Entrées/Sorties

Jean-Marie Lagniez

LPDIOC : Algorithmique et Programmation
IUT Lens

Département Informatique

Table des matières

Sommaire

Dans ce cours nous allons voir :

- Les flux d'entrées/sorties : **I/O stream**
- La gestion des fichiers et répertoires
- La sérialisation

Pré-requis :

- Héritage, exception, création d'objets, polymorphisme et classes abstraites

Table des matières

I/O streams

- Un flux d'entrée/sortie (**I/O stream**)
- Un **flux** peut **représenter différent type de données** : fichier, périphérique, ...
- Les données peuvent être de différents types : bit, caractère, objets
- Quel que soit le contexte un flux est une séquence de données

I/O streams

- Un flux d'entrée/sortie (**I/O stream**)
- Un **flux** peut **représenter différent type de données** : fichier, périphérique, ...
- Les données peuvent être de différents types : bit, caractère, objets
- Quel que soit le contexte un flux est une séquence de données

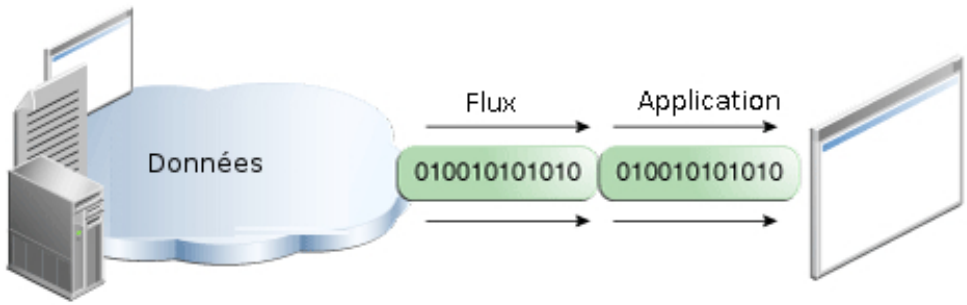


FIGURE – Un programme lit un flux d'entrée (*input stream*) un élément à chaque fois

I/O streams

- Un flux d'entrée/sortie (**I/O stream**)
- Un **flux** peut **représenter différent type de données** : fichier, périphérique, ...
- Les données peuvent être de différents types : bit, caractère, objets
- Quel que soit le contexte un flux est une séquence de données

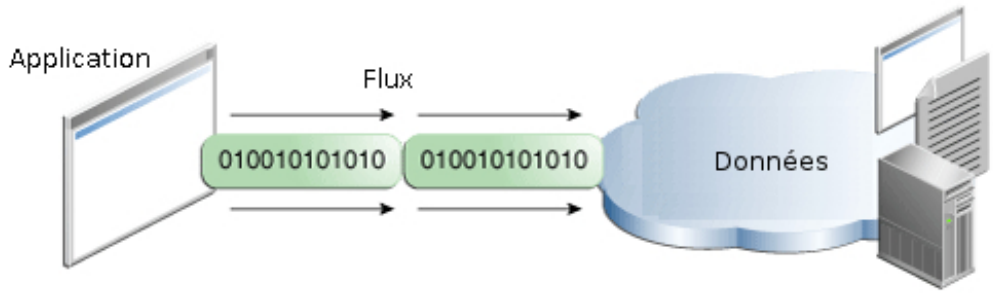
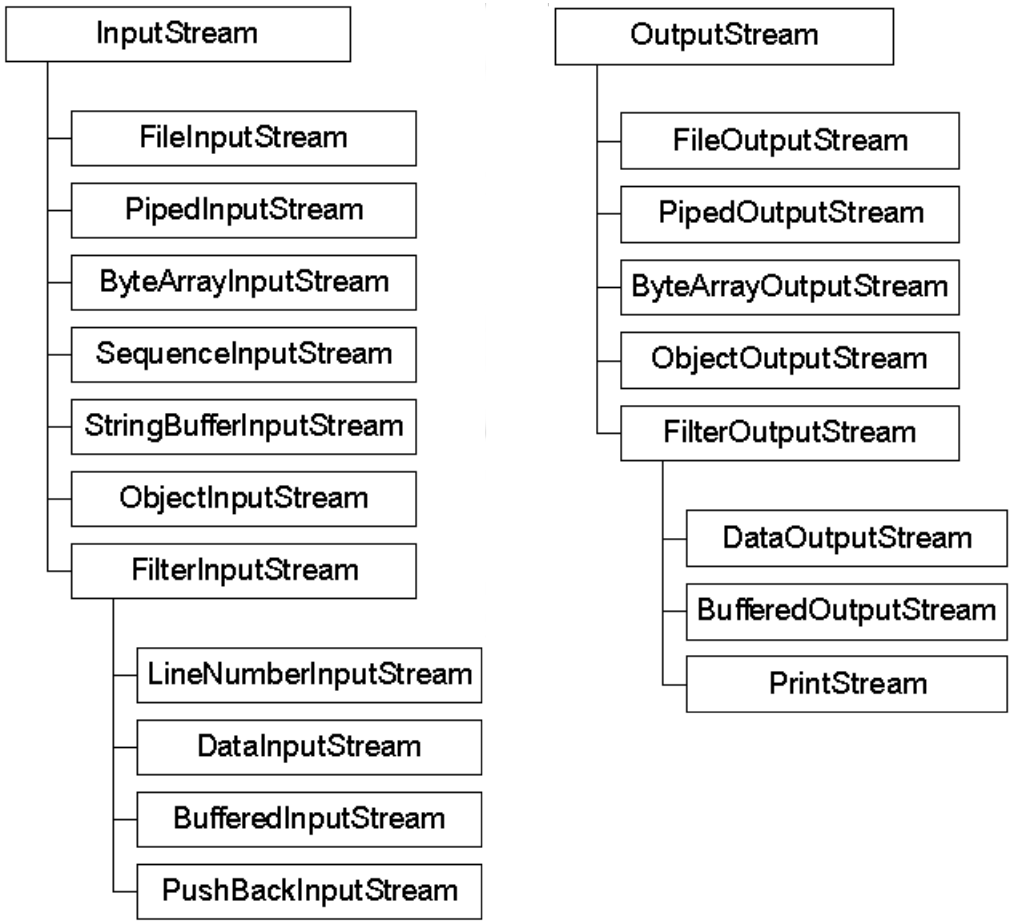


