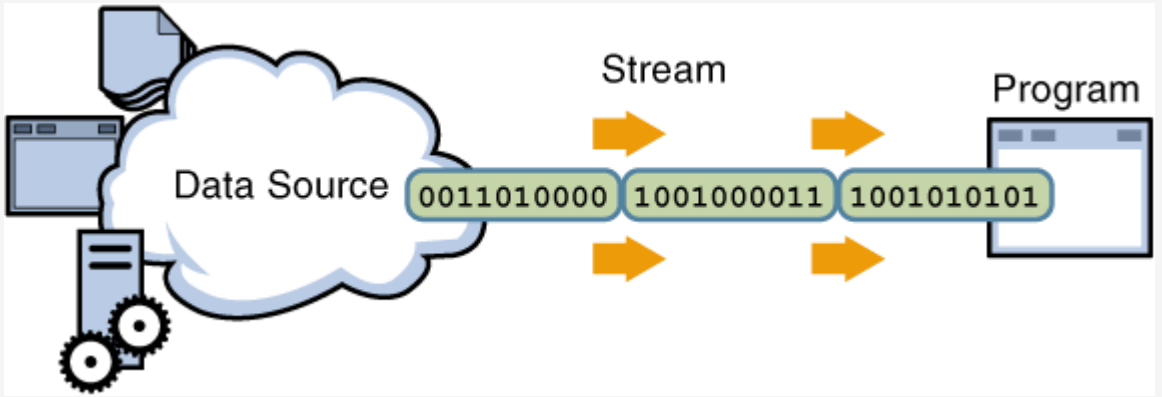
java.io.*

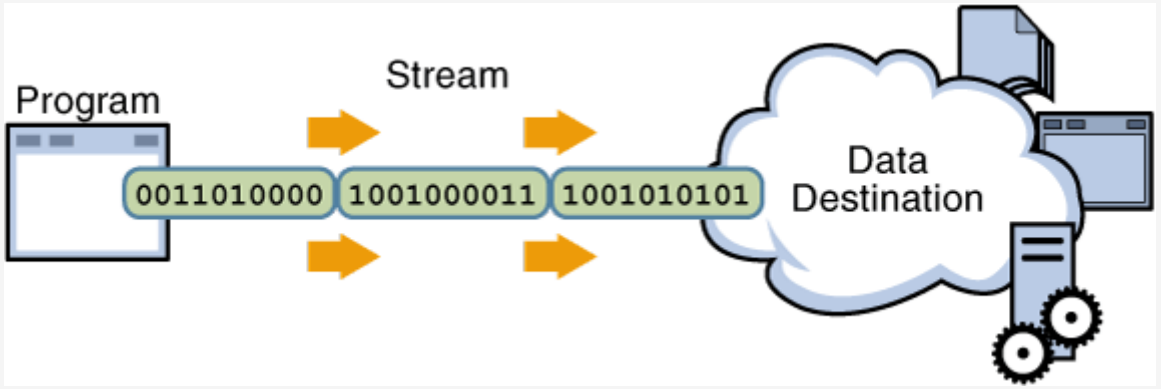
java.io: les bases

- Notion de « **I/O Stream** » (courant,ruisseau, flux)

input stream



output stream



Byte streams

- Les programmes utilisent des bytes de 8-bits pour réaliser des entrées/sorties de byte streams
- Toutes les classes de byte streams héritent de ***java.io.InputStream*** ou ***java.io.OutputStream***
- Il en existe de nombreuses, pour différents types de flux (audio, fichiers, objets, etc.)

java.io.OutputStream

java.io

Class OutputStream

java.lang.Object
  └ java.io.OutputStream

All Implemented Interfaces:
Closeable, Flushable

Direct Known Subclasses:
ByteArrayOutputStream, FileOutputStream, FilterOutputStream, ObjectOutputStream, OutputStream, PipedOutputStream

---

```
public abstract class OutputStream
extends Object
implements Closeable, Flushable
```

This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Applications that need to define a subclass of `OutputStream` must always provide at least a method that writes one byte of output.

java.io.OutputStream

Constructor Summary

OutputStream()

Method Summary

void	close() Closes this output stream and releases any system resources associated with this stream.
void	flush() Flushes this output stream and forces any buffered output bytes to be written out.
void	write(byte[] b) Writes b.length bytes from the specified byte array to this output stream.
void	write(byte[] b, int off, int len) Writes len bytes from the specified byte array starting at offset off to this output stream.
abstract void	write(int b) Writes the specified byte to this output stream.

java.io.InputStream

java.io

Class InputStream

java.lang.Object
  └ **java.io.InputStream**

All Implemented Interfaces:

Closeable

Direct Known Subclasses:

AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, InputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream

---

```
public abstract class InputStream
extends Object
implements Closeable
```

This abstract class is the superclass of all classes representing an input stream of bytes.

