

# Patrons : *Strategy, Decorator,* *Composite*

Théorie et pratique de la programmation  
Michel Schinz - 2013-04-15

# Patron n° 4 :

## *Strategy*

# Illustration du problème

On désire écrire une méthode de tri permettant de trier des listes d'éléments quelconques :

```
<E> void sort(List<E> list)
```

Pour que cette méthode soit flexible, il faut que l'utilisateur puisse spécifier l'ordre à utiliser pour le tri. En effet, même pour un type comme les chaînes de caractères il existe plusieurs ordres possibles.

Comment faire ?

# Solution

La solution consiste bien entendu à utiliser un comparateur pour spécifier l'ordre de tri.

Pour ce faire, il suffit d'ajouter un paramètre représentant ce comparateur à notre méthode :

```
<E> void sort(List<E> list,  
               Comparator<E> comparator)
```

Chaque fois que la méthode de tri doit comparer deux éléments, elle utilise le comparateur.

# Tri pour cruciverbiste

On peut par exemple utiliser cette méthode pour trier des mots par ordre de longueur - ce qui est utile aux cruciverbistes - au moyen du comparateur suivant :

```
class StringLengthFirstComparator
    implements Comparator<String> {
    public int compare(String s1, String s2){
        int d = Integer.compare(s1.length(),
                               s2.length());
        return (d != 0) ? d : s1.compareTo(s2);
    }
}
```

# Généralisation

Chaque fois qu'une partie d'un algorithme doit être interchangeable, il est possible de la représenter au moyen d'un objet.

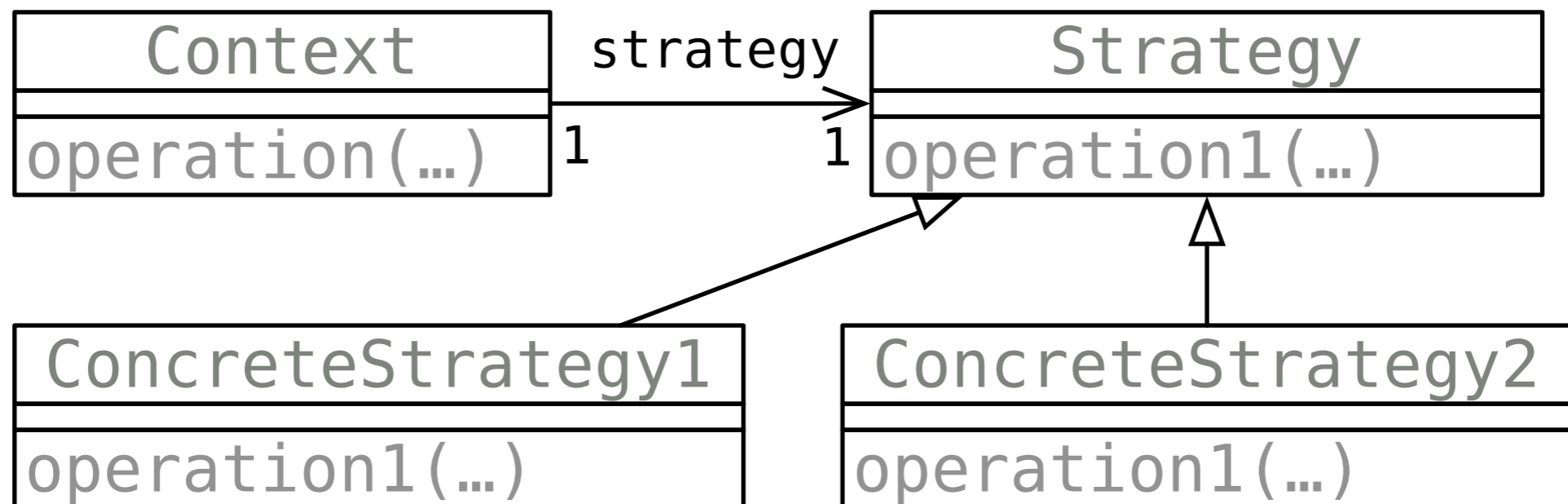
Dans notre exemple, la comparaison de deux objets était la partie variable de l'algorithme de tri, et le comparateur était l'objet la représentant.

Cette solution est décrite par le patron de conception **Strategy** - aussi appelé **Policy**.

# Diagramme de classes

algorithme  
à partie  
variable

partie  
variable de  
l'algorithme



# Exemples réels

La méthode `sort` de la classe `Collections` (paquetage `java.util`) utilise un comparateur pour connaître l'ordre de tri à utiliser.

Le patron *Strategy* est aussi souvent utilisé pour spécifier des politiques de sécurité, qui définissent par exemple les opérations qu'un programme a le droit d'effectuer. C'est cette utilisation qui vaut son second nom (*Policy*) au patron.



# Patron n°4 :

## *Decorator*

# Illustration du problème

Comme nous venons de le voir, la méthode `sort` de la bibliothèque Java permet de trier une liste selon un ordre spécifié par un comparateur.

Cette méthode trie la liste dans l'ordre croissant. Toutefois, il peut être parfois utile de la trier par ordre décroissant.

Etant donné une liste et un comparateur pour les éléments de cette liste, comment peut-on trier la liste par ordre décroissant ?

# Solution

Pour trier la liste par ordre décroissant selon un comparateur  $C$ , il suffit de passer à la méthode `sort` un comparateur qui soit l'inverse de  $C$ .

Nous dirons que deux comparateurs  $C_1$  et  $C_2$  sont inverses l'un de l'autre si  $C_1$  retourne une valeur positive quand  $C_2$  en retourne une négative, et inversement.

Peut-on programmer une méthode qui, étant donné un comparateur, retourne son inverse ?

# Inverseur de comparateur

Pour pouvoir définir une telle méthode, la première chose à faire est de définir un nouveau type de comparateur, qui se comporte comme l'inverse d'un autre comparateur.

Cela s'avère assez simple : la méthode de comparaison de notre inverseur de comparateur doit appeler celle du comparateur d'origine en échangeant les arguments.

# Comparateur inversant

```
class InvComparator<E>
    implements Comparator<E> {
    private Comparator<E> c;
    public InvComparator(Comparator<E> c) {
        this.c = c;
    }
    @Override
    public int compare(E s1, E s2) {
        return c.compare(s2, s1);
    }
}
<E> Comparator<E> inv(Comparator<E> c) {
    return new InvComparator<E>(c);
}
```

# Inverseur de comparateur

Bien entendu, il serait également possible d'utiliser une classe imbriquée anonyme pour définir la méthode `inv` :

```
<E> Comparator<E> inv(final Comparator<E> c) {  
    return new Comparator<E>() {  
        @Override  
        public int compare(E s1, E s2) {  
            return c.compare(s2, s1);  
        }  
    };  
}
```

# Utilisation de l'inverseur

Pour trier une liste par ordre décroissant, on peut alors définir la méthode suivante :

```
<E> void invSort(List<E> l,  
                  Comparator<E> c) {  
    Collections.sort(l, inv(c));  
}
```

Cette méthode utilise la méthode de tri de la bibliothèque Java, en inversant au préalable le comparateur passé en argument.

