FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Gestion des fichiers et IO

Licence Professionnelle : Développement Informatique

Pr: Youssef Ouassit

Plan Chapitre 5: Gestion des fichiers et IO

1. Lire et écrire dans des fichiers avec les classes File, FileReader, FileWriter.
2. Introduction aux flux (streams) : flux d'entrée/sortie, flux bufferisés.
3. Utilisation des classes Scanner et PrintWriter.

Les entrées / sorties

- Dans la plupart des langages de programmation les notions d'entrées/sorties sont considérées comme une technique de base, car les manipulations de fichiers, notamment, sont très fréquentes.

- En Java, et pour des raisons de sécurité, on distingue deux cas :
  - le cas des applications Java autonomes, où, comme dans n'importe quel autre langage, il est généralement fait un usage important de fichiers,
  - le cas des applets Java qui, ne peuvent pas, en principe, accéder, tant en écriture qu'en lecture, aux fichiers de la machine sur laquelle s'exécute le navigateur (machine cliente).

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 :

Méthodes	Description
File(String pathname)	Ce constructeur crée une nouvelle instance de File en fonction du chemin spécifié sous forme de chaîne de caractères (String). Le chemin peut être absolu ou relatif.
File(URI uri)	Ce constructeur crée une instance de File en fonction d'un objet URI. Il est particulièrement utile lorsque vous travaillez avec des chemins sous forme d'URI.
File(String parent, String child)	Ce constructeur permet de spécifier un chemin parent et un nom de fichier ou de répertoire enfant. Le chemin complet sera le résultat de la concaténation du chemin parent et du nom de l'enfant.
File(File parent, String child)	Ce constructeur est similaire au précédent, mais le chemin parent est représenté par un objet File au lieu d'une chaîne de caractères.

La gestion des fichiers (3)

Méthodes de création et suppression de la classe File :

Méthodes	Description
boolean **createNewFile**()	Crée un nouveau fichier vide s'il n'existe pas déjà.
boolean **mkdir**()	Crée un répertoire unique
boolean **mkdirs**()	Crée les répertoires manquants dans le chemin spécifié.
boolean **delete**()	Supprime le fichier ou le répertoire. Pour les répertoires, il doit être vide.

La gestion des fichiers (4)

Méthodes d'information de la classe File :

Méthodes	Description
boolean **exists**()	Vérifie si le fichier ou le répertoire existe
String **getName**()	Renvoie le nom du fichier ou du répertoire
String **getPath**()	Renvoie le chemin spécifié lors de la création de l'objet File
String **getAbsolutPath**()	Renvoie le chemin absolu
long **length**()	Renvoie la taille du fichier en octets
long **lastModified**()	Renvoie la date de la dernière modification du fichier

La gestion des fichiers (5)

Méthodes d'information et permission de la classe File :

Méthodes	Description
boolean **exists**()	Vérifie si le fichier ou le répertoire existe
String **getName**()	Renvoie le nom du fichier ou du répertoire
String **getPath**()	Renvoie le chemin spécifié lors de la création de l'objet File
String **getAbsolutPath**()	Renvoie le chemin absolu
long **length**()	Renvoie la taille du fichier en octets
long **lastModified**()	Renvoie la date de la dernière modification du fichier
boolean **isFile**()	Vérifie si l'objet est un fichier
boolean **isDirectory**()	Vérifie si l'objet est un répertoire
boolean **canRead**()	Vérifient les permissions de lecture
boolean **canWrite**()	Vérifient les permissions d’écriture

La gestion des fichiers (6)

Méthodes pour lister les fichiers et répértoires de la classe File :

Méthodes	Description
String[] **list**()	Renvoie un tableau de chaînes contenant les noms des fichiers dans le répertoire
File[] **listFiles**()	Renvoie un tableau d'objets File représentant les fichiers et sous-répertoires dans le répertoire

La gestion des fichiers (7)

```
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 gestion des fichiers (8)

