

Les entrées /sorties en Java

La gestion des fichiers (1)

- **La gestion de fichiers proprement dite se fait par l'intermédiaire de la classe File.**
- **Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation.**
- **Un objet de la classe File peut représenter un fichier ou un répertoire.**

La gestion des fichiers (2)

- **Voici un aperçu de quelques constructeurs et méthodes de la classe File :**
 - **File** (String name)
 - **File** (String path, String name)
 - **File** (File dir, String name)
 - boolean **isFile**() / boolean **isDirectory**()
 - boolean **mkdir**()
 - boolean **exists**()
 - boolean **delete**()
 - boolean **canWrite**() / boolean **canRead**()
 - File **getParentFile**()
 - long **lastModified**()

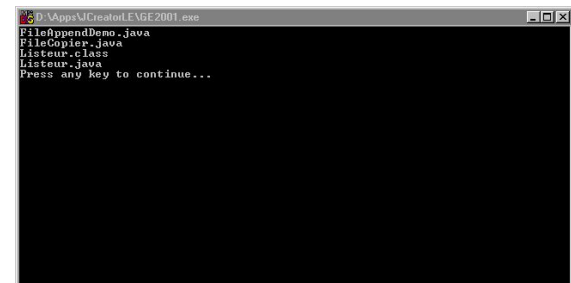
La gestion des fichiers (3)

```
import java.io.*; ←
public class Liseur
{
    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 un fichier ou un répertoire



```
D:\Apps\JCreator\FNGE2001.exe
FileAppendDemo.java
FileCopier.java
Liseur.class
Liseur.java
Press any key to continue...
```

La gestion des fichiers (4)

```
import java.io.*;
public class Listeur
{
    public static void main(String[] args)
    { litrep(new File( "c:\\"));}

