

SRCS :

Les Entrées/Sorties en Java.

Version 15.02

Julien Sopena¹

¹julien.sopena@lip6.fr

Équipe REGAL - INRIA Rocquencourt
LIP6 - Université Pierre et Marie Curie

Master SAR 1ère année - SRCS - 2016/2017

Grandes lignes du cours

L'abstraction

Les types de flux

Les sources/destinations

Les filtres

Les filtres personnalisés

Outline

L'abstraction

Les types de flux

Les sources/destinations

Les filtres

Les filtres personnalisés

Les Entrées/Sorties

Tous les programmes reposent sur des interactions avec le monde extérieur :

- ▶ leurs actions dépendent d'ordres, de données et de paramètres reçus
- ▶ leurs calculs produisent un ensemble de données et d'évènements transmis à l'utilisateur ou à d'autres programmes.

Définition

*On appelle **Entrées/Sorties (E/S)** l'ensemble des actions qui servent respectivement à recevoir et produire des données quel que soit le type d'interaction utilisée.*

De multiples types d'interaction

Les Entrées/Sorties utilisent de nombreux supports (fichiers, réseaux, mémoire, ...) et leurs actions peuvent sembler assez éloignées.

De multiples types d'interaction

Les Entrées/Sorties utilisent de nombreux supports (fichiers, réseaux, mémoire, ...) et leurs actions peuvent sembler assez éloignées.

Éditer un fichier →  → Lire un fichier

De multiples types d'interaction

Les Entrées/Sorties utilisent de nombreux supports (fichiers, réseaux, mémoire, ...) et leurs actions peuvent sembler assez éloignées.

Éditer un fichier →  → Lire un fichier

Envoyer un message →  → Recevoir un message

De multiples types d'interaction

Les Entrées/Sorties utilisent de nombreux supports (fichiers, réseaux, mémoire, ...) et leurs actions peuvent sembler assez éloignées.

Éditer un fichier →  → Lire un fichier

Envoyer un message →  → Recevoir un message

Entrer une valeur →  → Consulter une valeur

De multiples types d'interaction

Les Entrées/Sorties utilisent de nombreux supports (fichiers, réseaux, mémoire, ...) et leurs actions peuvent sembler assez éloignées.

Éditer un fichier →  → Lire un fichier

Envoyer un message →  → Recevoir un message

Entrer une valeur →  → Consulter une valeur

Modifier une chaîne →  → Récupérer une chaîne

Les flux : abstractions d'interactions

Définition

Les **flux** (*streams* en anglais) permettent d'encapsuler les envois et les réceptions de données de façon à cacher le support utilisé. Ainsi un même code utilisera indifféremment : un fichier, le réseau ou la mémoire.



Attention

Les flux traitent toujours les données de façon séquentielle.

Encapsulation des entrées/sorties

Grâce à l'utilisation des flux on peut masquer l'hétérogénéité des sources et des destinations.

Éditer un fichier →  → Lire un fichier

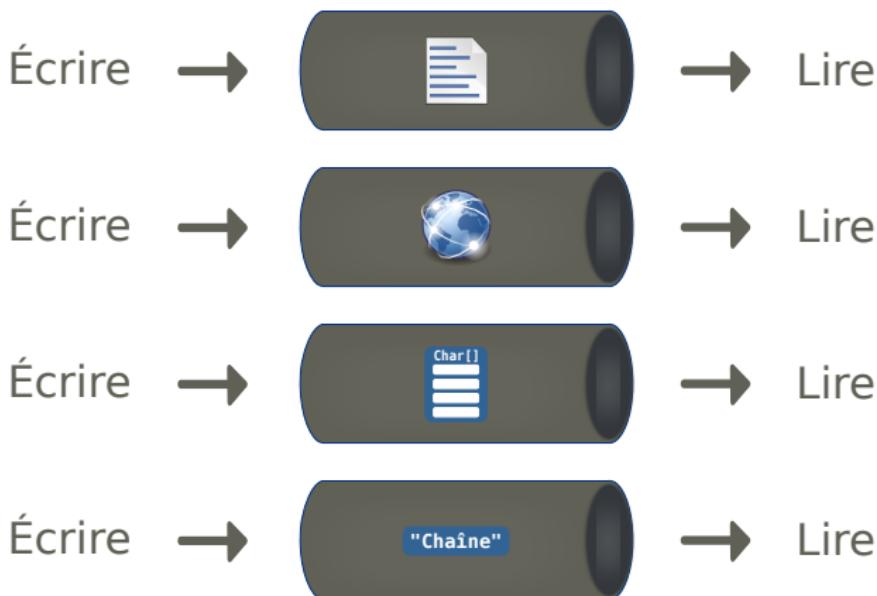
Envoyer un message →  → Recevoir un message