```
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());
               }
          }
     }
}
```

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)

- L'intérêt de la notion de flux est qu'elle permet une gestion homogène :
  - quelle que soit la ressource associée au flux de données,
  - quel que soit le flux (entrée ou sortie).
- Certains flux peuvent être associés à des filtres
  - Combinés à des flux d'entrée ou de sortie, ils permettent de traduire les données.

Notion de flux (4)

- Les flux sont regroupés dans le paquetage java.io
- 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

En Java, pour exploiter un flux, les étapes générales consistent à ouvrir, lire (ou écrire), puis fermer le flux. Voici les étapes détaillées :

1. Ouverture d'un flux
2. Lecture/Ecriture depuis ou dans le flux
3. Fermeture du flux

! Prévoir une gestion des exceptions

Flux d'octets et flux de caractères

Les flux sont décomposés en deux grandes familles:

- Les flux de caractères
  - classes abstraites **Reader** et **Writer** et leurs sous-classes concrètes respectives.

- Les flux d'octets
  - classes abstraites **InputStream** et **OutputStream** et leurs sous-classes concrètes respectives,

Flux d'octets et flux de caractères

La différence principale entre les classes **Reader** et **InputStream** en Java repose sur le type de données qu'elles gèrent :

- Utilisation :

  - Utilisez **InputStream** pour les données binaires (par exemple, des fichiers audio, des images, des fichiers binaires).

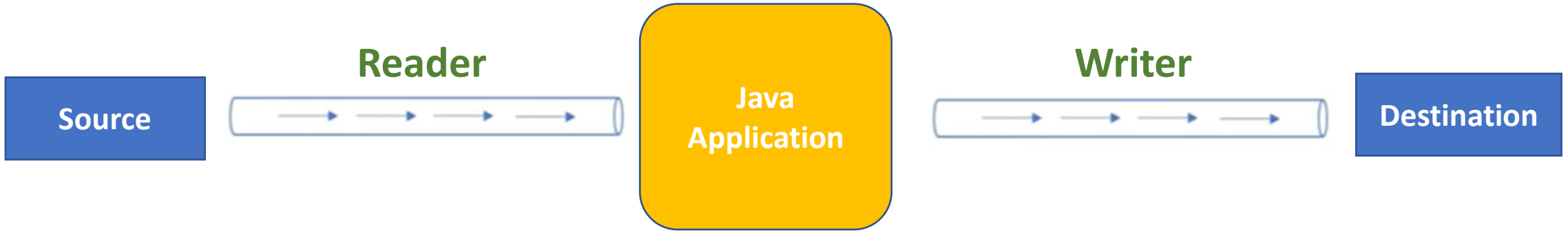
  - Utilisez **Reader** pour des données textuelles (par exemple, des fichiers texte, des flux de caractères).

- Encodage :

  - **InputStream** ne gère pas l'encodage, il lit des octets bruts.

  - **Reader** gère l'encodage des caractères, ce qui est important pour la lecture de texte dans des formats comme UTF-8 ou UTF-16.

Flux de caractères (1)



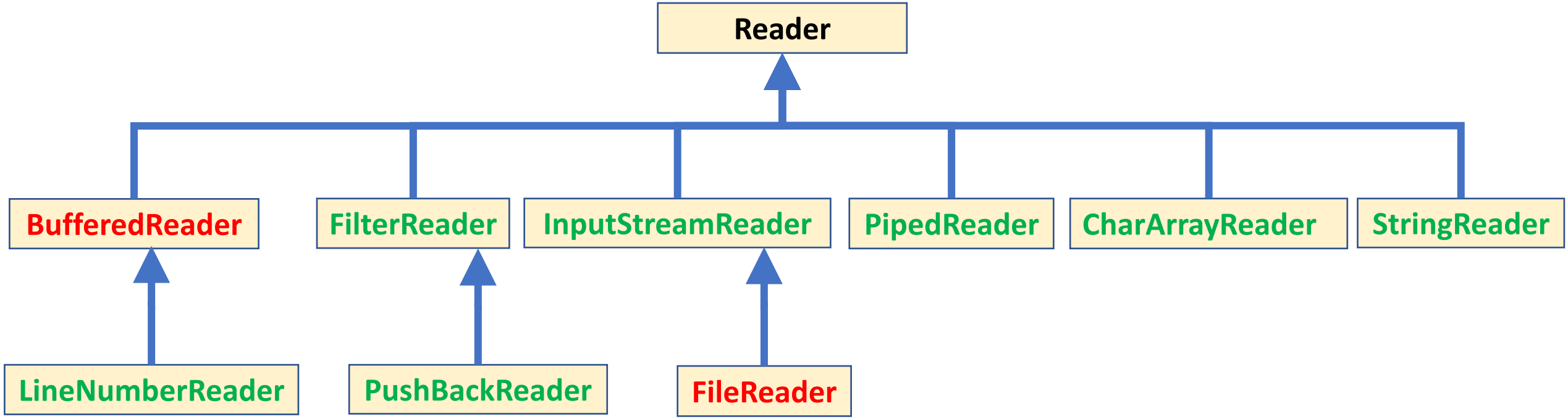
Méthodes de lecture :

int read() : Lit un caractère à la fois et retourne un entier représentant la valeur du caractère, ou -1 si la fin du flux est atteinte.
int read(char[] cbuf) : Lit des caractères dans un tableau et retourne le nombre de caractères lus.
void close() : Ferme le flux et libère les ressources associées.

Méthodes d'écriture:

void write(int c) : Écrit un seul caractère.
void write(char[] cbuf): Écrit un tableau de caractères.
void write(String str): Écrit une chaîne de caractères.
void close() : Ferme le flux d'écriture et libère les ressources.

Flux de caractères - Flux d'entré



Flux de caractères : La classe FileReader

FileReader est une classe de la bibliothèque Java qui permet de lire des fichiers texte. Elle fait partie du package java.io et est utilisée pour lire des caractères à partir d'un fichier.

Flux de caractères : La classe FileReader

