

# **Programmation Orienté Objet**

## **Gestion Des Fichiers**

## **IO/NIO/NIO II**

**Pr. Omar El MIDAOUI**



# JAVA.IO

- Tous les classes traitant des entrées/sorties se trouvent dans le package **java.io**
- Ce sont des sous classes héritant des classes :
  - ✓ **InputStream**, pour les classes gérant les flux d'entrée ;
  - ✓ **OutputStream**, pour les classes gérant les flux de sortie.
- Le Processus de gestion 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.**

# JAVA.IO

- L'objet **File** de la classe **java.io.File** :
- L'objet représentant un Fichier dans le package **java.io**
- Cet objet présente des méthodes qui peuvent s'avérer très utiles parmi lesquelles :

```
import java.io.File;

public class Main {
    public static void main(String[] args) {

        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());

    }
}
```

- Vous pouvez aussi effacer le fichier grâce la méthode **delete()**,
- créer des répertoires avec la méthode **mkdir()** ...etc.

# JAVA.IO

- L'objet **FileInputStream** et **FileOutputStream** :
  - Par le biais de ces deux objets on peut :
    - ✓ lire dans un fichier ;
    - ✓ écrire dans un fichier.
  - Ces classes héritent des classes abstraites **InputStream** et **OutputStream**, présentes dans le package **java.io** :
  - Il existe une hiérarchie de classes pour les traitements **in** et une autre pour les traitements **out** :
    - ✓ les classes héritant d'InputStream sont destinées à la lecture
    - ✓ et les classes héritant d'OutputStream se chargent de l'écriture !
  - NB : Il faut situez les flux par rapport 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**
    - ✓ Au contraire, lorsqu'il va écrire dans un fichier (ou à l'écran), il va faire sortir des informations ; donc, pour le programme, ce flux de données correspond à une sortie : **out**.

# JAVA.IO

- Exemple : lecture du contenu d'un fichier et le copier dans un autre, dont nous spécifierons le nom :

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class InOutPutStreamTest {
    public static void main(String[] args) {
        // Define our two objects out of the try catch bloc
        FileInputStream fis = null;
        FileOutputStream fos = null;
```

- Notez bien les imports de java.io pour pouvoir utiliser ces objets.

# JAVA.IO

- 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 !

```
try {  
  
    // instantiation of two Stream objects :  
    // FIS to read from a File & FOS to write on a File  
    fis = new FileInputStream (new File("fichierSource.txt"));  
    fos = new FileOutputStream (new File("fichierDestination.txt"));  
  
    // We create an array of byte to define the number of bytes to read in each iteration  
    byte[] buf = new byte[20];  
  
    // the variable n contain the value of the byte (= -1 at the end of the file)  
    int n = 0;
```

# JAVA.IO

```
// We create an array of byte to define the number of bytes to read in each iteration
    byte[] buf = new byte[20];

// the variable n contain the value of the byte (= -1 at the end of the file)
    int n = 0;

// while we don't reach the end of the file yet we continue looping
while ((n = fis.read(buf)) >= 0) {

    // we write on the second File the 20 bytes red from the first file
    fos.write(buf);

    // we print the bytes retrieved from the reading as a byte and corresponding char
    for (byte bit : buf)
        System.out.print("\t" + bit + "(" + (char) bit + ")");

    System.out.println("");


    //we initialize the buffer to empty for the next bytes to read next data
    // (assure none existence of duplicate data)
    buf = new byte[20];

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

# JAVA.IO

```
        } catch (FileNotFoundException e) { e.printStackTrace(); //if FileInputStream didn't found the file
        } catch (IOException e) {e.printStackTrace(); // if there is a problem while reading or writing
        } finally { // Closing the stream
            try { if (fis != null) fis.close(); } catch (IOException e) { e.printStackTrace();}
            try { if (fos != null) fos.close(); } catch (IOException e) { e.printStackTrace();}
        }
    }
}
```

- La gestion des flux peut engendrer la levée d'exceptions :  
**FileNotFoundException, IOException.**
- À l'exécution de ce code, vous pouvez voir que le fichier **fichier-Destination.txt** a bien été créé et qu'il contient exactement la même chose que **fichier-Source.txt**