Entrer une valeur →  → Consulter une valeur

Modifier une chaîne →  "Chaîne" → Récupérer une chaîne

Encapsulation des entrées/sorties

Grâce à l'utilisation des flux on peut masquer l'hétérogénéité des sources et des destinations.



Les classes d'Entrée/Sortie de Java

Java encapsule les entrées/sorties dans trois types de classes :

Les flux de base dont les types de la classe indiquent :

- ▶ le sens du flux (Entrée ou Sortie)
- ▶ le type des données (Octet ou Caractère)

Les constructeurs dont les noms <Support><TypeFlux>
indiquent :

- ▶ le support (ficher, tableau, chaîne, tube, ...)
- ▶ le flux de base utilisé

Les filtres dont les noms <Fonctionnalité><TypeFlux> indiquent :

- ▶ la fonctionnalité (buffer, serialisation, compression, ...)
- ▶ le flux de base utilisé

La suite du cours présente ces trois familles de Classe.

Outline

L'abstraction

Les types de flux

Les sources/destinations

Les filtres

Les filtres personnalisés

Les deux familles de flux : Octets / Caractères

Définition

Java distingue deux sortes de flux suivant le type de données manipulées :

Flux d'octets : **InputStream** en entrée et **OutputStream** en sortie ;

Flux de caractères : **Reader** en entrée et **Writer** en sortie ;

Pour éditer un fichier on peut, suivant ce que l'on veut stocker, utiliser :

OutputStream

Écrire des octets



Lire des octets

InputStream

Écrire des caractères



Lire des caractères

Reader

API des flux d'entrée en octets : InputStream

`int read()` : lecture d'un octet

- ▶ **bloquant** tant qu'il n'y a rien à lire et que le flux reste ouvert.
- ▶ l'octet lu est retourné sous forme d'un **int**
- ▶ retourne -1 si le flux est fermé

`int read(byte[] tab)` : lecture d'un tableau d'octet

- ▶ **bloquant** tant qu'il n'y a rien à lire et que le flux reste ouvert.
- ▶ retourne le nombre d'octet lu ou -1 si le flux est fermé
- ▶ Attention : le tableau doit être alloué avant l'appel à read

`int read(byte[] tab, int offset, int length)` :

- ▶ idem avec au plus length octets enregistrés à partir de la case offset

`long skip(long n)` : "saute" n octets (sans les retourner)

`int available()` : retourne le nombre d'octets disponible

- ▶ l'appel est **non bloquant**.

`void close()` : fermeture du flux

API des flux de sortie en octets : OutputStream

`void write(byte b)` : écriture d'un octet dans le flux

- ▶ Lève une exception en cas d'erreur.

`void write(byte[] tab)` : écriture d'un tableau d'octet

`void write(byte[] tab, int offset, int length)` : écriture de length octets à partir de la case offset

`void flush()` : vide les buffers en forçant l'écriture

`void close()` : fermeture du flux

Exemple d'utilisation des flux d'octets

OutputStream

Écrire des octets → ? → Lire des octets

InputStream

```
// Récupération du flux  
OutputStream os = ?????? ;  
  
// Utilisation du flux  
for (byte b=0; b<5 ;b++) {  
    os.write(b);  
    os.write(2*b+1);  
}  
  
// Fermeture du flux  
os.close();
```

```
// Récupération du flux  
InputStream is = ?????? ;
```

```
// Utilisation du flux  
int x,y ;  
while ((x=is.read())!=-1) {  
    y = is.read();  
    dessin.trace(x,y);  
}
```

```
// Fermeture du flux  
is.close();
```

Exemple d'utilisation des flux de caractères

Writer

Écrire des caractères



Lire des caractères

Reader

```
// Récupération d'un flux  
Writer w = ?????? ;
```

```
// Utilisation du flux  
w.write('a');  
w.write('b');  
w.write("cdef");
```

```
// Fermeture du flux  
w.close();
```

```
// Récupération d'un flux  
Reader r = ?????? ;
```

```
// Utilisation du flux  
StringBuffer buff = ...  
int i;  
while ((i=r.read())!=-1) {  
    buff.append((char)i);  
}
```

```
// Fermeture du flux  
r.close();
```

Outline

L'abstraction

Les types de flux

Les sources/destinations

Les filtres

Les filtres personnalisés

Où trouve t-on les flux ?

