# NFP121 — Programmation Avancée
—
# Les entrées/sorties en Java

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux

# Objectifs

- Comprendre les entrées/sorties
- Savoir utiliser l'API IO de Java
- Comprendre et savoir mettre en œuvre le patron *Décorateur*
- Savoir traiter un fichier texte en utilisant les API Java
- Savoir que l'API Java des expressions régulières existe

# Sommaire

# Entrées et sorties variées

**Entrées/sorties :** communication d'un programme avec son environnement

**Exemples d'entrées :**

- clavier, souris, joystick
- scanner, caméra, etc.
- lecture d'un fichier sur disque
- lecture des informations produites par un capteur
- réception d'une page web depuis un serveur distant
- ...

**Exemples de sorties :**

- affichage sur un écran
- écriture d'un fichier sur un disque local
- envoi d'une requête à un serveur
- envoi d'une commande à un robot, un actionneur, etc.
- impression d'un document vers un fax, une imprimante, etc.

# Autres difficultés

**Diversité des systèmes d'exploitation**

- Java se veut portable : le même code s'exécute partout avec le même comportement

**Fichiers de natures différentes :**

- texte : code source, script, configuration, courrier...
- binaire : exécutables, fichiers compressés, etc.
- spéciaux : /dev/ (périphériques)
- répertoires : contient des références à d'autres fichiers
- liens symboliques : autre accès à un même fichier

**Différents codages (fichier texte) :**

- latin1, utf8, ASCII, etc.
- un caractère codé sur 1 octet, 2 octets, 4 octets, voir un nombre variable
- recode : 281 codages pris en charge

# Sommaire



- Les classes abstraites
- Les concrétisations
- Les filtres

# Solution : Abstraction des E/S

- **Constatations :**
  - Toute E/S peut être représentée par une suite de bits
  - Les bits sont regroupés en octets (bytes)
- **Abstraction des entrées/sorties : flux (stream) :**
  - accès séquentiel aux données
  - flux d'entrée : InputStream
  - flux de sortie : OutputStream
- **Abstraction des opérations possibles**
  - opérations d'entrée (lecture : read)
  - opérations de sortie (écriture : write)
- **Concrétisation des flux :** Fichiers, Tableau, Pipe, etc.

**Remarque :** L'application doit transformer les informations en octets et les octets en information !

# Les entrées/sorties en Java

L'API d'entrée/sortie est définie dans le paquetage `java.io`.

- elle fournit une interface standard pour gérer les *flux* d'entrée/sortie
- elle libère le programmeur des détails d'implantation liés à une plateforme particulière

**Flux :**

- Séquence ordonnée de données (octets ou caractères)
- qui a une *source* (*input stream*)
- ou une *destination* (*output stream*).

**Classes de base :** Java propose 4 classes principales (abstraites) pour les entrées/sorties qui seront ensuite spécialisées en fonction de la nature de la source ou de la destination.

| | Caractères | Octets |
|---|---|---|
| Entrée | `Reader` | `InputStream` |
| Sortie | `Writer` | `OutputStream` |

# La classe InputStream

**But :** Lire des octets (bytes) depuis un flux d'entrée.

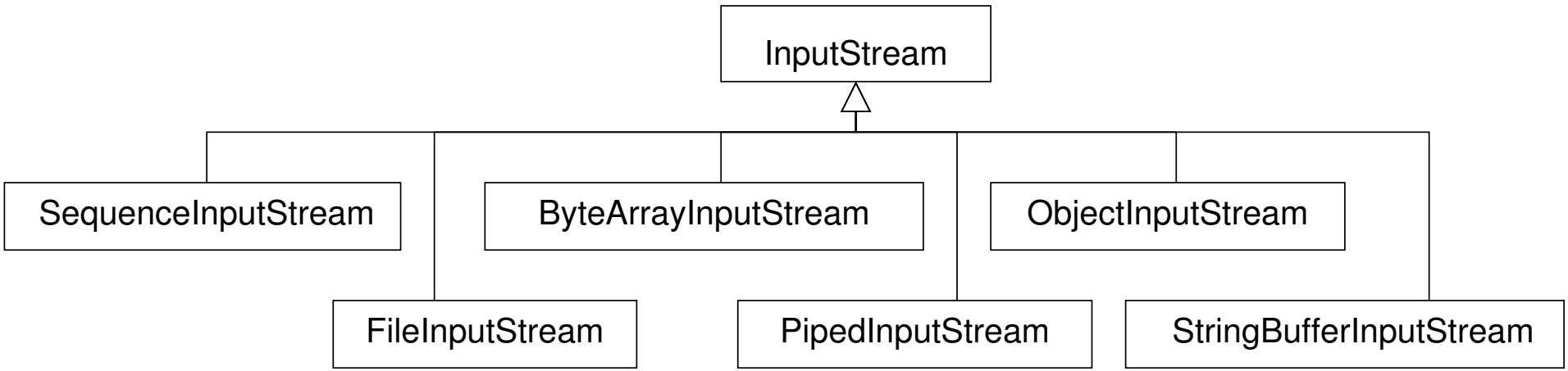
```
1  public abstract class InputStream {
2    public abstract int read() throws IOException;
3      // Lire un octet renvoyé sous la forme d'un entier entre 0 et 255.
4      // Retourne -1 si le flux d'entrée est terminé.  Bloquante.
5
6    public int read(byte[] buf, int off, int len) throws IOException;
7      // Lire au plus len octets et les stocker dans buf à partir de off.
8      // Retourne le nombre d'octets effectivement lus (-1 si fin de flux).
9      // @throws IndexOutOfBoundsException, NullPointerException...
10
11   public int read(byte[] b) throws IOException;
12     // Idem read(b, 0, b.length)
13
14   public long skip(long n) throws IOException
15     // Sauter (et supprimer) n octets du flux.
16     // Retourne le nombre d'octets effectivement sautés.
17
18   public void close() throws IOException;
19     // Fermer le flux et libérer les ressources système associées.
20
21   public int available() throws IOException;
22     // nombre d'octets disponibles (sans bloquer)
23 }
```

# La classe OutputStream

**But :** Écrire des octets (bytes) dans un flux de sortie.

```
1  public abstract class OutputStream {
2
3    public abstract void write(int b) throws IOException;
4      // Écrire b dans ce flux (seuls les 8 bits de poids faible).
5
6    public void write(byte[] buf, int off, int len) throws IOException;
7      // Écrire len octets de buf[off] à buf[off+len-1] dans ce flux.
8      // @throws IndexOutOfBoundsException, NullPointerException...
9
10   public void write(byte[] b) throws IOException;
11     // Idem write(b, 0, b.length)
12
13   public void flush() throws IOException
14     // Vider ce flux.
15     // Si des octets ont été bufférisés, ils sont effectivement écrits.
16
17   public void close() throws IOException;
18     // Fermer le flux et libérer les ressources système associées.
19 }
```