# JAVA.IO

fichier-Source.txt

```
this is An example of a text file
```

fichier-Destination.txt

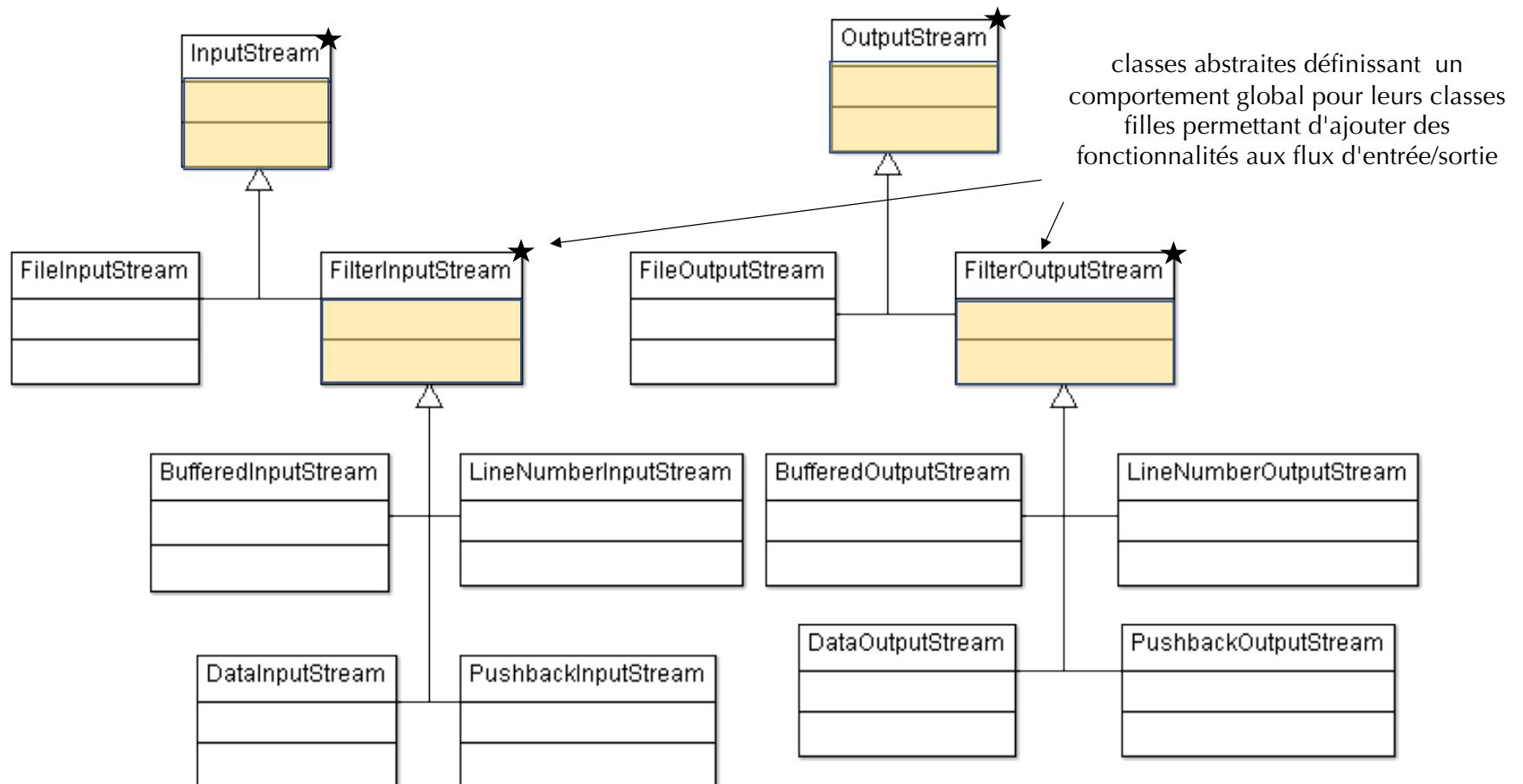
```
this is An example of a text file
```