```
try {
	// Ouvrir le fichier avec FileReader
	FileReader reader = new FileReader("example.txt");
	int character;
	// Lire chaque caractère du fichier
	while ((character = reader.read()) != -1) {
		// Afficher le caractère (converti en char)
		System.out.print((char) character);
	}
} catch (IOException e) {
	e.printStackTrace();
}
```

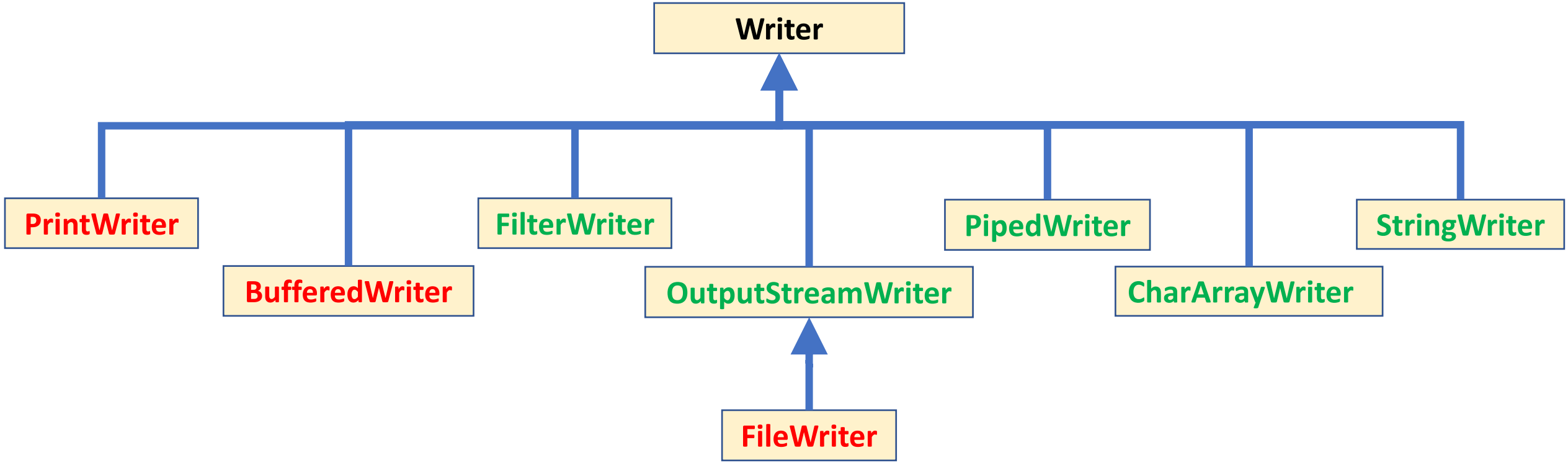
Flux de caractères : La classe BufferedReader

BufferedReader est une classe de la bibliothèque Java qui permet de lire des caractères à partir d'un flux d'entrée de manière plus efficace, en utilisant un tampon (buffer). Elle fait partie du package java.io et est utilisée pour lire des données texte de manière rapide et optimisée, notamment pour les fichiers texte ou les flux de données.

Flux de caractères : La classe BufferedReader