FIGURE – Un programme écrit sur un flux de sortie (*output stream*) un élément à chaque fois

Les flux d'octets

- Les programmes utilisant les flux d'octets (ou *Byte stream*) écrivent ou lisent des mots de 8-bits
- Tous les objets de ce type sont des descendant des classes **`InputStream`** et **`OutputStream`**. Il en existe beaucoup :



Exemple d'utilisation d'un flux d'octets

- Nous allons simplement copier le contenu d'un fichier dans un autre (facile :))

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

public class CopieOctets {
  public static void main(String[] args) throws IOException
      {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
      in = new FileInputStream("in.txt");
      out = new FileOutputStream("out.txt");
      int c;
      while ((c = in.read()) != -1) out.write(c);
    } finally {
      if (in != null) in.close();
      if (out != null) out.close();
    }
  }
}
```

- Remarque très importante : toujours fermer les flux ouverts !
- Bien qu'en pratique on utilise rarement les flux d'octets (car ils sont de bas niveau), **les autres classes permettant la manipulation de flux hérite de ces derniers**

Exercice

- Écrire un programme permettant de connaître la taille en `ko` du fichier `in.txt`

Exercice

- Écrire un programme permettant de connaître la taille en `ko` du fichier `in.txt`

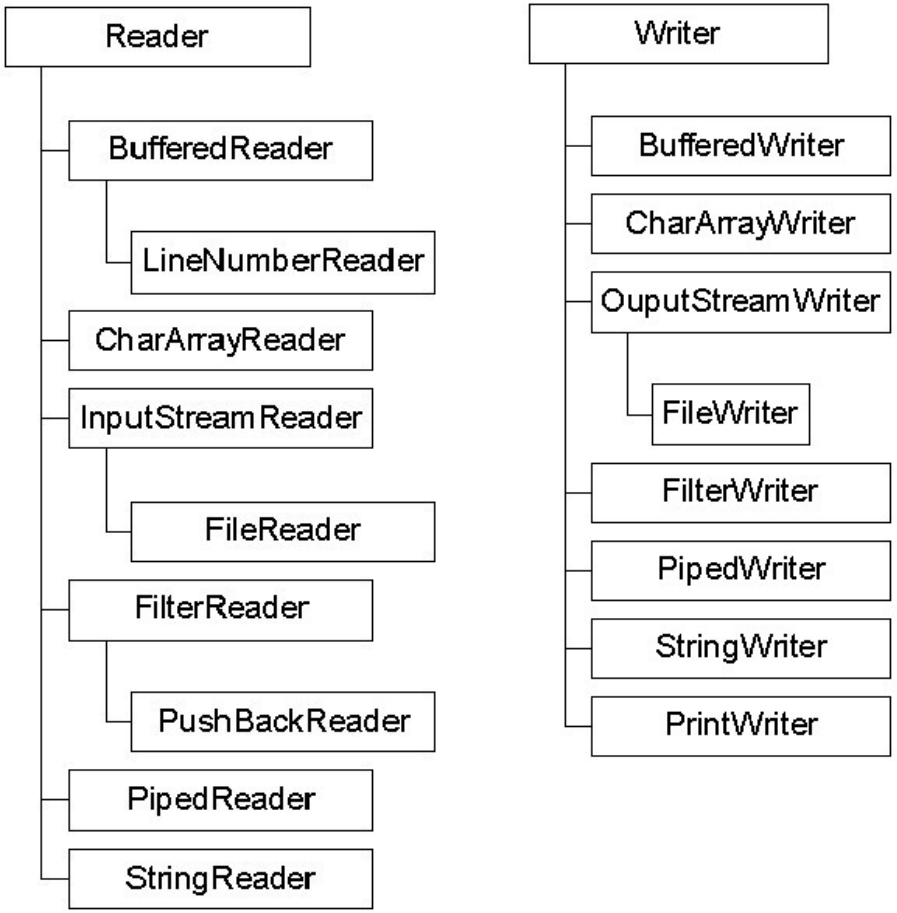
```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class SizeKO {
  public static void main(String[] args) throws IOException
      {
    FileInputStream in = null;
    int size = 0;