    public static void litrep(File rep)
    {
        File r2;
        if (rep.isDirectory())
        {String t[]=rep.list();
        for (int i=0;i<t.length;i++)
        {
            r2=new File(rep.getAbsolutePath()+"\\"+t[i]);
            if (r2.isDirectory()) litrep(r2); ←
            else System.out.println(r2.getAbsolutePath());
        }
    }
}
```

Le nom complet du fichier
est rep\fichier

Pour chaque fichier,
on regarde s'il est
un répertoire.

Si le fichier est un
répertoire
litrep s'appelle
récursivement
elle-même

Notion de flux (1)

- **Les E / S sont gérées de façon portable (selon les OS) grâce à la notion de flux (*stream* en anglais).**
- **Un flux est en quelque sorte un canal dans lequel de l'information transite. L'ordre dans lequel l'information y est transmise est respecté.**
- **Un flux peut être :**
 - Soit une source d'octets à partir de laquelle il est possible de lire de l'information. On parle de flux d'entrée.
 - Soit une destination d'octets dans laquelle il est possible d'écrire de l'information. On parle de flux de sortie.

Notion de flux (2)

- **Certains flux de données peuvent être associés à des ressources qui fournissent ou reçoivent des données comme :**
 - les fichiers,
 - les tableaux de données en mémoire,
 - les lignes de communication (connexion réseau)

Notion de flux (3)

- **Certains flux peuvent être associés à des filtres**
 - Combinés à des flux d'entrée ou de sortie, ils permettent de traduire les données.
- **Les flux sont regroupés dans le paquetage java.io**

Notion de flux (4)

- **Il existe de nombreuses classes représentant les flux**
 - il n'est pas toujours aisé de se repérer.
- **Certains types de flux agissent sur la façon dont sont traitées les données qui transitent par leur intermédiaire :**
 - E / S bufferisées, traduction de données, ...
- **Il va donc s'agir de combiner ces différents types de flux pour réaliser la gestion souhaitée pour les E / S.**

Flux d'octets et flux de caractères

- **Les flux sont décomposés en deux grandes familles**
 - Les flux d'octets (Binary)
 - classes abstraites **InputStream** et **OutputStream** et leurs sous-classes concrètes respectives,
 - Les flux de caractères (Text)
 - classes abstraites **Reader** et **Writer** et leurs sous-classes concrètes respectives.

Flux d'octets et flux de caractères

Streams

InputStream

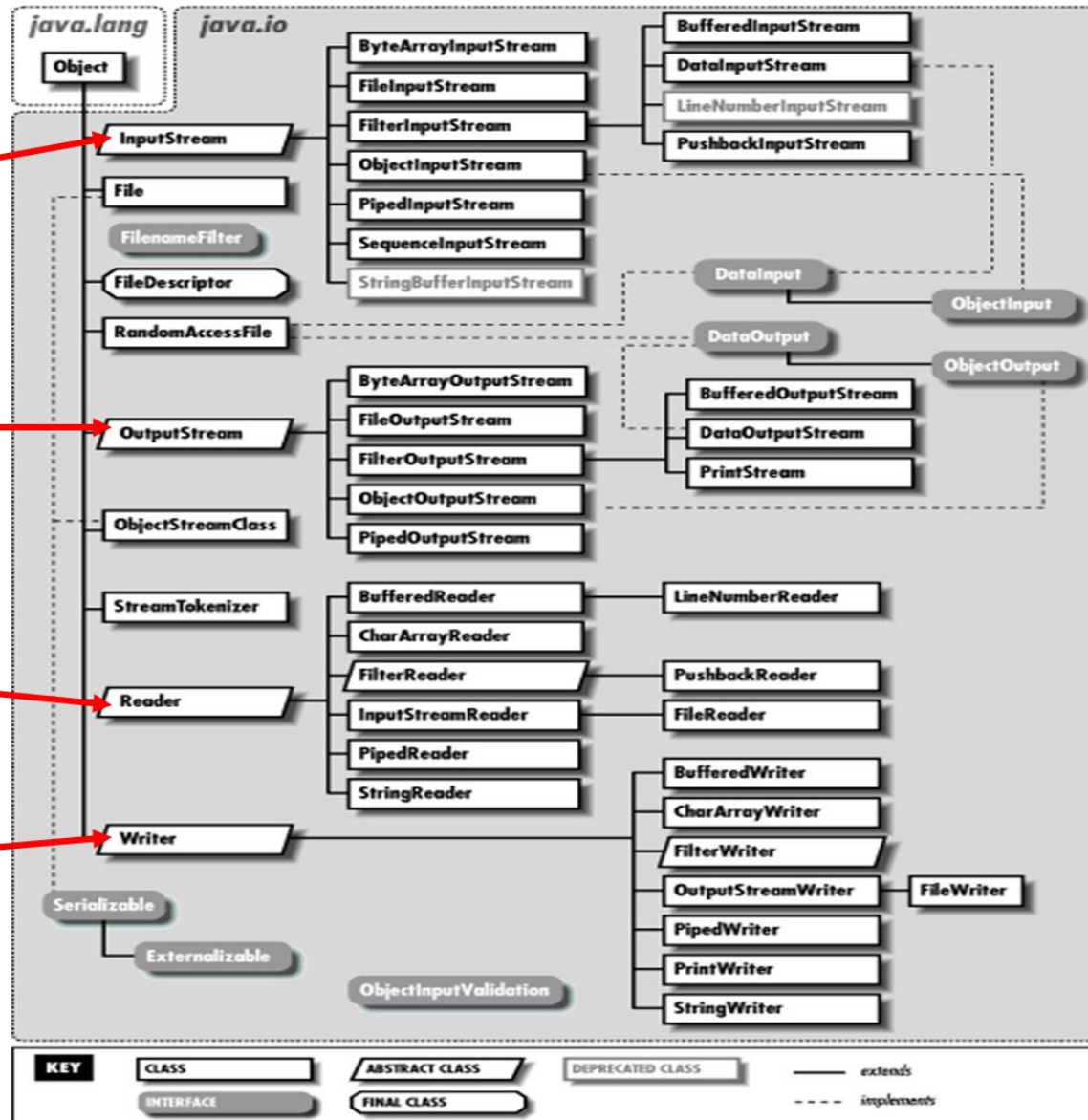
OutputStream

binary

Reader

Writer

text

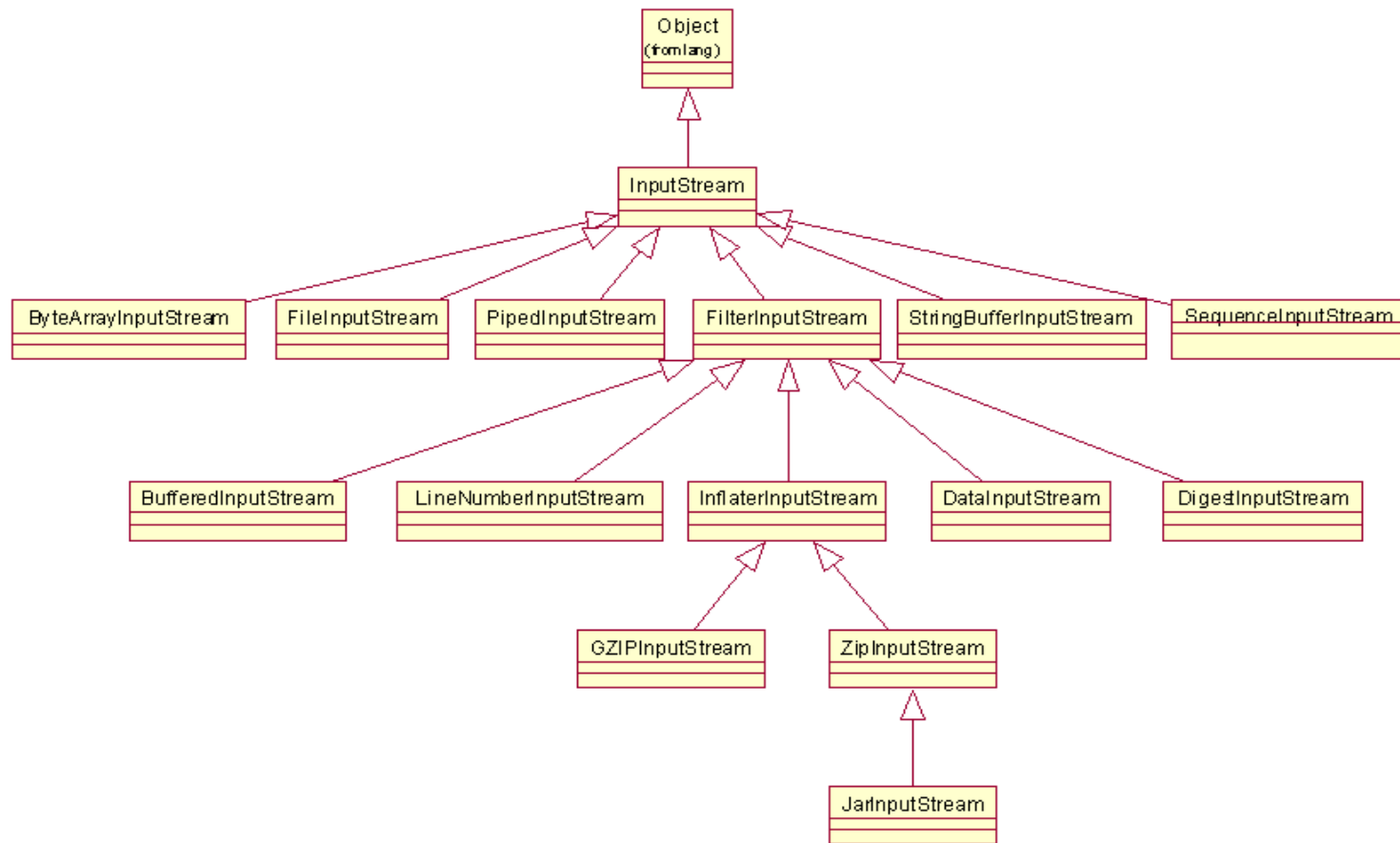


Les flux d'octets :

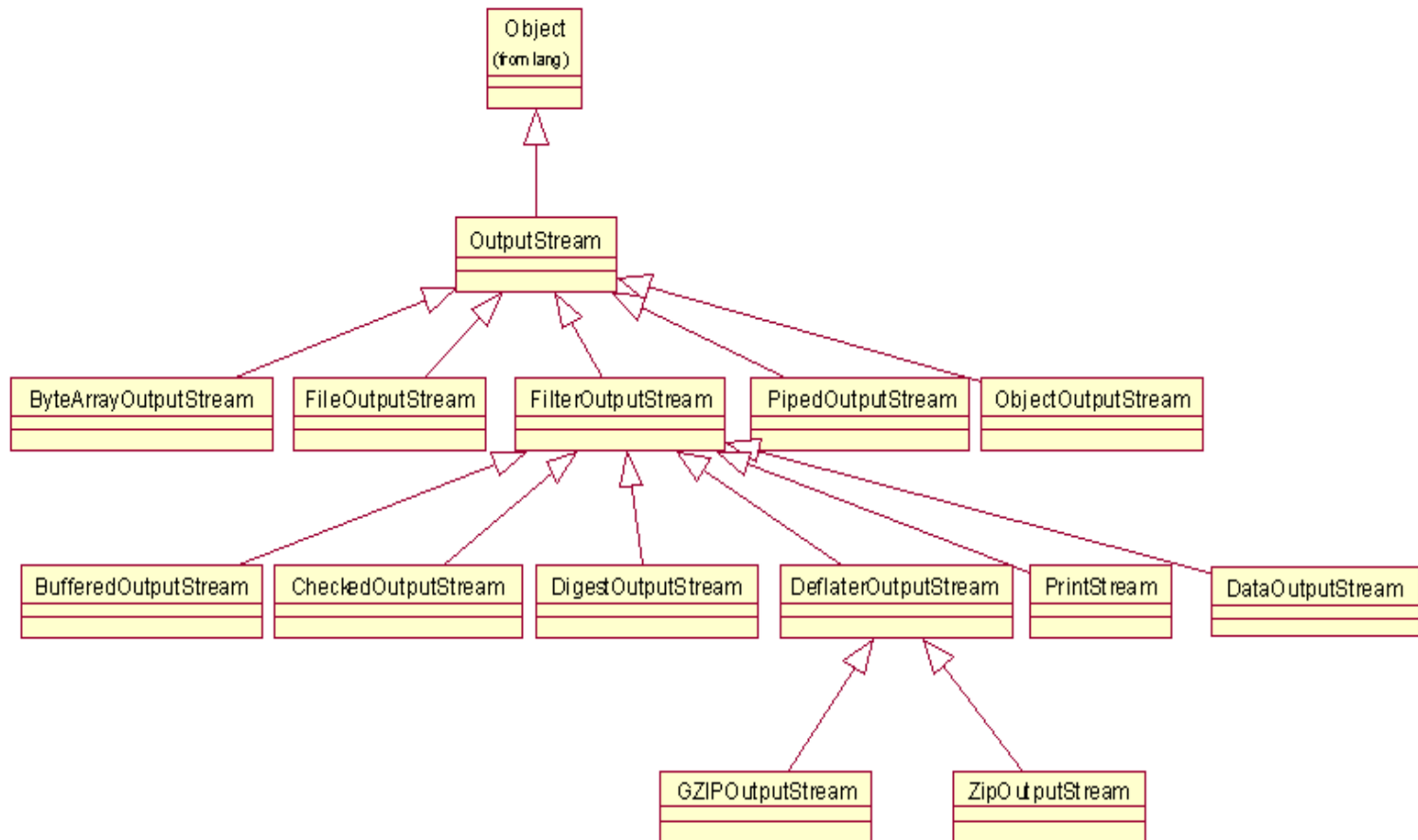
La classe `InputStream` (1)

- **Un `InputStream` est un flux de lecture d'octets.**
- **`InputStream` est une classe abstraite.**
 - Ses sous-classes concrètes permettent une mise en œuvre pratique.
 - Par exemple, **`FileInputStream`** permet la lecture d'octets dans un fichier.

La hiérarchie des flux d'octets en entrée



La hiérarchie des flux d'octets en sortie



La classe InputStream (2)

- **Les méthodes principales qui peuvent être utilisées sur un InputStream sont :**
