

# Les entrée/sortie en Java

Une entrée/sortie en Java consiste en un échange de données entre le programme et une autre source, par exemple la mémoire, un fichier, le programme lui-même... Pour réaliser cela, Java emploie ce qu'on appelle un *stream* (qui signifie « flux »). Celui-ci joue le rôle de médiateur entre la source des données et sa destination. Nous allons voir que Java met à notre disposition toute une panoplie d'objets permettant de communiquer de la sorte. Toute opération sur les entrées/sorties doit suivre le schéma suivant : ouverture, lecture, fermeture du flux.

Je ne vous cache pas qu'il existe une foule d'objets qui ont chacun leur façon de travailler avec les flux. Sachez que Java a décomposé les objets traitant des flux en deux catégories :

- les objets travaillant avec des flux d'entrée (`in`), pour la lecture de flux ;
- les objets travaillant avec des flux de sortie (`out`), pour l'écriture de flux.

## Utilisation de java.io

### L'objet `File`

Avant de commencer, créez un fichier avec l'extension que vous voulez et enregistrez-le à la racine de votre projet Eclipse. Personnellement, je me suis fait un fichier `test.txt` dont voici le contenu :

```
Voici une ligne de test.
```

```
Voici une autre ligne de test.
```

```
Et comme je suis motivé, en voici une troisième !
```

Dans votre projet Eclipse, faites un clic droit sur le dossier de votre projet, puis `New > File`. Vous pouvez nommer votre fichier ainsi qu'y taper du texte !

Le nom du dossier contenant mon projet s'appelle « IO » et mon fichier texte est à cette adresse

`:D:\Mes documents\Codage\SDZ\Java-SDZ\IO\test.txt`. Nous allons maintenant voir ce dont l'objet `File` est capable. Vous remarquerez que cet objet est très simple à utiliser et que ses méthodes sont très explicites.

```
//Package à importer afin d'utiliser l'objet File

import java.io.File;

public class Main {

    public static void main(String[] args) {

        //Création de l'objet File

        File f = new File("test.txt");

        System.out.println("Chemin absolu du fichier : " + f.getAbsolutePath());

        System.out.println("Nom du fichier : " + f.getName());

        System.out.println("Est-ce qu'il existe ? " + f.exists());

        System.out.println("Est-ce un répertoire ? " + f.isDirectory());

        System.out.println("Est-ce un fichier ? " + f.isFile());

        System.out.println("Affichage des lecteurs à la racine du PC : ");

        for(File file : f.listRoots())

        {
```

```

System.out.println(file.getAbsolutePath());

try {

    int i = 1;

    //On parcourt la liste des fichiers et répertoires

    for(File nom : file.listFiles()){

        //S'il s'agit d'un dossier, on ajoute un "/"

        System.out.print("\t\t" + ((nom.isDirectory()) ? nom.getName()+"/" : nom.getName()));

        if((i%4) == 0){

            System.out.print("\n");

        }

        i++;

    }

    System.out.println("\n");

} catch (NullPointerException e) {

    //L'instruction peut générer une NullPointerException

    //s'il n'y a pas de sous-fichier !

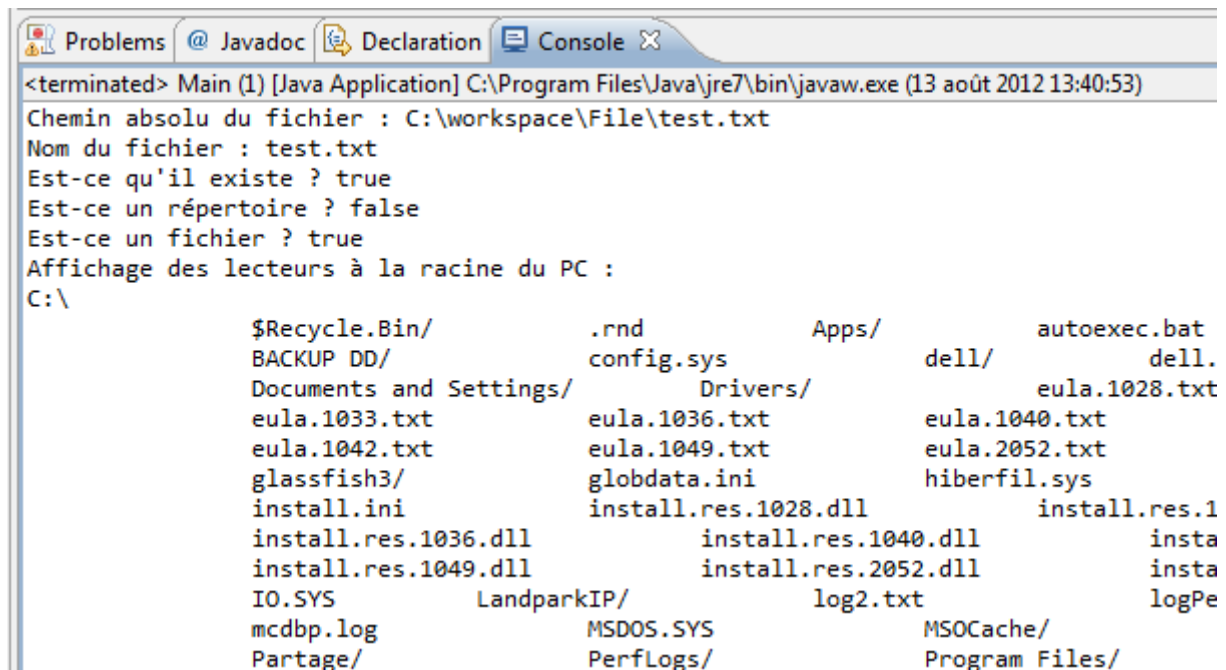
}

}

}

```

Le résultat est bluffant (voir figure suivante) !



```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (13 août 2012 13:40:53)
Chemin absolu du fichier : C:\workspace\File\test.txt
Nom du fichier : test.txt
Est-ce qu'il existe ? true
Est-ce un répertoire ? false
Est-ce un fichier ? true
Affichage des lecteurs à la racine du PC :
C:\
$Recycle.Bin/      .rnd      Apps/      autoexec.bat
BACKUP DD/      config.sys      dell/      dell.
Documents and Settings/      Drivers/      eula.1028.txt
eula.1033.txt      eula.1036.txt      eula.1040.txt
eula.1042.txt      eula.1049.txt      eula.2052.txt
glassfish3/      globdata.ini      hiberfil.sys
install.ini      install.res.1028.dll      install.res.1
install.res.1036.dll      install.res.1040.dll      insta
install.res.1049.dll      install.res.2052.dll      insta
IO.SYS      LandparkIP/      log2.txt      logPe
mcdbp.log      MSDOS.SYS      MSOCache/
Partage/      PerfLogs/      Program Files/
```

Test de l'objet File

Vous conviendrez que les méthodes de cet objet peuvent s'avérer très utiles ! Nous venons d'en essayer quelques-unes et nous avons même listé les sous-fichiers et sous-dossiers de nos lecteurs à la racine du PC.

Vous pouvez aussi effacer le fichier grâce la méthode `delete()`, créer des répertoires avec la méthode `mkdir()` (le nom donné à ce répertoire ne pourra cependant pas contenir de point (« . »)) etc.

Maintenant que vous en savez un peu plus sur cet objet, nous pouvons commencer à travailler avec notre fichier !

## Les objets `FileInputStream` et `FileOutputStream`

C'est par le biais des objets `FileInputStream` et `FileOutputStream` que nous allons pouvoir :

- lire dans un fichier ;
- écrire dans un fichier.

Ces classes héritent des classes abstraites `InputStream` et `OutputStream`, présentes dans le package `java.io`.

Comme vous l'avez sans doute deviné, il existe une hiérarchie de classes pour les traitements `in` et une autre pour les traitements `out`. Ne vous y trompez pas, les classes héritant d'`InputStream` sont destinées à la lecture et les classes héritant d'`OutputStream` se chargent de l'écriture !

Vous auriez dit le contraire ? Comme beaucoup de gens au début. Mais c'est uniquement parce que vous situez les flux par rapport à vous, et non à votre programme ! Lorsque ce dernier va lire des informations dans un fichier, ce sont des informations qu'il *reçoit*, et par conséquent, elles s'apparentent à une entrée `:in` (sachez tout de même que lorsque vous tapez au clavier, cette action est considérée comme un flux d'entrée !).

Au contraire, lorsqu'il va écrire dans un fichier (ou à l'écran, souvenez-vous de `System.out.println()`), par exemple, il va faire sortir des informations ; donc, pour lui, ce flux de données correspond à une sortie `:out`.

Nous allons enfin commencer à travailler avec notre fichier. Le but est d'aller en lire le contenu et de le copier dans un autre, dont nous spécifierons le nom dans notre programme, par le biais d'un programme Java.