```
try {
        // Ouvrir le fichier
        BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
        String line;
        // Lire chaque ligne du fichier
        while ((line = reader.readLine()) != null) {
                // Traiter la ligne
                String[] values = line.split(",");
                for (String value : values) {
                        System.out.print(value + " ");
                }
                System.out.println();
        }
} catch (IOException e) {
        e.printStackTrace();
}
```

Flux de caractères - Flux de sortie



Flux de caractères : La classe FileWriter

FileWriter est la manière la plus basique pour écrire dans un fichier texte. Si le fichier n'existe pas, il sera créé. Si le fichier existe, son contenu sera écrasé.

java

Flux de caractères (5) : La classe FileWriter

```
String text = "Ligne 1 : Bonjour\nLigne 2 : Ceci est un exemple.";
try {
        // Créer un objet FileWriter pour écrire dans le fichier
        FileWriter writer = new FileWriter("output.txt");
        // Écrire du texte dans le fichier
        writer.write(text);
        // Fermer le FileWriter après utilisation
        writer.close();
        System.out.println("Écriture réussie dans le fichier.");
} catch (IOException e) {
        System.out.println("Erreur lors de l'écriture dans le fichier : " + e.getMessage());
}
```

Flux de caractères : La classe BufferedWriter

BufferedWriter est une version plus performante de FileWriter car il utilise un tampon (buffer) pour écrire les données. Cela permet d'écrire plus efficacement, en particulier lorsqu'on travaille avec de grandes quantités de données.

Flux de caractères : La classe BufferedWriter

```
String text = "Ligne 1 : Bonjour\nLigne 2 : Ceci est un exemple.";
try {
        // Créer un BufferedWriter pour écrire dans le fichier avec un tampon
        BufferedWriter writer = new BufferedWriter(new FileWriter("outputBuffered.txt"));
        // Écrire du texte dans le fichier
        writer.write(text);
        // Ajouter une nouvelle ligne
        writer.newLine();
        writer.write("Ceci est une nouvelle ligne.");
        // Fermer le BufferedWriter après utilisation
        writer.close();
        System.out.println("Écriture réussie avec BufferedWriter.");
} catch (IOException e) {
        System.out.println("Erreur lors de l'écriture dans le fichier : " + e.getMessage());
}
```

Flux de caractères : La classe PrintWriter

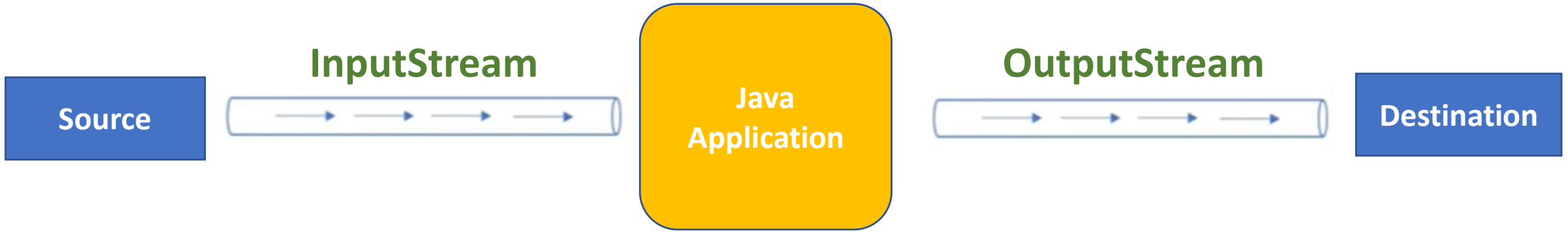
PrintWriter est une autre classe pratique pour écrire dans un fichier. Elle permet d'écrire des chaînes de caractères et d'autres types de données, tout en offrant une gestion facile des nouvelles lignes.

Flux de caractères : La classe PrintWriter