# Concrétisations de `InputStream`



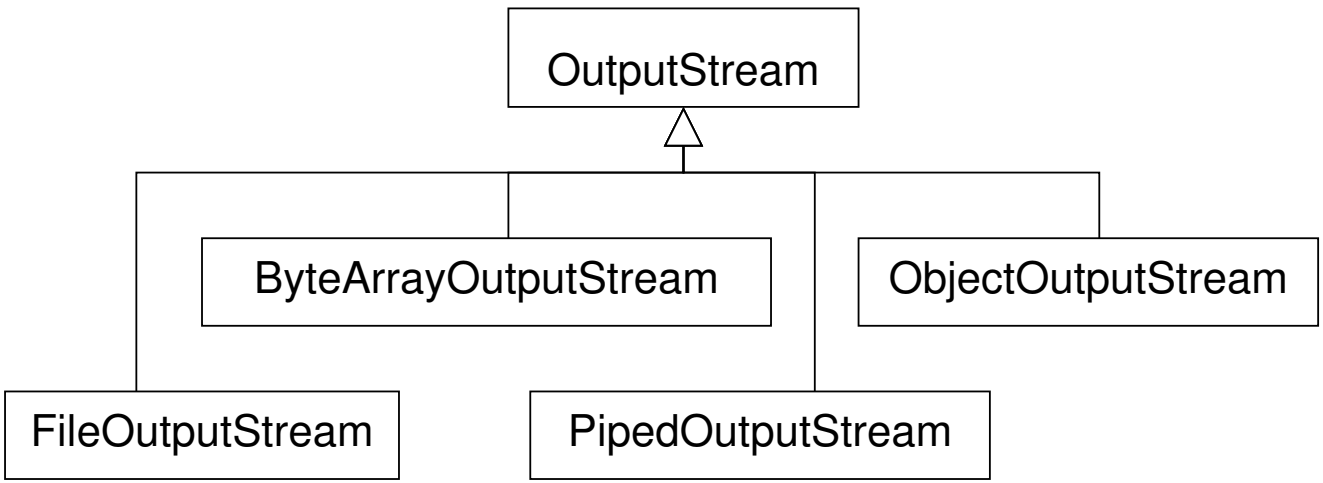

**Principe :** une sous-classe de `InputStream`...

- doit définir la méthode `int read();`
- devrait redéfinir `read(byte[], int, int)` pour en donner une version plus efficace

**Remarque :** `StringBufferInputStream` est obsolète :

- on ne doit pas traiter des caractères avec des `InputStream` !

# Concrétisations de `OutputStream`




**Principe :** Une sous-classe de `OutputStream`...

- doit définir la méthode `write(int)` ;
- devrait redéfinir `write(byte[], int, int)` pour en donner une version plus efficace

# Exemples de concrétisations

- **File* :** Flux sur des fichiers
  - Constructeur avec le nom du fichier, `File` ou `FileDescriptor`
  - Peut lever les exceptions :

```
FileNotFoundException   // le fichier n'existe pas
SecurityException       // pas de droit en lecture/écriture
```

- **ByteArray* :** Flux dans un tableau d'octets
- **Object* :** Flux d'objets (sérialisation)
- **Piped* :** Flux sur des tubes (connexion entre threads)
- **Mais aussi :**
  - **StringBufferInputStream :** Flux vers des chaînes de caractères (*deprecated*)
  - **SequenceInputStream :** Lecture depuis plusieurs flux successivement

# Exemple : Copier des flux quelconques

```
1     public static void copier(OutputStream out, InputStream in) throws IOException {
2         int c = in.read();
3         while (c != -1) {
4             out.write(c);
5             c = in.read();
6         }
7     }
```

- Fonctionne quelque soit l'`InputStream` et l'`OutputStream` !
- C'est l'intérêt du sous-typage !
- On constate l'intérêt du type **`int`** pour le paramètre de `write` !
- Cette méthode ne s'occupe pas de la création des flux (normal) ni de leur fermeture !

```
1     public static void copierTab(OutputStream out, InputStream in) throws IOException {
2         byte tampon[] = new byte[256];
3         int nb; // nombre d'octets lus
4         while ((nb = in.read(tampon)) >= 0) {
5             out.write(tampon, 0, nb);
6         }
7     }
```

# Exemple : Copier un fichier

```
1  public static void copier(String destination, String source) throws IOException {
2      FileInputStream in = null;
3      FileOutputStream out = null;
4      try {
5          in = new FileInputStream(source);
6          out = new FileOutputStream(destination);
7          copier(out, in);
8      } finally {
9          if (in != null) in.close();      // gain de place !
10         if (out != null) out.close();    // gain de place !
11 }   }
```

- **if** pour contrôler que le `Stream` avait bien été ouvert
- **finally** : garantir la fermeture des `Stream`... ou try-with-resources (AutoCloseable, Java7)

```
1  public static void copierJava7(String destination, String source) throws IOException {
2      try (
3          FileInputStream in = new FileInputStream(source);
4          FileOutputStream out = new FileOutputStream(destination);
5      ) {
6          copier(out, in);
7  }   }
```

- Il existe un autre constructeur de `FileOutputStream`

```
FileOutputStream(String name, boolean append)
    // append == true : écrit à la fin du fichier
    // append == false : écrit au début (fichier est écrasé)
```

# Quelques extensions...

**Exercice 1** Envisageons de nouveaux besoins.

**1.1** On souhaite pouvoir changer la casse des caractères lus depuis un fichier.
Expliquer comment faire.

**1.2** On veut pouvoir décrypter le contenu d'un fichier crypté en utilisant un chiffrement par décalage avec une clé valant 2. Expliquer comment faire.

**1.3** On veut d'abord décrypter, puis changer la casse du contenu d'un fichier.
Expliquer comment faire.

**1.4** Et pourquoi se limiter aux fichiers ? On veut aussi pouvoir considérer les autres sources.
Expliquer comment faire.

# Autres fonctionnalités souhaitées

Les classes précédentes fournissent **peu de fonctionnalités** :

- seulement lecture/écriture d'un octet ou plusieurs dans des flux
- on peut simplement lire ou écrire dans des flux variées