Ce code est assez compliqué, donc accrochez-vous à vos claviers !

```
//Packages à importer afin d'utiliser les objets

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;
```

```
import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        // Nous déclarons nos objets en dehors du bloc try/catch

        FileInputStream fis = null;

        FileOutputStream fos = null;

        try {

            // On instancie nos objets :

            // fis va lire le fichier

            // fos va écrire dans le nouveau !

            fis = new FileInputStream(new File("test.txt"));

            fos = new FileOutputStream(new File("test2.txt"));

            // On crée un tableau de byte pour indiquer le nombre de bytes lus à

            // chaque tour de boucle

            byte[] buf = new byte[8];

            // On crée une variable de type int pour y affecter le résultat de

            // la lecture

            // Vaut -1 quand c'est fini

            int n = 0;

            // Tant que l'affectation dans la variable est possible, on boucle

            // Lorsque la lecture du fichier est terminée l'affectation n'est

            // plus possible !

            // On sort donc de la boucle

            while ((n = fis.read(buf)) >= 0) {

                // On écrit dans notre deuxième fichier avec l'objet adéquat

                fos.write(buf);

                // On affiche ce qu'a lu notre boucle au format byte et au

                // format char
```

```

        for (byte bit : buf) {

            System.out.print("\t" + bit + "(" + (char) bit + ")");

        }

        System.out.println("");

        //Nous réinitialisons le buffer à vide

        //au cas où les derniers byte lus ne soient pas un multiple de 8

        //Ceci permet d'avoir un buffer vierge à chaque lecture et ne pas avoir de doublon en fin de fichier

        buf = new byte[8];

    }

    System.out.println("Copie terminée !");

} catch (FileNotFoundException e) {

    // Cette exception est levée si l'objet FileInputStream ne trouve

    // aucun fichier

    e.printStackTrace();

} catch (IOException e) {

    // Celle-ci se produit lors d'une erreur d'écriture ou de lecture

    e.printStackTrace();

} finally {

    // On ferme nos flux de données dans un bloc finally pour s'assurer

    // que ces instructions seront exécutées dans tous les cas même si

    // une exception est levée !

    try {

        if (fis != null)

            fis.close();

    } catch (IOException e) {

        e.printStackTrace();

    }

}

try {

    if (fos != null)

        fos.close();

```

```

    } catch (IOException e) {

        e.printStackTrace();

    }

}

}

```

Pour que l'objet `FileInputStream` fonctionne, le fichier doit exister ! Sinon

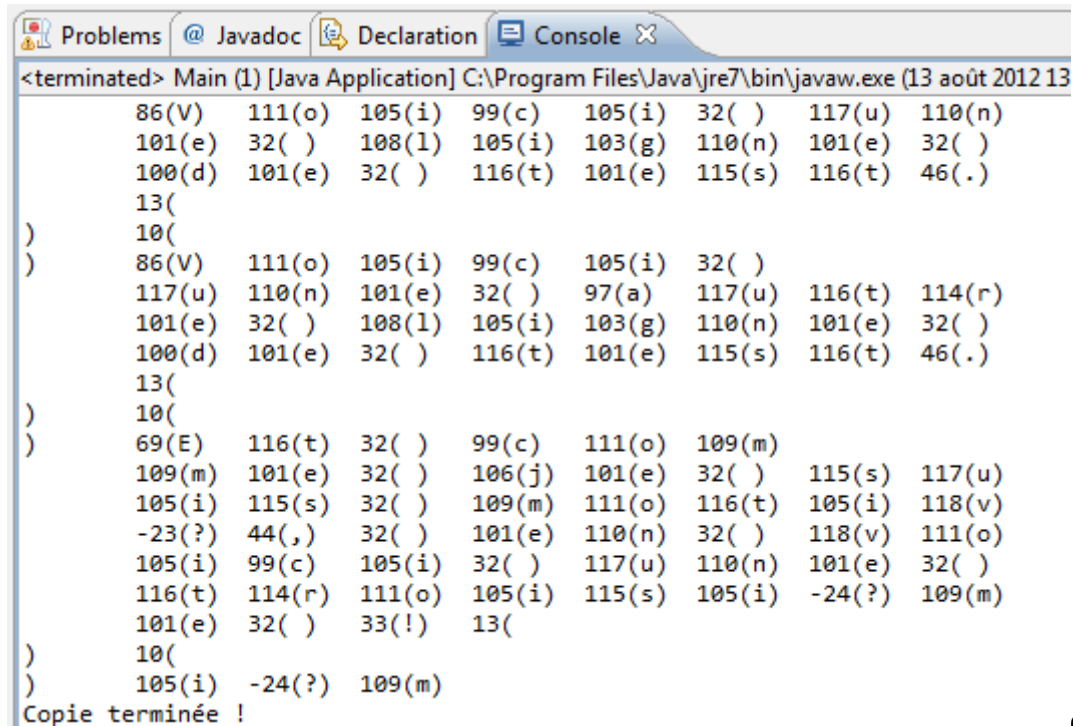
l'exception `FileNotFoundException` est levée. Par contre, si vous ouvrez un flux en écriture

(`FileOutputStream`) vers un fichier inexistant, celui-ci sera créé automatiquement !

Notez bien les imports pour pouvoir utiliser ces objets. Mais comme vous le savez déjà, vous pouvez taper votre code et faire ensuite **CTRL + SHIFT + O** pour que les imports soient automatiques.

À l'exécution de ce code, vous pouvez voir que le fichier `test2.txt` a bien été créé et qu'il contient exactement la même chose que `test.txt` ! De plus, j'ai ajouté dans la console les données que votre programme va utiliser (lecture et écriture).

La figure suivante représente le résultat de ce code.



fichier

Copie de

Le bloc `finally` permet de s'assurer que nos objets ont bien fermé leurs liens avec leurs fichiers respectifs, ceci afin de permettre à Java de détruire ces objets pour ainsi libérer un peu de mémoire à votre ordinateur.

En effet, les objets utilisent des ressources de votre ordinateur que Java ne peut pas libérer de lui-même, vous devez être sûr que la vanne est fermée ! Ainsi, même si une exception est levée, le contenu du bloc `finally` sera exécuté et nos ressources seront libérées. Par contre, pour alléger la lecture, **je ne mettrai plus ces blocs dans les codes à venir mais pensez bien à les mettre dans vos codes.**

Les objets `FileInputStream` et `FileOutputStream` sont assez rudimentaires, car ils travaillent avec un nombre déterminé d'octets à lire. Cela explique pourquoi ma condition de boucle était si tordue...

Lorsque vous voyez des caractères dans un fichier ou sur votre écran, ils ne veulent pas dire grand-chose pour votre PC, car il ne comprend que le binaire (vous savez, les suites de 0 et de 1). Ainsi, afin de pouvoir afficher et travailler avec des caractères, un système d'encodage (qui a d'ailleurs fort évolué) a été mis au point.

Sachez que chaque caractère que vous saisissez ou que vous lisez dans un fichier correspond à un code binaire, et ce code binaire correspond à un code décimal (voir la table de correspondance (table ASCII)) <http://www.table-ascii.com/>.

Cependant, au début, seuls les caractères de a à z, de A à Z et les chiffres de 0 à 9 (les 127 premiers caractères de la table ASCII) étaient codés (UNICODE 1), correspondant aux caractères se trouvant dans la langue anglaise. Mais ce

codage s'est rapidement avéré trop limité pour des langues comportant des caractères accentués (français, espagnol...). Un jeu de codage de caractères étendu a donc été mis en place afin de pallier ce problème.

Chaque code binaire UNICODE 1 est codé sur 8 bits, soit 1 octet. Une variable de type `byte`, en Java, correspond en fait à 1 octet et non à 1 bit !

Les objets que nous venons d'utiliser emploient la première version d'UNICODE 1 qui ne comprend pas les caractères accentués, c'est pourquoi ces caractères ont un code décimal négatif dans notre fichier. Lorsque nous définissons un tableau de `byte` à 8 entrées, cela signifie que nous allons lire 8 octets à la fois.

Vous pouvez voir qu'à chaque tour de boucle, notre tableau de `byte` contient huit valeurs correspondant chacune à un code décimal qui, lui, correspond à un caractère (valeur entre parenthèses à côté du code décimal).

Vous pouvez voir que les codes décimaux négatifs sont inconnus, car ils sont représentés par des « ? » ; de plus, il y a des caractères invisibles (les 32 premiers caractères de la table ASCII sont invisibles !) dans notre fichier :

- les espaces : `SP` pour « Space », code décimal 32 ;
- les sauts de lignes : `LF` pour « Line Feed », code décimal 13 ;
- les retours chariot : `CR` pour « Carriage Return », code décimal 10.