Applications that need to define a subclass of `InputStream` must always provide a method that returns the next byte of input.

java.io.InputStream

Constructor Summary

`InputStream()`

Method Summary

int	`available()` Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.
void	`close()` Closes this input stream and releases any system resources associated with the stream.
void	`mark(int readlimit)` Marks the current position in this input stream.
boolean	`markSupported()` Tests if this input stream supports the `mark` and `reset` methods.
abstract int	`read()` Reads the next byte of data from the input stream.
int	`read(byte[] b)` Reads some number of bytes from the input stream and stores them into the buffer array `b`.
int	`read(byte[] b, int off, int len)` Reads up to `len` bytes of data from the input stream into an array of bytes.
void	`reset()` Repositions this stream to the position at the time the `mark` method was last called on this input stream.
long	`skip(long n)` Skips over and discards `n` bytes of data from this input stream.

Byte streams

- Il en existe de nombreuses, pour différents types de flux (audio, fichiers, objets, etc.)

- En particulier, il existe deux classes de byte stream spécialisées pour le traitement de fichier :

  – java.io.FileInputStream

  – java.io.FileOutputStream

java.io.FileInputStream

java.io

Class FileInputStream

java.lang.Object
  └ java.io.InputStream
    └ **java.io.FileInputStream**

All Implemented Interfaces:

Closeable

```
public class FileInputStream
extends InputStream
```

A `FileInputStream` obtains input bytes from a file in a file system. What files are available depends on the host environment.

`FileInputStream` is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using `FileReader`.

java.io.FileInputStream

read

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

Reads a byte of data from this input stream. This method blocks if no input is yet available.

Specified by:

read in class InputStream

Returns:

the next byte of data, or -1 if the end of the file is reached.

Throws:

IOException - if an I/O error occurs.

FileOutputStream

write

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

Writes the specified byte to this file output stream. Implements the `write` method of `OutputStream`.

Specified by:

`write` in class `OutputStream`

Parameters:

`b` - the byte to be written.

Throws:

`IOException` - if an I/O error occurs.

FileOutputStream

java.io

Class FileOutputStream

java.lang.Object
  └ java.io.OutputStream
    └ **java.io.FileOutputStream**

All Implemented Interfaces:

Closeable, Flushable

```
public class FileOutputStream
extends OutputStream
```

A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one `FileOutputStream` (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