```
String text = "Ceci est écrit dans le fichier avec PrintWriter.";
try {
        // Créer un PrintWriter pour écrire dans le fichier
        PrintWriter writer = new PrintWriter("outputPrintWriter.txt");

        // Écrire du texte dans le fichier
        writer.println(text);
        writer.println("Une nouvelle ligne de texte.");

        // Fermer le PrintWriter après utilisation
        writer.close();

        System.out.println("Écriture réussie avec PrintWriter.");
} catch (FileNotFoundException e) {
        System.out.println("Erreur lors de la création du fichier : " + e.getMessage());
}
```

Gestion des fichiers et IO

Flux d'octets



Méthodes de lecture :

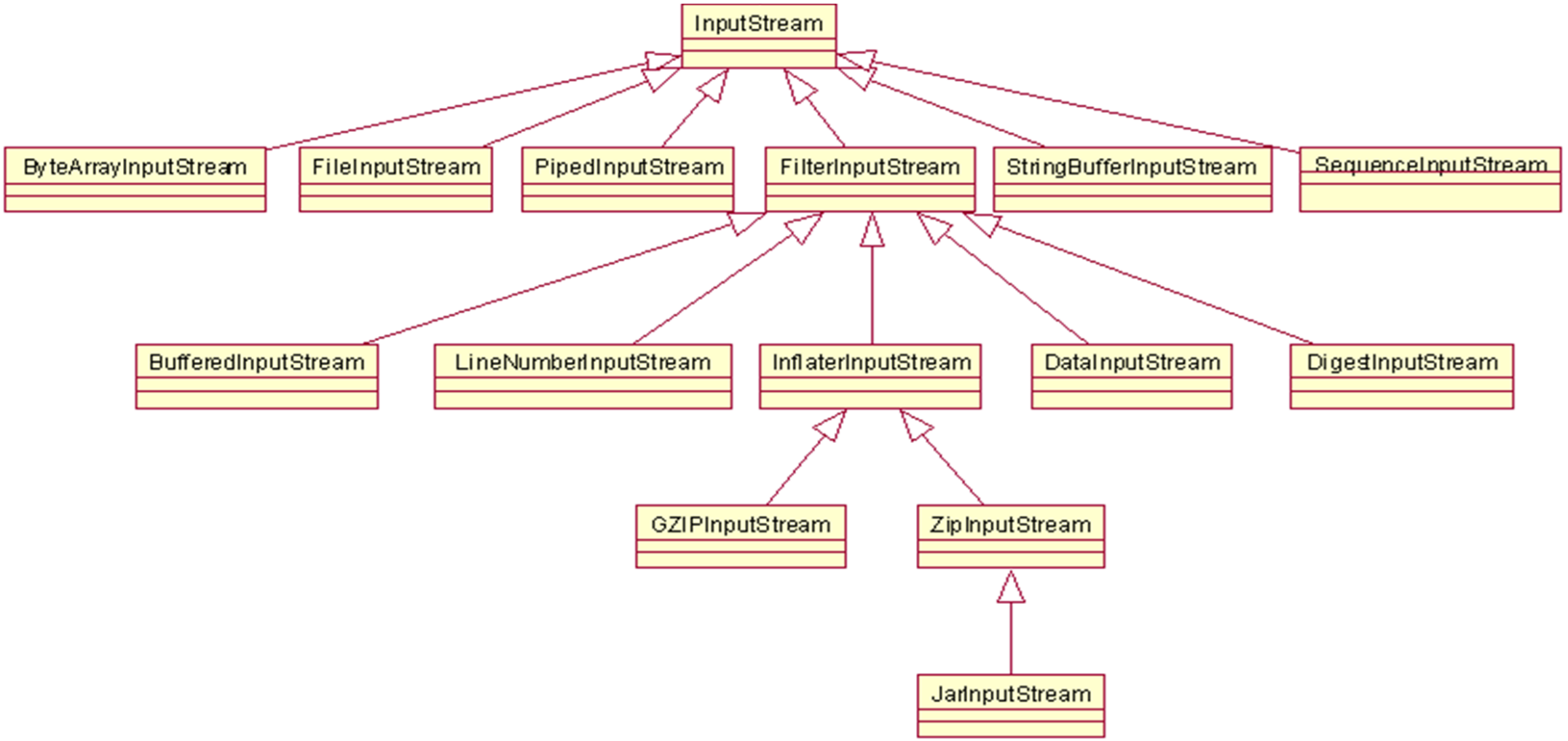
int read() : Lit et renvoie un octet du flux de données, ou -1 si la fin du flux est atteinte.
int read(byte[] b) : Lit des octets dans un tableau et retourne le nombre d'octets lus.