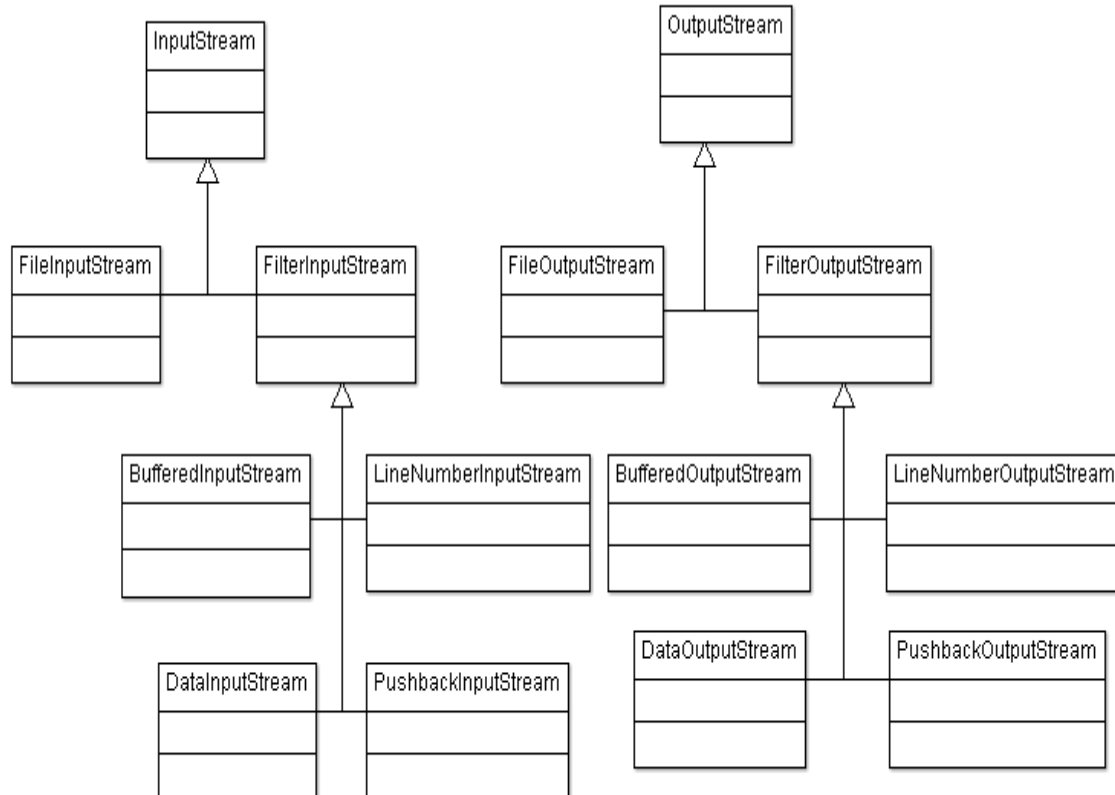
Vous voyez que les traitements des flux suivent une logique et une syntaxe précises ! Lorsque nous avons copié notre fichier, nous avons récupéré un certain nombre d'octets dans un flux entrant que nous avons passé à un flux sortant. À chaque tour de boucle, les données lues dans le fichier source sont écrites dans le fichier défini comme copie.

Il existe à présent des objets beaucoup plus faciles à utiliser, mais qui travaillent néanmoins avec les deux objets que nous venons d'étudier. Ces objets font également partie de la hiérarchie citée précédemment. Seulement, il existe une superclasse qui les définit.

## Les objets `FilterInputStream` et `FilterOutputStream`

Ces deux classes sont en fait des classes abstraites. Elles définissent un comportement global pour leurs classes filles qui, elles, permettent d'ajouter des fonctionnalités aux flux d'entrée/sortie !

La figure suivante représente un diagramme de classes schématisant leur hiérarchie.



Hiérarchie des classes du package `java.io`

Vous pouvez voir qu'il existe quatre classes filles héritant de `FilterInputStream` (de même pour `FilterOutputStream` (les classes dérivant de `FilterOutputStream` ont les mêmes fonctionnalités, mais en écriture)):

- `DataInputStream`: offre la possibilité de lire directement des types primitifs (`double`, `char`, `int`) grâce à des méthodes comme `readDouble()`, `readInt()` ...
- `BufferedInputStream`: cette classe permet d'avoir un tampon à disposition dans la lecture du flux. En gros, les données vont tout d'abord remplir le tampon, et dès que celui-ci est plein, le programme accède aux données.
- `PushbackInputStream`: permet de remettre un octet déjà lu dans le flux entrant.
- `LineNumberInputStream`: cette classe offre la possibilité de récupérer le numéro de la ligne lue à un instant T.

Ces classes prennent en paramètre une instance dérivant des classes `InputStream` (pour les classes héritant de `FilterInputStream`) ou de `OutputStream` (pour les classes héritant de `FilterOutputStream`).

Puisque ces classes acceptent une instance de leur superclasse en paramètre, vous pouvez cumuler les filtres et obtenir des choses de ce genre :

```
FileInputStream fis = new FileInputStream(new File("toto.txt"));

DataInputStream dis = new DataInputStream(fis);

BufferedInputStream bis = new BufferedInputStream(dis);

//Ou en condensé :

BufferedInputStream bis = new BufferedInputStream(

    new DataInputStream(

        new FileInputStream(

            new File("toto.txt"))));
```

Afin de vous rendre compte des améliorations apportées par ces classes, utilisez un fichier texte volumineux (plusieurs Mo). Faire passer ce fichier en lecture de façon conventionnelle avec l'objet vu précédemment, puis grâce à un buffer.

Récupérez le fichier compressé grâce à un logiciel de compression/décompression et remplacez le contenu de votre fichier `test.txt` par le contenu de ce fichier. Maintenant, voici un code qui permet de tester le temps d'exécution de la lecture :

```
//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.DataInputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        //Nous déclarons nos objets en dehors du bloc try/catch

        FileInputStream fis;
```



```

BufferedInputStream bis;

try {

    fis = new FileInputStream(new File("test.txt"));

    bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));

    byte[] buf = new byte[8];

    //On récupère le temps du système

    long startTime = System.currentTimeMillis();

    //Inutile d'effectuer des traitements dans notre boucle

    while(fis.read(buf) != -1);

    //On affiche le temps d'exécution

    System.out.println("Temps de lecture avec FileInputStream : " + (System.currentTimeMillis() - startTime));

    //On réinitialise

    startTime = System.currentTimeMillis();

    //Inutile d'effectuer des traitements dans notre boucle

    while(bis.read(buf) != -1);

    //On réaffiche

    System.out.println("Temps de lecture avec BufferedInputStream : " + System.currentTimeMillis() - startTime));

    //On ferme nos flux de données

    fis.close();

    bis.close();

} catch (FileNotFoundException e) {

    e.printStackTrace();

} catch (IOException e) {

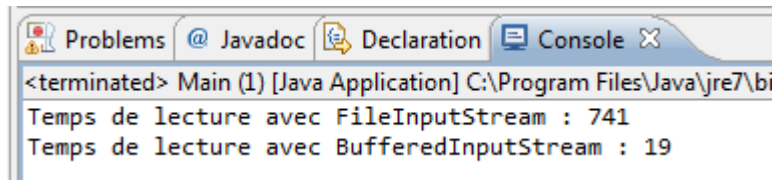
    e.printStackTrace();

}

}

```

Et le résultat, visible à la figure suivante, est encore une fois bluffant.



#### Comparatif de lecture avec et sans filtre

La différence de temps est vraiment énorme : 1,578 seconde pour la première méthode et 0,094 seconde pour la deuxième ! Vous conviendrez que l'utilisation d'un buffer permet une nette amélioration des performances de votre code. Faisons donc sans plus tarder le test avec l'écriture :

```
//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        //Nous déclarons nos objets en dehors du bloc try/catch

        FileInputStream fis;

        FileOutputStream fos;

        BufferedInputStream bis;

        BufferedOutputStream bos;

        try {

            fis = new FileInputStream(new File("test.txt"));

            fos = new FileOutputStream(new File("test2.txt"));

            bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));

            bos = new BufferedOutputStream(new FileOutputStream(new File("test3.txt")));

            byte[] buf = new byte[8];

            //On récupère le temps du système

            long startTime = System.currentTimeMillis();
```

```

while(fis.read(buf) != -1){

    fos.write(buf);

}

//On affiche le temps d'exécution

System.out.println("Temps de lecture + écriture avec FileInputStream et FileOutputStream : " +
(System.currentTimeMillis() - startTime));

//On réinitialise

startTime = System.currentTimeMillis();

while(bis.read(buf) != -1){

    bos.write(buf);

}

//On réaffiche

System.out.println("Temps de lecture + écriture avec BufferedInputStream et BufferedOutputStream : " +
(System.currentTimeMillis() - startTime));

//On ferme nos flux de données

fis.close();

bis.close();

fos.close();

bos.close();

} catch (FileNotFoundException e) {

    e.printStackTrace();

} catch (IOException e) {

    e.printStackTrace();

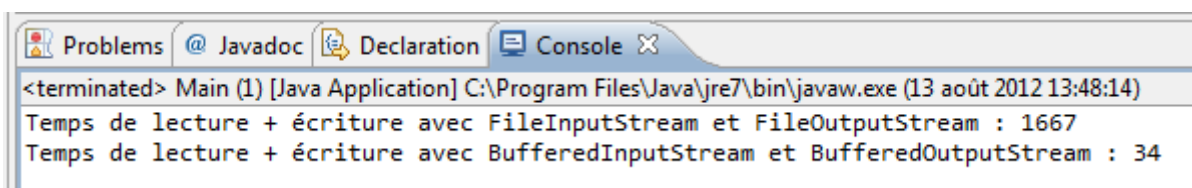
}

}

}

```

Là, la différence est encore plus nette, comme le montre la figure suivante.