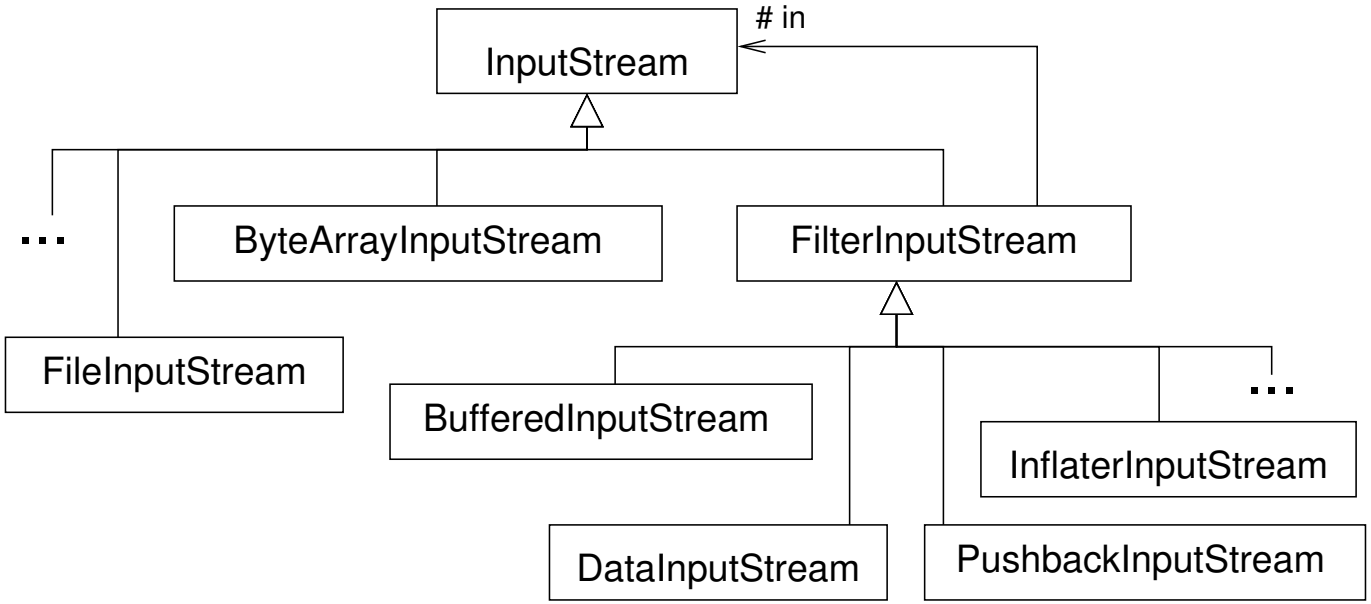
**Autres fonctionnalités souhaitées :**

- gérer un buffer pour des accès plus efficaces
- pouvoir compresser ou décompresser les données d'un flux
- crypter/décrypter les données dans un flux
- connaître le numéro de ligne et colonne (flux texte)
- remettre dans le flux des octets/caractères déjà lus
- ...
- et, surtout, pouvoir combiner les fonctionnalités précédentes

**Comment les mettre en œuvre pour les différents types de flux ?**

# Filtres : FilterInputStream et FilterOutputStream

**Principe :** Ajouter de nouvelles fonctionnalités à des flux existants




```
1 public class FilterInputStream extends InputStream {
2   protected InputStream in; // le flux filtré (décoré)
3   protected FilterInputStream(InputStream in)   // filtrer un flux existant
4     // protégé ==> ne peut être appelé que depuis une sous-classe
5 ... toutes les méthodes m(...) de InputStream sont redirigées sur in.m(...)
```

**Remarque :** Même principe pour `FilterOutputStream`

# Principaux filtres

**Buffered* :** Ajouter la bufferisation sur un flux existant

- Les informations ne sont pas directement écrites dans le flux
- En entrée : ajoute les fonctionalités `mark(int)` et `reset` sur BufferedReader, ajoute `readLine()`
- En sortie : opération `flush()` pour forcer l'écriture dans le flux

**PrintStream et PrintWriter :** Définit `print`/`println` pour les types primitifs et Object.

**Data* :** Écrit les types primitifs Java de manière portable.

**Compression de flux :**

- Entrée : `Inflater` avec comme spécialisations GZIP, Zip, Jar
- Sortie : `Deflater` avec comme spécialisations GZIP, Zip, Jar

**Pushback* :** Remettre (unread) des éléments dans le flux d'entrée.

**Remarque :** ObjectInputStream et ObjectOutputStream ne sont pas des filtres... mais ils pourraient (même principe).
Ces classes permettent la sérialisation (java.io.Serializable) : readObject and writeObject.

# Définir un FilterInputStream

```
1  import java.io.*;
2
3  public class FlipCaseInputStream extends FilterInputStream {
4      // Attention : pas la bonne façon de faire.  Devrait être FilterReader !
5      // On considère les octets comme des caractères !
6
7      public FlipCaseInputStream(InputStream in) {
8          super(in);
9      }
10
11     public int read() throws IOException {
12         int c = super.read();   // idem in.read()
13         int newC = c;
14         if (Character.isUpperCase(c)) {
15             newC = Character.toLowerCase(c);
16         } else if (Character.isLowerCase(c)) {
17             newC = Character.toUpperCase(c);
18         }
19         return newC;
20     }
21
22     // ...
23     // Définir aussi les deux autres méthodes read !  Pourquoi ?
24
25 }
```

**Question :** Pourquoi faut-il redéfinir les autres métodes `read(...)` ?

# Définir un FilterInputStream : Utilisation

```
1  import java.io.*;
2
3  public class TestFlipCaseInputStream {
4
5      public static void afficher(InputStream in) throws IOException {
6          int c;
7          while ((c = in.read()) != -1) {
8              System.out.print((char) c);
9      }   }
10
11     public static void main(String[] args) throws IOException { // Aucun contrôle
12         try (InputStream inStream = new FlipCaseInputStream(
13                 new FileInputStream(args[0]))
14         ) {
15             afficher(inStream);
16 }   }   }
```

Fichier initial en Latin 1 (et locales en Latin 1) :

```
Où était-il ? À l'école␣?
```

devient :

```
oÙ ÉTAIT-IL ? à L'ÉCOLE␣?
```

Ne marche pas si le fichier est en UTF 8 !

# Exemple de combinaison de filtres

## Écrire des données de manière portable

```
1  import java.io.*;
2  public class TestFiltres {
3      public static void main(String[] args) throws IOException {
4          ecrire("/tmp/fichier.out");
5          lire("/tmp/fichier.out");
6      }
7      public static void ecrire(String fileName) throws IOException {
8          try (DataOutputStream out =
9                  new DataOutputStream(
10                     new BufferedOutputStream(
11                         new FileOutputStream(fileName)));
12         ) {
13             out.writeUTF("ABCDEF");
14             out.writeInt(421);
15             out.writeBoolean(true);
16             out.writeInt(10);
17             out.writeDouble(12.5);
18         }
19     }
```

Résultat de : `xxd /tmp/fichier.out`

```
00000000: 0006 4142 4344 4546 0000 01a5 0100 0000  ..ABCDEF........
00000010: 0a40 2900 0000 0000 00                   .@).....
```

# Exemple de combinaison de filtres

Et leur lecture !

```
1     public static void lire(String fileName) throws IOException {
2         try (DataInputStream in =
3                 new DataInputStream(new FileInputStream(fileName));
4         ) {
5             System.out.println(in.readUTF());
6             System.out.println(in.readInt());
7             System.out.println(in.readBoolean());
8             System.out.println(in.readInt());
9             System.out.println(in.readDouble());
10        }
11    }
12 }
```