Console

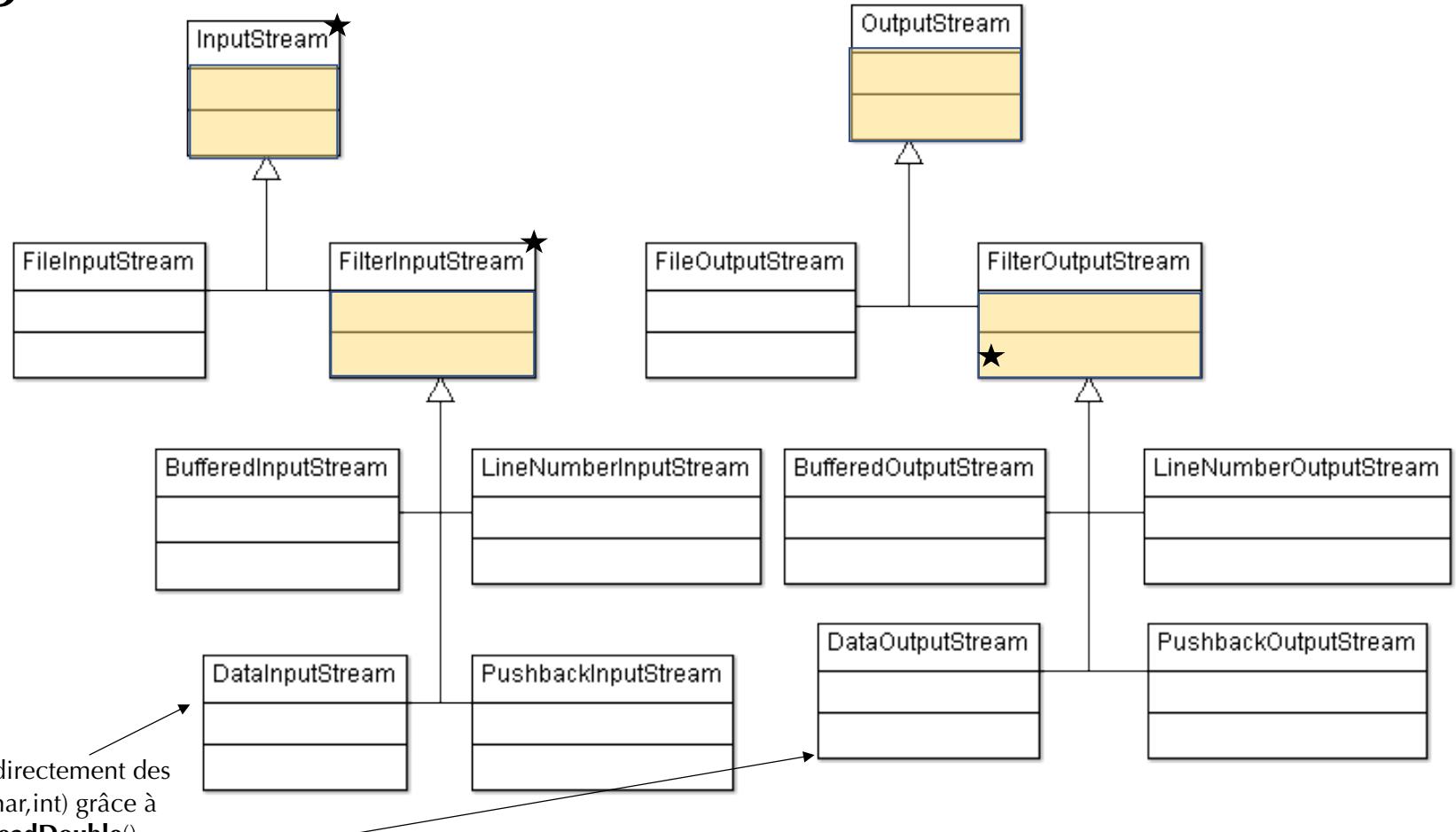
```
116(t) 104(h) 105(i) 115(s) 32( ) 105(i) 115(s) 32( )
65(A) 110(n) 32( ) 101(e) 120(x) 97(a) 109(m) 112(p)
108(l) 101(e) 32( ) 111(o) 102(f) 32( ) 97(a) 32( )
116(t) 101(e) 120(x) 116(t) 32( ) 102(f) 105(i) 108(l)
101(e) 0() 0() 0() 0() 0() 0() 0()
```

```
=> Copie terminée !
```