    try {
      in = new FileInputStream("in.txt");
      while (in.read() != -1) size++;
    } finally {
      if (in != null) in.close();
    }
    System.out.println("Size : " + size/1000 + " ko");
  }
}
```

Les flux de caractères

- Le flux de données en entrée est encodé suivant le format **Unicode** (Java les gèrent avec le format Unicode qui code les caractères sur 2 octets)
- Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites **`Reader`** ou **`Writer`**. Il en existe beaucoup :



Exemple d'utilisation d'un flux de caractères

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
  public static void main(String[] args) throws IOException
      {

    FileReader inputStream = null;
    FileWriter outputStream = null;

    try {
      inputStream = new FileReader("in.txt");
      outputStream = new FileWriter("out.txt");

      int c;
      while ((c = inputStream.read()) != -1) {
        outputStream.write(c);
      }
    } finally {
      if (inputStream != null) {
        inputStream.close();
      }
      if (outputStream != null) {
        outputStream.close();
      }
    }
  }
}
```

Exercice

- Écrire un programme permettant de connaître le nombre de caractères, de mots et de lignes du fichier texte `in.txt`

Exercice

- Écrire un programme permettant de connaître le nombre de caractères, de mots et de lignes du fichier texte `in.txt`

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Test {
  public static void main(String[] args) throws IOException
      {
    FileReader inputStream = null;
    int nbChar = 0, nbMot = 0, nbLigne = 0, pc = -1;

    try {
      inputStream = new FileReader("in.txt");
      int c;
      while ((c = inputStream.read()) != -1) {
        nbChar++;
        if((c == '\n' || c == ' ') && pc != ' ' && pc !=
            '\n') nbMot++;
        if(c == '\n') nbLigne++;
        pc = c;
      }
    } finally {
      if (inputStream != null) inputStream.close();
    }
    System.out.println(nbChar + " " + nbMot + " " +
        nbLigne);
  }
}
```

Les flux tamponnés avec un fichier

- La lecture ou l'écriture de données est souvent une **opération coûteuse**
- Pour **améliorer les performances** de ces opérations les données à lire ou à écrire sont mises dans **une mémoire tampon**
- On dit alors que la lecture (ou l'écriture) est bufferisée
- On peut **transformer** un flux de données **non bufferisé** en un autre qui est **bufferisé**

```
inputStream = new BufferedReader(new
    FileReader("in.txt"));
outputStream = new BufferedWriter(new
    FileWriter("out.txt"));
```

- Il existe 4 classes permettant de gérer les flux bufferisés :

  → `BufferedInputStream` et `BufferedOutputStream` pour les octets

  → `BufferedReader` et `BufferedWriter` pour les caractères

- Il est possible de **forcer l'écriture** d'un buffer en utilisant la méthode **`flush`**

Saisie et formatage des données

- Les programmes utilisant des entrées/sorties prennent souvent des données **formatées** par des **humains**

- Afin de faciliter la manipulation de ces types de données, Java met a disposition de l'utilisateur deux APIs :
  - L'API de saisie des données qui découpe en tokens une suite de bits
  - L'API de formatage des données qui permet d'assembler des données afin qu'elles soient facilement lisible par un humain

La saisie

- La classe **`Scanner`** a pour objectif de transformer les **données en tokens**
- Par défaut le délimiteur utilisé est tous les caractères tels que la méthode **`Character.isWhitespace`** retourne `true`

```
import java.io.*;
import java.util.Scanner;

public class ScanXan {
  public static void main(String[] args) throws IOException
      {
    Scanner s = null;
    try {
      s = new Scanner(new BufferedReader(new
          FileReader("in.txt")));
      while (s.hasNext()) System.out.println(s.next());
    } finally {
      if (s != null) s.close();
    }
  }
}
```

- Il est possible de définir d'**autres séparateurs**, pour cela il faut utiliser la méthode **`useDelimiter`** de la classe `Scanner`

La saisie d'éléments de types primitifs

- La classe `Scanner` permet de **découper** (le flux qui est un flux de caractères) suivant les **types primitifs** de Java (excepté `char`) ainsi que `BigInteger` et `BigDecimal`
- Il faut noter que les nombres ne s'écrivent pas de la même manière dans tous les pays : choix du format avec la méthode **`useLocal`**

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;

public class ScanSum {
  public static void main(String[] args) throws IOException {
    Scanner s = null;
    double sum = 0;
    try {
      s = new Scanner(new BufferedReader(new FileReader("in.txt")));
      s.useLocale(Locale.FRENCH);
      while (s.hasNext()) {
        if (s.hasNextDouble()) sum += s.nextDouble();
        else s.next();
      }
    } finally { s.close(); }
    System.out.println(sum);
  }
}
```

- Attention, la sortie ne respectera pas forcément le format local sélectionné

Exercice

- Écrire un programme qui affiche toutes les valeurs entières qui sont écrites dans le fichier `in.txt` (`hasNextInt` et `nextInt` permet de gérer des entiers)