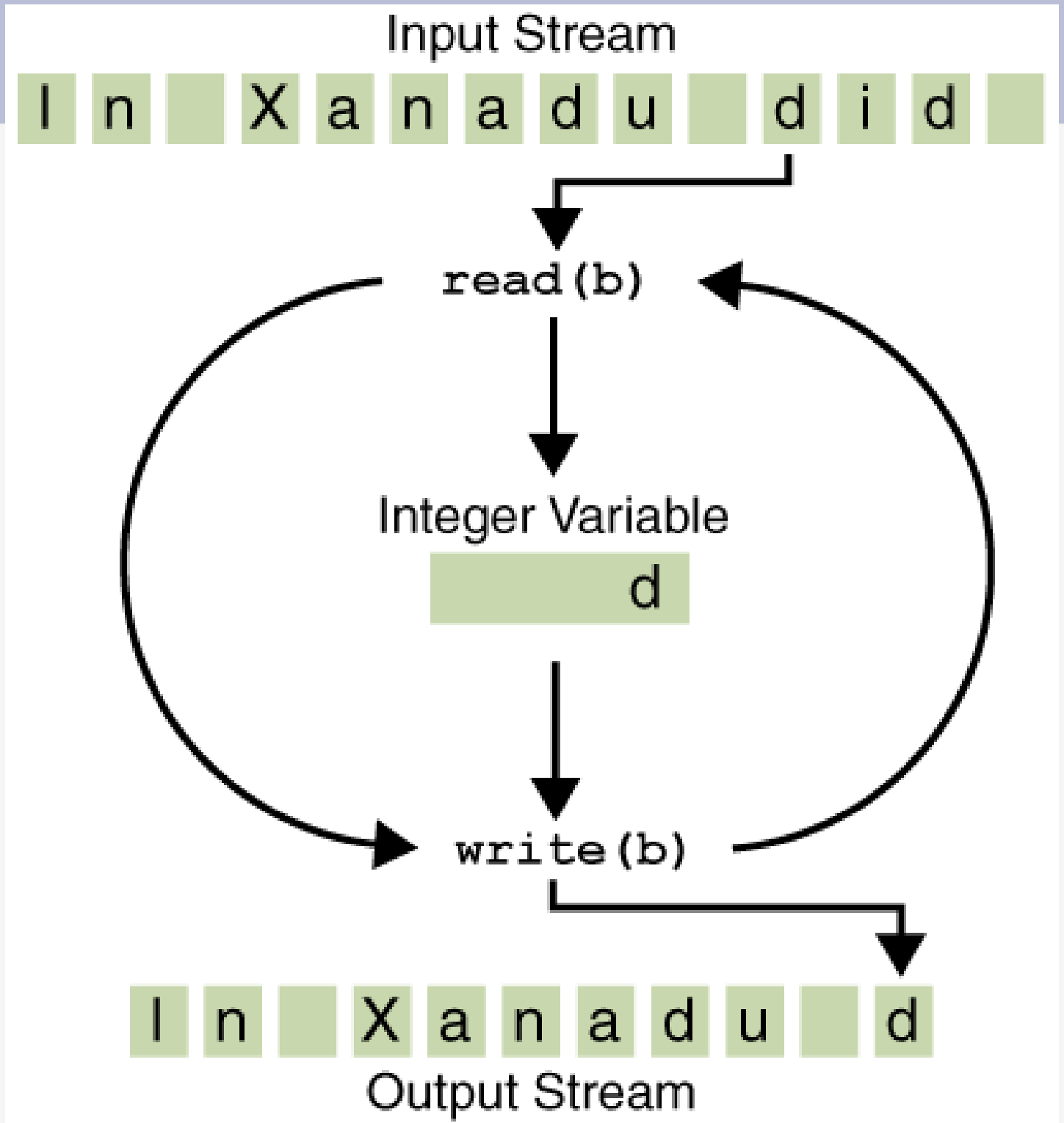
`FileOutputStream` is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using `FileWriter`.

Byte streams : exemple

- soit le fichier suivant, xanadu.txt :

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

```java
package test;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```


Input Stream
I n X a n a d u d i d
read(b)
Integer Variable
d
write(b)
I n X a n a d u d
Output Stream

Attention

Fermer un stream qui a été ouvert est **extrêmement important** !

```java
package test;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Mais en fait ...

- *CopyBytes*, bien qu'il fonctionne, utilise des I/O de très bas niveaux (byte stream).
- Eviter le plus possible d'utiliser ce type de stream.
- Par exemple, *xanadu.txt* est un fichier contenant des données de type « caractère ».
- → Il faut donc utiliser les streams appropriés : des *character streams.*
- Alors pourquoi parler de byte streams ?
- → Tous les autres streams sont basés sur eux !

Character Streams

Character Streams

- Java utilise les conventions unicode pour manipuler les variables de type caractère
- Les character streams convertissent automatiquement ce format interne en fonction du codage local (internationalisation).
- Toutes les classes de Character streams héritent de ***java.io.Reader*** ou ***java.io.Writer***
- Comme pour les byte streams, il existe deux classes spécialisées pour le traitement de fichier :
  - ***java.io.FileReader***
  - ***java.io.FileWriter***

```java
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Aucune différence ?

- *CopyBytes* et *CopyCharacters* sont très similaires
- La seule différence, au niveau du code, est que *CopyCharacters* utilise *FileReader* et *FileWriter* à la place de *FileInputStream FileOutputStream*
- Les deux utilisent un int pour lire et écrire, MAIS, dans *CopyCharacters* la valeur du int correspond à 16 bits (contre 8 bits dans *CopyCharacters)*

D'une manière générale

- Les character streams utilisent des byte streams pour les opérations IO : ils font le pont entre le byte stream et le format souhaité (« wrapper »).
- Par exemple, *FileReader* utilise *FileInputStream* et *FileWriter* utilise *FileOutputStream*
- Il existe 2 classes génériques qui permettent de faire le pont byte stream → character streams :
  - *java.io.InputStreamReader*
  - *java.io.OutputStreamWriter*
  - Utile lorsqu'il n'y pas de classe de character stream qui correspond au besoin (socket)

Character Streams : I/O de lignes

- Les I/O de caractères sont en général utilisées sur de plus grosses unités que le caractère.
- Une unité couramment utilisée est la ligne : i.e. une chaîne de caractères terminée par un caractère de terminaison :
  - **\r** : Carriage return « CR », ASCII 13, Hexa 0D (mac)
  - ou **\n** : Line feed « LF », ASCII 10, Hexa 0A (unix)
  - ou encore la séquence **\r\n** (windows cf. notepad)
- Deux classes permettent de faire des I/O utilisant la ligne comme unité :
  - *java.io.BufferedReader* et *java.io.PrintWriter*

java.io

Class BufferedReader

java.lang.Object
  └ java.io.Reader
    └ **java.io.BufferedReader**

readLine