Comparatif d'écriture avec et sans filtre

Si avec ça, vous n'êtes pas convaincus de l'utilité des buffers...

Je ne vais pas passer en revue tous les objets cités un peu plus haut, mais vu que vous risquez d'avoir besoin des objets `Data (Input/Output) Stream`, nous allons les aborder rapidement, puisqu'ils s'utilisent comme les objets `BufferedInputStream`. Je vous ai dit plus haut que ceux-ci ont des méthodes de lecture pour chaque type primitif : il faut cependant que le fichier soit généré par le biais d'un `DataOutputStream` pour que les méthodes fonctionnent correctement.

Nous allons donc créer un fichier de toutes pièces pour le lire par la suite.

```
//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        //Nous déclarons nos objets en dehors du bloc try/catch

        DataInputStream dis;

        DataOutputStream dos;

        try {

            dos = new DataOutputStream(

                new BufferedOutputStream(

                    new FileOutputStream(

                        new File("sdz.txt"))));

            //Nous allons écrire chaque type primitif

            dos.writeBoolean(true);

            dos.writeByte(100);

            dos.writeChar('C');

            dos.writeDouble(12.05);

            dos.writeFloat(100.52f);

            dos.writeInt(1024);
```

```

dos.writeLong(123456789654321L);

dos.writeShort(2);

dos.close();

//On récupère maintenant les données !

dis = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream(
            new File("sdz.txt"))));

System.out.println(dis.readBoolean());

System.out.println(dis.readByte());

System.out.println(dis.readChar());

System.out.println(dis.readDouble());

System.out.println(dis.readFloat());

System.out.println(dis.readInt());

System.out.println(dis.readLong());

System.out.println(dis.readShort());

} catch (FileNotFoundException e) {

    e.printStackTrace();

} catch (IOException e) {

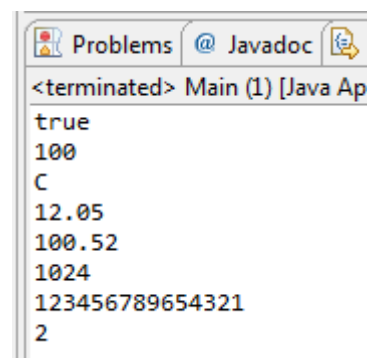
    e.printStackTrace();

}

}
}

```

La figure suivante correspond au résultat de ce code.



Test avec les DataInputStream — DataOutputStream

Le code est simple, clair et concis. Vous avez pu constater que ce type d'objet ne manque pas de fonctionnalités ! Jusqu'ici, nous ne travaillions qu'avec des types primitifs, mais il est également possible de travailler avec des objets !

## Les objets `ObjectInputStream` et `ObjectOutputStream`

Vous devez savoir que lorsqu'on veut écrire des objets dans des fichiers, on appelle ça la « sérialisation » : c'est le nom que porte l'action de sauvegarder des objets ! Cela fait quelque temps déjà que vous utilisez des objets et, j'en suis sûr, vous avez déjà souhaité que certains d'entre eux soient réutilisables. Le moment est venu de sauver vos objets d'une mort certaine ! Pour commencer, nous allons voir comment sérialiser un objet de notre composition.

Voici la classe avec laquelle nous allons travailler :

```
//Package à importer

import java.io.Serializable;

public class Game implements Serializable{

    private String nom, style;

    private double prix;

    public Game(String nom, String style, double prix) {

        this.nom = nom;

        this.style = style;

        this.prix = prix;

    }

    public String toString(){

        return "Nom du jeu : " + this.nom + "\n

        Style de jeu : " + this.style + "\n

        Prix du jeu : " + this.prix + "\n";

    }

}
```

Qu'est-ce que c'est que cette interface ? Tu n'as même pas implémenté de méthode !

En fait, cette interface n'a pas de méthode à redéfinir : l'interface `Serializable` est ce qu'on appelle une « interface marqueur ». Rien qu'en implémentant cette interface dans un objet, Java sait que cet objet peut être sérialisé. Et j'irai même plus loin : si vous n'implémentez pas cette interface dans vos objets, ceux-ci ne pourront pas être sérialisés ! En revanche, si une superclasse implémente l'interface `Serializable`, ses enfants seront considérés comme sérialisables.

Voici ce que nous allons faire :

- nous allons créer deux ou trois objets `Game`;
- nous allons les sérialiser dans un fichier de notre choix ;
- nous allons ensuite les désérialiser afin de pouvoir les réutiliser.

Vous avez sûrement déjà senti comment vous allez vous servir de ces objets, mais travaillons tout de même sur l'exemple que voici :

```
//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedReader;
```

```
import java.io.BufferedOutputStream;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

public class Main {

    public static void main(String[] args) {

        //Nous déclarons nos objets en dehors du bloc try/catch

        ObjectInputStream ois;

        ObjectOutputStream oos;

        try {

            oos = new ObjectOutputStream(

                new BufferedOutputStream(

                    new FileOutputStream(

                        new File("game.txt"))));

            //Nous allons écrire chaque objet Game dans le fichier

            oos.writeObject(new Game("Assassin Creed", "Aventure", 45.69));

            oos.writeObject(new Game("Tomb Raider", "Plateforme", 23.45));

            oos.writeObject(new Game("Tetris", "Stratégie", 2.50));

            //Ne pas oublier de fermer le flux !

            oos.close();

            //On récupère maintenant les données !

            ois = new ObjectInputStream(

                new BufferedInputStream(

                    new FileInputStream(
```

```

        new File("game.txt"))));

try {

    System.out.println("Affichage des jeux :");

    System.out.println("*****\n");

    System.out.println(((Game)ois.readObject()).toString());

    System.out.println(((Game)ois.readObject()).toString());

    System.out.println(((Game)ois.readObject()).toString());

} catch (ClassNotFoundException e) {

    e.printStackTrace();

}

ois.close();

} catch (FileNotFoundException e) {

    e.printStackTrace();

} catch (IOException e) {

    e.printStackTrace();

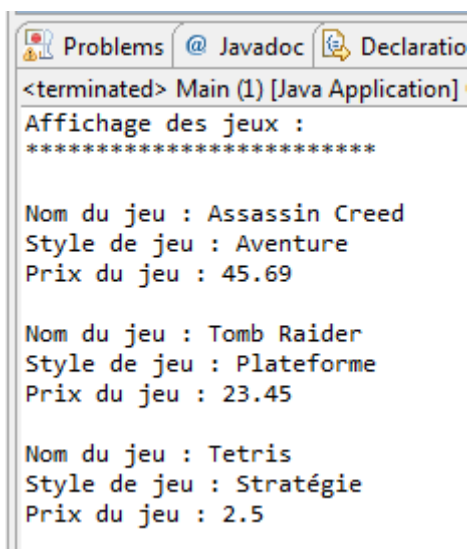
}

}

}

```

La désérialisation d'un objet peut engendrer une `Class Not FoundException`, pensez donc à la capturer !  
Et voyez le résultat en figure suivante.



Sérialisation — désérialisation



Ce qu'il se passe est simple : les données de vos objets sont enregistrées dans le fichier. Mais que se passerait-il si notre objet `Game` avait un autre objet de votre composition en son sein ? Voyons ça tout de suite. Créez la classe `Notice` comme suit :

```
public class Notice {

    private String langue ;

    public Notice(){

        this.langue = "Français";

    }

    public Notice(String lang){

        this.langue = lang;

    }

    public String toString() {

        return "\t Langue de la notice : " + this.langue + "\n";

    }

}
```

Nous allons maintenant implémenter une notice par défaut dans notre objet `Game`. Voici notre classe modifiée :

```
import java.io.Serializable;

public class Game implements Serializable{

    private String nom, style;

    private double prix;

    private Notice notice;

    public Game(String nom, String style, double prix) {

        this.nom = nom;

        this.style = style;

        this.prix = prix;

        this.notice = new Notice();

    }

    public String toString(){

        return "Nom du jeu : " + this.nom + "\n

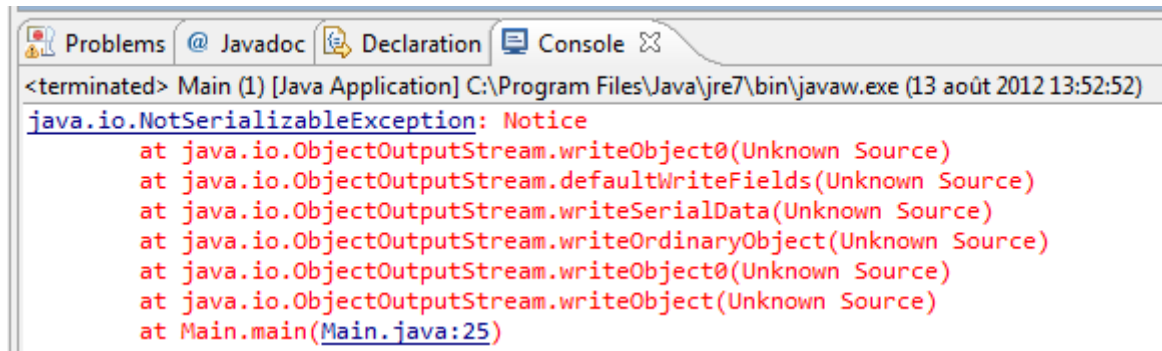
                Style de jeu : " + this.style + "\n

                Prix du jeu : " + this.prix + "\n";

    }

}
```

Réessayez votre code sauvegardant vos objets Game. La figure suivante nous montre le résultat obtenu.