 - **public abstract int read () throws IOException** qui retourne l'octet lu ou -1 si la fin de la source de données est atteinte. C'est cette méthode qui doit être définie dans les sous-classes concrètes et qui est utilisée par les autres méthodes définies dans la classe **InputStream**.
 - **int read (byte[] b)** qui emplit un tableau d'octets et retourne le nombre d'octets lus
 - **int read (byte [] b, int off, int len)** qui emplit un tableau d 'octets à partir d 'une position donnée et sur une longueur donnée
 - **void close ()** qui permet de fermer un flux,
 - Il faut fermer les flux dès qu'on a fini de les utiliser. En effet, un flux ouvert consomme des ressources du système d'exploitation qui sont en nombre limité.

La classe InputStream (3)

- **Les méthodes principales qui peuvent être utilisées sur un InputStream sont (suite) :**
 - **int available ()** qui retourne le nombre d'octets prêts à être lus dans le flux,
 - Attention : Cette fonction permet d'être sûr qu'on ne fait pas une tentative de lecture bloquante. Au moment de la lecture effective, il se peut qu'il y ait plus d'octets de disponibles.
 - **long skip (long n)** qui permet d'ignorer un certain nombre d'octets en provenance du flot. Cette fonction renvoie le nombre d'octets effectivement ignorés.

La classe OutputStream (1)

- **Un OutputStream est un flot d'écriture d'octets.**
- **La classe OutputStream est abstraite.**
- **Les méthodes principales qui peuvent être utilisées sur un OutputStream sont :**
 - **public abstract void write (int) throws IOException** qui écrit l'octet passé en paramètre,
 - **void write (byte[] b)** qui écrit les octets lus depuis un tableau d'octets,
 - **void write (byte [] b, int off, int len)** qui écrit les octets lus depuis un tableau d'octets à partir d'une position donnée et sur une longueur donnée,

La classe OutputStream (2)

- **Les méthodes principales qui peuvent être utilisées sur un OutputStream sont (suite) :**
 - **void close ()** qui permet de fermer le flux après avoir éventuellement vidé le tampon de sortie,
 - **flush ()** qui permet de purger le tampon en cas d'écritures bufferisées.

Les flux d'octets

- **Classe DataInputStream**
 - sous classes de InputStream permet de lire tous les types de base de Java.
- **Classe DataOutputStream**
 - sous classes de OutputStream permet d'écrire tous les types de base de Java.
- **Classes ZipOutputStream et ZipInputStream**
 - permettent de lire et d'écrire des flux dans le format de compression zip.

Empilement de flux filtrés (1)

- **En Java, chaque type de flux est destiné à réaliser une tâche.**
- **Lorsque le programmeur souhaite un flux qui ait un comportement plus complexe, il "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.**

Empilement de flux filtrés (2)

- **FileInputStream**
 - permet de lire depuis un fichier mais ne sait lire que des octets.
- **DataInputStream**
 - permet de combiner les octets pour fournir des méthodes de lecture de plus haut niveau (pour lire un double par exemple), mais ne sait pas lire depuis un fichier.
- **Une combinaison des deux permet de combiner leurs caractéristiques :**