```
public String readLine()
                throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

java.io

Class PrintWriter

java.lang.Object
  └ java.io.Writer
    └ **java.io.PrintWriter**

println

```
public void println(String x)
```

Prints a String and then terminates the line. This method behaves as though it invokes print(String) and then println().

Parameters:

x - the String value to be printed

```java
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

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

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

Buffered Streams

Buffered Streams

- La plupart des exemples précédents utilisent des I/O non « bufferisées » → chaque requête I/O est traitée immédiatement par l'OS.
- Cela peut rendre un programme peu efficace :
  - accès disque à répétition
  - surcharge d'activité réseau
  - etc.
- Solution → Utiliser des ***Buffered I/O Streams*** :
  - Ils travaillent sur une zone mémoire tampon : **buffer**
  - → les API natives de l'OS ne sont appelées que lorsque le tampon est vide (input) ou plein (output).

Buffered Streams

- Pour convertir un stream non bufferisé, il suffit d'utiliser les classes adéquates en passant au constructeur le stream à bufferiser.
- Par exemple, dans *CopyCharacters* :

```
BufferedReader inputStream = null;
BufferedWriter outputStream = null;
try {

    inputStream =
        new BufferedReader(new FileReader("xanadu.txt"));
    outputStream =
        new BufferedWriter(new FileWriter("characteroutput.txt"));
```

Buffered Streams

- Il existe 4 classes permettant cette conversion :
- Pour les byte streams :

java.io

Class BufferedInputStream

java.lang.Object
  └ java.io.InputStream
    └ java.io.FilterInputStream
      └ **java.io.BufferedInputStream**

java.io

Class BufferedOutputStream

java.lang.Object
  └ java.io.OutputStream
    └ java.io.FilterOutputStream
      └ **java.io.BufferedOutputStream**

- Pour les character streams :

java.io

Class BufferedReader

java.lang.Object
  └ java.io.Reader
    └ **java.io.BufferedReader**

java.io

Class BufferedWriter

java.lang.Object
  └ java.io.Writer
    └ **java.io.BufferedWriter**

« *Flushing* Buffered Streams »

- Il peut très utile de vider le buffer, en certains points de l'application, sans attendre qu'il soit plein.
- → vider le buffer : « ***flushing*** the buffer »
- Invoquer la méthode ***flush*** des classes de sorties bufferisées pour réaliser le flushing manuellement.
- Certaines classes bufferisées supportent l'autoflush, activé via un argument du constructeur.
- Lorsque l'autoflush est actif, certains événements clés déclenchent le vidage du buffer :
  - e.g. *PrintWriter* à chaque invocation de *println* et de *format.*

Scanning and Formatting

- Programmer des I/O implique souvent de traduire des données de et/ou vers des formats facilement compréhensible par l'homme.
- Pour faciliter cette tâche, java fournit 2 APIs :
  - L'API ***scanner*** : découpe une entrée en signes individuels et les transforme en données suivant leur type.
  - L'API ***formatting*** qui assemble les données pour les mettre sous une forme facilement compréhensible par l'homme.

java.util.Scanner

- Les objets de type ***java.util.Scanner*** servent à convertir une entrée formatée en données individuelles suivant leur type.
- Par défaut, un scanner utilise les espaces pour découper une entrée (blanc, tabulation et terminaison de ligne → *Character.isWhitespace*)

```java
import java.io.*;
import java.util.Scanner;

public class ScanXan {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

```
In
Xanadu
did
Kubla
Khan
A
stately
pleasure-dome
...
```

java.util.Scanner

- Un scanner n'est pas un stream mais il est nécessaire d'utiliser *close()* pour indiquer qu'on en a fini avec le stream utilisé dans la construction.
- Pour utiliser un autre symbole de séparation, on utilise la méthode *useDelimiter(...)*
- Par exemple, pour découper suivant une virgule, optionnellement suivi d'un espace :
  - ***s.useDelimiter(",\\s*");*** (expression régulière)
- *ScanXan* traite tous les symboles en entrée comme des strings. Il est cependant possible de les onvertir dans n'importe quel type primitif java.