On distingue deux méthodes pour obtenir la référence d'un flux :

- 1. Par accesseur** : certaines classes de Java possèdent des flux pour produire ou recevoir des données. Ceux-ci sont accessibles directement (**System.in** et **System.out**) ou au travers d'accesseurs (**getInputStream()** et **getOutputStream()**).
- 2. Par constructeur** : pour les objets qui ne possèdent pas directement de flux, Java offre pour les construire des classes spécifiques à chaque support. Toutes ces classes héritent d'un des 4 flux de base : `InputStream`, `OutputStream`, `Reader` ou `Writer`.

Dans tous les cas, une fois obtenu la référence :
on utilise le flux indifféremment du type de sources/destinations.

Les flux standard : la console et le clavier

La classe **System** est pourvue de trois attributs de classe :

- in** : un flux qui correspond à l'entrée standard (par défaut le clavier)
- out** : un flux qui correspond à la sortie standard (par défaut la console)
- err** : un flux qui correspond à la sortie d'erreur (par défaut la console)

```
// Lecture sur le clavier
int x = System.in.read() ;

// Écriture sur la console
System.out.write(x);
```

Les flux standard : la console et le clavier

La classe **System** est pourvue de trois attributs de classe :

in : un flux qui correspond à l'entrée standard (par défaut le clavier)

out : un flux qui correspond à la sortie standard (par défaut la console)

err : un flux qui correspond à la sortie d'erreur (par défaut la console)

```
// Lecture sur le clavier  
int x = System.in.read();  
  
// Ecriture sur la console  
System.out.write(x);
```

Attention

En réalité les sorties **out** et **err** utilisent un filtre **PrintStream** (voir plus loin) qui permet l'utilisation des fonctions **print** et **println**.

Exemple de classe munie d'un flux : Process

Dans le cas de la classe **Process** les flux permettent d'écrire (*resp.* de lire) sur l'entrée standard (*resp.* la sortie standard) du processus.

```
// Lancement du programme beep
Process p =
    Runtime.getRuntime().exec("beep -c -f 400 -D 50 -l 10");

// Récupération du flux d'entrée standard du programme
// ATTENTION : Pour nous c'est un flux de sortie (out)
OutputStream os = p.getOutputStream();

// Utilisation du flux : comme avec tout OutputStream
int x;
while ( (x = System.in.read()) != -1 ) {
    os.write(x);
}

// Fermeture du flux (=> arrêt du programme beep)
os.close();
```

Classes permettant de construire des flux



Source / Destination <i>(Défini avec un préfixe)</i>	Entrée en Octets <i>(Hérite de InputStream)</i>	Sortie en Octets <i>(Hérite de OutputStream)</i>
Tableau d'octets en mémoire :	ByteArrayInputStream	ByteArrayOutputStream
Fichier :	FileInputStream	FileOutputStream
Pipeline entre deux threads :	PipedInputStream	PipedOutputStream
Chaîne de caractères :	StringBufferInputStream	StringBufferOutputStream



Source / Destination <i>(Défini avec un préfixe)</i>	Entrée en Caractères <i>(Hérite de Reader)</i>	Sortie en Caractères <i>(Hérite de Writer)</i>
Tableau de caractères en mémoire :	CharArrayReader	CharArrayWriter
Fichier :	FileReader	FileWriter
Pipeline entre deux threads :	PipedReader	PipedWriter
Chaîne de caractères :	StringReader	StringWriter

Exemple d'utilisation d'un fichier d'octets

OutputStream

Écrire des octets



?