Erreur de sérialisation

Eh non, votre code ne compile plus ! Il y a une bonne raison à cela : votre objet `Notice` n'est pas sérialisable, une erreur de compilation est donc levée. Maintenant, deux choix s'offrent à vous :

1. soit vous faites en sorte de rendre votre objet sérialisable ;
2. soit vous spécifiez dans votre classe `Game` que la variable `notice` n'a pas à être sérialisée.

Pour la première option, c'est simple, il suffit d'implémenter l'interface sérialisable dans notre classe `Notice`. Pour la seconde, il suffit de déclarer votre variable : `transient`; comme ceci :

```
import java.io.Serializable;

public class Game implements Serializable{

    private String nom, style;

    private double prix;

    //Maintenant, cette variable ne sera pas sérialisée

    //Elle sera tout bonnement ignorée !

    private transient Notice notice;

    public Game(String nom, String style, double prix) {

        this.nom = nom;

        this.style = style;

        this.prix = prix;

        this.notice = new Notice();

    }

    public String toString(){

        return "Nom du jeu : " + this.nom + "\n

        Style de jeu : " + this.style + "\n

        Prix du jeu : " + this.prix + "\n";

    }

}
```

```
}  
}
```

Vous aurez sans doute remarqué que nous n'utilisons pas la variable `notice` dans la méthode `toString()` de notre objet `Game`. Si vous faites ceci, que vous sérialisez puis désérialisez vos objets, la machine virtuelle vous renverra l'exception `NullPointerException` à l'invocation de ladite méthode. Eh oui ! L'objet `Noticee` est ignoré : il n'existe donc pas !

## Les objets `CharArray(Writer/Reader)` et `String(Writer/Reader)`

Nous allons utiliser des objets :

- `CharArray(Writer/Reader);`
- `String(Writer/Reader).`

Ces deux types jouent quasiment le même rôle. De plus, ils ont les mêmes méthodes que leur classe mère. Ces deux objets n'ajoutent donc aucune nouvelle fonctionnalité à leur objet mère.

Leur principale fonction est de permettre d'écrire un flux de caractères dans un buffer adaptatif : un emplacement en mémoire qui peut changer de taille selon les besoins (nous n'en avons pas parlé dans le chapitre précédent afin de ne pas l'alourdir, mais il existe des classes remplissant le même rôle que ces classes-ci : `ByteArray(Input/Output)Stream`).

Commençons par un exemple commenté des objets `CharArray(Writer/Reader)` :

```
//Packages à importer afin d'utiliser l'objet File  
  
import java.io.CharArrayReader;  
  
import java.io.CharArrayWriter;  
  
import java.io.IOException;  
  
public class Main {  
  
    public static void main(String[] args) {  
  
        CharArrayWriter caw = new CharArrayWriter();  
  
        CharArrayReader car;  
  
        try {  
  
            caw.write("Coucou les Zéros");  
  
            //Appel à la méthode toString de notre objet de manière tacite  
  
            System.out.println(caw);  
  
            //caw.close() n'a aucun effet sur le flux  
  
            //Seul caw.reset() peut tout effacer  
  
            caw.close();  
  
            //On passe un tableau de caractères à l'objet qui va lire le tampon  
  
            car = new CharArrayReader(caw.toCharArray());  

```

```

    int i;

    //On remet tous les caractères lus dans un String

    String str = "";

    while(( i = car.read()) != -1)

        str += (char) i;

    System.out.println(str);

} catch (IOException e) {

    e.printStackTrace();

}

}

}

```

Je vous laisse le soin d'examiner ce code ainsi que son effet. Il est assez commenté pour que vous en compreniez toutes les subtilités. L'objet `String` (`Writer/Reader`) fonctionne de la même façon :

```

//Packages à importer afin d'utiliser l'objet File

import java.io.IOException;

import java.io.StringReader;

import java.io.StringWriter;

public class Main {

    public static void main(String[] args) {

        StringWriter sw = new StringWriter();

        StringReader sr;

        try {

            sw.write("Coucou les Zéros");

            //Appel à la méthode toString de notre objet de manière tacite

            System.out.println(sw);

            //caw.close() n'a aucun effet sur le flux

            //Seul caw.reset() peut tout effacer

            sw.close();

```

```

//On passe un tableau de caractères à l'objet qui va lire le tampon

sr = new StringReader(sw.toString());

int i ;

//On remet tous les caractères lus dans un String

String str = "";

while(( i = sr.read()) != -1)

    str += (char) i;

System.out.println(str);

} catch (IOException e) {

    e.printStackTrace();

}

}

}

```

En fait, il s'agit du même code, mais avec des objets différents ! Vous savez à présent comment écrire un flux de texte dans un tampon de mémoire. Je vous propose maintenant de voir comment traiter les fichiers de texte avec des flux de caractères.

## Les classes `File` (`Writer/Reader`) et `Print` (`Writer/Reader`)

Comme nous l'avons vu, les objets travaillant avec des flux utilisent des flux binaires.

La conséquence est que même si vous ne mettez que des caractères dans un fichier et que vous le sauvegardez, les objets étudiés précédemment traiteront votre fichier de la même façon que s'il contenait des données binaires ! Ces deux objets, présents dans le package `java.io`, servent à lire et écrire des données dans un fichier texte.

```

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        File file = new File("testFileWriter.txt");

        FileWriter fw;

        FileReader fr;

        try {

            //Création de l'objet

```

```

    fw = new FileWriter(file);

    String str = "Bonjour à tous, amis Zéros !\n";

    str += "\tComment allez-vous ? \n";

    //On écrit la chaîne

    fw.write(str);

    //On ferme le flux

    fw.close();

    //Création de l'objet de lecture

    fr = new FileReader(file);

    str = "";

    int i = 0;

    //Lecture des données

    while((i = fr.read()) != -1)

        str += (char)i;

    //Affichage

    System.out.println(str);

} catch (FileNotFoundException e) {

    e.printStackTrace();

} catch (IOException e) {

    e.printStackTrace();

}

}
}

```

Vous pouvez voir que l'affichage est bon et qu'un nouveau fichier (la lecture d'un fichier inexistant entraîne l'exception `FileNotFoundException`, et l'écriture peut entraîner une `IOException`) vient de faire son apparition dans le dossier contenant votre projet Eclipse !

Depuis le JDK 1.4, un nouveau package a vu le jour, visant à améliorer les performances des flux, buffers, etc. traités par `java.io`. En effet, vous ignorez probablement que le package que nous explorons depuis le début existe depuis la version 1.1 du JDK. Il était temps d'avoir une remise à niveau afin d'améliorer les résultats obtenus avec les objets traitant les flux. C'est là que le package `java.nio` a vu le jour !

## Utilisation de `java.nio`

Vous l'avez sûrement deviné, `nio` signifie « New I/O ». Comme je vous l'ai dit précédemment, ce package a été créé afin d'améliorer les performances sur le traitement des fichiers, du réseau et des buffers. Il permet de lire les données (nous nous intéresserons uniquement à l'aspect fichier) d'une façon différente. Vous avez constaté que les objets du

package `java.io` traitaient les données par octets. Les objets du package `java.nio`, eux, les traitent par blocs de données : la lecture est donc accélérée !

Tout repose sur deux objets de ce nouveau package : les *channels* et les *buffers*. Les channels sont en fait des flux, tout comme dans l'ancien package, mais ils sont amenés à travailler avec un buffer dont vous définissez la taille. Pour simplifier au maximum, lorsque vous ouvrez un flux vers un fichier avec un objet `FileInputStream`, vous pouvez récupérer un canal vers ce fichier. Celui-ci, combiné à un buffer, vous permettra de lire votre fichier encore plus vite qu'avec un `BufferedInputStream`!