long skip(long n) : Ignore (saute) jusqu'à n octets de données dans le flux d'entrée. Renvoie le nombre d'octets ignorés.
void close() : Ferme le flux et libère les ressources associées.

Méthodes d'écriture:

void write(int b) : Écrit un seul octet.
void write(byte[] b): Écrit un tableau d'octets.
void write(byte[] b, int off, int len): Écrit les octets du tableau b à partir de l'index off jusqu'à len octets dans le flux de sortie.
void close() : Ferme le flux d'écriture et libère les ressources.
flush(): qui permet de purger le tampon en cas d'écritures bufferisées.

Gestion des fichiers et IO

Flux d'octets - Flux d'entré



Flux de caractères : La classe FileInputStream

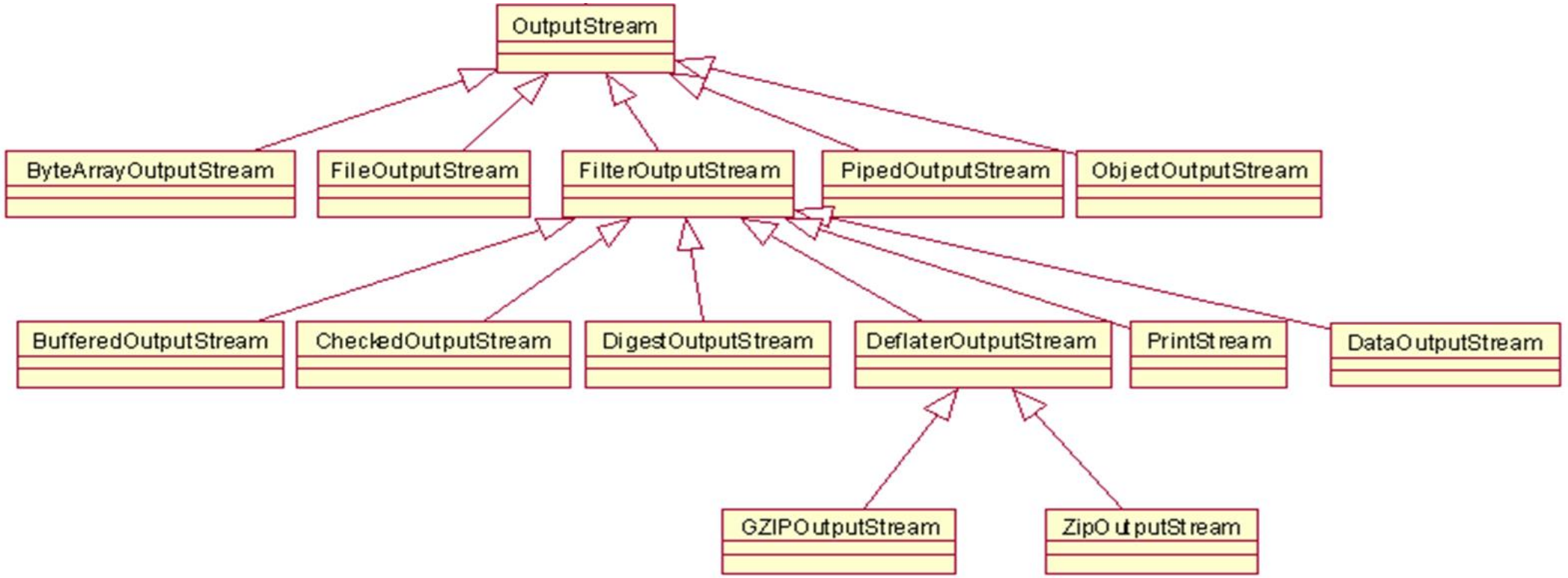
BufferedWriter est une version plus performante de FileWriter car il utilise un tampon (buffer) pour écrire les données. Cela permet d'écrire plus efficacement, en particulier lorsqu'on travaille avec de grandes quantités de données.

Flux de caractères : La classe FileInputStream

