedReader(new FileReader("fichier.txt"));
```

## À partir de JDK 7 :

- On peut utiliser la méthode `newBufferedReader()` de la classe `Files`.

```
BufferedReader in = Files.newBufferedReader(Paths.get("fichier.txt"),  
                                         Charset.forName("UTF-8"));
```

## Le try "avec ressources"

- Déclarer les ressources utilisées **dans le bloc try(...)** .
- Elles seront **automatiquement fermées** à la fin du bloc d'instructions.

```
try(FileInputStream fis = new FileInputStream("fichierEntree.txt");
    FileOutputStream fos = new FileOutputStream("fichierSortie.txt"))
```

- Séparer les différentes ressources utilisées par un " ; " .

# Le try "avec ressources" - Commentaires

## Avantages :

- Plus **clair** et **lisible** qu'avant.
- Plus besoin de prendre soin de les fermer dans le bloc **finally**.

Cependant, des problèmes peuvent arriver avec **res ressources encapsulées**.

```
try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("fichierEntree.txt"))) {  
//...  
}
```

- Si une exception est levée, le flux correspondant à l'objet **FileInputStream** ne sera pas fermé.
- Pour cette raison, il faut découper toutes les ressources à utiliser.

```
try (FileInputStream fis = new FileInputStream("fichierEntree.txt");  
ObjectInputStream ois = new ObjectInputStream(fis)) {  
//...  
}
```