```
FileInputStream fic = new FileInputStream ("fichier");  
DataInputStream din = new DataInputStream (fic);  
double d = din.readDouble ();
```

Empilement de flux filtrés (3)

Lecture bufferisée de nombres depuis un fichier

```
DataInputStream din = new DataInputStream(new  
BufferedInputStream( new FileInputStream ("monfichier")));
```

Lecture de nombre dans un fichier au format

```
zip  
ZipInputStream zin = new ZipInputStream (  
    new FileInputStream ("monfichier.zip"));  
DataInputStream din = new DataInputStream (zin);
```

Flux de fichiers à accès direct (1)

- **La classe `RandomAccessFile`**
 - permet de lire ou d'écrire dans un fichier à n'importe quel emplacement (par opposition aux fichiers à accès séquentiels).
- **Elle implémente les interfaces `DataInput` et `DataOutput`**
 - permettent de lire ou d'écrire tous les types Java de base, les lignes, les chaînes de caractères ascii ou unicode, etc ...

Flux de fichiers à accès direct (2)

- **Un fichier à accès direct peut être**
 - ouvert en lecture seule (option "r") ou
 - en lecture / écriture (option "rw").
- **Ces fichiers possèdent un pointeur de fichier qui indique constamment la donnée suivante.**
 - La position de ce pointeur est donnée par **long getFilePointer ()** et celui-ci peut être déplacé à une position donnée grâce à **seek (long off)**.

Les flux de caractères (1)

- **Ce sont des sous-classes de Reader et Writer.**
- **Ces flux utilisent le codage de caractères Unicode.**
- **Exemples**
 - conversion des caractères saisis au clavier en caractères dans le codage par défaut

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

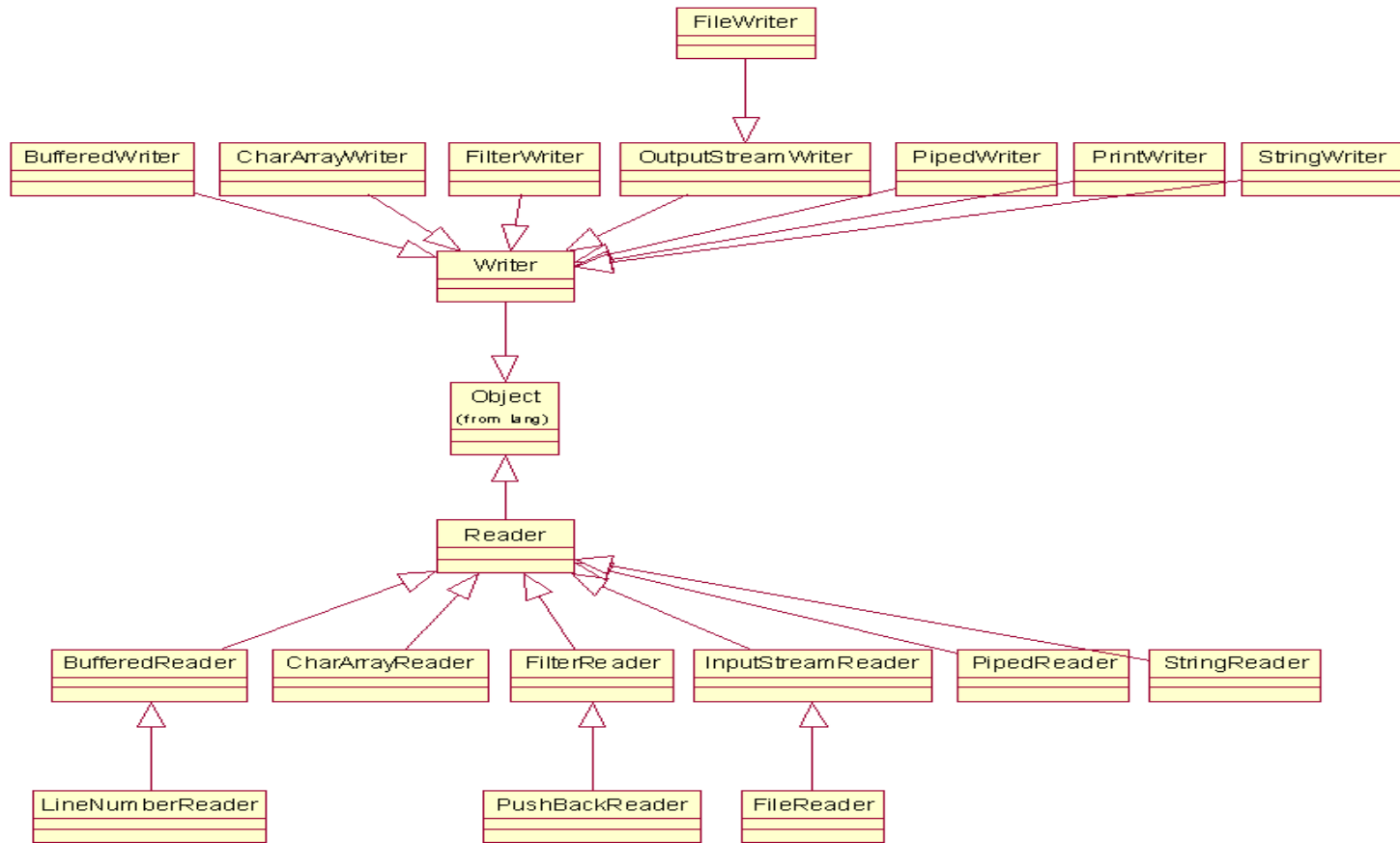
Conversion des caractères d'un fichier
avec un codage explicitement indiqué