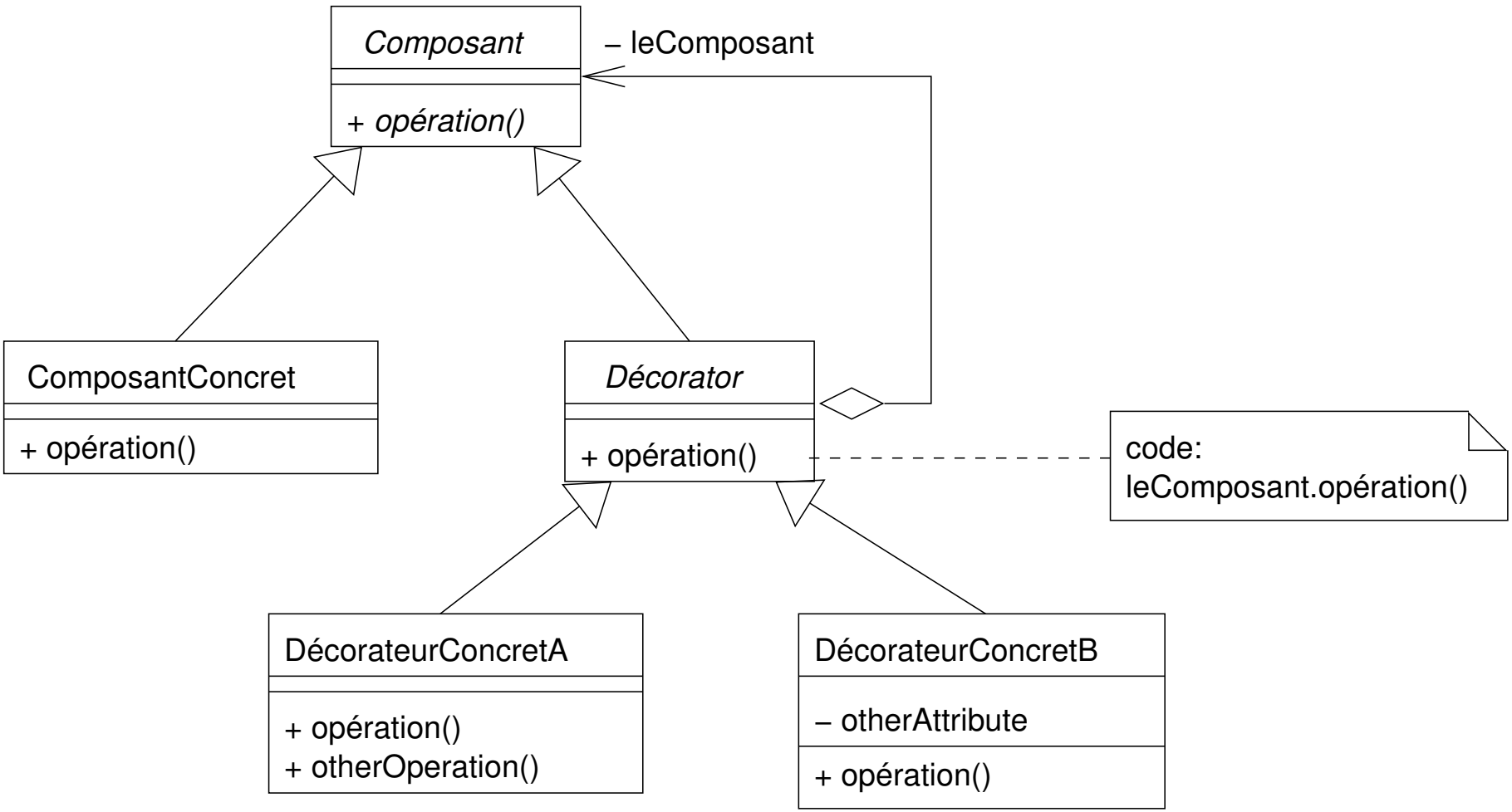
Résultat de : `java TestFiltres`

```
ABCDEF
421
true
10
12.5
```

**Question :** Modifier les deux programmes pour que le fichier soit compressé.

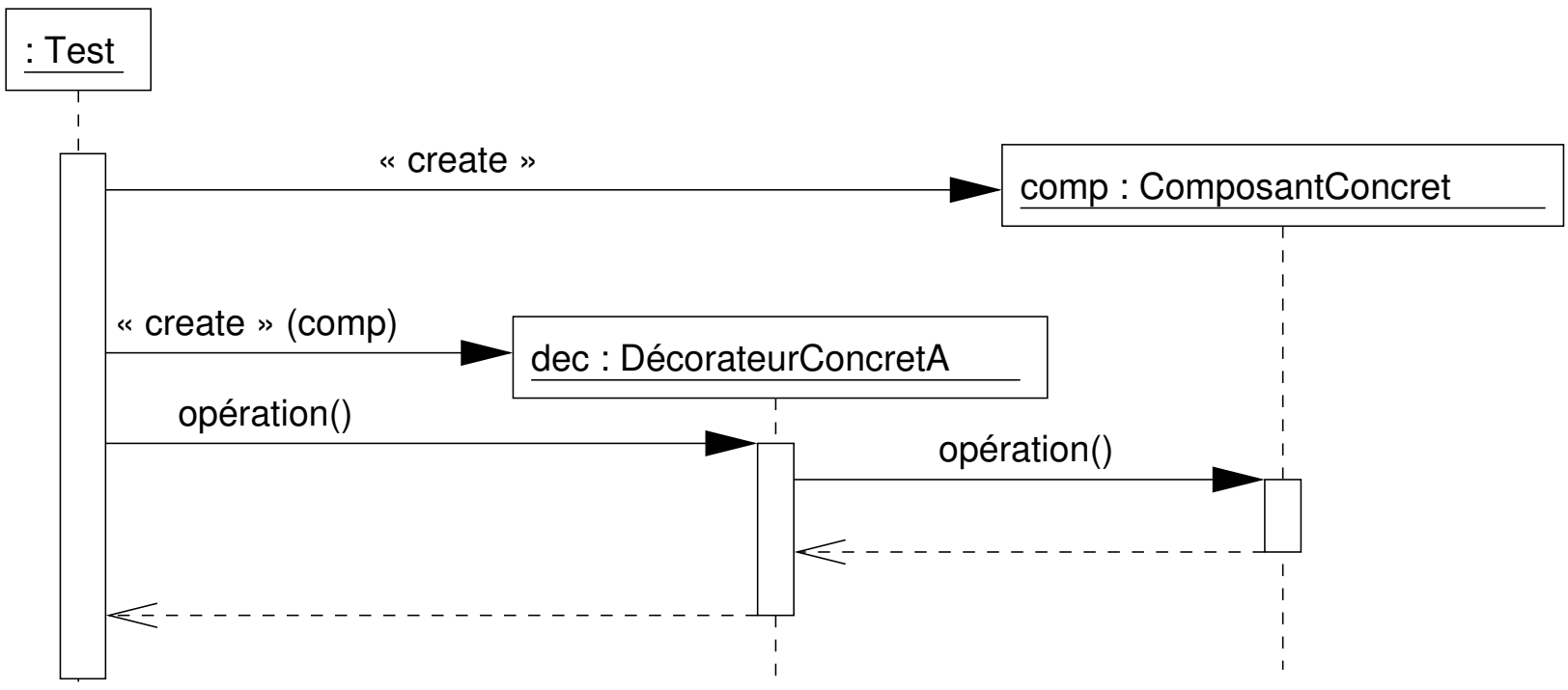
# Décorateur (Decorator)