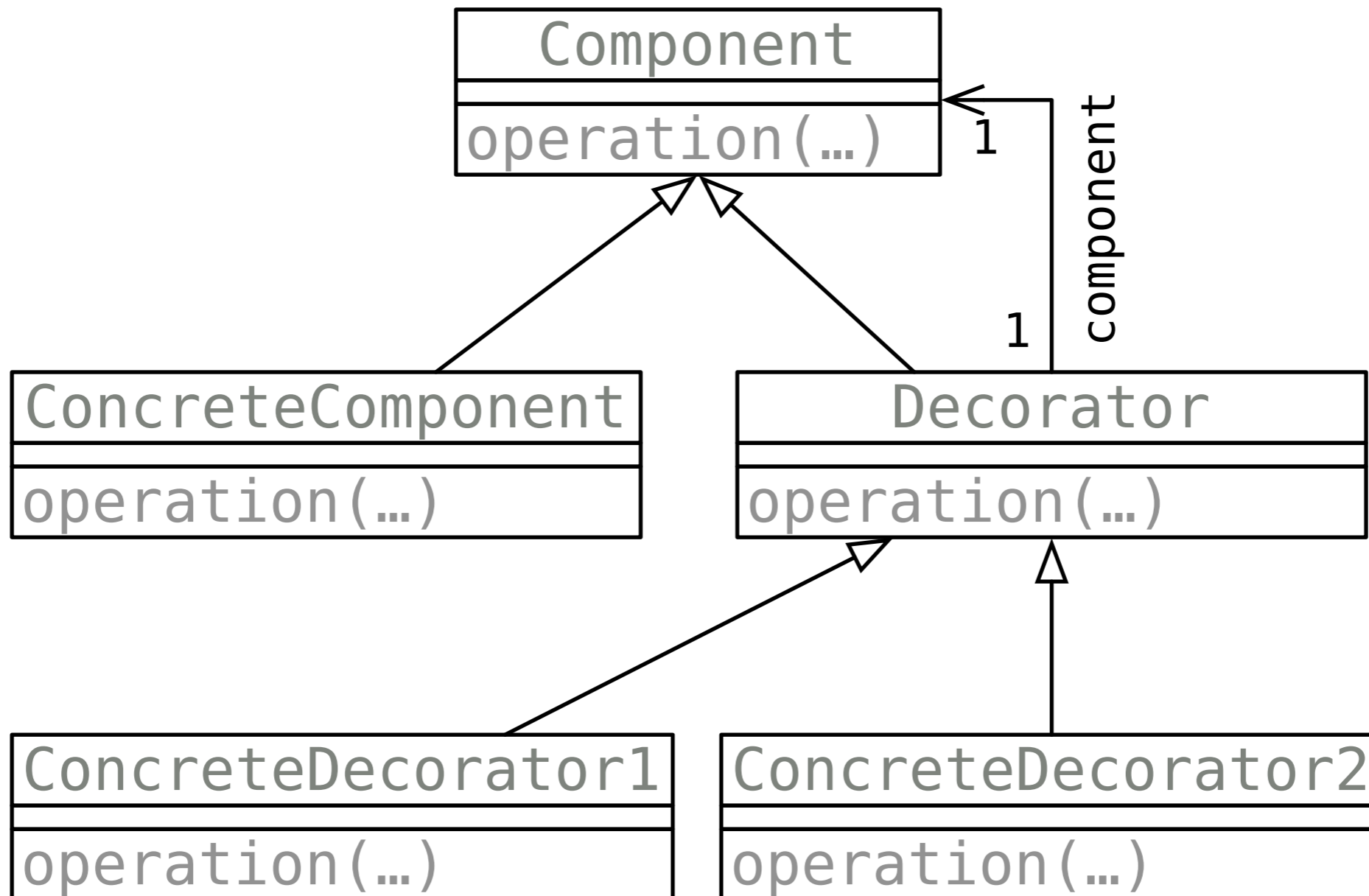
# Généralisation

Chaque fois que l'on désire changer le comportement d'un objet sans changer son interface, on peut « l'emballer » dans un objet ayant la même interface mais un comportement différent. L'objet « emballant » laisse l'objet emballé faire le gros du travail, mais modifie son comportement lorsque cela est nécessaire.

Cette solution est décrite par le patron **Decorator** - aussi appelé **Wrapper**.



# Diagramme de classes



# Exemples réels

Les décorateurs sont utilisés dans les bibliothèques de gestion d'interfaces graphiques pour ajouter des ornements aux éléments graphiques : bordures, barres de défilement, etc. Le nom *Decorator* vient de cette utilisation.

Comme nous allons le voir maintenant, les décorateurs sont utilisés intensivement dans la bibliothèque Java pour gérer les entrées-sorties.

# Entrées/sorties en Java (digression)

# Flots

Dans la bibliothèque Java, les classes gérant les entrées-sorties sont regroupées dans le paquetage `java.io`.

La notion centrale de ce paquetage est celle de **flot** (*stream*), qui représente une séquence de valeurs à accès séquentiel. C'est-à-dire qu'il n'est en général pas possible de lire ou écrire une valeur à une position quelconque dans un flot, mais seulement à la position courante.

# Sortes de flots

Il existe deux sortes de flots : les flots d'entrée (*input streams*) et de sortie (*output streams*).

Les flots d'entrée permettent de lire des valeurs depuis une source - p.ex. un fichier - pour les traiter dans l'application.

Les flots de sortie permettent d'écrire des valeurs de l'application dans un puits - p.ex. un fichier.

# Flots d'octets/de caractères

Java distingue deux genres de flots, en fonction du type de données qu'ils transportent : les flots de caractères et les flots d'octets.

- Les flots de caractères sont représentés par les classes abstraites `Reader` (entrée) et `Writer` (sortie) et leurs sous-classes.
- Les flots d'octets sont représentés par les classes abstraites `InputStream` (entrée) et `OutputStream` (sortie) et leurs sous-classes.

# Flots concrets

La classe `InputStream` possède plusieurs sous-classes concrètes qui lisent des octets de plusieurs sources différentes, p.ex. :

- `FileInputStream` lit les octets d'un fichier,
- `ByteArrayInputStream` lit les octets d'un tableau.