```
InputStreamReader in = new InputStreamReader (  
    new FileInputStream ("chinois.txt"), "ISO2022CN");
```

Les flux de caractères (2)

- **Pour écrire des chaînes de caractères et des nombres sous forme de texte**
 - on utilise la classe **PrintWriter** qui possède un certain nombre de méthodes **print (...)** et **println (...)**.
- **Pour lire des chaînes de caractères sous forme texte, il faut utiliser, par exemple,**
 - **BufferedReader** qui possède une méthode **readLine()** .
 - Pour la lecture de nombres sous forme de texte, il n'existe pas de solution toute faite : il faut par exemple passer par des chaînes de caractères et les convertir en nombres.

La hiérarchie des flux de caractères



Exemple : lecture de fichier

```
import java.io.*;
public class LireLigne
{
    public static void main(String[] args)
    {
        try
        {
            FileReader fr=new FileReader("c:\\windows\\system.ini");
            BufferedReader br= new BufferedReader(fr);
            while (br.ready())
                System.out.println(br.readLine());
            br.close();
        }
        catch (Exception e)
        {System.out.println("Erreur "+e);}
    }
}
```

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileReader puis à partir de celui-ci, on crée un BufferedReader

Dans l'objet BufferedReader on dispose d'une méthode readLine()

Ecriture dans un fichier

```
import java.io.*;
public class Ecrire
{
    public static void main(String[] args)
    {
        try
        {
            FileWriter fw=new FileWriter("c:\\temp\\essai.txt");
            BufferedWriter bw= new BufferedWriter(fw);
            bw.write("Ceci est mon fichier");
            bw.newLine();
            bw.write("Il est à moi...");
            bw.close();
        }
        catch (Exception e)
        { System.out.println("Erreur "+e);}
    }
}
```

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileWriter puis à partir de celui-ci, on crée un BufferedWriter

Attention, lorsque l'on a écrit, il ne faut pas oublier de fermer le fichier

Les flux de données prédéfinis (1)

Il existe 3 flux prédéfinis :

- l'entrée standard **System.in** (instance de `InputStream`)
- la sortie standard **System.out** (instance de `PrintStream`)
- la sortie standard d'erreurs **System.err**(instance de `PrintStream`)

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

Les flux de données prédéfinis (2)

La classe **InputStream** ne propose que des méthodes élémentaires. Préférez la classe **BufferedReader**.qui permet de récupérer des chaînes de caractères.

```
try {  
    Reader reader = new InputStreamReader(System.in);  
    BufferedReader keyboard = new BufferedReader(reader);  
  
    System.out.print("Entrez une ligne de texte : ");  
    String line = keyboard.readLine();  
    System.out.println("Vous avez saisi : " + line);  
} catch(IOException e) {  
    System.out.print(e);}
```

Les flux de données prédéfinis (3)

L'utilisation de flux “bufferisés” permet d'améliorer considérablement les performances

```
import java.io.*;

class TestVitesseFlux {
public static void main(String[] args) {
    FileInputStream fis; BufferedInputStream bis;
    try {fis = new FileInputStream(new File("test.txt"));
        bis = new BufferedInputStream(new FileInputStream(new
File("test.txt")));
        byte[] buf = new byte[8];
        long startTime = System.currentTimeMillis();
        while(fis.read(buf) != -1);
        System.out.println("Temps de lecture avec FileInputStream :
" + (System.currentTimeMillis() - startTime));
        startTime = System.currentTimeMillis();
        while(bis.read(buf) != -1);
        System.out.println("Temps de lecture avec
BufferedInputStream : " + (System.currentTimeMillis() - startTime));
        fis.close(); bis.close(); }
    catch (FileNotFoundException e) { e.printStackTrace(); } catch
(IOException e) { e.printStackTrace(); } }
```


La sérialisation (1)

La sérialisation consiste à prendre un objet en mémoire et à en sauvegarder l'état sur un flux de données (vers un fichier, par exemple).

Ce concept permet aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement.

La sérialisation peut donc être considérée comme une forme de persistance des données.

La sérialisation (2)

2 classes **ObjectInputStream** et **ObjectOutputStream** proposent, respectivement, les méthodes **readObject** et **writeObject**

Par défaut, les classes ne permettent pas de sauvegarder l'état d'un objet sur un flux de données. Il faut implémenter l'interface **java.io.Serializable**.

Il faut que la classe n'ait pas supprimé le constructeur par défaut

Exemple de sérialisation