Exercice

- Écrire un programme qui affiche toutes les valeurs entières qui sont écrites dans le fichier `in.txt` (`hasNextInt` et `nextInt` permet de gérer des entiers)

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;

public class Test {
  public static void main(String[] args) throws IOException {
    Scanner s = null;
    int nbInt = 0;
    try {
      s = new Scanner(new BufferedReader(new FileReader("in.txt")));
      s.useLocale(Locale.FRENCH);
      while (s.hasNext()) {
        if (s.hasNextInt()){
          s.nextInt();
          nbInt++;
        }
        else s.next();
      }
    } finally { s.close(); }
    System.out.println(nbInt);
  }
}
```

Le formatage

- Les classes qui implémentent le formatage des données sont soient des instances de la classe **`PrintWriter`** (pour les caractères) ou **`PrintStream`** (pour les octets)

- Comme tous les flux qui manipule des octets ou des caractères, les instances des classes **`PrintWriter`** et **`PrintStream`** implémentent un ensemble de méthodes pour l'écriture des données : `write`

- De plus, ils implémentent les même méthodes pour formatter les données

- Deux niveaux de formattage sont disponibles :
  - `print` et `println` formatent les données de manières standard
  - `format` permet de fournir un formattage des données plus précis

Les méthodes `print` et `println`

- Un petit exemple :

```
public class Test {
  public static void main(String[] args) {
    int i = 2;
    double r = Math.sqrt(i);

    System.out.print("La racine carrée ");
    System.out.print(i);
    System.out.print(" est ");
    System.out.print(r);
    System.out.println(".");

    i = 5;
    r = Math.sqrt(i);
    System.out.println("Racine carrée de " + i + "=" + r);
  }
}
```

- Les variables `i` et `r` sont automatiques converties via la méthode `toString`

La méthode `format`

- La méthode `format` est une méthode a argument variables qui s'appuie sur une chaîne de caractères et des éléments de formatage afin de savoir comment fomater les données

```
public class Test {
  public static void main(String[] args) {
    int i = 2;
    double r = Math.sqrt(i);
    System.out.format("La racine carrée de %d is %f.%n", i,
        r);
  }
}
```

- Tous les caractères de formatage commencent par % :
  - `d` : écrit un entier
  - `f` : écrit un double
  - `n` : écrit un terminateur de ligne (en fonction de la plate-forme)
  - `x` : écrit un entier en hexadécimal
  - `s` : écrit une chaîne de caractère
  - `tB` : écrit le mois de l'année associé à un entier
  - ...
- Voir la documentation de Java pour une liste complète

Un peu plus de formatage

- En plus de la conversion des données, il est possible de **spécifier finement** comment on veut voir afficher les données

```
public class Test {
  public static void main(String[] args) {
    System.out.format("%f, %1$20.10f %n", Math.PI);
  }
}
```

- Les éléments ajoutés sont **optionnels** mais permettent de **décrire finement** ce que l'on souhaite récupérer et comment on l'affiche
  - `i$` : permet de spécifier que l'on souhaite afficher le $i^{ème}$ argument
  - `x.y` : `x` donne le taille minimal de la partie entière et `y` permet de donner la précision
- Il existe de nombreuses options : allez voir la documentation !

Exercice

- Écrire une méthode static qui prend en entrée un tableau deux dimensions d'entiers compris entre -10000 et 10000, et affiche le résultat sous la forme d'un tableau de manière à ce que les éléments soient alignés

Exercice

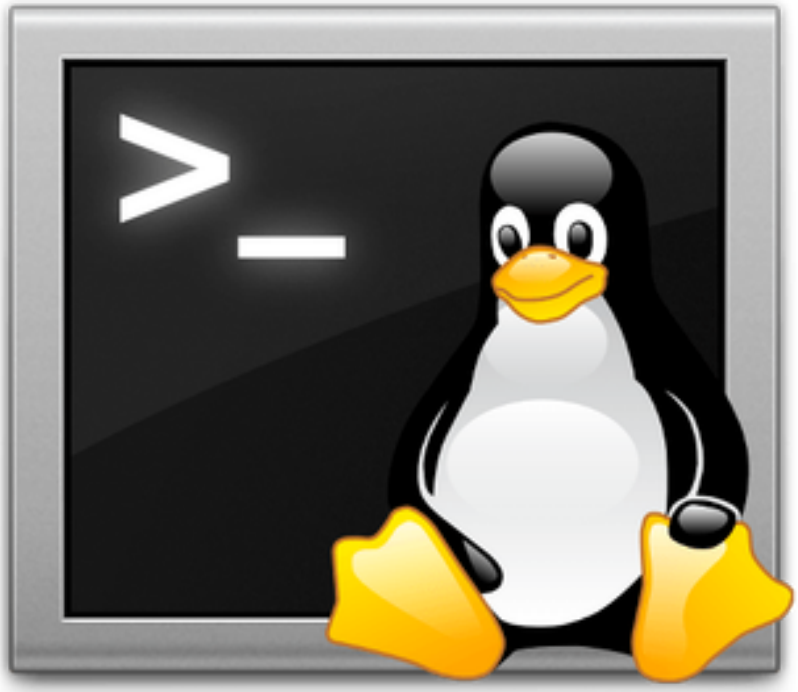
- Écrire une méthode static qui prend en entrée un tableau deux dimensions d'entiers compris entre -10000 et 10000, et affiche le résultat sous la forme d'un tableau de manière à ce que les éléments soient alignés