## Diagramme de classe



# Décorateur (Decorator)

## Diagramme de séquence



# Décorateur (Decorator)

- **Intention**
  - Attache dynamiquement des responsabilités supplémentaires à un objet
  - Alternative souple à l'héritage pour ajouter de nouvelles fonctionnalités
- **Alias :** Emballeur (Wrapper)
- **Indications d'utilisation**
  - ajouter dynamiquement de nouvelles fonctionnalités, de manière transparente (sans changement d'interface)
  - définir des responsabilités qui peuvent être ajoutées/retirées
  - éviter un héritage impossible à cause du trop grand nombre d'extensions *indépendantes* possibles
- **Conséquences :**
  - (+) plus de souplesse que l'héritage (qui est statique)
  - (+) évite de surcharger en fonctionnalités les classes de haut niveau
  - (–) composant et composant décoré n'ont pas la même identité (adresse) !
  - (–) de nombreux petits objets !

# Exemple : Compresser un fichier (GZIP)

```
1  import java.io.*;
2  import java.util.zip.GZIPOutputStream;
3
4  public class GZipper {
5
6      public static void main(String[] args) throws IOException {
7          for (String nomFichier : args) {
8              zipper(nomFichier);
9          }
10     }
11
12     public static void zipper(String nom) throws IOException {
13         String nomSortie = nom + ".gz";
14         try (GZIPOutputStream sortie = new GZIPOutputStream(
15                 new FileOutputStream(nomSortie)))
16         {
17             try (InputStream entree = new FileInputStream(nom)) {
18                 int c;
19                 while ((c = entree.read()) != -1) {
20                     sortie.write(c);
21     }   }   }   }
22 }
```

# Exemple : Compresser une archive Zip

```
1  import java.io.*;
2  import java.util.zip.*;
3
4  public class Zipper {
5
6      public static void zipper(String nomArchive, String[] noms)
7                  throws IOException {
8          try (ZipOutputStream sortie = new ZipOutputStream(
9                  new FileOutputStream(nomArchive));
10         ) {
11             for (String nom: noms) {    // ajouter une entrée
12                 sortie.putNextEntry(new ZipEntry(nom));
13                 try (InputStream entree = new FileInputStream(nom)) {
14                     int c;
15                     while ((c = entree.read()) != -1) {
16                         sortie.write(c);
17                     }
18                 }
19                 sortie.closeEntry();
20     }   }   }
21
22     public static void main(String[] args) throws IOException {
23         zipper("/tmp/tout.zip", args);
24     }
25 }
```

# Exemple : Décompresser un fichier Zip

```
1  import java.io.*;
2  import java.util.zip.*;
3
4  public class Dezipper {
5      public static void main(String[] args) throws IOException {
6          try (ZipInputStream zin = new ZipInputStream(
7                      new FileInputStream(args[0]));
8          ) {
9              ZipEntry entree = zin.getNextEntry();
10             while (entree != null) {
11                 String nomEntree = "/tmp/" + entree.getName();
12                 try (FileOutputStream out = new FileOutputStream(nomEntree)) {
13                     Copier.copier(out, zin);
14                 }
15                 zin.closeEntry();
16                 entree = zin.getNextEntry();
17             }
18         }
19     }
20 }
```

# Sommaire

# La classe java.io.File

- **But :** Représentation abstraite des chemins d'accès aux fichiers/répertoires
- **Attention :** Ne permet ni de lire, ni d'écrire le contenu d'un fichier !

```
1  public class File implements Comparable<File> {
2    File(String nomChemin)
3    File(File parent, String fils)
4    File(String cheminParent, String fils)
5    File(URI uri)
6
7    // droits d'accès (1.6)
8    boolean canExecute()
9    boolean canRead()
10   boolean canWrite()
11   boolean setWritable(boolean writable, boolean ownerOnly)
12   boolean setWritable(boolean writable)     // setWritable(writable, true)
13   ... idem pour setExecutable, setReadable
14
15   // le nom du chemin
16   String getName()          // équivalent de basename
17   String getParent()        // équivalent de dirname
18   String getPath()          // ...
19   String getAbsolutePath()  // le chemin absolu correspondant
20   String getCanonicalPath() // absolu et unique
21   File getAbsoluteFile()    // ce fichier avec un chemin absolu
22   File getCanonicalFile()   // ce fichier avec un chemin canonique
```

# La classe java.io.File (2)

```
23   URI getURI()              // URI désignant ce fichier
24   boolean isAbsolute()      // est un chemin absolu
25
26   // nature et caractéristiques
27   boolean isDirectory()     // répertoire ?
28   boolean isFile()          // fichier ?
29   boolean isHidden()        // caché ?
30   long length()             // longueur en octet
31   long lastModified()       // date de dernière modification
32   boolean setLastModified(long)
33
34   // accès au contenu d'un répertoire
35   String[] list()               // noms des éléments contenus ds ce répertoire (ou null)
36   String[] list(FilenameFilter) // avec filtre
37   File[] listFiles()            // éléments contenus dans ce répertoire (ou null)
38   File[] listFiles(FilenameFilter)  // avec filtre
39   static File[] listRoots()     // liste les racines
40
41   // espace disponible
42   long getTotalSpace()      // taille de la partition
43   long getFreeSpace()       // nombre d'octets non alloués sur cette partition
44   long getUsableSpace()     // octets disponibles sur partition pour JVM (1.6)
45
46   // liens avec le système de gestion de fichier
47   boolean exists()      // est-ce que le fichier existe ?
48   boolean createNewFile()   // crée un fichier avec ce nom ssi n'existe pas
49   boolean delete()          // détruit le fichier correspondant
```

# La classe java.io.File (3)

```
50    boolean deleteOnExit()    // fichier doit être détruit quand la JVM finit
51    boolean mkdir()           // créer le répertoire correspondant à ce fichier
52    boolean mkdirs()          // ... y compris les répertoires parents inexistants
53    boolean renameTo(File dest)   // dépendant de la plate-forme
54
55    // création de nom de fichiers temporaires
56    static File createTempFile(String prefix, String suffix, File directory)
57    static File createTempFile(String prefix, String suffix)
58 }
```

# Exemple d'utilisation de File

**But :** afficher les fichiers et le contenu des répertoires récursivement

```
1  public class Lister {
2      public static void lister(java.io.File file) {
3          System.out.print(file.getName());
4          if (!file.exists()) {
5              System.out.println(":␣fichier␣ou␣répertoire␣inexistant␣!");
6          } else if (file.isDirectory()) {
7              System.out.println(":");
8              lister(file, file.list());
9              System.out.println();
10         } else {
11             if (file.canExecute()) {
12                 System.out.print("*");
13             }
14             System.out.println();
15     }   }
16
17     public static void lister(File parent, String[] noms) {
18         for (String nom: noms) {
19             lister(new java.io.File(parent, nom));
20     }   }
21
22     public static void main(String[] args) {
23         lister(null, args);
24 }   }
```