Reprenez le gros fichier que je vous ai fait créer dans la sous-section précédente : nous allons maintenant le relire avec ce nouveau package en comparant le buffer conventionnel et la nouvelle façon de faire.

```
//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.channels.FileChannel;

public class Main {

    public static void main(String[] args) {

        FileInputStream fis;

        BufferedInputStream bis;

        FileChannel fc;

        try {

            //Création des objets

            fis = new FileInputStream(new File("test.txt"));

            bis = new BufferedInputStream(fis);

            //Démarrage du chrono

            long time = System.currentTimeMillis();

            //Lecture

            while(bis.read() != -1);

            //Temps d'exécution

            System.out.println("Temps d'exécution avec un buffer conventionnel : " + (System.currentTimeMillis() - time));

            //Création d'un nouveau flux de fichier
```

```

    fis = new FileInputStream(new File("test.txt"));

    //On récupère le canal

    fc = fis.getChannel();

    //On en déduit la taille

    int size = (int)fc.size();

    //On crée un buffer correspondant à la taille du fichier

    ByteBuffer bBuff = ByteBuffer.allocate(size);

    //Démarrage du chrono

    time = System.currentTimeMillis();

    //Démarrage de la lecture

    fc.read(bBuff);

    //On prépare à la lecture avec l'appel à flip

    bBuff.flip();

    //Affichage du temps d'exécution

    System.out.println("Temps d'exécution avec un nouveau buffer : " + (System.currentTimeMillis() - time));

    //Puisque nous avons utilisé un buffer de byte afin de récupérer les données

    //Nous pouvons utiliser un tableau de byte

    //La méthode array retourne un tableau de byte

    byte[] tabByte = bBuff.array();

} catch (FileNotFoundException e) {

    e.printStackTrace();

} catch (IOException e) {

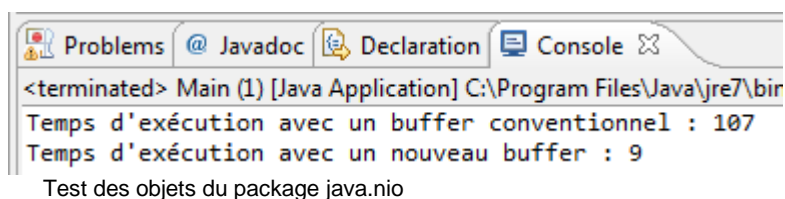
    e.printStackTrace();

}

}
}

```

La figure suivante vous montre le résultat.



Vous constatez que les gains en performance ne sont pas négligeables. Sachez aussi que ce nouveau package est le plus souvent utilisé pour traiter les flux circulant sur les réseaux. Je ne m'attarderai pas sur le sujet, mais une petite



présentation est de mise. Ce package offre un buffer par type primitif pour la lecture sur le channel, vous trouverez donc ces classes :

- `IntBuffer`;
- `CharBuffer`;
- `ShortBuffer`;
- `ByteBuffer`;
- `DoubleBuffer`;
- `FloatBuffer`;
- `LongBuffer`.

Je ne l'ai pas fait durant tout le chapitre afin d'alléger un peu les codes, mais si vous voulez être sûrs que votre flux est bien fermé, utilisez la clause `finally`. Par exemple, faites comme ceci :

```
//Packages à importer afin d'utiliser l'objet File
//...

public class Main {

    public static void main(String[] args) {

        //Nous déclarons nos objets en dehors du bloc try / catch

        ObjectInputStream ois;

        ObjectOutputStream oos;

        try {

            //On travaille avec nos objets

        } catch (FileNotFoundException e) {

            //Gestion des exceptions

        } catch (IOException e) {

            //Gestion des exceptions

        }

        finally{

            if(ois != null)ois.close();

            if(oos != null)oos.close();

        }

    }

}
```

Avec l'arrivée de Java 7, quelques nouveautés ont vu le jour pour la gestion des exceptions sur les flux. Contrairement à la gestion de la mémoire (vos variables, vos classes, etc.) qui est déléguée au *garbage collector* (ramasse miette), plusieurs types de ressources doivent être gérées manuellement. Les flux sur des fichiers en font parti mais, d'un point de vue plus général, toutes les ressources que vous devez fermer manuellement (les flux réseaux, les connexions à une base de données...). Pour ce genre de flux, vous avez vu qu'il vous faut déclarer une variable en dehors d'un bloc `try{...} catch{...}` afin qu'elle soit accessible dans les autres blocs d'instructions, le bloc `finally` par exemple.

Java 7 initie ce qu'on appelle vulgairement le « *try-with-resources* ». Ceci vous permet de déclarer les ressources utilisées directement dans le bloc `try (...)`, ces dernières seront automatiquement fermées à la fin du bloc d'instructions ! Ainsi, si nous reprenons notre code de début de chapitre qui copie notre fichier `test.txt` vers `test2.txt`, nous aurons ceci :

```
try(FileInputStream fis = new FileInputStream("test.txt");
    FileOutputStream fos = new FileOutputStream("test2.txt")) {

    byte[] buf = new byte[8];

    int n = 0;

    while((n = fis.read(buf)) >= 0){

        fos.write(buf);

        for(byte bit : buf)

            System.out.print("\t" + bit + "(" + (char)bit + ")");

        System.out.println("");

    }

    System.out.println("Copie terminée !");

} catch (IOException e) {

    e.printStackTrace();

}
```

Notez bien que les différentes ressources utilisées sont séparées par un « ; » dans le bloc `try`!

C'est tout de même beaucoup plus clair et plus lisible qu'avant, surtout que vous n'avez plus à vous soucier de la fermeture dans le bloc `finally`. Il faut cependant prendre quelques précautions notamment pour ce genre de déclaration :

```
try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.txt"))) {

    //...

}
```

Le fait d'avoir des ressources encapsulées dans d'autres ne rend pas « visible » les ressources encapsulées. Dans le cas précédent, si une exception est levée, le flux correspondant à l'objet `FileInputStream` ne sera pas fermé. Pour pallier ce problème il suffit de bien découper toutes les ressources à utiliser, comme ceci :

```
try (FileInputStream fis = new FileInputStream("test.txt");

    ObjectInputStream ois = new ObjectInputStream(fis)) {

    //...

}
```

Eh ! Avant tu utilisais l'objet `File` dans l'instanciation de tes objets `FileInputStream` et `FileOutputStream`!

Rien ne vous échappe ! Si j'ai changé de façon de faire c'est parce qu'il y a une restriction sur ce mode de fonctionnement. Pour rendre la fermeture automatique possible, les développeurs de la plateforme Java 7 ont créé une nouvelle interface : `java.lang.AutoCloseable`. Seuls les objets implémentant cette interface peuvent être utilisés de la sorte ! Vous pouvez voir la liste des classes autorisées [à cette adresse](#) (et vous constaterez que la classe `File` n'en fait pas parti).

## Depuis Java 7 : nio II

L'une des grandes nouveautés de Java 7 réside dans NIO.2 avec un nouveau package `java.nio.file` en remplacement de la classe `java.io.File`. Voici un bref listing de quelques nouveautés :

- une meilleure gestion des exceptions : la plupart des méthodes de la classe `File` se contentent de renvoyer une valeur nulle en cas de problème, avec ce nouveau package, des exceptions seront levées permettant de mieux cibler la cause du (ou des) problème(s) ;
- un accès complet au système de fichiers (support des liens/liens symboliques, etc.) ;
- l'ajout de méthodes utilitaires tels que le déplacement/la copie de fichier, la lecture/écriture binaire ou texte...
- récupérer la liste des fichiers d'un répertoire via un flux ;
- remplacement de la classe `java.io.File` par l'interface `java.nio.file.Path`.

Je vous propose maintenant de jouer avec quelques nouveautés. Commençons par le commencement : ce qui finira par remplacer la classe `File`. Afin d'être le plus souple et complet possible, les développeurs de la plateforme ont créé une interface `java.nio.file.Path` dont le rôle est de récupérer et manipuler des chemins de fichiers de dossier et une classe `java.nio.file.Files` qui contient tout un tas de méthodes qui simplifient certaines actions (copie, déplacement, etc.) et permet aussi de récupérer tout un tas d'informations sur un chemin.

Afin d'illustrer ce nouveau mode de fonctionnement, je vous propose de reprendre le premier exemple de ce chapitre, celui qui affichait différentes informations sur notre fichier de test.

```
Path path = Paths.get("test.txt");

System.out.println("Chemin absolu du fichier : " + path.toAbsolutePath());

System.out.println("Est-ce qu'il existe ? " + Files.exists(path));

System.out.println("Nom du fichier : " + path.getFileName());

System.out.println("Est-ce un répertoire ? " + Files.isDirectory(path));
```

La classe `Files` vous permet aussi de lister le contenu d'un répertoire mais *via* un objet `DirectoryStream` qui est un itérateur. Ceci évite de charger tous les fichiers en mémoire pour récupérer leurs informations. Voici comment procéder :

```
//On récupère maintenant la liste des répertoires dans une collection typée

//Via l'objet FileSystem qui représente le système de fichier de l'OS hébergeant la JVM

Iterable<Path> roots = FileSystems.getDefault().getRootDirectories();

//Maintenant, il ne nous reste plus qu'à parcourir

for(Path chemin : roots){

    System.out.println(chemin);

    //Pour lister un répertoire, il faut utiliser l'objet DirectoryStream

    //L'objet Files permet de créer ce type d'objet afin de pouvoir l'utiliser

    try(DirectoryStream<Path> listing = Files.newDirectoryStream(chemin)){

        int i = 0;

        for(Path nom : listing){

            System.out.print("\t\t" + ((Files.isDirectory(nom)) ? nom+"/" : nom));

            i++;

            if(i%4 == 0)System.out.println("\n");

        }

    }

}
```

```

    }

    } catch (IOException e) {

        e.printStackTrace();

    }

}

```

Vous avez également la possibilité d'ajouter un filtre à votre listing de répertoire afin qu'il ne liste que certains fichiers :

```

try(DirectoryStream<Path> listing = Files.newDirectoryStream(chemin, "*.txt")){ ... }

//Ne prendra en compte que les fichier ayant l'extension .txt

```

C'est vrai que cela change grandement la façon de faire et elle peut paraître plus complexe. Mais l'objet `Files` simplifie aussi beaucoup de choses. Voici quelques exemple de méthodes utilitaires qui, je pense, vont vous séduire.