```
public class Test {
  public static void printTab(int [][] t){
    for(int i = 0 ; i<t.length ; i++)
    {
      for(int j = 0 ; j<t[i].length ; j++)
      System.out.format("%10d ", t[i][j]);
      System.out.println();
    }
  }

  public static void main(String[] args){
    int t[][] = {{1,2},{3,4},{5,6}};
    printTab(t);
  }
}
```

Les entrées/sorties via le terminal

- La plupart des programmes que vous allez exécuter seront lancés via un terminal :



- Java propose deux modes : `Standard Stream` et `Console`

Les flux standards

- La lecture sur l'entrée standard est quelque chose de classique
- Par défaut, l'entrée est saisie via le **clavier** et est écrite dans le **terminal**
- Il est aussi possible d'utiliser des tubes (*pipes*) entre différents programmes
- Trois types de flux standards sont supportés par Java :
  - entrée standard : `System.in`
  - sortie standard : `System.out`
  - sortie standard des erreurs : `System.err`
- Ces objets sont définis automatiquement au lancement du programme
- Les sorties standards sont des flux de caractères définies comme des objets de types `PrintStream`
- L'entrée standard est quant à elle un flux d'octets. Pour utiliser l'entrée standard comme un flux de caractères il faut l'encapsuler en `InputStream` :

```
-> InputStreamReader cin=new
    InputStreamReader(System.in);
```

Exercice

- Lire l'entrée standard et afficher le résultat dans `out.txt`

Exercice

- Lire l'entrée standard et afficher le résultat dans `out.txt`

```
import java.io.InputStreamReader;
import java.io.FileWriter;
import java.io.IOException;

public class Test {
  public static void main(String[] args) throws IOException
      {

    InputStreamReader inputStream = null;
    FileWriter outputStream = null;

    try {
      inputStream = new InputStreamReader(System.in);
      outputStream = new FileWriter("out.txt");

      int c;
      while ((c = inputStream.read()) != -1) {
        outputStream.write(c);
      }
    } finally {
      if (inputStream != null) inputStream.close();
      if (outputStream != null) outputStream.close();
    }
  }
}
```

Le mode console

- Une autre alternative afin **d'interagir avec l'utilisateur via le terminal** est d'utiliser le mode **console** proposé par Java
- Ce mode console est particulièrement utilisé pour la lecture mot de passe

```
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;

public static void main (String args[]) throws IOException {

  Console c = System.console();
  if (c == null) {
    System.err.println("Pas de console.");
    System.exit(1);
  }

  String login = c.readLine("Login: ");
  char [] password = c.readPassword("Password: ");
  System.out.println("Personne n'a vu mon password :P");
}
```

- Pour plus d'information allez voir la documentation Java !

Data Streams

- Il est possible de lire et écrire des types primitifs : `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double` et `String`
- Les flux de `data` implémente une des interfaces : **`DataInput`** ou **`DataOutput`**

```
import java.io.*;

public class Test {
  public static void main(String args[])throws IOException {
    DataOutputStream dataOut = new DataOutputStream(new
          FileOutputStream("out.txt"));
    dataOut.writeUTF("hello");
    dataOut.writeInt(1);

    // on lit dans le même fichier
    DataInputStream dataIn = new DataInputStream(new FileInputStream("out.txt"));
    while(dataIn.available()>0) {
      String k = dataIn.readUTF();
      int i = dataIn.readInt();
      System.out.print(k + " " + i);
    }
  }
}
```

- Attention : dans ce cas l'élément qui est lu ou écrit n'est pas forcément lisible dans le terminal

Object Streams

- Il est aussi possible de sauvegarder des objets. **Ils doivent implémenter l'interface `Serializable`**
- La plupart, mais pas tous, des objets standards implémentent cette interface

```
import java.io.Serializable;
class Point implements Serializable{
  public int x, y;
  public Point(int vx, int vy){x = vx; y = vy;}
}

import java.io.*;
public class Test {

  public static void main(String args[])throws IOException {
    Point p = new Point(1, 2);

    try{
      FileOutputStream outStream = new FileOutputStream("out.txt");
      ObjectOutputStream output = new ObjectOutputStream(outStream);
      output.writeObject(p);
      output.close();