# Sommaire

# La classe `java.io.RandomAccessFile`

La classe `RandomAccessFile` (sous classe de `Object`) permet de manipuler un **fichier**, vu comme un tableau d'octets stockés dans le système de gestion de fichiers.
Elle permet de :

- se positionner dans le fichier (seek)
- lire et écrire des informations (méthodes similaire à `InputStream` et `OutputStream`)
- permet de manipuler les types primitifs (tout type readXXX et writeXXX)

# Sommaire



- Les Reader
- Les Writer
- Passerelle entre Stream et Reader/Writer
- PushbackReader

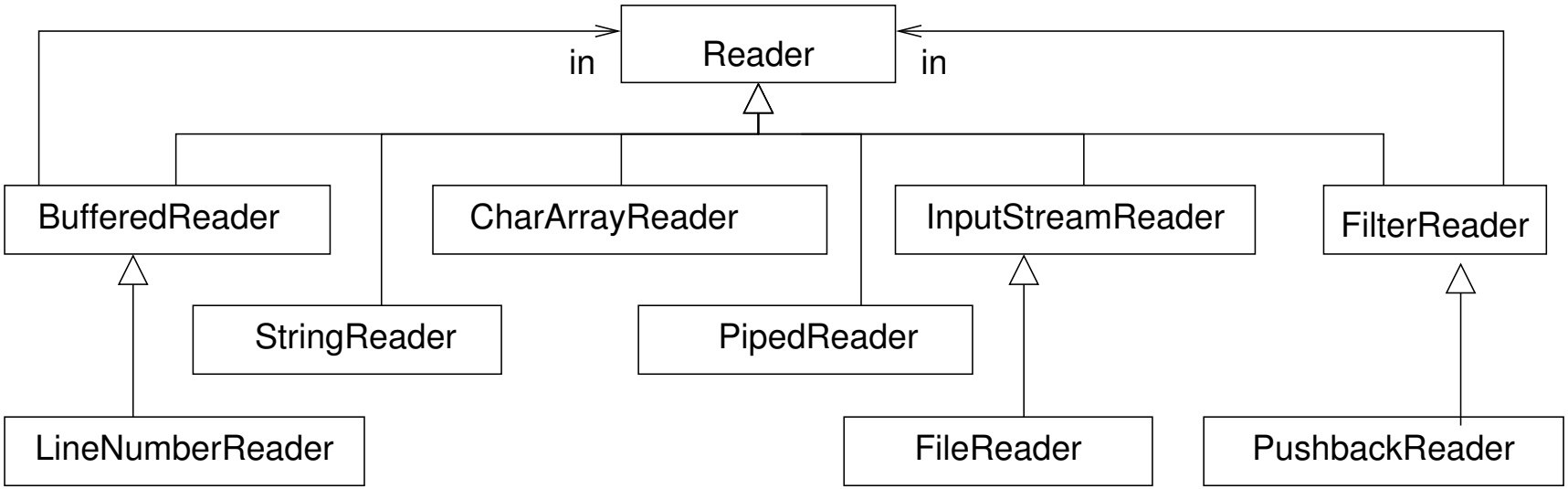
# Les classes Reader et Writer

- **Principe :** Équivalentes à `InputStream` et `OutputStream` mais avec des caractères et non des octets.
- **`int`** pour stocker 16 bits d'un caractère ou -1 pour fin du flux.

```
1 public abstract class Reader {
2   public int read() throws IOException;
3   public abstract int read(char[] buf, int off, int len) throws IOException;
4   public int read(char[] b) throws IOException;
5   public long skip(long n) throws IOException
6   public abstract void close() throws IOException;
7   public boolean ready() throws IOException;    // prêt à être lu ?
8 }
```

```
1 public abstract class Writer {
2   public void write(int b) throws IOException;
3   public abstract void write(char[] buf, int off, int len) throws IOException;
4   public void write(char[] b) throws IOException;
5   public void write(String str) throws IOException;
6   public void write(String str, int off, int len) throws IOException;
7   public void flush() throws IOException
8   public abstract void close() throws IOException;
9 }
```

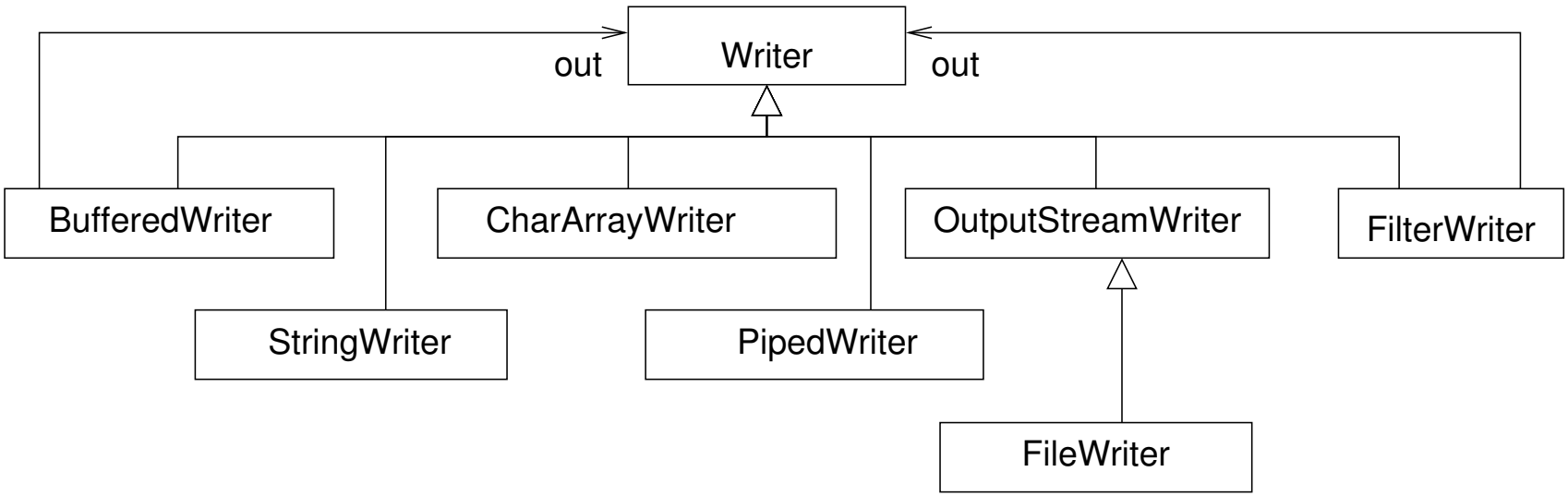
# Les Reader




Architecture proche de celle de `InputStream` mais :

- `BufferedReader` n'est pas défini comme sous-classe de `FilterReader`... même s'il a les caractéristiques d'un filtre
- On peut donc quand même le combiner avec les autres filtres !

# Les Writer




Architecture proche de celle de `OutputStream`, calquée sur celle de `Reader`.