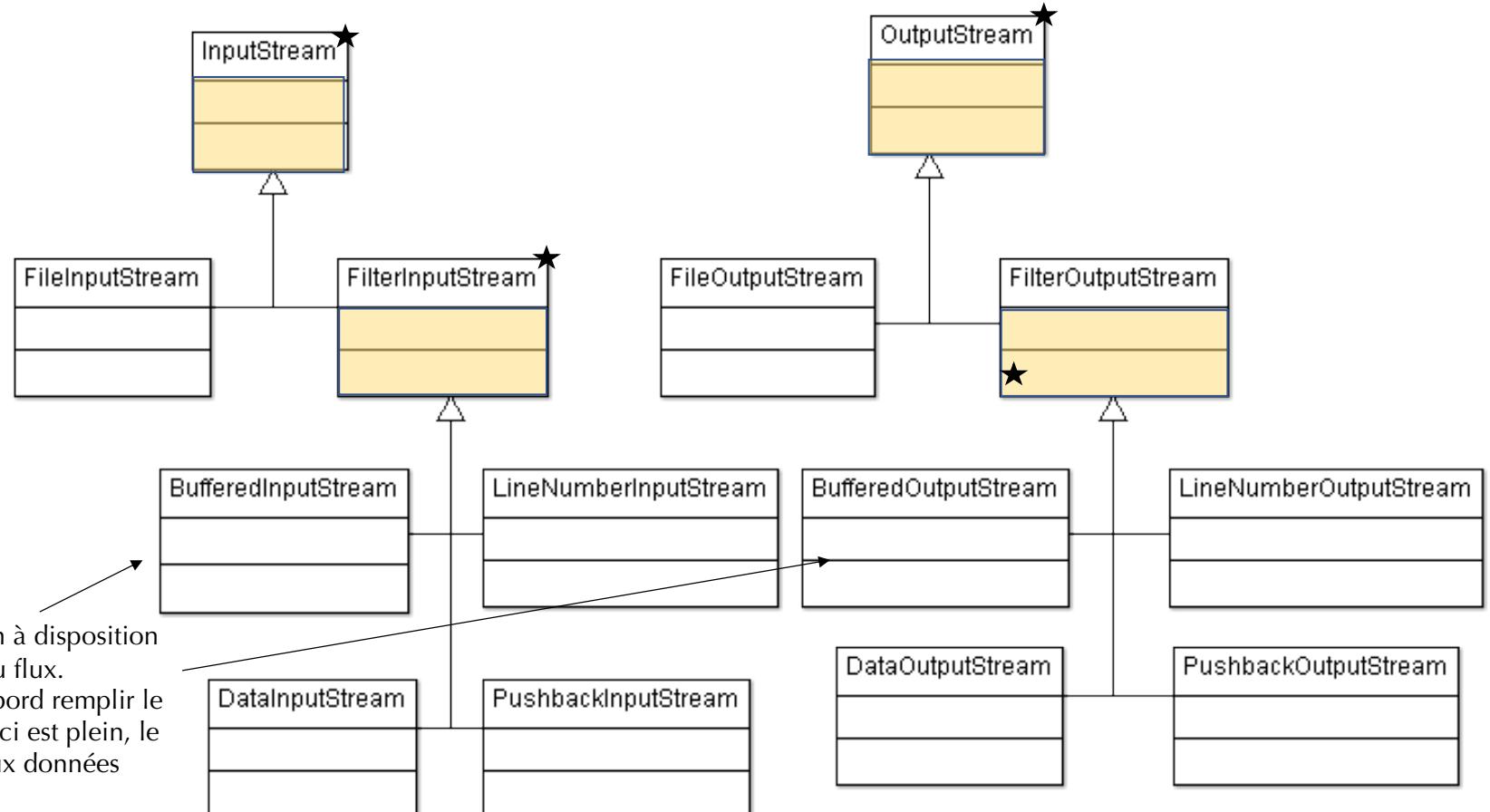
## package JAVA.IO



## package JAVA.IO

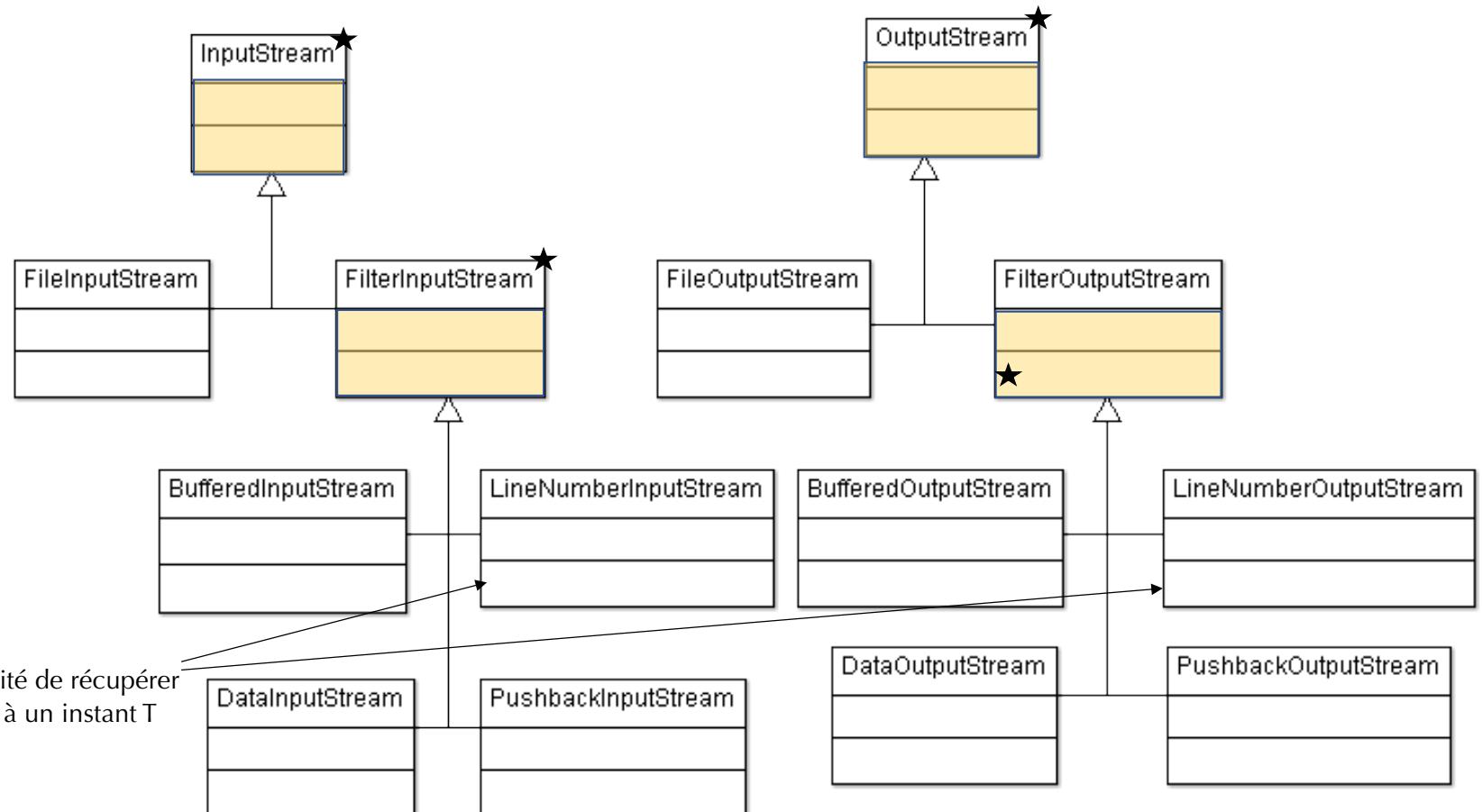


## package JAVA.IO



permet d'avoir un tampon à disposition  
dans la lecture du flux.  
les données vont tout d'abord remplir le  
tampon, et dès que celui-ci est plein, le  
programme accède aux données

## package JAVA.IO



cette classe offre la possibilité de récupérer  
le numéro de la ligne lue à un instant T

# JAVA.IO

- 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 plus de fonctionnalités comme suit :

## Utiliser l'objet : **BufferedInputStream**

- Afin de vous rendre compte des améliorations apportées par la classe **BufferedInputStream**, nous allons lire un énorme fichier texte (3,6 Mo) de façon conventionnelle avec l'objet vu précédemment, puis grâce à un **BufferedInputStream** :  
⇒ On essaye de lire le fichier **dictionnaire.txt** par les deux objets

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BufferedVsFileInput {

    public static void main(String[] args) {

        FileInputStream fis;
        BufferedInputStream bis;
        try {
            fis = new FileInputStream(new File("dictionnaire.txt"));
            bis = new BufferedInputStream(new FileInputStream(new File("dictionnaire.txt")));

            byte[] buf = new byte[8];
        }
    }
}
```

```
byte[] buf = new byte[8];

// we retrieve the System's time
long startTime = System.currentTimeMillis();

// read with a FileInputStream
// not necessary to add processing during the reading
while(fis.read(buf) != -1);

//print the execution time
System.out.println("Temps de lecture avec FileInputStream : "
+ (System.currentTimeMillis() - startTime) + "ms");