```
String text = "Ligne 1 : Bonjour\nLigne 2 : Ceci est un exemple.";
try {
        // Créer un BufferedWriter pour écrire dans le fichier avec un tampon
        BufferedWriter writer = new BufferedWriter(new FileWriter("outputBuffered.txt"));
        // Écrire du texte dans le fichier
        writer.write(text);
        // Ajouter une nouvelle ligne
        writer.newLine();
        writer.write("Ceci est une nouvelle ligne.");
        // Fermer le BufferedWriter après utilisation
        writer.close();
        System.out.println("Écriture réussie avec BufferedWriter.");
} catch (IOException e) {
        System.out.println("Erreur lors de l'écriture dans le fichier : " + e.getMessage());
}
```

Gestion des fichiers et IO

Flux d'octets - Flux de sortie



Flux d'octets : La classe FileOutputStream

BufferedWriter est une version plus performante de FileWriter car il utilise un tampon (buffer) pour écrire les données. Cela permet d'écrire plus efficacement, en particulier lorsqu'on travaille avec de grandes quantités de données.

Flux d'octets : La classe FileOutputStream

```
String text = "Ligne 1 : Bonjour\nLigne 2 : Ceci est un exemple.";
try {
        // Créer un BufferedWriter pour écrire dans le fichier avec un tampon
        BufferedWriter writer = new BufferedWriter(new FileWriter("outputBuffered.txt"));
        // Écrire du texte dans le fichier
        writer.write(text);
        // Ajouter une nouvelle ligne
        writer.newLine();
        writer.write("Ceci est une nouvelle ligne.");
        // Fermer le BufferedWriter après utilisation
        writer.close();
        System.out.println("Écriture réussie avec BufferedWriter.");
} catch (IOException e) {
        System.out.println("Erreur lors de l'écriture dans le fichier : " + e.getMessage());
}
```

Empilement de flux:

- 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:

- 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:

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);
```

La sérialisation

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 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;}
}
```

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)

La class Scanner

- Une classe injustement méconnue du JDK est Scanner (depuis la verison 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();
```

La class Scanner

- 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.

La class Scanner

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());
}
```