# Lien entre Stream en Reader/Writer

## InputStreamReader et OutputStreamWriter

- InputStreamReader permet de considérer un InputStream comme un Reader
  - C'est un Adaptateur !
- InputStreamReader est une spécialisation de Reader.

```
1  public class InputStreamReader extends java.io.Reader {
2
3      public InputStreamReader(InputStream in, String charSetName)
4          throws UnsupportedEncodingException
5      // Le flux d'entrée est in. charSetName est le nom du codage
6      // utilisé (US-ASCII, ISO-8859-1, UTF-8...)
7
8      public InputStreamReader(InputStream in);
9      // Le flux est in avec le jeu de caractères par défaut
10 }
```

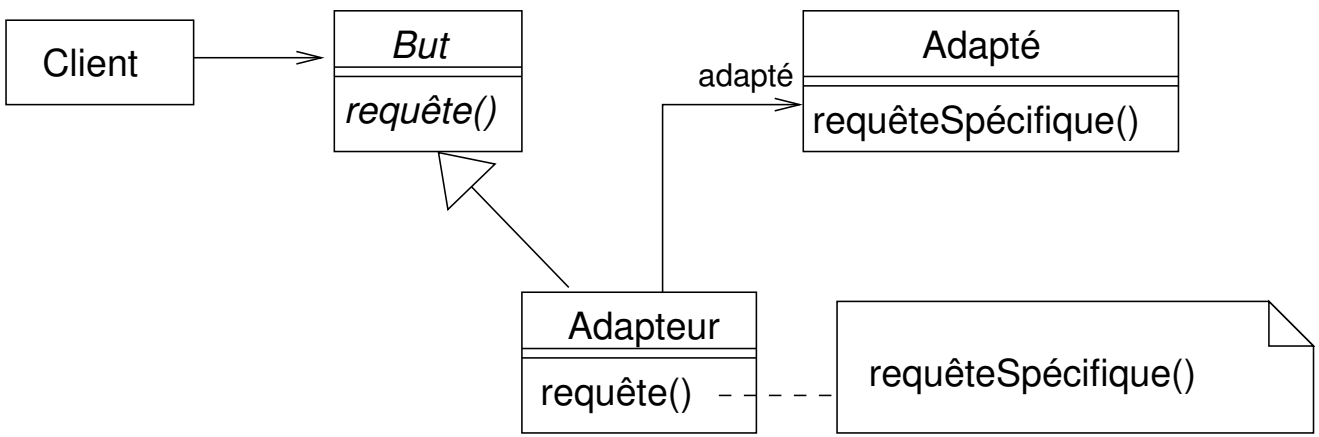
- **Exemple :**

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

# Le patron Adaptateur

**Intention**

- Convertit l'interface d'une classe en une autre conforme à l'attente d'un client.
- Permet de faire collaborer des classes aux interfaces incompatibles.




**Question :** Expliquer en quoi `InputStreamReader` est un Adaptateur.

# Un mini-recode !

Convertir un fichier latin1 en fichier UTF8 (avec l'extension .utf8).

```
1  import java.io.*;
2
3  public class MiniRecode {
4
5     public static void main(String[] args) throws IOException {
6        assert args.length == 1;
7        String name = args[0];
8        try (
9           Reader in = new InputStreamReader(new FileInputStream(name), "latin1");
10          Writer out = new OutputStreamWriter(new FileOutputStream(name + ".utf8"), "UTF8");
11       ) {
12          int c;
13          while ((c = in.read()) >= 0) {
14             out.write(c);
15 }  }  }  }
```

`FileReader` ou `FileWriter` ne permettent pas de sélectionner l'encodage.

# PushbackReader : pouvoir remettre des caractères dans le flux

Permet de remettre des caractères dans le flux d'entrée :

```
1 public void unread(int c) throws IOException
2     // remettre c dans ce flux d'entrée
3
4 public void unread(char[] cbuf, int off, int len) throws IOException
5
6 public void unread(char[] cbuf) throws IOException
```

Exemple : Définir un mini-scanner...

# MiniScanner

```
1  import java.io.*;
2
3  public class MiniScanner {
4
5      private PushbackReader input;
6
7      public MiniScanner(Reader r) {
8          this.input = new PushbackReader(r);
9      }
10
11     public MiniScanner(InputStream in) {
12         this(new InputStreamReader(in));
13     }
14
15     public char readChar() throws IOException {
16         return (char) input.read();
17     }
18
19     public void skipWhite() throws IOException {
20         char c;
21         while (Character.isWhitespace(c = this.readChar())) {
22             // Rien à faire
23         }
24         this.input.unread(c);   // lu en avant !
25     }
26
27     public int readInt() throws IOException {   // très simplifié !
28         this.skipWhite();
```

# MiniScanner (2)

```
29
30          // reconnaître un entier
31          int c;
32          int nb = 0;
33          while (Character.isDigit(c = this.readChar())) {
34              nb = (nb * 10) + (c - '0');
35          }
36          this.input.unread(c);   // lu en avant !
37          return nb;
38      }
39
40  }
```

```
1  import java.io.*;
2  public class ExempleMiniScanner {
3      public static void main(String[] args) throws IOException {
4          Reader reader = new StringReader("␣␣305m␣3x10");
5          MiniScanner scanner = new MiniScanner(reader);
6          System.out.println(scanner.readInt());
7          System.out.println(scanner.readChar());
8          System.out.println(scanner.readInt());
9          System.out.println(scanner.readChar());
10         System.out.println(scanner.readInt());
11     }
12 }
```

```
1 305
2 m
3 3
4 x
5 10
```

# Sommaire



- String
- Expressions régulières
- Scanner
- StreamTokenizer

# String : opérations de manipulation de chaînes

La classe `String` propose de nombreuse méthodes pour travailler avec les chaînes.
En voici quelques unes :

```
1  boolean startsWith(String prefix)       // est ce que this commence par prefix ?
2  void trim()                             // supprimer les espaces en début et fin
3
4  boolean matches(String regexp)          // est-ce que this correspond à regexp ?
5
6  String replaceFirst(String regex, String replacement)
7  String replaceAll(String regex, String replacement)
8          // remplace la première (toutes les) occurrence(s) de regex par la
9          // chaîne replacement
10
11 String[] split(String regexp)   // découpe this en un tableau de String
12 String[] split(String regexp, int limit)
13             // le tableau créé a au plus limit éléments (si limit > 0)
```

Bon nombre de ces méthodes s'appuient sur des **expressions régulières** (regexp).

# Lire des propriétés

Contenu d'un fichier de propriétés :

```
1 no-space:without spaces around semicolon
2       first : whith many spaces
3
4 empty:
5
6 semicolons: :::: one with many semicolons :::::
7 spaces: with spaces at the end of the value
```

Comment faire pour le décoder ?