Des classes équivalentes existent pour les flots de sortie, et pour les flots de caractères.

# Exemple

```
// Détermine si un fichier est un fichier classe Java
public static void main(String[] args) {
    try {
        FileInputStream s = new FileInputStream(args[0]);
        byte[] b = new byte[4];
        int read = s.read(b);
        boolean isClassFile =
            (read == b.length
             && b[0] == (byte)0xCA && b[1] == (byte)0xFE
             && b[2] == (byte)0xBA && b[3] == (byte)0xBE);
        System.out.println(isClassFile);
        s.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



# Transformation des données

Lors d'opérations d'entrée-sortie, il est fréquent de devoir transformer les données lues ou écrites. On peut par exemple vouloir les (dé)compresser, les (dé)crypter ou les (dé)coder.

Pour ce faire, la bibliothèque Java fournit plusieurs classes décorateurs qui transforment les données lues ou écrites dans des flots.

# Décorateur de compression

Par exemple, pour compresser les données au format gzip, il existe la paire de décorateurs `GZIPInputStream` et `GZIPOutputStream`, du paquetage `java.util.zip`. Des décorateurs similaires existent pour d'autres formats de compression.

# Mémoire tampon

En plus des décorateurs transformant les données, la bibliothèque Java offre des décorateurs plaçant une mémoire tampon (*buffer*) à l'entrée ou à la sortie d'un flot, pour améliorer les performances.

L'idée d'une mémoire tampon est de stocker une certaine quantité de données en mémoire avant de les transférer plus loin en une seule fois, p.ex. sur disque. Cela évite les accès trop fréquents au disque, et améliore donc les performances.

# Décorateurs tampons

La classe `BufferedInputStream` est un décorateur qui ajoute une mémoire tampon à un flot d'entrée.

La classe `BufferedOutputStream` est un décorateur qui ajoute une mémoire tampon à un flot de sortie.

# Données complexes

La bibliothèque Java fournit également des décorateurs permettant d'écrire ou de lire des données plus complexes que de simples octets.

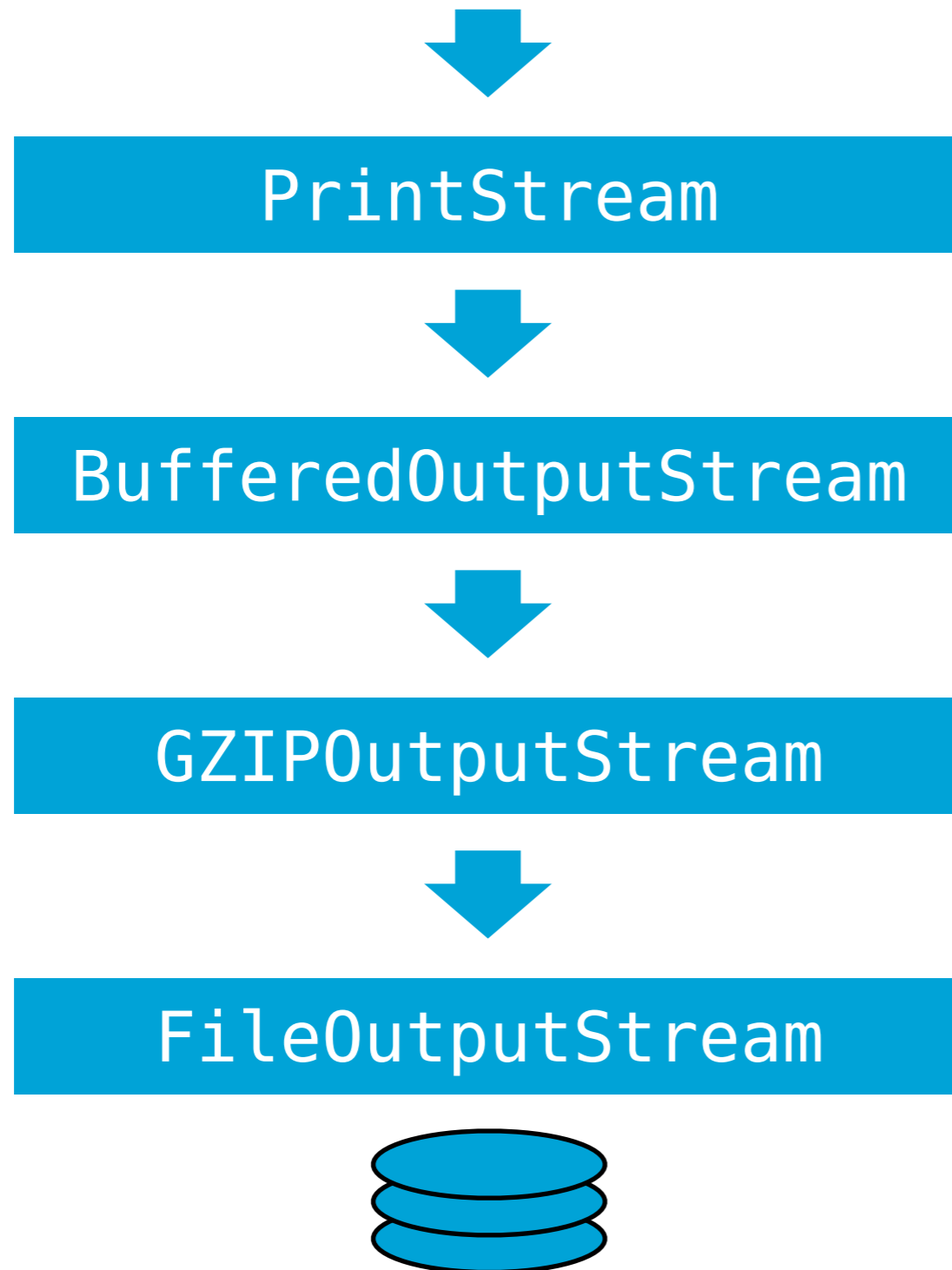
Les classes `DataInputStream` et `DataOutputStream` permettent les entrées-sorties, sous forme binaire, de données Java - entiers, booléens, etc.

La classe `PrintStream` ajoute la possibilité d'imprimer des données sous forme textuelle dans un flot de sortie.

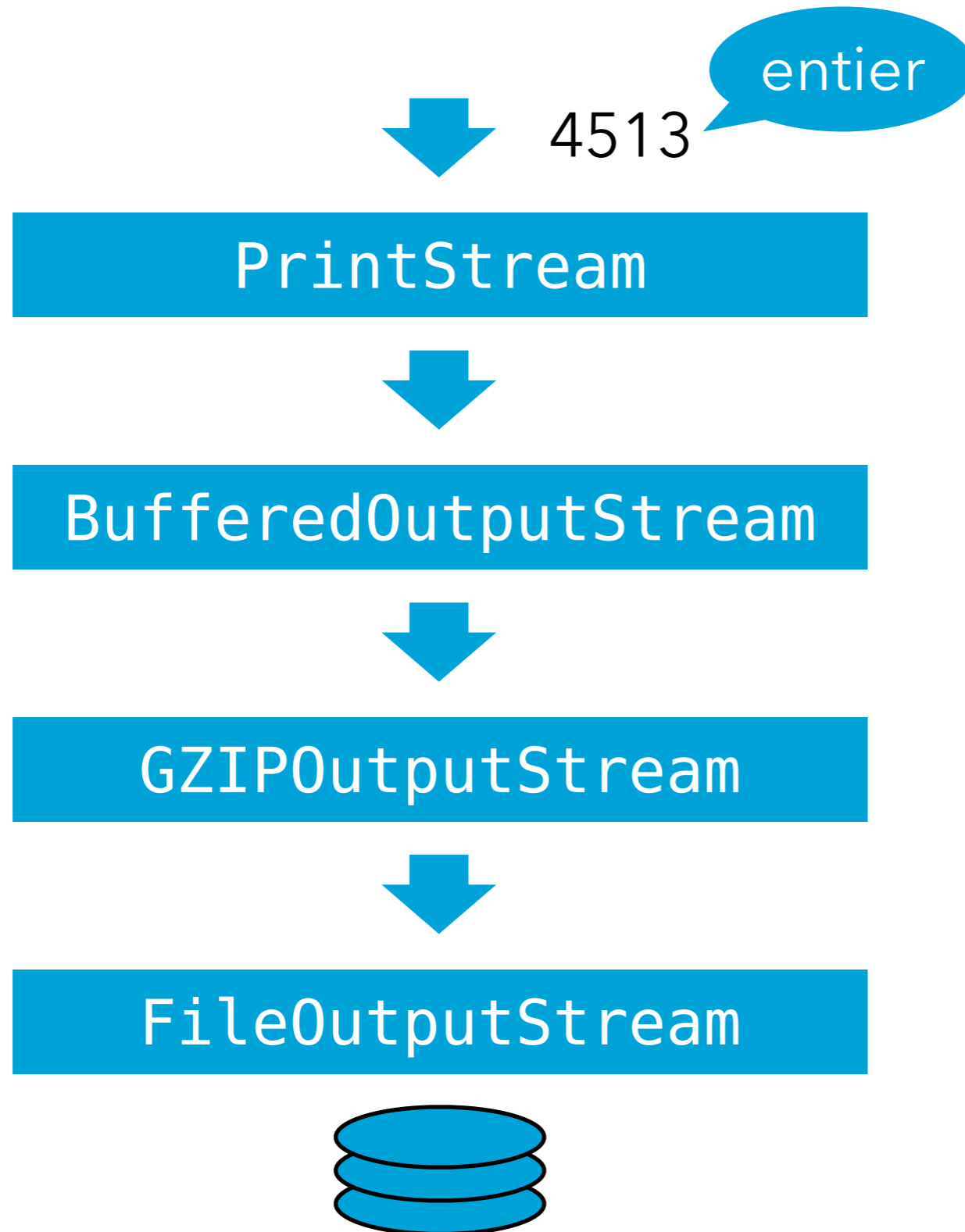
# Exemple