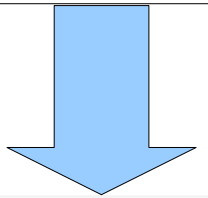
java.util.Scanner

- Cependant, attention à la *locale* utilisée !!
- Par exemple dans une locale us "32,767" est considéré comme un entier alors qu'il s'agit d'un double avec une locale fr !!
- Soit le fichier suivant usnumbers.txt :

```
8.5
32,767
3.14159
1,000,000.1
```

```
public class ScanSum {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        double sum = 0;
        try {
            s = new Scanner(
                    new BufferedReader(new FileReader("usnumbers.txt")));
            s.useLocale(Locale.US);

            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                        sum += s.nextDouble();
                    } else {
                        s.next();
                    }
            }
        } finally {
            s.close();
        }

        System.out.println(sum);
    }
}
```

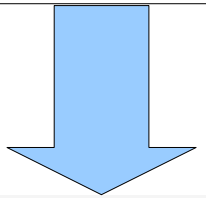
8.5
32,767
3.14159
1,000,000.1



1032778.74159

```
public class ScanSum {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        double sum = 0;
        try {
            s = new Scanner(
                    new BufferedReader(new FileReader("usnumbers.txt")));


            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                        sum += s.nextDouble();
                    } else {
                        s.next();
                    }
            }
        } finally {
            s.close();
        }

        System.out.println(sum);
    }
}
```

8.5
32,767
3.14159
1,000,000.1



? 32.767

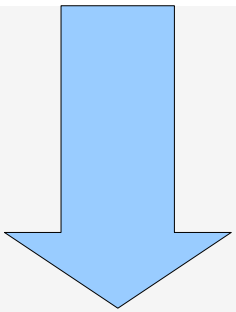
A propos de java.util.Scanner

- Contrairement à ce qu'on peut parfois lire (et faire), Scanner n'est pas une classe appropriée pour récupérer les entrées utilisateurs depuis la ligne de commande (elle n'a pas été créée pour ça) :
  - A cause des réglages par défaut (espaces)
  - L'entrée est supposée formatée : délimiteurs, type des données → faire des suppositions sur ce que va renter l'utilisateur peut compromettre la robustesse du programme
  - Par défaut, elle gère les exceptions en arrière plan.
  - Comme nous allons le voir, il faut lui préférer un InputStreamReader (character stream) pour récupérer les entrées de l'utilisateur.

Formatting

- Les objets stream qui implémentent des méthodes de formatage sont des instances de, soit
  - java.io.PrintWriter (character streams)
  - java.io.PrintStream(byte streams)
- Note: Les seuls objets de type *PrintStream* qu'on utilise généralement sont *System.out* et *System.err*
- Ainsi, lorsqu'on a besoin de créer un nouveau stream de sortie formaté on hérite de *PrintWriter*, pas de *PrintStream*
- *PrintWriter* et *PrintStream* définissent un ensemble de méthodes qui permettent le formatage :
  - *print* et *println* (standard) et *format* (pour les nombres)

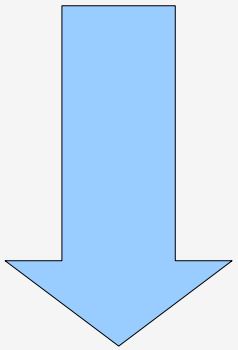
```java
public class Root {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");
    }
}
```



```
The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.
```