Lire des octets

InputStream

```
// Récupération du flux  
OutputStream os =  
????????????????????? ;
```

```
// Utilisation du flux  
for (byte b=0; b<5 ;b++) {  
    os.write(b);  
    os.write(2*b+1);  
}
```

```
// Fermeture du flux  
os.close();
```

```
// Récupération du flux  
InputStream is =  
????????????????????? ;
```

```
// Utilisation du flux  
int x,y ;  
while ((x=is.read())!=-1) {  
    y = is.read();  
    dessin.trace(x,y);  
}
```

```
// Fermeture du flux  
is.close();
```

Exemple d'utilisation d'un fichier d'octets

OutputStream

Écrire des octets



Lire des octets

InputStream

```
// Récupération du flux
OutputStream os =
    new FileOutputStream("foo");

// Utilisation du flux
for (byte b=0; b<5 ;b++) {
    os.write(b);
    os.write(2*b+1);
}

// Fermeture du flux
os.close();
```

```
// Récupération du flux
InputStream is =
    new FileInputStream("foo");

// Utilisation du flux
int x,y ;
while ((x=is.read())!=-1) {
    y = is.read();
    dessin.trace(x,y);
}

// Fermeture du flux
is.close();
```

Exemple d'utilisation d'un tableau d'octets

OutputStream

Écrire des octets



Lire des octets

InputStream

```
// Récupération du flux
OutputStream os =
    new ByteArrayOutputStream(t);

// Utilisation du flux
for (byte b=0; b<5 ;b++) {
    os.write(b);
    os.write(2*b+1);
}

// Fermeture du flux
os.close();
```

```
// Récupération du flux
InputStream is =
    new ByteArrayInputStream(t);

// Utilisation du flux
int x,y ;
while ((x=is.read())!=-1) {
    y = is.read();
    dessin.trace(x,y);
}

// Fermeture du flux
is.close();
```

Exemple d'utilisation d'un fichier de caractères

Writer

Écrire des caractères



Lire des caractères

Reader

```
// Recupération d'un flux
Writer w =
????????????????????? ;

// Utilisation du flux
w.write('a');
w.write('b');
w.write("cdef");

// Fermeture du flux
w.close();
```

```
// Récupération d'un flux
Reader r =
```

```
????????????????????? ;
```

```
// Utilisation du flux
```

```
StringBuffer buff = ...
```

```
int i;
```

```
while ((i=r.read())!=-1) {
```

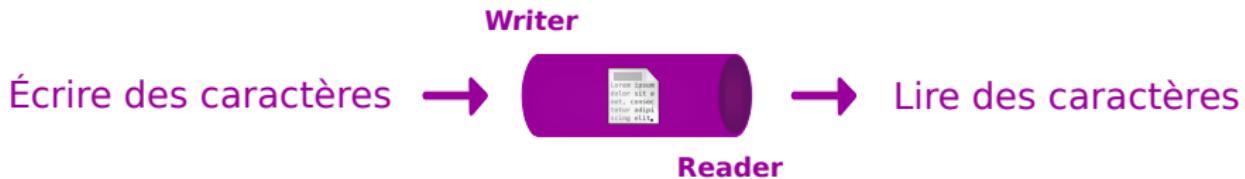
```
buff.append((char)i);
```

```
}
```

```
// Fermeture du flux
```

```
r.close();
```

Exemple d'utilisation d'un fichier de caractères



```
// Récupération d'un flux
Writer w =
    new FileWriter("foo.txt");

// Utilisation du flux
w.write('a');
w.write('b');
w.write("cdef");

// Fermeture du flux
w.close();
```

```
// Récupération d'un flux
Reader r =
    new FileReader("foo.txt");

// Utilisation du flux
StringBuffer buff = ...
int i;
while ((i=r.read())!=-1) {
    buff.append((char)i);
}

// Fermeture du flux
r.close();
```

Exemple d'utilisation d'un tableau de caractères



```
// Récupération d'un flux
Writer w =
    new CharArrayWriter(t);

// Utilisation du flux
w.write('a');
w.write('b');
w.write("cdef");

// Fermeture du flux
w.close();
```

```
// Récupération d'un flux
Reader r =
    new CharArrayReader(t);

// Utilisation du flux
StringBuffer buff = ...
int i;
while ((i=r.read())!=-1) {
    buff.append((char)i);
}

// Fermeture du flux
r.close();
```

Outline

L'abstraction

Les types de flux

Les sources/destinations

Les filtres

Les filtres personnalisés

Les filtres de base en Java

Définition

*Pour ajouter des fonctionnalités aux flux, Java utilise des classes appelées **filtre** suivant le patron de conception (design-pattern) du **décorateur**.*

Le nom des filtres Java est suffixé par l'un des quatre flux de base :

- ▶ ils héritent de **Filter<FluxBase>** qui hérite de **<FluxBase>** ;
- ▶ ils prennent ce flux comme paramètre de leur constructeur.