```
// Création d'un fichier compressé
// contenant 100000 nombres aléatoires
// sous forme textuelle.
FileOutputStream fos =
    new FileOutputStream("rand.gz");
GZIPOutputStream zos =
    new GZIPOutputStream(fos);
BufferedOutputStream bos =
    new BufferedOutputStream(zos);
PrintStream ps =
    new PrintStream(bos);
Random r = new Random();
for (int i = 0; i < 100000; ++i)
    ps.println(r.nextInt());
ps.close();
```

# Exemple

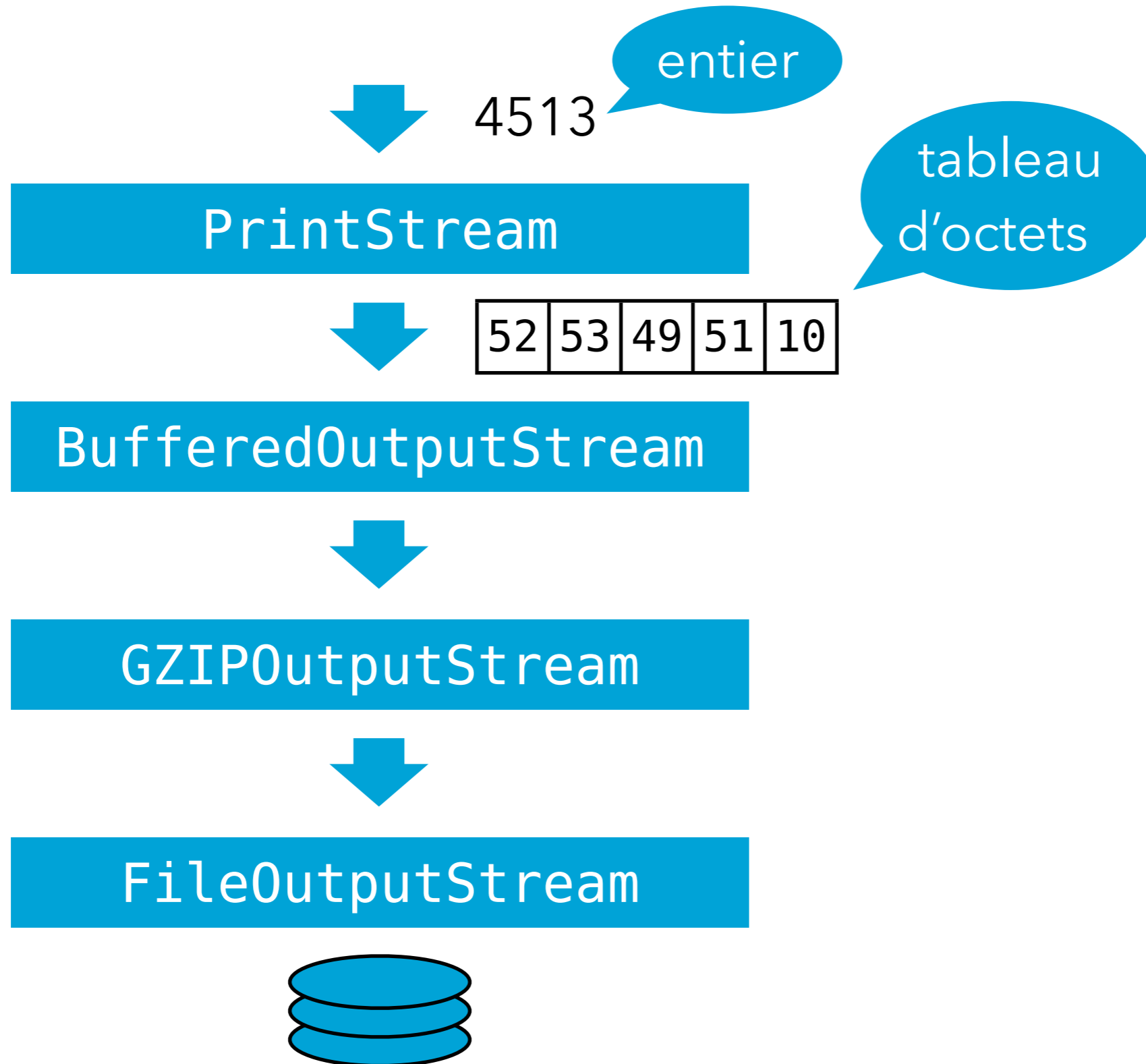


# Exemple

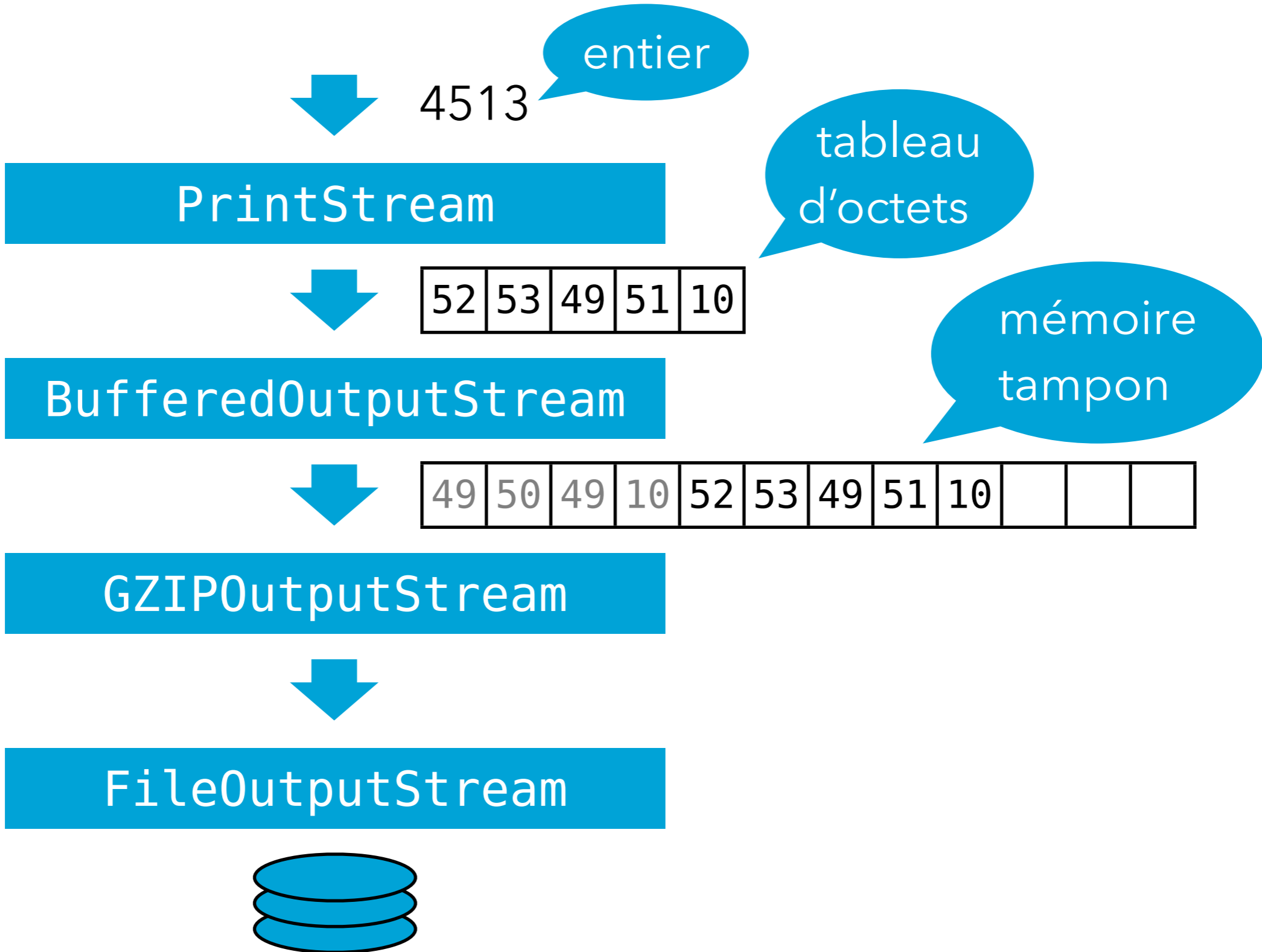




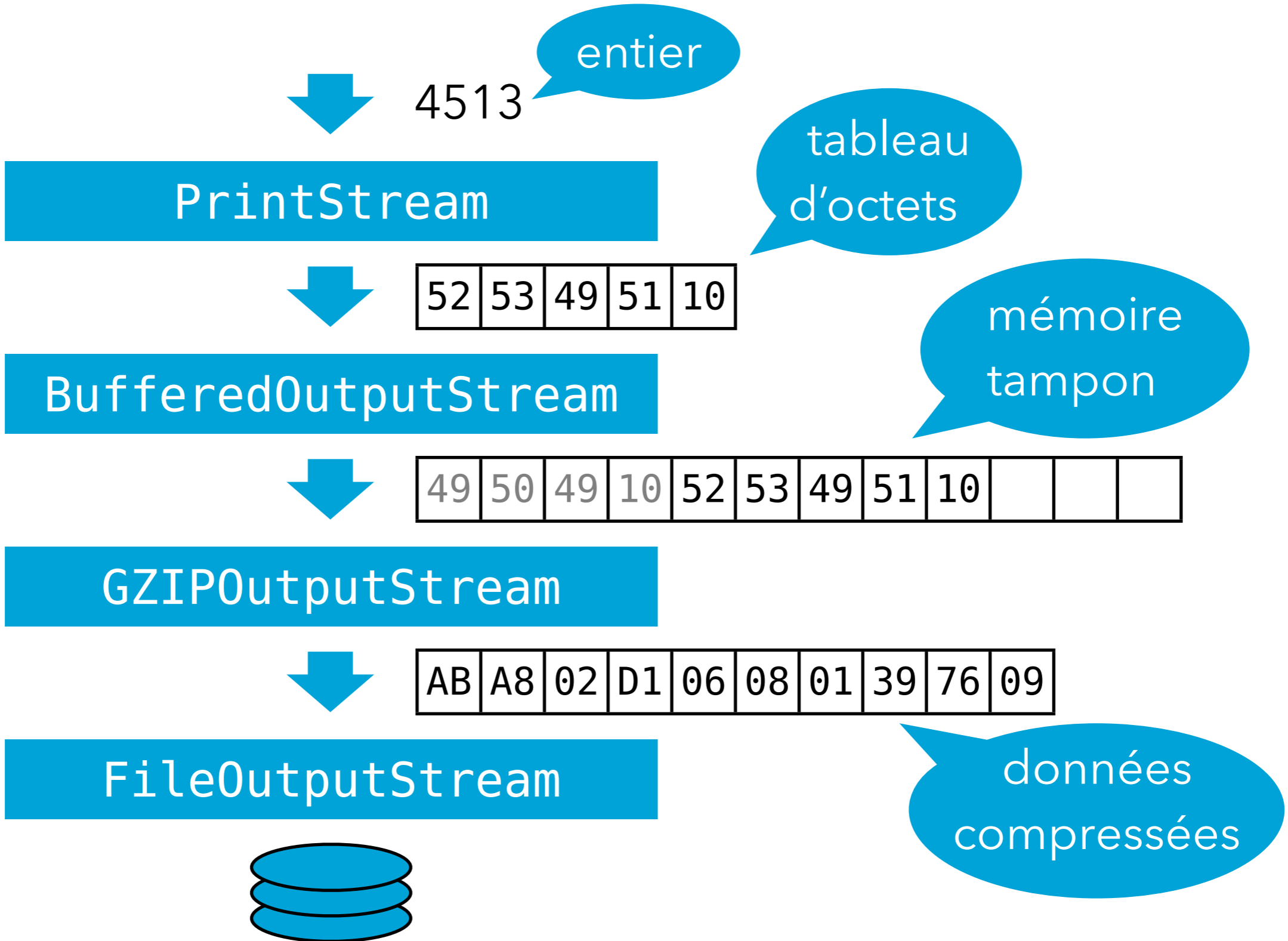
# Exemple



# Exemple



# Exemple



# Patron n°5 :

## *Composite*

# Illustration du problème

Dans un programme de gestion de base de données, il est fréquent de laisser l'utilisateur spécifier un ordre pour trier les données.

Cet ordre peut généralement comporter plusieurs critères. Par exemple, on peut vouloir d'abord trier par nom de famille, puis par prénom.

En interne, un ordre simple peut être représenté par un comparateur. Mais comment représenter les ordres composés de plusieurs critères ?

# Solution

Une solution consiste à définir une classe pour représenter un « macro-comparateur » composé de deux comparateurs  $C_1$  et  $C_2$  à appliquer en séquence. Si  $C_1$  déclare que les objets à comparer sont différents, on utilise son résultat, sinon on utilise celui de  $C_2$ .

Pour qu'un tel macro-comparateur soit utilisable comme les autres, il faut qu'il implémente l'interface **Comparator**.

# Compateur composé