format : exemple

```
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```



```
The square root of 2 is 1.414214
```

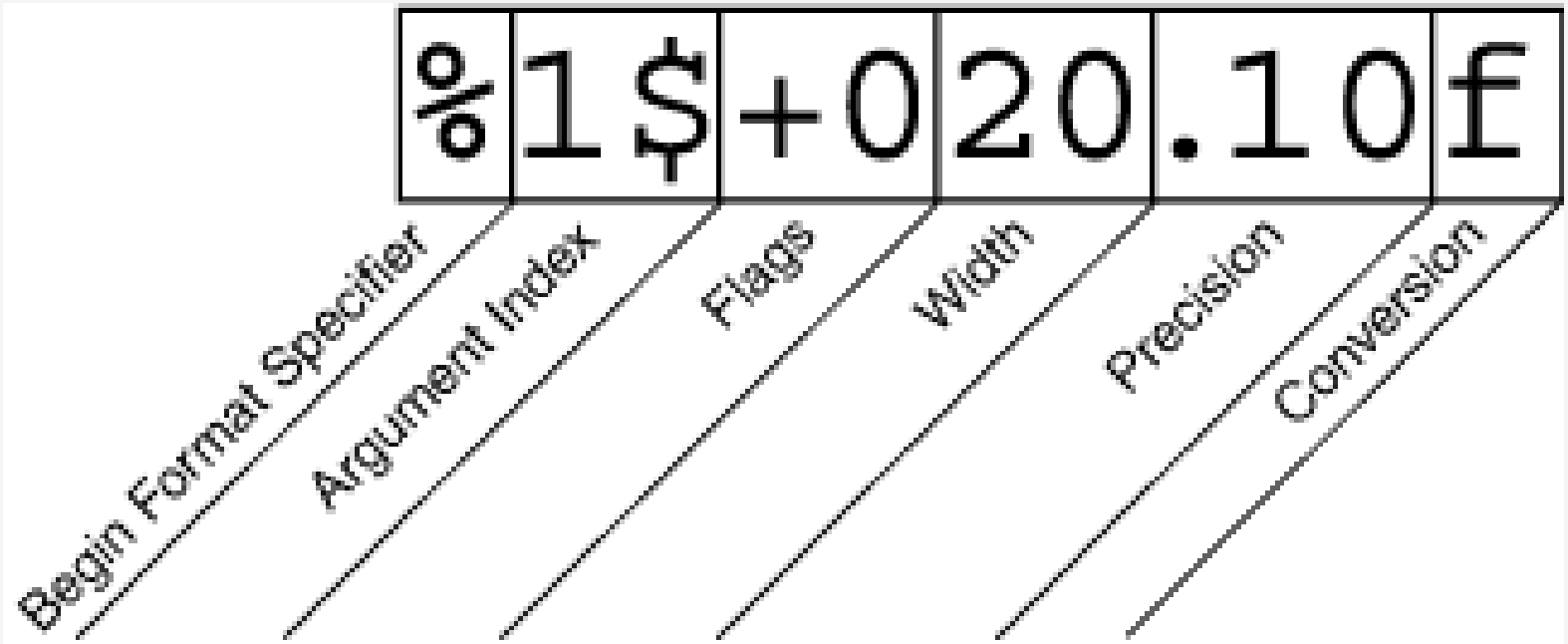
format : exemple

- %d : formate un entier comme une valeur décimale
- %f : formate un flottant comme une valeur décimale
- %n : caractère de terminaison de ligne de l'OS
- %x : entier → hexadécimal
- %s : n'importe quoi → string
- %tB : entier → le nom du mois (avec la locale)
- etc.
- Attention: excepté pour %% et %n, tous les % doivent correspondre à un argument → Exception !

format : exemple

```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

3.141593, +00000003.1415926536



I/O depuis la ligne de commande

- Un programme est souvent lancé depuis la ligne de commande et interagit parfois avec l'utilisateur depuis celle-ci.

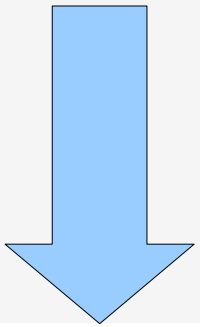
- Java permet ce type d'interaction de 2 manières :
  - Avec les streams standards
  - Avec la classe ***java.io.Console*** (**Java 6**)

Utilisation des streams standards

- Java définit 3 streams standards :
  - l'entrée : *Standard Input* → ***System.in***
  - la sortie standard : *Standard Output* → ***System.out***
  - la sortie d'erreur : Standard Error → ***System.err***
- Ces objets sont définis automatiquement au lancement et n'ont pas besoin d'être ouverts.

out, err et in → byte streams

- Ce sont des byte streams : *out* et *err* sont de type *PrintStream* mais il utilisent un objet interne qui émule les fonctionnalités des character streams
- A contrario, Sytem.in doit être explicitement converti en *InputStreamReader* pour bénéficier de ces fonctionnalités



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

```
private String saisieClavier(){
    try{
    BufferedReader clavier = new BufferedReader(new InputStreamReader(System.in));
    return clavier.readLine();
    }
    catch(IOException e){
        e.printStackTrace();
        System.exit(0);
        return null;
    }
}
```

java.io.Console (Java 6)

- Alternative aux streams standards → *Console*
- Cet objet prédéfini possède la plupart des fonctionnalités fournies par les streams standards et en ajoute d'autres
- Il est particulièrement utile pour réaliser une entrée sécurisée de mot de passe
- Console fournit des I/O basées sur de vrais character streams grâce à ces méthodes *reader* et *writer*
- Avant d'utiliser la console, un programme doit la récupérer en invoquant System.console()
  - Elle peut être null (OS ou environnement incompatibles)

java.io.Console (Java 6)

- La console supporte les entrées sécurisées de mot de passe → méthode *readPassword*
  - pas d'écho
  - retourne un tableau de caractères, pas une string → il peut être facilement enlever de la mémoire.