# Lire des propriétés

```
1  import java.io.*;
2  import java.util.*;
3
4  public class Properties {
5      private Map<String, String> maps = new TreeMap<>();
6
7      public static Properties load(Reader r) throws IOException {
8          Properties props = new Properties();
9          BufferedReader in = new BufferedReader(r);
10         String line;
11         while ((line = in.readLine()) != null) { // décoder la ligne
12             String[] parts = line.split("\\s*:\\s*", 2);
13             if (parts.length == 2) {
14                 String name = parts[0].trim();
15                 String value = parts[1];
16                 props.maps.put(name, value);
17         }   }
18         return props;
19     }
20
21     public String get(String name)      { return this.maps.get(name); }
22     @Override public String toString()  { return "" + this.maps; }
23
24     public static void main(String[] args) throws IOException {
25         assert args.length == 1;
26         System.out.println(Properties.load(new FileReader(args[0])));
27     }
28 }
```

# MiniGrep

Exemple d'utilisation : `java MiniGrep motif MiniGrep.java`

```
MiniGrep.java:5         String motif = args[0];
MiniGrep.java:7             grep(".*" + motif + ".*", args[i]);
MiniGrep.java:10    public static void grep(String motif, String fichier) throws IOExceptior
MiniGrep.java:15                if (ligne.matches(motif)) {
```

```java
import java.io.*;
public class MiniGrep {
    public static void main(String[] args) throws IOException {
        assert args.length > 0;
        String motif = args[0];
        for (int i = 1; i < args.length; i++) {
            grep(".*" + motif + ".*", args[i]);
    }   }

    public static void grep(String motif, String fichier) throws IOException {

        try (LineNumberReader in = new LineNumberReader(new FileReader(fichier))) {
            String ligne;
            while ((ligne = in.readLine()) != null) {
                if (ligne.matches(motif)) {
                    System.out.println(fichier + ":" +in.getLineNumber() + "␣" + ligne);
}   }   }   }   }
```

# Expressions régulières en Java

- Paquetage : java.util.regexp (depuis java 1.4)
- classe Matcher : réalise des opérations d'appariement d'une séquence de caractères avec un motif (pattern)
  - matches : la séquence entière correspond au motif
  - lookingAt : le début de la séquence correspond au motif
  - find : analyse la séquence pour trouver la prochaine sous-séquence qui correspond au motif
- Pattern : représentation compilée d'un motif (efficace si plusieurs utilisations)
- Pattern ne définit pas de constructeur $\Longrightarrow$ méthodes statiques

```
1 Pattern p = Pattern.compile("a*b");
2 Matcher m = p.matcher("aaaaab");
3 boolean b = m.matches();
```

Voir la documentation (javadoc) pour la syntaxe des expressions régulières.

# Scanner

Scanner : un analyseur lexical simple qui traite les types primitifs et les chaînes (String) grâce à des expressions régulières.

lire une information d'un type primitif : nextXXX

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

interroger sur la présence d'un type primitif dans le flux hasNextXXX

```
1 Scanner sc = new Scanner(new File("myNumbers"));
2 while (sc.hasNextLong()) {
3     long aLong = sc.nextLong();
4 }
```

Les délimiteurs peuvent être changés :

```
1 String input = "1␣fish␣2␣fish␣red␣fish␣blue␣fish";
2 Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
3 System.out.println(s.nextInt());
4 System.out.println(s.nextInt());
5 System.out.println(s.next());
6 System.out.println(s.next());
7 s.close();
```

donne : 1 2 red blue

# StreamTokenizer : Analyseur lexical simplifié

- Reconnaît 5 types de lexèmes (token) :
  - TT_EOL : fin de ligne (si eolIsSignificant(true))
  - TT_EOF : fin de fichier
  - TT_WORD : un mot (String), nval est le mot lu
  - TT_NUMBER : un nombre (double), nval et le nombre lu
  - une « quote string » : le token est le caractère qui délimite la chaîne
- Principales méthodes

```
1  StreamTokenizer(Reader)          // Constructeur
2  void commentChar(int ch)         // commentaire de ch à EOL
3  void eolIsSignificant(boolean flag)     // considérer EOL comme un token
4  int lineno()                     // numéro de ligne
5  int nextToken()                  // lexème suivant
6          // voir sval ou nval suivant le cas
7  void ordinaryChar(int ch)        // rendre ch un caractère normal
8  void ordinaryChars(int low, int hi)     // low..hi deviennent normaux
9  void parseNumbers()              // demander à analyser les nombre (défaut)
10 void pushBack()                  // remettre le dernier lexème dans le flux
11 void quoteChar(int ch)           // caractère pour quote string (plusieurs possibles)
12 void whitespaceChars(int low, int hi)   // caractères blancs
13 void wordChars(int low, int hi)         // caractères font partie de WORD
14 void slashStarComments(boolean flag)    // commentaires //
15 void slashSlashComments(boolean flag)   // commentaires /* ... */
```

# Exemple d'utilisation de StreamTokenizer

```
1  import java.io.*;
2  public class EssaiStreamTokenizer {
3      public static void main(String[] args) throws IOException {
4          Reader r = new StringReader("Ceci < est > MA\n \"Chaine\"\n"
5                  +"--- 0 1 10.5 -4 ---\n 'YYY' X ! FIN\nY");
6          StreamTokenizer s = new StreamTokenizer(r);
7          s.eolIsSignificant(true);   // EOL est un lexème
8          s.commentChar('!');         // commentaire de ! à EOL
9          s.lowerCaseMode(true);      // tout en minuscule
10         int token;
11         while ((token = s.nextToken()) != StreamTokenizer.TT_EOF) { // Pas fini !
12             switch (token) {
13                 case StreamTokenizer.TT_WORD:
14                     System.out.println("mot = " + s.sval + "    @" + s.lineno());
15                     break;
16                 case StreamTokenizer.TT_NUMBER:
17                     System.out.println("nombre = " + s.nval); break;
18                 case StreamTokenizer.TT_EOL:
19                     System.out.println("EOL : numero ligne = " + s.lineno()); break;
20                 case '"':
21                     System.out.println("\"string = \"" + s.sval + '"'); break;
22                 case '\'':
23                     System.out.println("'string = '" + s.sval + "'"); break;
24                 default:
25                     System.out.println("char = '" + (char) s.ttype + "'"); break;
26 }   }   }   }
```

# Résultat de l'exécution de EssaiStreamTokenizer

```
1  mot = ceci  @1
2  char = '<'
3  mot = est   @1
4  char = '>'
5  mot = ma    @1
6  EOL : numero ligne = 2
7  "string = "Chaine"
8  EOL : numero ligne = 3
9  char = '-'
10 char = '-'
11 char = '-'
12 nombre = 0.0
13 nombre = 1.0
14 nombre = 10.5
15 nombre = -4.0
16 char = '-'
17 char = '-'
18 char = '-'
19 EOL : numero ligne = 4
20 'string = 'YYY'
21 mot = x @4
22 EOL : numero ligne = 5
23 mot = y @5
```

- `StreamTokenizer` est assez limité !
- Comment traiter le fichier /etc/hosts

```
1 # Fichier hosts donné en exemple par man hosts
2 127.0.0.1       localhost
3 192.168.1.10    toto.mondomaine.org    toto
4 192.168.1.13    titi.mondomaine.org    titi
5 146.82.138.7    master.debian.org      master
6 209.237.226.90  www.opensource.org
```