```
void sauvegarde(String s) {  
    try {FileOutputStream f = new FileOutputStream(new File(s));  
        ObjectOutputStream oos = new  
ObjectOutputStream(f);  
        oos.writeObject(this);  
        oos.close();}  
    catch (Exception e)  
        { System.out.println("Erreur "+e);}  
}  
  
static Object relecture(String s) {  
    try {FileInputStream f = new FileInputStream(new File(s));  
        ObjectInputStream oos = new ObjectInputStream(f);  
        Object o=oos.readObject();  
        oos.close();  
        return o;}  
    catch (Exception e)  
        { System.out.println("Erreur "+e);  
        return null;}  
}
```

Class Scanner (1)

- Une classe injustement méconnue du JDK est Scanner (depuis la version 5 de Java)
- fonctionnalités très intéressantes pour parser des chaînes de caractères, et en extraire et convertir les composants.
- Un Scanner peut se brancher sur à peu près n'importe quelle source : **InputStream**, **Readable** (et donc **Reader**), **File...** et bien sûr une simple **String**.
- Ensuite utiliser les méthodes de type **hasNext...()** / **next...()**, ou alors les méthodes de type **find...()** / **match()** / **group()**.

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

Class Scanner (2)

- Méthode `hasNext()` / `next()`
- 1. découper la chaîne de caractères en *tokens* grâce à un délimiteur ; il s'agit par défaut d'un caractère "blanc" (espace, tabulation, retour à la ligne...), mais il est évidemment possible de fournir sa propre expression via la méthode `useDelimiter(expression)`.
- 2. utiliser les méthodes de type `hasNext...()` et `next...()` pour parcourir, récupérer et convertir ces tokens.
- Les méthodes de type `hasNext...()` (`hasNextInt()`, `hasNextFloat()`...) fonctionnent sur le même principe qu'un `Iterator`.

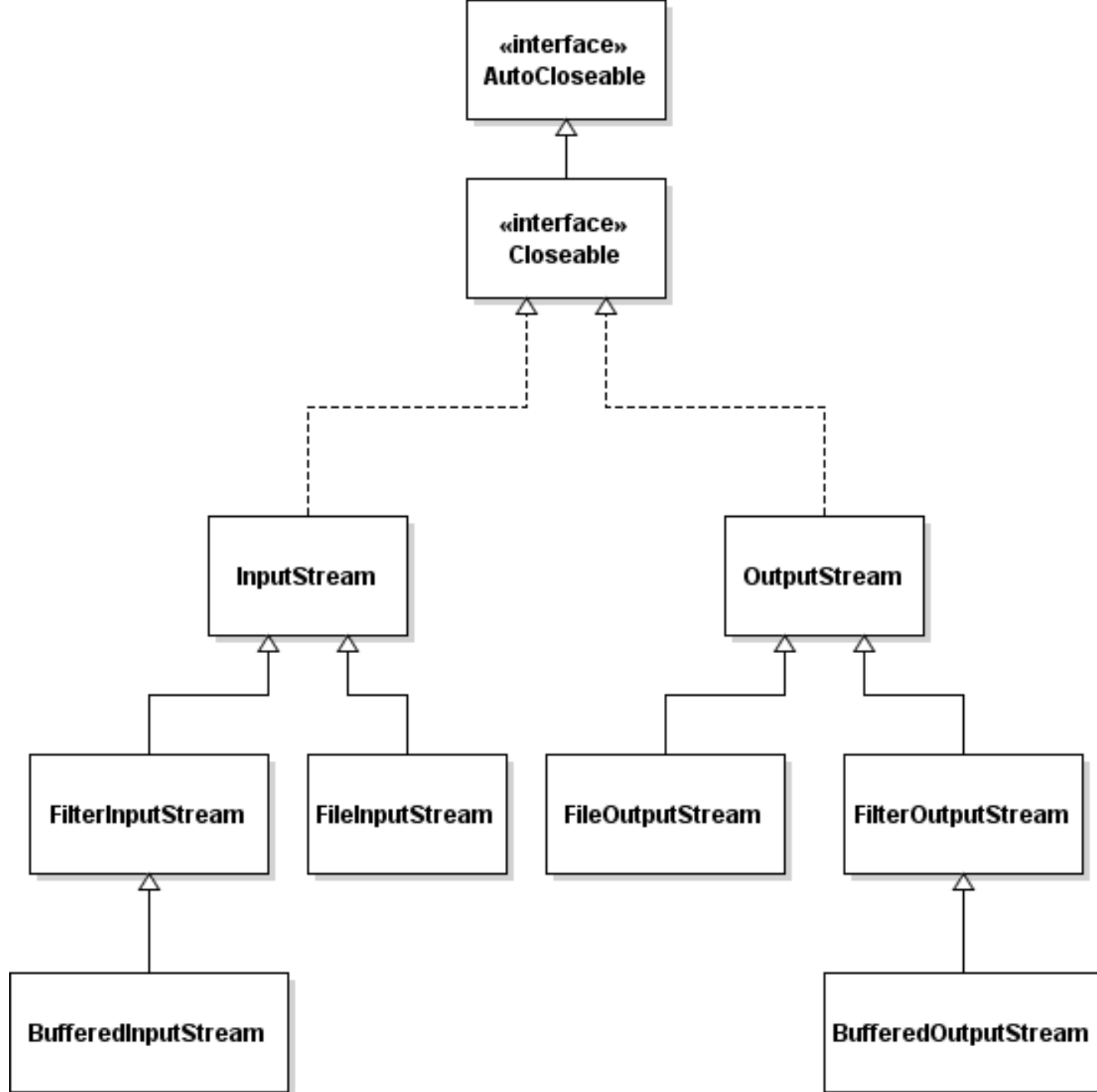
Class Scanner(3)

- **A l'aide de ces méthodes, il est très facile de parser une chaîne dont vous maîtrisez parfaitement le format, par exemple un fichier .CSV :**

```
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());  
}
```

Fichier Binaire

- Utilisez `FileInputStream` / `FileOutputStream` pour ouvrir le fichier.
- Optionnellement, encapsulez-le dans un `BufferedInputStream` / `BufferedOutputStream` pour améliorer les performances.
- Lire les données avec `read()` / `read(byte[])`.
- Ecrire avec `write()` / `write(byte[])`.



Fermeture automatique des ressources :

- Toute ressource qui implémente l'interface `AutoCloseable` (comme `InputStream`, `OutputStream`, `Scanner`, etc.) peut être utilisée avec try-with-resources.
- Ces ressources sont fermées automatiquement lorsque le bloc `try` se termine.