//reinitializing the start time
startTime = System.currentTimeMillis();
// read with a BufferedInputStream
while(bis.read(buf) != -1);
//print the execution time again
System.out.println("Temps de lecture avec BufferedInputStream : "
+ (System.currentTimeMillis() - startTime)+ "ms");

//close streams
fis.close();
bis.close();

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

}
```

- Et le résultat, visible à la figure suivante :

```
Temps de lecture avec FileInputStream : 562ms  
Temps de lecture avec BufferedInputStream : 18ms
```

## Utiliser l'objet : `BufferedInputStream` + `BufferedOutputStream`

```
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 BufferedVsFileInput {

    public static void main(String[] args) {

        FileInputStream fis;
        BufferedInputStream bis;
        FileOutputStream fos;
        BufferedOutputStream bos;

        try {
            fis = new FileInputStream(new File("dictionnaire.txt"));
            bis = new BufferedInputStream(new FileInputStream(new File("dictionnaire.txt")));

            fos = new FileOutputStream(new File("copieDictionnaire1.txt"));
            bos = new BufferedOutputStream(new FileOutputStream(new File("copieDictionnaire2.txt")));

            byte[] buf = new byte[8];
        }
    }
}
```

```
byte[] buf = new byte[8];

// we retrieve the System's time
long startTime = System.currentTimeMillis();

// read with a FileInputStream
while(fis.read(buf) != -1) { fos.write(buf);}

//print the execution time
System.out.println("\n\tTemps de lecture et écriture avec FileInputStream : "
+ (System.currentTimeMillis() - startTime) + "ms");

//reinitializing the start time
startTime = System.currentTimeMillis();
// read with a BufferedInputStream
while(bis.read(buf) != -1) { bos.write(buf);}

//print the execution time again
System.out.println("\n\tTemps de lecture et écriture avec BufferedInputStream : "
+ (System.currentTimeMillis() - startTime)+ "ms");

//close streams
fis.close();
bis.close();
fos.close();
bos.close();

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

}
```

- Et le résultat, visible à la figure suivante :

```
Temps de lecture et écriture avec FileInputStream : 2523ms
```

```
Temps de lecture et écriture avec BufferedInputStream : 27ms
```

## Utiliser l'objet : Data(Input/Output)Stream

- 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.

```
import java.io.BufferedInputStream;[]

public class DataOutputStreamClass {
    public static void main(String[] args) {

        DataInputStream dis;
        DataOutputStream dos;

        try {
            dos = new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream(
                        new File("A.txt"))));
            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();
        }
    }
}
```

## Utiliser l'objet : DataInputStream

```
dis = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream(
            new File("Variables.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();
}
```

```
true
100
C
12.05
100.52
1024
123456789654321
2
```

- 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 !

# Utiliser l'objet : Object(Input/Output)Stream

- La « **sérialisation** » : c'est le nom que porte l'action de sauvegarder des objets dans des fichiers.

```
package model;

import java.io.Serializable;

public class CD implements Serializable{
    private static final long serialVersionUID = 1L;

    private String titre, auteur;
    private double prix;

    public CD(String titre, String auteur, double prix) {
        this.titre = titre;
        this.auteur = auteur;
        this.prix = prix;
    }

    public String getTitre() { return titre; }
    public String getAuteur() { return auteur; }
    public double getPrix() { return prix; }

    public void setTitre (String titre) { this.titre = titre; }
    public void setAuteur (String auteur) { this.auteur = auteur;}
    public void setPrix (double prix) { this.prix = prix; }

    @Override
    public String toString() {
        return "CD [Titre : " + titre + ", Auteur : " + auteur + ", Prix : " + prix + "DH]";
    }
}
```

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.

## Utiliser l'objet : Object(InputStream/OutputStream)

```
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;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class CDReaderWriter {
    public static void main(String[] args) {

        ObjectInputStream ois;
        ObjectOutputStream oos;

        try {
            oos = new ObjectOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream(
                        new File("CDs.txt"))));

            oos.writeObject(new CD("Assassin Creed", "EA games", 45.69));
            oos.writeObject(new CD("Tomb Raider", "WB", 23.45));
            oos.writeObject(new CD("Tetris Game", "Stratégie", 2.50));

            //don't forget to close the Stream
            oos.close();
        }
    }
}
```

- ✓ On peut créer deux ou trois objets **CD**;
- ✓ les sérialiser dans un fichier de notre choix ;
- ✓ Ensuite les désérialiser afin de pouvoir les réutiliser.

## Utiliser l'objet : ObjectInputStream