## La copie de fichier

Pour copier le fichier `test.txt` vers un fichier `test2.txt`, il suffit de faire :

```

Path source = Paths.get("test.txt");

Path cible = Paths.get("test2.txt");

try {

    Files.copy(source, cible, StandardCopyOption.REPLACE_EXISTING);

} catch (IOException e) { e.printStackTrace(); }

```

Le troisième argument permet de spécifier les options de copie. Voici celles qui sont disponibles :

- `StandardCopyOption.REPLACE_EXISTING`: remplace le fichier cible même s'il existe déjà ;
- `StandardCopyOption.COPY_ATTRIBUTES`: copie les attributs du fichier source sur le fichier cible (droits en lecture etc.) ;
- `StandardCopyOption.ATOMIC_MOVE`: copie atomique ;
- `LinkOption.NOFOLLOW_LINKS`: ne prendra pas en compte les liens.

## Le déplacement de fichier

Pour déplacer le fichier `test2.txt` vers un fichier `test3.txt`, il suffit de faire :

```

Path source = Paths.get("test2.txt");

Path cible = Paths.get("test3.txt");

try {

    Files.move(source, cible, StandardCopyOption.REPLACE_EXISTING);

} catch (IOException e) { e.printStackTrace(); }

```

Dans le même genre vous avez aussi :

- une méthode `Files.delete(path)` qui supprime un fichier ;
- une méthode `Files.createFile(path)` qui permet de créer un fichier vide.

## Ouvrir des flux

Ceci est très pratique pour lire ou écrire dans un fichier. Voici comment ça se traduit :

```

Path source = Paths.get("test.txt");

//Ouverture en lecture :

try ( InputStream input = Files.newInputStream(source) ) { ... }

```

```

//Ouverture en écriture :

try ( OutputStream output = Files.newOutputStream(source) ) { ... }

//Ouverture d'un Reader en lecture :

try ( BufferedReader reader = Files.newBufferedReader(source, StandardCharsets.UTF_8) ) { ... }

//Ouverture d'un Writer en écriture :

try ( BufferedWriter writer = Files.newBufferedWriter(source, StandardCharsets.UTF_8) ) { ... }

```

Java 7 vous permet également de gérer les fichier ZIP grâce à l'objet `FileSystem`:

```

// Création d'un système de fichiers en fonction d'un fichier ZIP

try (FileSystem zipFS = FileSystems.newFileSystem(Paths.get("monFichier.zip"), null)) {

    //Suppression d'un fichier à l'intérieur du ZIP :

    Files.deleteIfExists( zipFS.getPath("test.txt") );

    //Création d'un fichier à l'intérieur du ZIP :

    Path path = zipFS.getPath("nouveau.txt");

    String message = "Hello World !!!";

    Files.write(path, message.getBytes());

    //Parcours des éléments à l'intérieur du ZIP :

    try (DirectoryStream<Path> stream = Files.newDirectoryStream(zipFS.getPath("/"))) {

        for (Path entry : stream) {

            System.out.println(entry);

        }

    }

    //Copie d'un fichier du disque vers l'archive ZIP :

    Files.copy(Paths.get("fichierSurDisque.txt"), zipFS.getPath("fichierDansZIP.txt"));

}

```

Il est également possible d'être averti via l'objet `WatchService` lorsqu'un fichier est modifié, de gérer des entrées/sorties asynchrones via les objets `AsynchronousFileChannel`, `AsynchronousSocketChannel` ou `AsynchronousServerSocketChannel`. Ceci permet de faire les actions en tâche de fond, sans bloquer le

code pendant l'exécution. Il est aussi possible d'avoir accès aux attributs grâce à 6 vues permettant de voir plus ou moins d'informations, à savoir :

- `BasicFileAttributeView` permet un accès aux propriétés généralement communes à tous les systèmes de fichiers ;
- `DosFileAttributeView` ajoute le support des attributs MS-DOS (`readonly`, `hidden`, `system`, `archive`) à l'objet ci-dessus ;
- `PosixFileAttributeView` ajoute les permissions POSIX du monde Unix au premier objet cité ;
- `FileOwnerAttributeView` permet de manipuler le propriétaire du fichier ;
- `AclFileAttributeView` permet de manipuler les droits d'accès au fichier ;
- `UserDefinedFileAttributeView` : permet de définir des attributs personnalisés.

## Le pattern decorator

Vous avez pu remarquer que les objets de ce chapitre utilisent des instances d'objets de même supertype dans leur constructeur. Rappelez-vous cette syntaxe :

```
DataInputStream dis = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream(  
            new File("sdz.txt"))));
```

La raison d'agir de la sorte est simple : c'est pour ajouter de façon dynamique des fonctionnalités à un objet. En fait, dites-vous qu'au moment de récupérer les données de notre objet `DataInputStream`, celles-ci vont d'abord transiter par les objets passés en paramètre. Ce mode de fonctionnement suit une certaine structure et une certaine hiérarchie de classes : c'est le *pattern decorator*.

Ce pattern de conception permet d'ajouter des fonctionnalités à un objet sans avoir à modifier son code source. Afin de ne pas trop vous embrouiller avec les objets étudiés dans ce chapitre, je vais vous fournir un autre exemple, plus simple, mais gardez bien en tête que les objets du package `java.io` utilisent ce pattern. Le but du jeu est d'obtenir un objet auquel nous pourrions ajouter des choses afin de le « décorer »... Vous allez travailler avec un objet `Gateau` qui héritera d'une classe abstraite `Patisserie`. Le but du jeu est de pouvoir ajouter des couches à notre gâteau sans avoir à modifier son code source.

Vous avez vu avec le pattern strategy que la composition (« A un ») est souvent préférable à l'héritage (« Est un ») : vous aviez défini de nouveaux comportements pour vos objets en créant un supertype d'objet par comportement. Ce pattern aussi utilise la composition comme principe de base : vous allez voir que nos objets seront composés d'autres objets. La différence réside dans le fait que nos nouvelles fonctionnalités ne seront pas obtenues uniquement en créant de nouveaux objets, mais en associant ceux-ci à des objets existants. Ce sera cette association qui créera de nouvelles fonctionnalités !

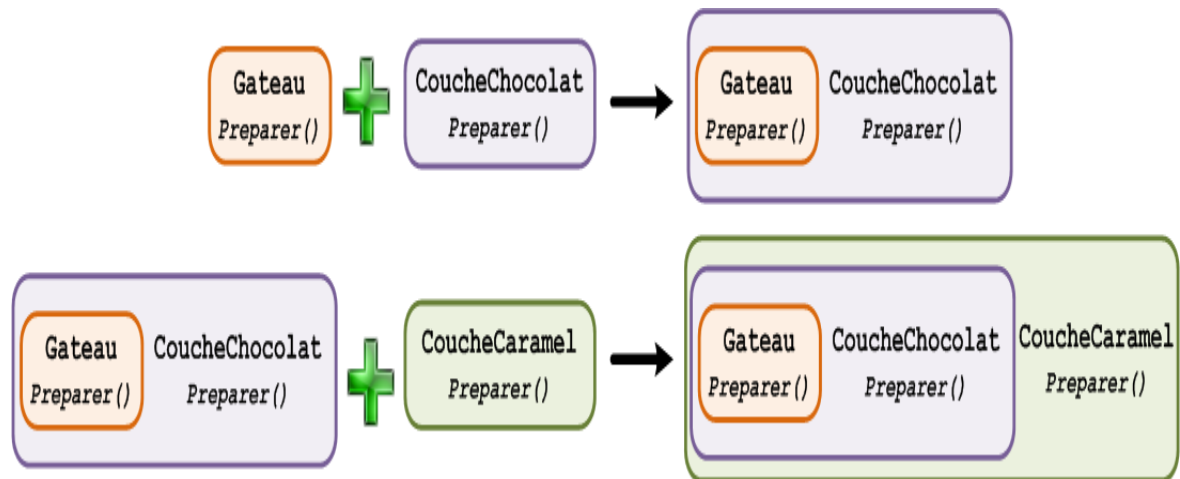
Nous allons procéder de la façon suivante :

- nous allons créer un objet `Gateau`;
- nous allons lui ajouter une `Couche Chocolat`;
- nous allons aussi lui ajouter une `Couche Caramel`;
- nous appellerons la méthode qui confectionnera notre gâteau.

Tout cela démarre avec un concept fondamental : l'objet de base et les objets qui le décorent *doivent* être du même type, et ce, toujours pour la même raison, le polymorphisme, le polymorphisme, et le polymorphisme !