```
FileInputStream inputStream = null;
try {
    inputStream = new FileInputStream("fichier.txt");
    // Traitement du fichier
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (inputStream != null) {
        try {
            inputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
try (FileInputStream inputStream = new
FileInputStream("fichier.txt")) {
    // Traitement du fichier
} catch (IOException e) {
    e.printStackTrace();
}
// Pas besoin de bloc finally pour fermer
le flux !
```

```

import java.io.*;

public class CopyFiles {
    public static void main(String[] args) {

        if (args.length < 2) {
            System.out.println("Please provide input and output files");
            System.exit(0);
        }

        String inputFile = args[0];
        String outputFile = args[1];

        try (
            InputStream inputStream = new FileInputStream(inputFile);
            OutputStream outputStream = new FileOutputStream(outputFile);
        ) {
            int byteRead = -1;

            while ((byteRead = inputStream.read()) != -1) {
                outputStream.write(byteRead);
            }

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

```

```

import java.io.*;

public class CopyFilesOne {
    public static void main(String[] args) {

        if (args.length < 2) {
            System.out.println("Please provide input and output files");
            System.exit(0);
        }

        String inputFile = args[0];
        String outputFile = args[1];

        try (
            InputStream inputStream = new FileInputStream(inputFile);
            OutputStream outputStream = new FileOutputStream(outputFile);
        ) {
            long fileSize = new File(inputFile).length();
            byte[] allBytes = new byte[(int) fileSize];

            int bytesRead = inputStream.read(allBytes);

            outputStream.write(allBytes, 0, bytesRead);

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

```

```

import java.io.*;

public class CopyFilesChunk {
    private static final int BUFFER_SIZE = 4096; // 4KB

    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Please provide input and output files");
            System.exit(0);
        }

        String inputFile = args[0];
        String outputFile = args[1];

        try (
            InputStream inputStream = new FileInputStream(inputFile);
            OutputStream outputStream = new FileOutputStream(outputFile);
        ) {
            byte[] buffer = new byte[BUFFER_SIZE];
            int bytesRead = -1;

            while ((bytesRead = inputStream.read(buffer)) != -1) {
                outputStream.write(buffer, 0, bytesRead);
            }

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

```

```

private static int[] pngSignature = {137, 80, 78, 71, 13, 10, 26, 10};
public static void main(String[] args) {
    if (args.length < 1) {
        System.out.println("Please provide the input file");
        System.exit(0);
    }
    String inputFile = args[0];
    try (
        InputStream inputStream = new FileInputStream(inputFile);
    ) {
        int[] headerBytes = new int[8];
        boolean isPNG = true;

        for (int i = 0; i < 8; i++) {

            headerBytes[i] = inputStream.read();

            if (headerBytes[i] != pngSignature[i]) {
                isPNG = false;
                break;
            }
        }
        System.out.println("Is PNG file? " + isPNG);
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

```

Type de fichier	Signature magique (Décimal)
PNG	137, 80, 78, 71, 13, 10, 26, 10
JPEG/JPG	255, 216, 255
GIF (GIF87a)	71, 73, 70, 56, 55, 97
GIF (GIF89a)	71, 73, 70, 56, 57, 97
BMP	66, 77
PDF	37, 80, 68, 70

- **MimetypesFileTypeMap** est utile pour obtenir rapidement un type MIME basé sur l'extension d'un fichier.
- Facile à personnaliser avec vos propres mappages MIME.

```
import javax.activation.MimetypesFileTypeMap;
import java.io.File;

public class MimeTypeExample {
    public static void main(String[] args) {
        File file = new File("example.pdf");

        MimetypesFileTypeMap fileTypeMap = new MimetypesFileTypeMap();

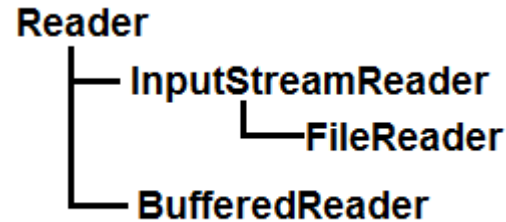
        String mimeType = fileTypeMap.getContentType(file);

        System.out.println("Type MIME : " + mimeType);
    }
}
```

BufferedInputStream / BufferedOutputStream pour améliorer les performances.

```
try (  
    InputStream inputStream = new BufferedInputStream(new  
FileInputStream(inputFile));  
    OutputStream outputStream = new BufferedOutputStream(new  
FileOutputStream(outputFile));  
) {  
    byte[] buffer = new byte[BUFFER_SIZE];  
    int bytesRead = -1;  
  
    while ((bytesRead = inputStream.read(buffer)) != -1) {  
        outputStream.write(buffer, 0, bytesRead);  
    }  
} catch (IOException ex) {  
    ex.printStackTrace();  
}
```

Fichier Texte



Reader :

- C'est la classe abstraite de base pour la lecture des flux de caractères en Java. Elle fournit les méthodes de base pour lire des caractères d'un flux.
- **Méthodes principales :**
 - `read()`: Lit un caractère.
 - `read(char[])`: Lit un tableau de caractères.
 - `skip(long)`: Sauter un certain nombre de caractères dans le flux.
 - `close()`: Ferme le flux.

Fichier Text

InputStreamReader :

- `InputStreamReader` est une classe qui fait la passerelle entre les flux de bytes (`InputStream`) et les flux de caractères (`Reader`).
- Elle convertit les octets en caractères à l'aide d'un encodage spécifié. par défaut sera l'encodage de caractères du système d'exploitation.
- **Utilisation** : Elle est utilisée lorsque les données à lire sont dans un format binaire (comme des fichiers ou des flux réseau en byte) et doivent être converties en texte.