```
ois = new ObjectInputStream(
    new BufferedInputStream(
        new FileInputStream(
            new File("CDs.txt"))));

try {
    System.out.println("Affichage des CDs :");
    System.out.println("*****\n");
    System.out.println(((CD)ois.readObject()).toString());
    System.out.println(((CD)ois.readObject()).toString());
    System.out.println(((CD)ois.readObject()).toString());
} catch (ClassNotFoundException e) {e.printStackTrace();}

ois.close();

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

- ✓ On peut créer deux ou trois objets **CD**;
- ✓ les sérialiser dans un fichier de notre choix ;
- ✓ Ensuite les désérialiser afin de pouvoir les réutiliser.

# JAVA.NIO

- 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 !
- Grace à ces deux objets de ce nouveau package :
  - **Les channels**
  - **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!**

## Lire un fichier avec un objet FileChannel :

```
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.channels.FileChannel;

public class Main {
    public static void main(String[] args) {

        FileInputStream      fis;
        BufferedInputStream   bis;
        FileChannel          fc;

        try {

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

            long starttime = System.currentTimeMillis();

            while(bis.read() != -1);
            System.out.println("Temps d'exécution avec un buffer conventionnel : "
                + (System.currentTimeMillis() - starttime));
        }
    }
}
```

```
fis = new FileInputStream(new File("dictionnaire.txt"));
// we retrieve the channel
fc = fis.getChannel();
//we get the size of the channel
int size = (int)fc.size();
//we create a buffer having the size of the channel
ByteBuffer bBuff = ByteBuffer.allocate(size);

starttime = System.currentTimeMillis();
fc.read(bBuff); // Start reading

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

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

## console

```
Temps d'exécution avec un buffer conventionnel : 32
Temps d'exécution avec un nouveau buffer : 5
```

# JAVA.NIO II

- un nouveau package **java.nio** :
  - une meilleure gestion des exceptions :
  - 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**.
- 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 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

# JAVA.NIO II

```
racine
subDirectory
file.txt
/Users/mac/Desktop/Main Folder/EMSI/Workspace Java/AA/racine/subDirectory/file.txt
false
false
java.nio.file.NoSuchFileException: racine/subDirectory/file.txt
```

# JAVA.NIO II

La classe **Files** vous permet aussi de lister le contenu d'un répertoire mais via un objet **DirectoryStream** qui est un **itérateur**.

```
Path path3 = Paths.get("BD");
System.out.println(path3.toString());
try(DirectoryStream<Path> listing = Files.newDirectoryStream(path3)){
    for(Path nom : listing){
        System.out.println("  " + ((Files.isDirectory(nom)) ? nom + "/" : "\t=> " + nom.getFileName()));
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

```
BD
      => cities.txt
BD/folder2/
BD/folder1/
      => outPut.txt
```

# JAVA.NIO II

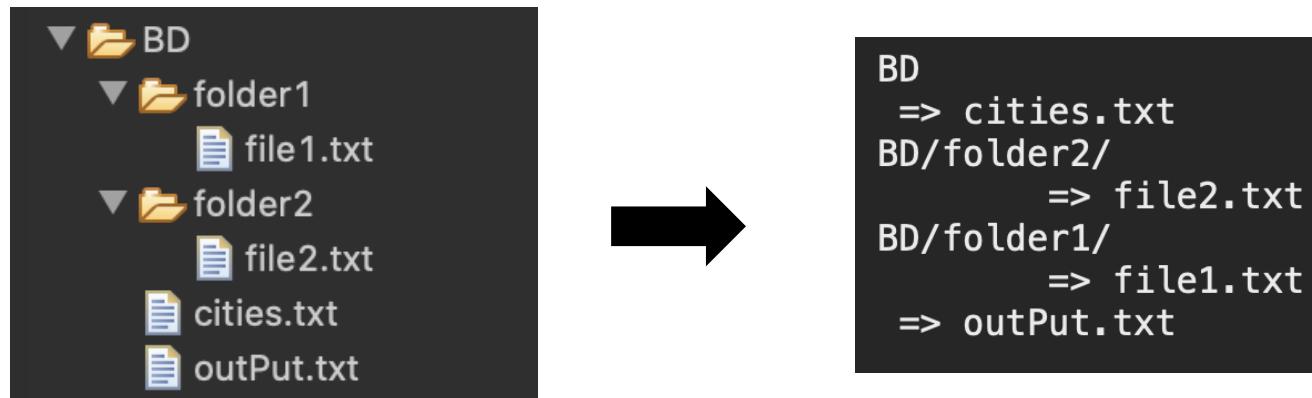
La classe **Files** vous permet aussi de lister le contenu d'un répertoire mais via un objet **DirectoryStream** qui est un **itérateur**.

```
Path path3 = Paths.get("BD");
System.out.println(path3.toString());
try(DirectoryStream<Path> listing = Files.newDirectoryStream(path3)){
    for(Path nom : listing){
        System.out.println("  " + ((Files.isDirectory(nom)) ? nom + "/" : " => " + nom.getFileName()));
        if(Files.isDirectory(nom)) {
            try(DirectoryStream<Path> Sublisting = Files.newDirectoryStream(nom)){
                for(Path name : Sublisting){
                    System.out.println("\t=> " + ((Files.isDirectory(name)) ? name + "/" : name.getFileName()));
                }
            }catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