```
class SeqComparator<E>
    implements Comparator<E> {
    private final Comparator<E> c1, c2;
    public SeqComparator(Comparator<E> c1,
                        Comparator<E> c2) {

        this.c1 = c1;
        this.c2 = c2;
    }
    @Override
    public int compare(E e1, E e2) {
        int c = c1.compare(e1, e2);
        return c != 0 ? c : c2.compare(e1, e2);
    }
}
```

# Comparateurs de base

```
class StringLengthComparator
    implements Comparator<String> {
    public int compare(String s1, String s2){
        return Integer.compare(s1.length(),
                               s2.length());
    }
}

class StringLexicalComparator
    implements Comparator<String> {
    public int compare(String s1, String s2){
        return s1.compareTo(s2);
    }
}
```



# Comparateur composé

Une fois les deux comparateurs de base définis, on peut les composer au moyen de `SeqComparator` pour obtenir le comparateur « du cruciverbiste » vu précédemment :

```
Comparator<String> c1 =  
    new StringLengthComparator();  
Comparator<String> c2 =  
    new StringLexicalComparator();  
Comparator<String> c =  
    new SeqComparator<String>(c1, c2);
```

# Exercice

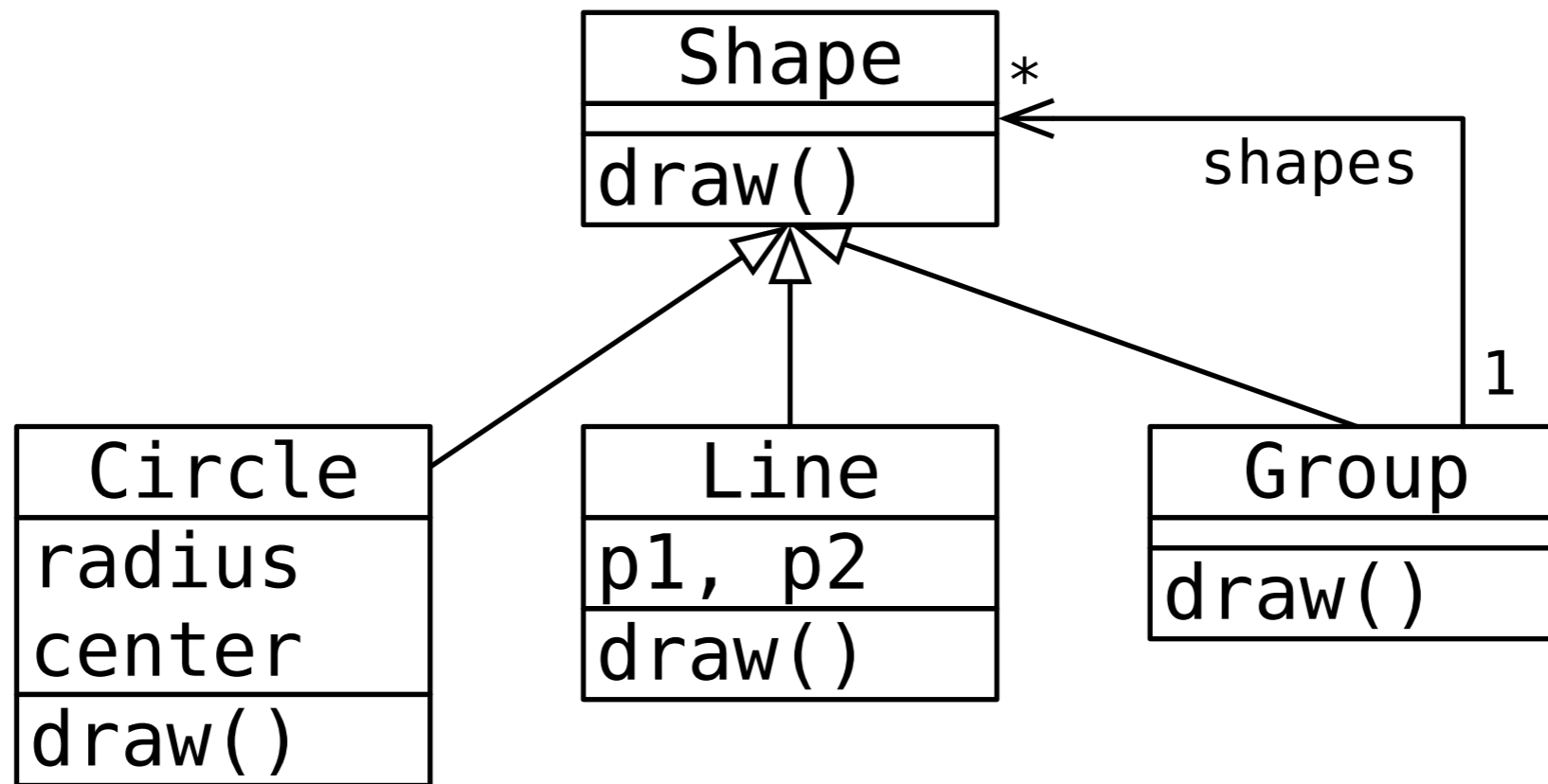
Définissez un comparateur ordonnant les chaînes par ordre *inverse* de longueur puis alphabétiquement.

# Programme de dessin

Dans les programmes de dessin vectoriels, il est fréquent d'offrir la possibilité de grouper plusieurs objets en un objet unique.

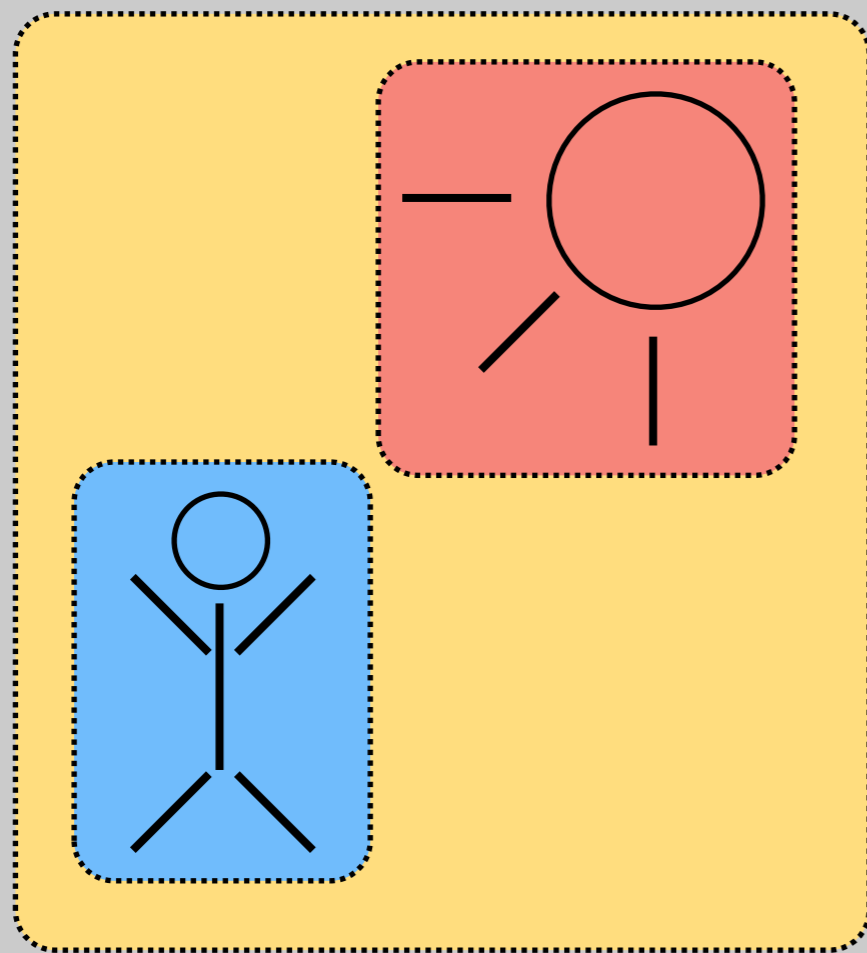
En interne, on peut imaginer avoir une interface **Shape** représentant les formes géométriques, et une classe l'implémentant par forme - ligne, cercle, etc. La classe des groupe de formes implémentera aussi **Shape**, pour faciliter la programmation.

# Diagramme de classes

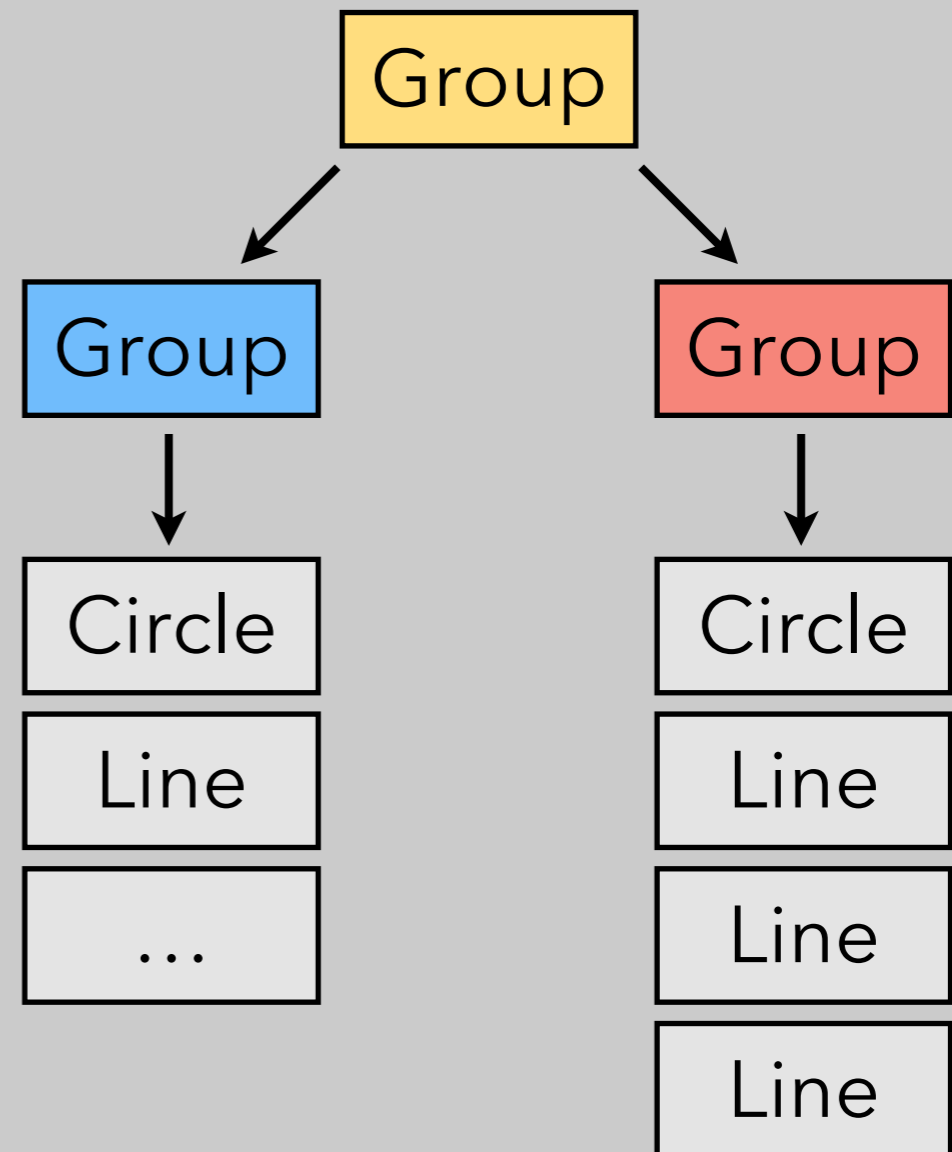


# Représentation mémoire

dessin



représentation



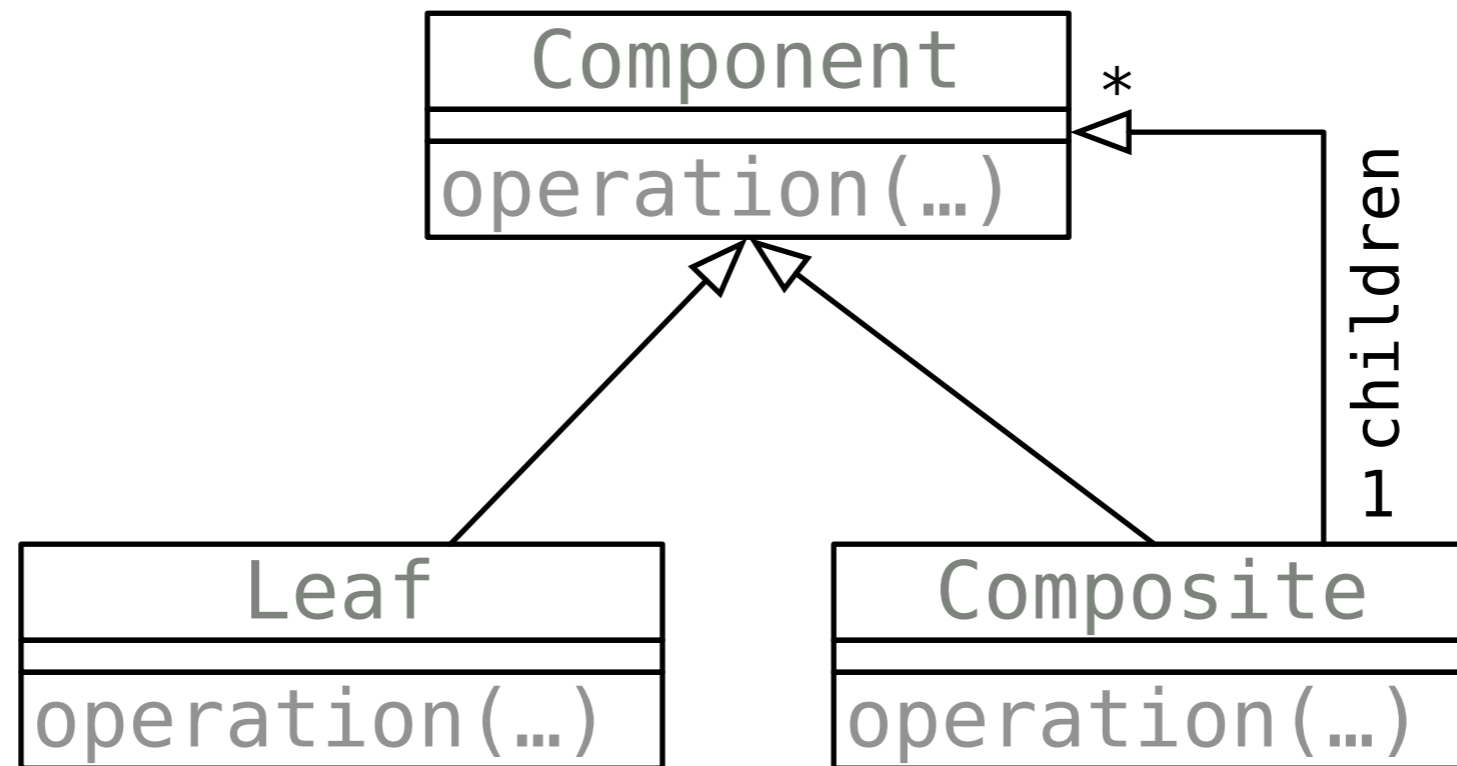
# Généralisation

Chaque fois qu'un programme permet à l'utilisateur de grouper plusieurs éléments en un macro-élément, il faut s'assurer que le type des éléments de base et des macro-éléments soit le même.

Cela permet de traiter tous les éléments de manière uniforme, et simplifie grandement l'organisation interne du programme.

C'est l'idée du patron **Composite**.

# Diagramme de classes



# Exemple réel

Les composants Swing d'une telle interface - bouton, barres de menus, etc. - sont tous modélisés par des sous-classes de la classe **JComponent**.

Swing inclut plusieurs composants composites comme le « panneau à onglets », **JTabbedPane**. Il présente à l'utilisateur un ensemble de sous-composants entre lesquels il est possible de naviguer au moyen d'onglets.

La classe **JTabbedPane** est elle-même une sous-classe de **JComponent**. Cela implique entre autres qu'il est possible d'imbriquer plusieurs panneaux à onglets.



# Résumé

Le patron *Strategy* permet de représenter une partie variable d'un algorithme par un objet.

Le patron *Decorator* (ou *Wrapper*) permet de changer le comportement d'un objet sans changer son interface.

Le patron *Composite* permet de représenter des objets composés comme des objets simples, simplifiant grandement leur manipulation.