      FileInputStream inStream = new FileInputStream(new File("out.txt"));
      ObjectInputStream input = new ObjectInputStream(inStream);
      Point q = (Point) input.readObject();
      System.out.println(q.x + " "  + q.y);
      input.close();
    }catch(Exception e){
      System.exit(1);
    }
  }
}
```

Table des matières

Manipulation du système de fichiers

- Il est aussi possible en Java de manipuler le système de fichiers via l'API `java.nio.file`
- Avant de commencer, quelques rappels sur la notion de chemin. On distingue deux expressions d'un chemin :
  - Le chemin d'accès **absolu** (chemin absolu)
  - Le chemin d'accès **relatif** (chemin relatif)
- Le **chemin absolu** commence par le symbole séparateur ("/" pour `unix` ou `C:\\` sous `win`), il exprime le chemin complet à partir de la racine de l'arborescence
- Le **chemin relatif** commence par un autre caractère que le caractère séparateur. Il indique un chemin à partir du répertoire de travail courant
- Il existe différent types de fichiers (tout est fichiers sous `unix`) :
  - Les fichiers réguliers
  - Les répertoires
  - Un lien symbolique est un fichier (de type **lien** qui contient le chemin et le nom d'un autre fichier
  - ...

La classe `Path`

- La classe `Path` permet de manipuler des chemins afin d'obtenir des informations (ex : la racine, un sous-chemin, ...)
- Un objet `Path` n'est pas directement associé à un fichier ou un répertoire
- Le chemin peut concerner plusieurs types d'élément : fichiers, répertoires, liens ou sous-chemin
- L'obtention d'une instance de type `Path` se fait simplement via l'utilisation de la méthode static `get` de la classes `Path` (qui invoque la méthode `FileSystems.getDefault().getPath()`)
- Il est possible de créer deux types de chemin :

```
Path relative = Paths.get("/a/b/c",
    "d/e.txt");
Path absolute = Paths.get(".");
```

- Nous allons voir par la suite plusieurs méthode pour manipuler ces chemins

L'obtention d'éléments du chemin

- L'interface `Path` propose plusieurs méthodes pour retrouver un élément particulier ou un sous-chemin composé de plusieurs éléments :
  - `String getFileName()` : retourne le nom du dernier élément du chemin
  - `Path getName(int index)` : retourne l'élément du chemin dont l'index est fourni en paramètre. Le premier élément possède l'index 0
  - `int getNameCount()` : retourne le nombre d'éléments du chemin
  - `Path getParent()` : retourne le chemin parent ou null s'il n'existe pas (dans ce cas, le chemin correspond à une racine)
  - `Path getRoot()` : retourne la racine d'un chemin absolu (par exemple C : sous Dos ou / sous Unix) ou null pour un chemin relatif
  - `String toString()` : retourner le chemin sous la forme d'une chaîne de caractères
  - `Path subPath(int beginIndex, int endIndex)` : retourner un sous-chemin correspondant aux deux index fournis en paramètres

```
Path path = Paths.get("/un/super/chemin/reste.txt");
System.out.format("toString: %s%n", path.toString());
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getName(0): %s%n", path.getName(0));
System.out.format("getNameCount: %d%n",
    path.getNameCount());
```

La manipulation d'un chemin

- L'interface `Path` propose plusieurs méthodes pour manipuler les chemins :
  - `Path normalize()` : nettoie le chemin en supprimant les éléments « . » et « .. » qu'il contient
  - `Path relativize(Path other)` : retourne le chemin relatif à celui fourni en paramètres
  - `Path resolve(Path)` : combine deux chemins. Attention, si le chemin fourni en paramètre contient un élément racine, alors la méthode resolve() renvoie le chemin fourni en paramètre

```
Path normalizePath = Paths.get("/tmp/jm/../../AppData/./././Local/");
System.out.println(normalizePath.normalize());
// --> /AppData/Local/

Path path = Paths.get("/tmp/jm/AppData/Local/");
System.out.println(path.resolve("Temp/monfichier.txt"));
// --> /tmp/jm/AppData/Local/Temp/monfichier.txt

System.out.println(path.resolve("/Temp"));
// --> /Temp

Path path1 = Paths.get("C:/Users/hello/jm");
Path path2 = Paths.get("C:/Users/test");
System.out.println(path1.relativize(path2));
// --> ../../test

System.out.println(path2.relativize(path1));
// --> ../hello/jm
```

La comparaison de chemins

- `Path` redéfinit la méthode `equals()` afin de comparer deux instances
- `Path` hérite de l'interface `Comparable`, ce qui permet le tri
- `Path` propose des méthodes pour comparer des sous-chemins
  - `int compareTo(Path other)` : compare le chemin avec celui fourni en paramètre
  - `boolean endsWith(Path other)` : compare la fin du chemin avec celui fourni en paramètre
  - `boolean endsWith(String other)` : compare la fin du chemin avec celui fourni en paramètre
  - `boolean startsWith(Path other)` : compare le début du chemin avec celui fourni en paramètre
  - `boolean startsWith(String other)` : compare le début du chemin avec celui fourni en paramètre

```
Path path1 = Paths.get("/Users/jm");
Path path2 = Paths.get("/");
System.out.println(path1.startsWith("/"));       // true
System.out.println(path1.startsWith("/Users")); // true
System.out.println(path1.startsWith(path2));     // true
System.out.println(path1.startsWith(""));        // false
System.out.println(path1.startsWith("Users"));   // false
System.out.println(path1.startsWith("/Users")); // true
```

La conversion d'un chemin

- Les chemins encapsulés dans une instance de type `Path` ne sont pas toujours complets ou linéaires (ex : un chemin relatif ne possède pas de racine)
- L'interface `Path` propose donc plusieurs méthodes pour convertir un chemin
  - `Path toAbsolutePath()` : retourne le chemin absolu du chemin
  - `Path toRealPath(LinkOption...)` : retourne le chemin physique du `Path` notamment en résolvant les liens symboliques selon les options fournies. Peut lever une exception si le fichier n'existe pas ou s'il ne peut pas être accédé
  - `URI toUri()` : retourne le chemin sous la forme d'une URI