Pour ajouter des fonctionnalités ils peuvent :

- ▶ redéfinir le fonctionnement des méthodes de base (`read`, `write`, ...)
- ▶ ajouter de nouvelles méthodes (`print`, `readLine`, ...)

Les filtres de base en Java

Définition

*Pour ajouter des fonctionnalités aux flux, Java utilise des classes appelées **filtre** suivant le patron de conception (design-pattern) du **décorateur**.*

Le nom des filtres Java est suffixé par l'un des quatre flux de base :

- ▶ ils héritent de **Filter<FluxBase>** qui hérite de **<FluxBase>** ;
- ▶ ils prennent ce flux comme paramètre de leur constructeur.

Pour ajouter des fonctionnalités ils peuvent :

- ▶ redéfinir le fonctionnement des méthodes de base (`read`, `write`, ...)
⇒ par **polymorphisme** l'utilisation est transparente et interchangeable
- ▶ ajouter de nouvelles méthodes (`print`, `readLine`, ...)

Les filtres de base en Java

Définition

*Pour ajouter des fonctionnalités aux flux, Java utilise des classes appelées **filtre** suivant le patron de conception (design-pattern) du **décorateur**.*

Le nom des filtres Java est suffixé par l'un des quatre flux de base :

- ▶ ils héritent de **Filter<FluxBase>** qui hérite de **<FluxBase>** ;
- ▶ ils prennent ce flux comme paramètre de leur constructeur.

Pour ajouter des fonctionnalités ils peuvent :

- ▶ redéfinir le fonctionnement des méthodes de base (`read`, `write`, ...)
⇒ par **polymorphisme** l'utilisation est transparente et interchangeable
- ▶ ajouter de nouvelles méthodes (`print`, `readLine`, ...)
⇒ nécessite d'adapter un peu les codes

Les filtres de base en Java

Les Filtres par flux

InputStream
OutputStream
Reader
Writer

Préfixe	Fonctionnalité	InputStream	OutputStream	Reader	Writer
Buffered :	Mise en tampon	✓	✓	✓	✓
Data :	Conversion des types primitifs	✓	✓	✗	✗
Object :	Sérialisation d'objet	✓	✓	✗	✗
Sequence :	Concaténation de flux	✓	✗	✗	✗
Print :	Fonctions de formatage print	✗	✓	✗	✓
PushBack :	Lecture avec remise dans le flux	✓	✗	✓	✗
LineNumber :	Numérotation des lignes	✓	✗	✓	✗
Digest :	Vérification par somme de contrôle	✓	✓	✗	✗
Cipher :	Chiffrage du flux	✓	✓	✗	✗

InputStreamReader et OutputStreamWriter

Attention

Il ne faut pas confondre les utilisations des termes **InputStream** et **OutputStream** pour désigner des flux, avec leurs utilisations comme nom des constructeurs permettant de construire des flux de caractères à partir de flux en caractères.

En tant que flux, il sont des suffixes comme dans :

- ▶ `BufferedInputStream`, `ObjectInputStream`, `DataInputStream`
- ▶ `BufferedOutputStream`, `ObjectOutputStream`, `DataOutputStream`

En tant que constructeurs, il sont des préfixes comme dans :

- ▶ `InputStreamReader`
- ▶ `OutputStreamWriter`

Composition de filtres



```
OutputStream os = ?????? ;  
  
for (byte b=0 ; b<3 ; b++) {  
    os.write(b);  
}  
os.close();  
  
InputStream is = ?????? ;
```

```
while ( (x = is.read()) != -1 ) {  
    System.out.println(x);  
}  
is.close();
```