Exemple

```
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;

public class Password {

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

        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        String login = c.readLine("Enter your login: ");
        char [] oldPassword = c.readPassword("Enter your old password: ");
```

Exemple, suite

```
    if (verify(login, oldPassword)) {
        boolean noMatch;
        do {
            char [] newPassword1 =
                c.readPassword("Enter your new password: ");
            char [] newPassword2 =
                c.readPassword("Enter new password again: ");
            noMatch = ! Arrays.equals(newPassword1, newPassword2);
            if (noMatch) {
                c.format("Passwords don't match. Try again.%n");
            } else {
                change(login, newPassword1);
                c.format("Password for %s changed.%n", login);
            }
            Arrays.fill(newPassword1, ' ');
            Arrays.fill(newPassword2, ' ');
        } while (noMatch);
    }
    Arrays.fill(oldPassword, ' ');
}
```

Exemple, suite

```
//Dummy verify method.
static boolean verify(String login, char[] password) {
    return true;
}

//Dummy change method.
static void change(String login, char[] password) {}
```

Data Streams

- Les data streams permettent des I/O binaires pour les données de type primitif (boolean, char, byte, short, int, long, float, et double) et String.
- Tous les data streams implémentent l'une des deux interfaces suivantes :
  - ***java.io.DataInput***
  - ***java.io.DataOutput***
- Les deux classes les plus utilisées sont :
  - ***java.io.DataInputStream***
  - ***java.io.DataOutputStream***

Data Streams : exemple

Order in record	Data type	Data description	Output Method	Input Method	Sample Value
1	double	Item price	DataOutputStream.writeDouble	DataInputStream.readDouble	19.99
2	int	Unit count	DataOutputStream.writeInt	DataInputStream.readInt	12
3	String	Item description	DataOutputStream.writeUTF	DataInputStream.readUTF	"Java T-Shirt"

```
public class DataStreams {
    static final String dataFile = "invoicedata";

    static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
    static final int[] units = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Java T-shirt",
            "Java Mug",
            "Duke Juggling Dolls",
            "Java Pin",
            "Java Key Chain" };
```

Data Streams : exemple

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

    DataOutputStream out = null;

    try {
        out = new DataOutputStream(new
                BufferedOutputStream(new FileOutputStream(dataFile)));

        for (int i = 0; i < prices.length; i ++) {
            out.writeDouble(prices[i]);
            out.writeInt(units[i]);
            out.writeUTF(descs[i]);
        }
    } finally {
        out.close();
    }
```

Data Streams : exemple

```
DataInputStream in = null;
double total = 0.0;
try {
    in = new DataInputStream(new
            BufferedInputStream(new FileInputStream(dataFile)));

    double price;
    int unit;
    String desc;