```
Path path = Paths.get("/Users/jm/AppData/Local/Temp/monfichier.txt");
System.out.println(path.toUri());
// --> file://Users/jm/AppData/Local/Temp/monfichier.txt

path = Paths.get("src/monfichier.txt");
System.out.println(path.toAbsolutePath());
// -->
      /home/lagniez/works/enseignement/LPDIOC/cours/coursIO/code/src/monfichier.txt

try {
  System.out.println(path.toRealPath(LinkOption.NOFOLLOW_LINKS));
  //-->
        /home/lagniez/works/enseignement/LPDIOC/cours/coursIO/code/src/monfichier.txt
} catch (IOException ex) {
  ex.printStackTrace();
}
```

Table des matières

La classe Files

- La classe **`java.nio.file.Files`** est un helper qui contient une cinquantaine de méthodes statiques permettant de réaliser des **opérations sur des fichiers ou des répertoires** dont le chemin est encapsulé dans un objet de type **`Path`**
- Elle permet de réaliser de nombreuses opérations :
  - La création d'éléments : createDirectory(), createFile(), createLink(), createSymbolicLink(), createTempFile(), createTempDirectory(), ...
  - La manipulation d'éléments : delete(), move(), copy(), ...
  - L'obtention du type d'un élément : isRegularFile(), isDirectory(), . . .
  - L'obtention de métadonnées et la gestion des permissions : getAttributes(), getPosixFilePermissions(), isReadable(), isWriteable(), size(), getFileAttributeView(), ...
- Les méthodes de la classe `Files` attendent généralement en paramètre au moins une instance de type Path

Les vérifications sur un fichier ou un répertoire

- La classe Files propose deux méthodes pour vérifier l'existence d'un élément dans le système de fichier :
  - `boolean exists(Path)` : true si le fichier `Path` fourni existe
  - `boolean notExists(Path)` : true si le fichier `Path` fourni n'existe pas
- Elle propose aussi des méthodes pour vérifier les droits d'un fichier
  - `boolean isReadable(Path)` : true si le fichier peut être lu
  - `boolean isWritable(Path)` : true si le fichier peut être modifié
  - `boolean isHidden(Path)` : true si le fichier est caché
  - `boolean isExecutable(Path)` : true si le fichier est exécutable
  - `boolean isRegularFile(Path p)` : true si `p` est un fichier
  - `boolean isDirectory(Path p)` : true si `p` est un répertoire
  - `boolean isSymbolicLink(Path path)` : true si `p` est un lien symbolique

```
Path monFichier = Paths.get("in.txt");
boolean estLisible = Files.isRegularFile(monFichier) &
Files.isReadable(monFichier);
System.out.println(monFichier + " est lisible :
    "+estLisible);
```

- Et encore plein d'autres . . .

La création d'un fichier

L'API permet la création de fichiers :

- `Path createFile(Path p, FileAttribute<?>... a)` : crée un fichier dont le chemin et les attributs sont fournis en paramètre

- Cette méthode attend en paramètres un objet `Path` et un varargs de type `FileAttribute< ?>` qui permet de préciser les attributs du fichier créé

```
Pathfichier=Paths.get("/home/jm/test.txt");
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rw-rw-rw-");
FileAttribute<Set<PosixFilePermission>>attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.createFile(fichier,attr);
```

- Par défaut (sans attributs) les attributs par défaut du système sont considérés

```
Path monFichier = Paths.get("/tmp/fichier.txt");
Path file = Files.createFile(monFichier);
```

- L'exception `FileAlreadyExistsException` est levée si le fichier existe déjà

La création d'un répertoire

L'API permet la création de repertoires :

- `Path createDirectory(Path d,FileAttribute<?>... a)` : crée un répertoire dont le chemin et les attributs sont fournis en paramètre
- `Path createDirectories(Path d, FileAttribute<?>... a)` : crée dans le répertoire `d` un sous-répertoire avec les attributs fournis (mkdir -p)
- Ces méthodes attendent en paramètres un objet `Path` et un varargs de type `FileAttribute<?>` qui permet de préciser les attributs du fichier créé

```
Path monRepertoire = Paths.get("C:/temp/mon_repertoire");
Path file = Files.createDirectory(monRepertoire);
```

- En ce qui concerne les attributs, cela fonctionne comme pour les fichiers

La création d'un fichier ou d'un répertoire temporaires

L'API permet la création de fichiers et répertoires temporaires :

- `createTempDirectory(Path d, String p,FileAttribute<?>...a)` : crée dans le répertoire `d` un sous-répertoire temporaire dont le nom utilisera le préfixe fourni
- `createTempDirectory(String p,FileAttribute<?>...a)` : crée dans `/tmp` un sous-répertoire temporaire dont le nom utilisera la préfixe fourni
- `createTempFile(Path d, String p,String s, FileAttribute<?>...a)` : crée dans le répertoire `d` un fichier temporaire dont le nom utilisera le préfixe fourni
- `createTempFile(String p, String s, FileAttribute<?>... a)` : crée dans `/tmp` un fichier temporaire dont le nom utilisera le préfixe et le suffixe fournis