Composition de filtres



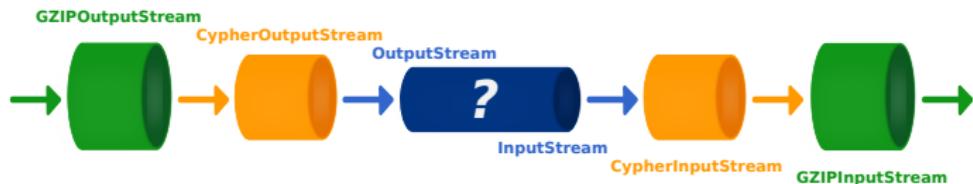
```
OutputStream os = ?????? ;
CipherOutputStream cos = new CipherOutputStream(os, cipher);

for (byte b=0 ; b<3 ; b++) {
    cos.write(b);
}
cos.close();

InputStream is = ?????? ;
CipherInputStream cis = new CipherInputStream(is, cipher);

while ( (x = cis.read()) != -1 ) {
    System.out.println(x);
}
cis.close();
```

Composition de filtres



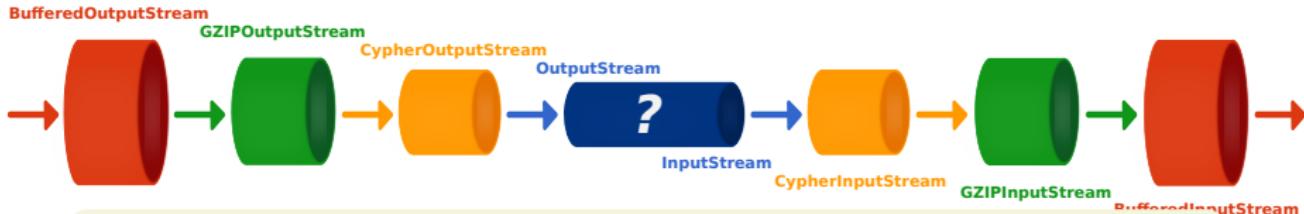
```
OutputStream os = ?????? ;
CipherOutputStream cos = new CipherOutputStream(os, cipher);
GZIPOutputStream zcos = new GZIPOutputStream(cos);

for (byte b=0 ; b<3 ; b++) {
    zcos.write(b);
}
zcos.close();

InputStream is = ?????? ;
CipherInputStream cis = new CipherInputStream(is, cipher);
GZIPInputStream zcis = new GZIPInputStream(cis);

while ( (x = zcis.read()) != -1 ) {
    System.out.println(x);
}
zcis.close();
```

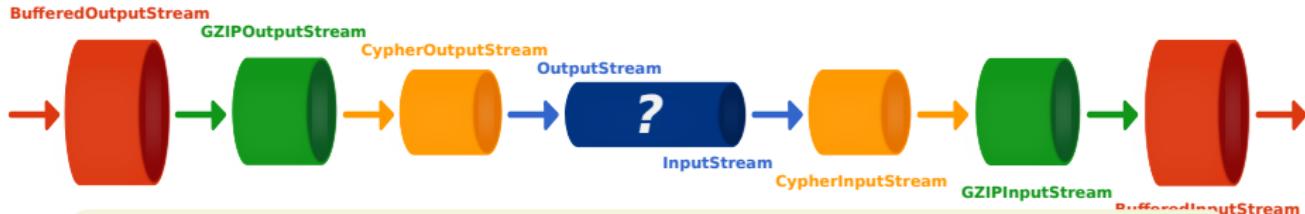
Composition de filtres



```
OutputStream os = ?????? ;
CipherOutputStream cos = new CipherOutputStream(os, cipher);
GZIPOutputStream zcos = new GZIPOutputStream(cos);
BufferedOutputStream bzcos = new BufferedOutputStream(zcos);
for (byte b=0 ; b<3 ; b++) {
    bzcos.write(b);
}
bzcos.close();
```

```
InputStream is = ?????? ;
CipherInputStream cis = new CipherInputStream(is, cipher);
GZIPInputStream zcis = new GZIPInputStream(cis);
BufferedInputStream bzcis = new BufferedInputStream(zcis);
while ( (x = bzcis.read()) != -1 ) {
    System.out.println(x);
}
bzcis.close();
```