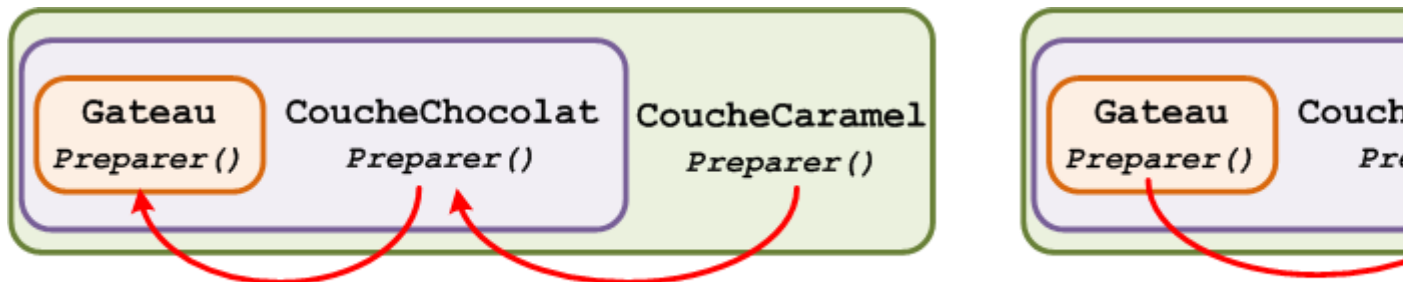
Vous allez comprendre. En fait, les objets qui vont décorer notre gâteau posséderont la même méthode `preparer()` que notre objet principal, et nous allons faire fondre cet objet dans les autres. Cela signifie que nos objets qui vont servir de décorateurs comporteront une instance de type `Patisserie`; ils vont englober les instances les uns après les autres et du coup, nous pourrions appeler la méthode `preparer()` de manière récursive !

Vous pouvez voir les décorateurs comme des poupées russes : il est possible de mettre une poupée dans une autre. Cela signifie que si nous décorons notre `gateau` avec un objet `CoucheChocolat` et un objet `CoucheCaramel`, la situation pourrait être symbolisée par la figure suivante.



Encapsulation des objets

L'objet `CoucheCaramel` contient l'instance de la classe `CoucheChocolat` qui, elle, contient l'instance de `Gateau`: en fait, on va passer notre instance d'objet en objet ! Nous allons ajouter les fonctionnalités des objets « décorants » en appelant la méthode `preparer()` de l'instance se trouvant dans l'objet avant d'effectuer les traitements de la même méthode de l'objet courant, comme à la figure suivante.



## Etape 1

Invocation des méthodes

Nous verrons, lorsque nous parlerons de la classe `Thread`, que ce système ressemble fortement à la pile d'invocations de méthodes. La figure suivante montre à quoi ressemble le diagramme de classes de notre exemple.

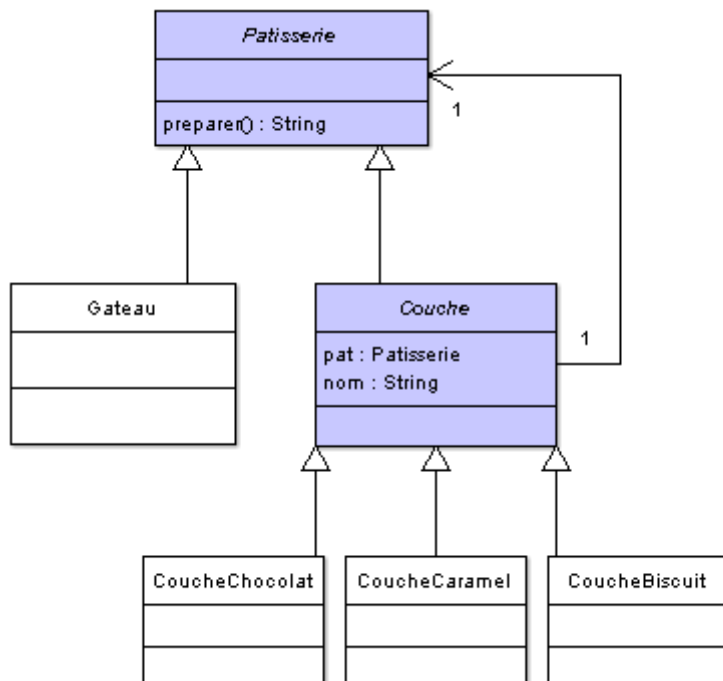


Diagramme de classes

Vous remarquez sur ce diagramme que notre classe mère `Patisserie` est en fait la *strategy* (une classe encapsulant un comportement fait référence au pattern strategy : on peut dire qu'elle est la *strategy* de notre hiérarchie) de notre structure, c'est pour cela que nous pourrions appeler la méthode `preparer()` de façon récursive afin d'ajouter des fonctionnalités à nos objets. Voici les différentes classes que j'ai utilisées (je n'ai utilisé que des `String` afin de ne pas surcharger les sources, et pour que vous vous focalisiez plus sur la logique que sur le code).

## Patisserie.java

```
public abstract class Patisserie {  
  
    public abstract String preparer();  
  
}
```

## Gateau.java

```
public class Gateau extends Patisserie{  
  
    public String preparer() {  
  
        return "Je suis un gâteau et je suis constitué des éléments suivants. \n";  
  
    }  
  
}
```

## Couche.java

```
public abstract class Couche extends Patisserie{  
  
    protected Patisserie pat;  
  
    protected String nom;  
  
    public Couche(Patisserie p){  
  
        pat = p;  
  
    }  
  
    public String preparer() {  
  
        String str = pat.preparer();  
  
        return str + nom;  
  
    }  
  
}
```

## CoucheChocolat.java

```
public class CoucheChocolat extends Couche{  
  
    public CoucheChocolat(Patisserie p) {  
  
        super(p);  
  
        this.nom = "\t- Une couche de chocolat.\n";  
  
    }  
  
}
```

## CoucheCaramel.java



```

public class CoucheCaramel extends Couche{

    public CoucheCaramel(Patisserie p) {

        super(p);

        this.nom = "\t- Une couche de caramel.\n";

    }

}

```

### CoucheBiscuit.java

```

public class CoucheBiscuit extends Couche {

    public CoucheBiscuit(Patisserie p) {

        super(p);

        this.nom = "\t- Une couche de biscuit.\n";

    }

}

```

Et voici un code de test ainsi que son résultat, représenté à la figure suivante.

```

public class Main{

    public static void main(String[] args){

        Patisserie pat = new CoucheChocolat(

            new CoucheCaramel(

                new CoucheBiscuit(

                    new CoucheChocolat(

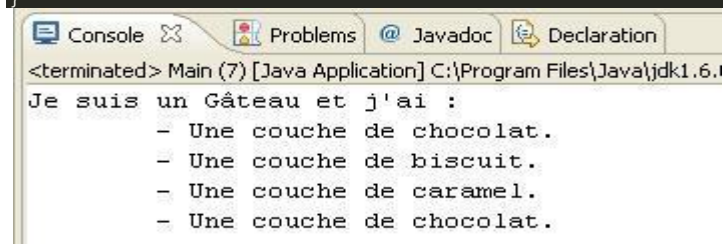
                        new Gateau()))));

        System.out.println(pat.preparer());

    }

}

```



Résultat du test

J'ai agrémenté l'exemple d'une couche de biscuit, mais je pense que tout cela est assez représentatif de la façon dont fonctionnent des flux d'entrée/sortie en Java. Vous devriez réussir à saisir tout cela sans souci. Le fait est que vous commencez maintenant à avoir en main des outils intéressants pour programmer, et c'est sans compter les outils du langage : vous venez de mettre votre deuxième pattern de conception dans votre mallette du programmeur.

Vous avez pu voir que l'invocation des méthodes se faisait en allant jusqu'au dernier élément pour remonter ensuite la pile d'invocations. Pour inverser ce fonctionnement, il vous suffit d'inverser les appels dans la méthode `preparer()` : affecter d'abord le nom de la couche et ensuite le nom du décorateur.

- Les classes traitant des entrées/sorties se trouvent dans le package `java.io`.

- Les classes que nous avons étudiées dans ce chapitre sont héritées des classes suivantes :
  - `InputStream`, pour les classes gérant les flux d'entrée ;
  - `OutputStream`, pour les classes gérant les flux de sortie.
- La façon dont on travaille avec des flux doit respecter la logique suivante :
  - ouverture de flux ;
  - lecture/écriture de flux ;
  - fermeture de flux.
- La gestion des flux peut engendrer la levée d'exceptions : `FileNotFoundException`, `IOException` etc.
- L'action de sauvegarder des objets s'appelle la « sérialisation ».
- Pour qu'un objet soit sérialisable, il doit implémenter l'interface `Serializable`.
- Si un objet sérialisable comporte un objet d'instance non sérialisable, une exception sera levée lorsque vous voudrez sauvegarder votre objet.
- L'une des solutions consiste à rendre l'objet d'instance sérialisable, l'autre à le déclarer `transient` afin qu'il soit ignoré à la sérialisation.
- L'utilisation de *buffers* permet une nette amélioration des performances en lecture et en écriture de fichiers.
- Afin de pouvoir ajouter des fonctionnalités aux objets gérant les flux, Java utilise le pattern « decorator ».
- Ce pattern permet d'encapsuler une fonctionnalité et de l'invoquer de façon récursive sur les objets étant composés de décorateurs.