```
InputStreamReader reader = new InputStreamReader(  
    new FileInputStream("MyFile.txt"), "UTF-16");
```

Fichier Text

FileReader :

- **FileReader** est une classe pratique pour lire des fichiers texte en utilisant l'encodage de caractères par défaut du système d'exploitation.
- Elle hérite de **InputStreamReader** et est spécialisée dans la lecture de fichiers. **FileReader** utilise l'encodage par défaut du système pour lire les caractères.
- **Méthodes supplémentaires** par rapport à **InputStreamReader** : elle ne demande pas explicitement un encodage, car elle utilise celui par défaut.

```
FileReader reader = new FileReader("MyFile.txt");
```

Fichier Text

BufferedReader :

- **BufferedReader** lit le texte d'un flux de caractères avec plus d'efficacité en utilisant un tampon interne. Cela signifie qu'elle permet de lire les données en bloc, ce qui réduit le nombre de lectures directes sur le flux de base, et donc les performances sont meilleures.
- Elle fournit des méthodes pratiques pour lire des lignes de texte (`readLine()`).
- **Utilisation** : **BufferedReader** est souvent utilisé avec **FileReader** ou **InputStreamReader** pour ajouter un tampon aux opérations de lecture.

```
InputStreamReader reader = new InputStreamReader(  
    new FileInputStream("MyFile.txt"), "UTF-16");
```

```
BufferedReader bufReader = new BufferedReader(reader);
```

```
package net.codejava.io;

import java.io.FileReader;
import java.io.IOException;

public class TextFileReadingExample1 {
    public static void main(String[] args) {
        try {
            FileReader reader = new FileReader("MyFile.txt");
            int character;

            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
            reader.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
package net.codejava.io;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class TextFileReadingExample2 {

    public static void main(String[] args) {
        try {
            FileInputStream inputStream = new FileInputStream("MyFile.txt");
            InputStreamReader reader = new InputStreamReader(inputStream,
"UTF-16");
            int character;

            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
            reader.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
package net.codejava.io;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TextFileReadingExample3 {

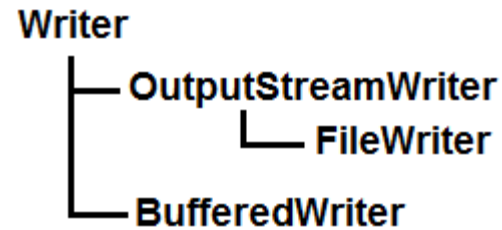
    public static void main(String[] args) {
        try {
            FileReader reader = new FileReader("MyFile.txt");
            BufferedReader bufferedReader = new BufferedReader(reader);

            String line;

            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Fichier Texte



La classe `Writer` est l'abstraction pour l'écriture dans des flux de caractères en Java. Elle implémente les méthodes fondamentales suivantes :

- `write(int)`: écrit un caractère unique.
- `write(char[])`: écrit un tableau de caractères.
- `write(String)`: écrit une chaîne de caractères.
- `close()`: ferme le flux.

Fichier Text

OutputStreamWriter

- `OutputStreamWriter` est un pont entre les flux d'octets et les flux de caractères. Les caractères sont convertis en octets à l'aide d'un jeu de caractères spécifié. Le jeu de caractères peut être l'encodage par défaut du système ou un jeu de caractères spécifique donné lors de la création de `OutputStreamWriter`.

```
OutputStreamWriter writer = new OutputStreamWriter(  
    new FileOutputStream("YourFile.txt"), "UTF-8");
```


Fichier Text

FileWriter

- `FileWriter` est une classe pratique pour écrire dans des fichiers texte en utilisant l'encodage de caractères par défaut du système d'exploitation.

```
FileWriter writer = new FileWriter("YourFile.txt");
```

Fichier Text

BufferedWriter

- `BufferedWriter` écrit du texte dans un flux de caractères avec efficacité (les caractères, les tableaux et les chaînes sont mis en tampon pour éviter l'écriture fréquente dans le flux sous-jacent). Elle fournit également une méthode pratique pour écrire un séparateur de ligne : `newLine()`.

```
OutputStreamWriter writer = new OutputStreamWriter(  
    new FileOutputStream("YourFile.txt"), "UTF-8");
```

```
BufferedWriter bufWriter = new BufferedWriter(writer);
```

```
package net.codejava.io;

import java.io.FileWriter;
import java.io.IOException;

public class TextFileWritingExample1 {

    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("MyFile.txt", true);
            writer.write("Hello World");
            writer.write("\r\n");    // write new line
            writer.write("Good Bye!");
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

package net.codejava.io;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class TextFileWritingExample2 {

    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("MyFile.txt", true);
            BufferedWriter bufferedWriter = new BufferedWriter(writer);

            bufferedWriter.write("Hello World");
            bufferedWriter.newLine();
            bufferedWriter.write("See You Again!");

            bufferedWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```
package net.codejava.io;
```

```
import java.io.BufferedWriter;
```

```
import java.io.FileOutputStream;
```

```
import java.io.IOException;
```

```
import java.io.OutputStreamWriter;
```

```
public class TextFileWritingExample3 {
```

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

```
        try {
```

```
            FileOutputStream outputStream = new FileOutputStream("MyFile.txt");
```

```
            OutputStreamWriter outputStreamWriter = new OutputStreamWriter(outputStream,  
16");
```

```
            BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);
```

```
            bufferedWriter.write("Xin chào");
```

```
            bufferedWriter.newLine();
```

```
            bufferedWriter.write("Hẹn gặp lại!");
```

```
            bufferedWriter.close();
```

```
        } catch (IOException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```