Composition de filtres



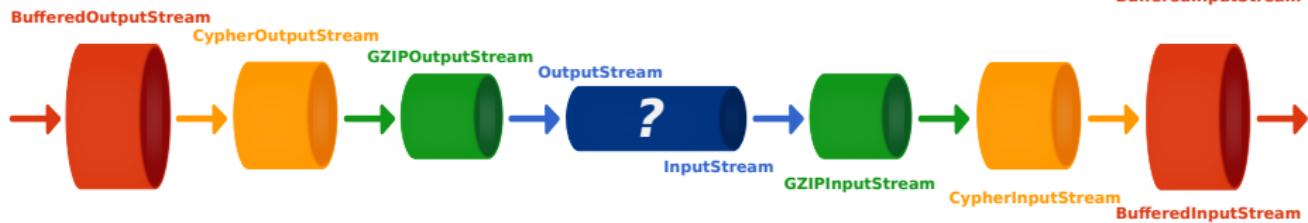
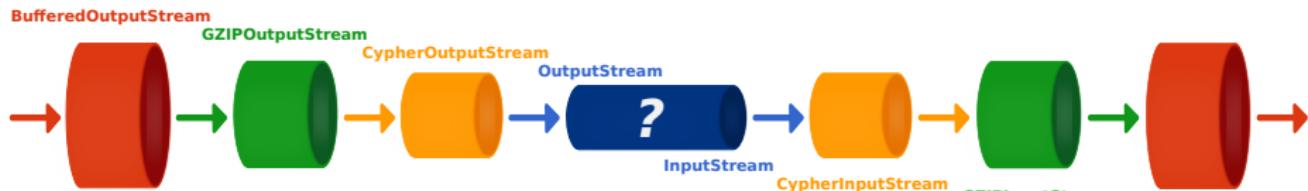
```
OutputStream os = new BufferedOutputStream(
    new GZIPOutputStream(
        new CipherOutputStream(???????, cipher)
    ));
for (byte b=0 ; b<3 ; b++) {
    os.write(b);
}
os.close();

InputStream is = new BufferedInputStream(
    new GZIPInputStream(
        new CipherInputStream(???????, cipher)
    ));
while ( (x = is.read()) != -1 ) {
    System.out.println(x);
}
is.close();
```

Attention à l'ordre des filtres : efficacité

L'ordre de mise en œuvre des filtres peut avoir une grande incidence sur les performances du programme.

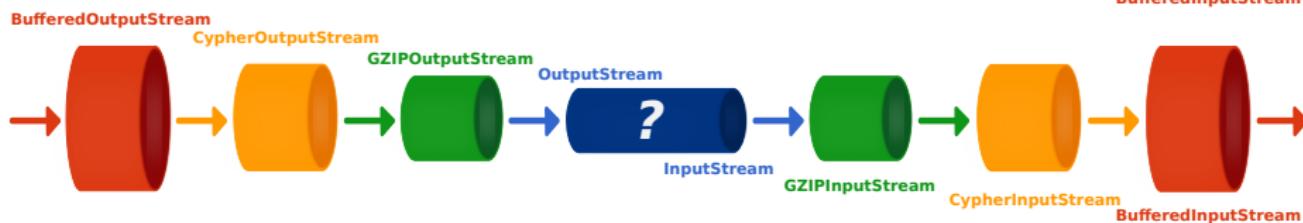
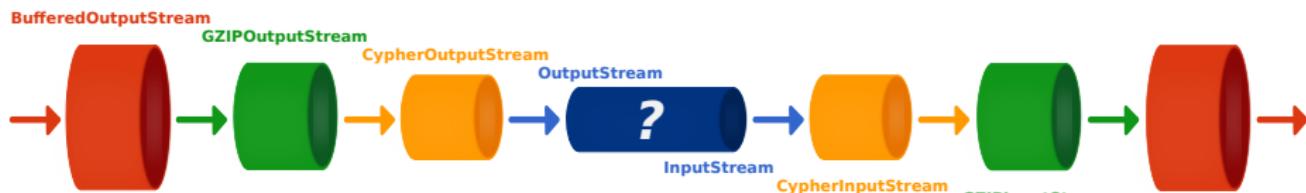
Comparons par exemple les séquences suivantes :



Attention à l'ordre des filtres : efficacité

L'ordre de mise en œuvre des filtres peut avoir une grande incidence sur les performances du programme.