    try {
        while (true) {
            price = in.readDouble();
            unit = in.readInt();
            desc = in.readUTF();
            System.out.format("You ordered %d units of %s at $%.2f%n",
                    unit, desc, price);
            total += unit * price;
        }
    } catch (EOFException e) { }
    System.out.format("For a TOTAL of: $%.2f%n", total);
}
finally {
    in.close();
}
```

Data Streams : exemple

- **Tous** les data streams détectent la fin d'un stream en attrapant l'exception *EOFException* (pas avec un code de return)
- A chaque méthode *write* correspond une méthode *read* correspondante
- C'est le programme qui doit s'assurer que les types correspondent : l'entrée contient uniquement des données binaires → rien n'indique le type d'une donnée, ni son début ou sa fin dans le stream

Object streams

- Les object streams permettent de réaliser des I/O d'objets java.
- La plupart (pas toute) des classes standards supportent la « sérialisation » de leur instances.
- En fait, seules les classes qui implémentent l'interface ***java.io.Serializable*** sont sérialisables
- Les classes d'Object streams sont :
  - ***java.io.ObjectInputStream***
  - ***java.io.ObjectOutputStream***
- Ce sont des interfaces qui héritent des interfaces ***DataInputStream et DataOutputStream***

Object streams : exemple

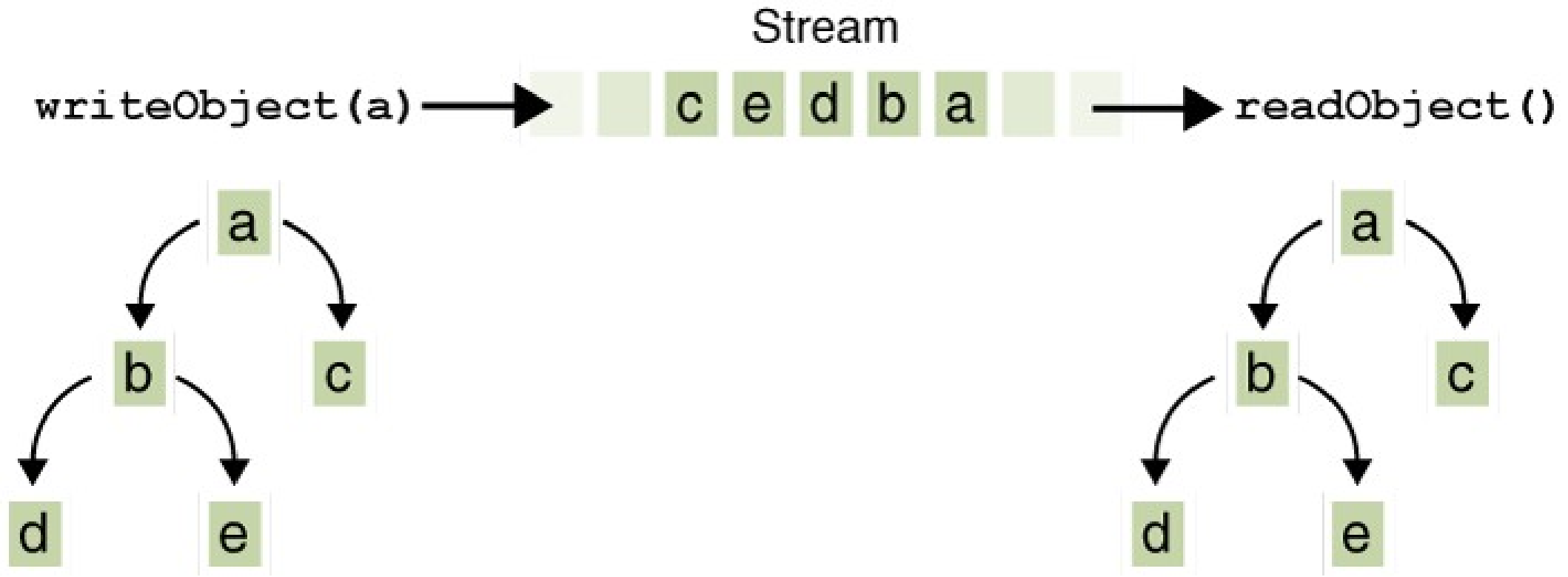
```
public class ObjectStreams {
    static final String dataFile = "invoicedata";

    static final BigDecimal[] prices = {
        new BigDecimal("19.99"),
        new BigDecimal("9.99"),
        new BigDecimal("15.99"),
        new BigDecimal("3.99"),
        new BigDecimal("4.99") };
    static final int[] units = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Java T-shirt",
            "Java Mug",
            "Duke Juggling Dolls",
            "Java Pin",
            "Java Key Chain" };
```

Object streams : exemple

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


    ObjectOutputStream out = null;
    try {
        out = new ObjectOutputStream(new
                BufferedOutputStream(new FileOutputStream(dataFile)));

        out.writeObject(Calendar.getInstance());
        for (int i = 0; i < prices.length; i ++) {
            out.writeObject(prices[i]);
            out.writeInt(units[i]);
            out.writeUTF(descs[i]);
        }
    } finally {
        out.close();
    }
```

```
ObjectInputStream in = null;
try {
    in = new ObjectInputStream(new
            BufferedInputStream(new FileInputStream(dataFile)));

    Calendar date = null;
    BigDecimal price;
    int unit;
    String desc;
    BigDecimal total = new BigDecimal(0);

    date = (Calendar) in.readObject();

    System.out.format ("On %tA, %<tB %<te, %<tY:%n", date);

    try {
        while (true) {
            price = (BigDecimal) in.readObject();
            unit = in.readInt();
            desc = in.readUTF();
            System.out.format("You ordered %d units of %s at $%.2f%n",
                    unit, desc, price);
            total = total.add(price.multiply(new BigDecimal(unit)));
        }
    } catch (EOFException e) {}
    System.out.format("For a TOTAL of: $%.2f%n", total);
} finally {
    in.close();
}
```

Object streams



Aller plus loin

Pour tout ce qui est manipulation de fichiers et de dossiers :
création, déplacement, copie, propriétés, etc. :

java.nio.*