```
BD
=> cities.txt
BD/folder2/
          => file2.txt
BD/folder1/
          => file1.txt
=> outPut.txt
```

# JAVA.NIO II

La classe **Files** vous permet aussi de lister le contenu d'un répertoire mais via un objet **DirectoryStream** qui est un **itérateur**.



# 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 **test.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.

# Lire tous les lignes d'un fichier

- Pour Lire le contenu d'un fichier **test.txt** vers une Liste il suffit de faire :

```
Path path = Paths.get("test.txt");
List<String> Lines = Files.readAllLines(path, StandardCharsets.UTF_8);
```

# Ecrire dans un fichier

- Pour écrire dans un fichier **test.txt** il suffit de faire :

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

String text = "Hi there !!!\n";
Files.write(path,
            text.getBytes(StandardCharsets.UTF_8),
            StandardOpenOption.CREATE,
            StandardOpenOption.WRITE,
            StandardOpenOption.APPEND);
```

# Exercice :

- fichier **input.txt**

Rank	City	Population	Region
1	Casablanca	3359818	Casablanca/Settat
2	Fez	1112072	Fès/Meknès
3	Tangier	947952	Tanger/Tetouan/Al Hoceima
4	Marrakesh	928850	Marrakesh/Safi
5	Salé	890403	Rabat/Salé/Kénitra
6	Meknes	632079	Fès/Meknès
7	Rabat	577827	Rabat/Salé/Kénitra
8	Oujda	494252	Oriental
9	Kenitra	431282	Rabat/Salé/Kénitra
10	Agadir	421844	Souss/Massa
11	Tetouan	380787	Tanger/Tetouan/Al Hoceima
12	Temara	313510	Rabat/Salé/Kénitra
13	Safi	308508	Marrakesh/Safi
14	Mohammedia	208612	Casablanca/Settat
15	Khouribga	196196	Béni Mellal/Khénifra

Ville
<ul style="list-style-type: none"><li>• id</li><li>• nom</li><li>• population</li><li>• région</li><li>• <b>toString()</b> : =&gt; Nom – population - Region</li></ul>

Utilitaire
LireFichierVilles( " chemin " ) : List<Ville>
LireFichierVilles( Scanner ) : List<Ville>
WriteVillesOutput(List<Ville> , " Output " ) : void

# Exercice :

- outputFile.txt

[ Agadir	-	421844	-	Souss/Massa	]
[ Ain Harrouda	-	62420	-	Casablanca/Settat	]
[ Al Hoceima	-	56716	-	Tanger/Tetouan/Al Hoceima]	
[ Azrou	-	54350	-	Fès/Meknès	]
[ Aït Melloul	-	171847	-	Souss/Massa	]
[ Ben Guerir	-	88626	-	Marrakesh/Safi	]
[ Beni Ansar	-	56582	-	Oriental	]
[ Beni Mellal	-	192676	-	Béni Mellal/Khénifra	]
[ Benslimane	-	57101	-	Casablanca/Settat	]
[ Berkane	-	109237	-	Oriental	]
[ Berrechid	-	136634	-	Casablanca/Settat	]
[ Bouskoura	-	103026	-	Casablanca/Settat	]
[ Casablanca	-	3359818	-	Casablanca/Settat	]
[ Dar Bouazza	-	151373	-	Casablanca/Settat	]
[ Dcheira El Ibadia	-	100336	-	Souss/Massa	]