Comparons par exemple les séquences suivantes :

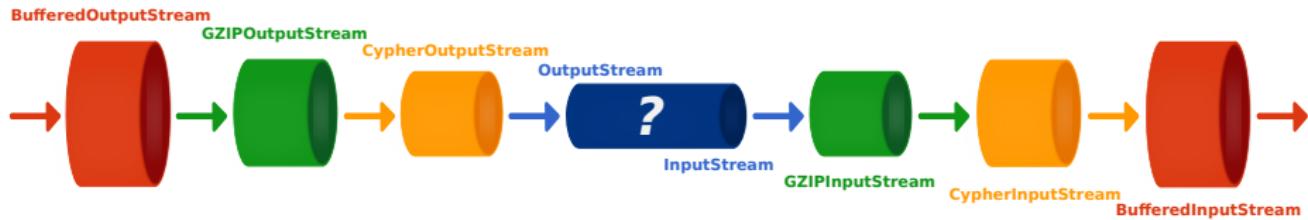


Les opérations cryptographiques sont très coûteuses en calculs : il est donc préférable de compresser le flux avant de le crypter.

Attention à l'ordre des filtres : cohérence

Attention

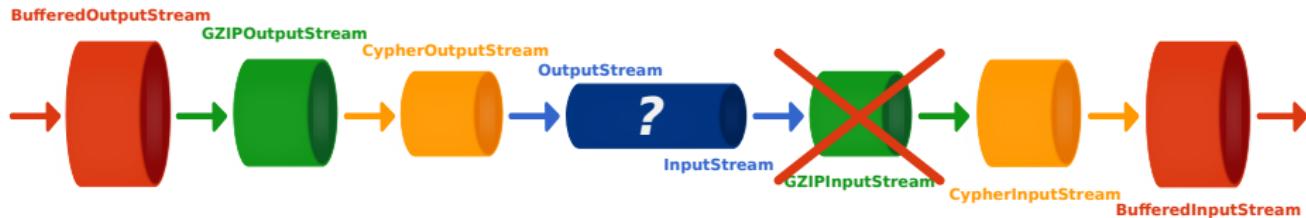
Attention l'ordre des filtres utilisés lors de la lecture du flux d'entrée doit être compatible avec celui utilisé pour produire les données dans le flux de sortie



Attention à l'ordre des filtres : cohérence

Attention

Attention l'ordre des filtres utilisés lors de la lecture du flux d'entrée doit être compatible avec celui utilisé pour produire les données dans le flux de sortie



```
java Reception
Exception in thread "main" java.util.zip.ZipException: Not in
GZIP format
    at java.util.zip.GZIPInputStream.readHeader(GZIPInputStream.java:164)
    at java.util.zip.GZIPInputStream.<init>(GZIPInputStream.java:78)
    at java.util.zip.GZIPInputStream.<init>(GZIPInputStream.java:90)
    at flux.FluxBZC.main(Reception.java:10)
```

Outline

L'abstraction

Les types de flux

Les sources/destinations

Les filtres

Les filtres personnalisés

Implémenter ses propres filtres

Les fonctionnalités offertes par Java sont insuffisantes :

⇒ implémentez un filtre en suivant le *design-pattern* du **décorateur**.

Pour un filtre personnalisé destiné à la lecture d'octets :

- ▶ définir une classe qui hérite de **FilterInputStream** ;
- ▶ implémenter un constructeur à un paramètre **InputStream** is :
 1. appeler à la première ligne du constructeur super(is)
 2. si besoin enregistrer les autres paramètres dans des attributs
- ▶ redéfinir la méthode **read()**
 1. utiliser super.read() pour obtenir la valeur avant transformation
 2. appliquer le filtre
 3. retourner la valeur modifiée

Attention, une fois implémentés vos filtres personnalisés devront s'utiliser comme tous les autres filtres standards de Java.

Exemple de filtre personnalisé : fonction linéaire

```
public class LinearInputStream extends FilterInputStream {  
    int a,b;  
    protected LinearInputStream(InputStream is,int a, int b) {  
        super(is);  
        this.a=a;  
        this.b=b;  
    }  
    @Override  
    public int read() throws IOException {  
        int x = super.read();  
        return a*x+b ;  
    }  
}
```

```
OutputStream os =  
    new FileOutputStream("f1");  
os.write(1);  
os.close();
```

```
InputStream is =  
    new FileInputStream("f1");  
LinearInputStream lis =  
    new LinearInputStream(is,2,3);  
int y = lis.read();  
System.out.println("y="+y);  
lis.close();
```