```
Path repertoireTemp = Files.createTempDirectory(null);
System.out.println(repertoireTemp);
// --> /tmp/6561063271512529198

repertoireTemp = Files.createTempDirectory("monApp_");
System.out.println(repertoireTemp);
// --> /tmp/monApp_3072294315038144833
```

La copie d'un fichier ou d'un répertoire

- `Files` propose plusieurs surcharges de la méthode **`copy()`** :
  - `Path copy(Path source, Path target, CopyOption...opts)` : copie un élément avec les options précisées
  - `long copy(InputStream in, Path t, CopyOption...opts)` : copie tous les octets d'un flux de type InputStream vers un fichier
  - `long copy(Path source, OutputStream out)` : copie tous les octets d'un fichier dans un flux de type OutputStream
- Les objets **`StandardCopyOption`** et **`LinkOption`** implémentent l'interface `CopyOption` :
  - `StandardCopyOption.COPY_ATTRIBUTES`
  - `StandardCopyOption.REPLACE_EXISTING`
  - `LinkOption.NOFOLLOW_LINKS`

```
import static java.nio.file.StandardCopyOption.*;
// ...
Path f=Files.copy(Paths.get("a.txt"),Paths.get("b.txt"),
      REPLACE_EXISTING);
```

- **Il est possible de combiner des `Streams` et des `Paths`**

```
Path cible = Paths.get("monfichier_copie.txt");
URI uri = new File("monfichier.txt").toURI();
try (InputStream in = uri.toURL().openStream()) {
  Files.copy(in, cible);
}
```

Le déplacement d'un fichier ou d'un répertoire

- `Files.move()` permet de déplacer ou de renommer un fichier :

  `move(Path source, Path target, CopyOption... options)`

- Plusieurs valeurs de `StandardCopyOption` implémentant l'interface `CopyOption` peuvent être utilisées avec la méthode move() :
  - `StandardCopyOption.REPLACE_EXISTING` : remplacement s'il existe
  - `StandardCopyOption.ATOMIC_MOVE` : assure que le déplacement est réalisé atomiquement (sinon `AtomicMoveNotSupportedException`)

```
import static java.nio.file.StandardCopyOption.*;
// Déplace a en c, si b existe alors
    FileAlreadyExistsException
Files.move(Paths.get("a"), Paths.get("c"));
// Force le remplacement de a par b même si b existe
Files.move(Paths.get("a"), Paths.get("b"),
    REPLACE_EXISTING);
```

- L'exécution de `move()` se fait de manière **synchrone et bloquante**
  - la copie échoue si le fichier cible existe déjà
  - les attributs du fichier sont conservés entièrement, partiellement ou pas du tout
  - lors de la copie d'un lien symbolique, c'est la cible qui est copiée et non le lien
  - lors du déplacement d'un lien symbolique, le lien est déplacé mais le fichier cible pas
  - un répertoire est déplacé s'il est vide ou si le déplacement consiste à le renommer

La suppression d'un fichier ou d'un répertoire

L'API permet la suppression de fichiers, de répertoires :

- `void delete(Path path)` : supprime un élément du système de fichiers
- `boolean deleteIfExist(Path path)` : supprime s'il existe

```
Path path = Paths.get("test.txt");
try {
  Files.delete(path);
} catch (NoSuchFileException nsfe) {
  System.err.println("Fichier ou repertoire " + path + "
      n'existe pas");
} catch (DirectoryNotEmptyException dnee) {
  System.err.println("Le repertoire " + path + " n'est pas
      vide");
} catch (IOException ioe) {
  System.err.println("Impossible de supprimer " + path + "
      : " + ioe);
}
```

- La méthode `boolean deleteIfExist(Path path)` est identique mais ne lève pas l'exception `NoSuchFileException`

L'obtention du type de fichier

- Il est possible d'obtenir le type du contenu d'un fichier :
  - `String probeContentType(Path path)` : retourne le type du contenu
- `probeContentType()` renvoie `null` si le type est indéterminé

```
try {
  Path s = Paths.get("monfichier.txt");
  String type = Files.probeContentType(s);

  if (type != null) System.out.println(s + " : " + type);
  else System.out.println(s + " : indéterminé");

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

// monfichier.txt : text/plain
```

Table des matières

- Un **glob** est un pattern qui est appliqué sur des noms de fichiers ou de répertoires : similaire aux **wildcards**
  - `Boolean matches(Path path)` : true si le chemin correspond au pattern
- Pour obtenir une instance de type `PathMatcher`, il faut invoquer la méthode `getPathMatcher()` de la classe `FileSystem`

```
import java.nio.file.PathMatcher;
import java.nio.file.Paths;
import java.nio.file.Path;
import java.nio.file.FileSystems;
import java.io.IOException;

public class Test {
  public static void main(String[] args) throws IOException{
    Path path = Paths.get("test.java");
    PathMatcher matcher =
        FileSystems.getDefault().getPathMatcher("glob:*.java");
    if (matcher.matches(path)) System.out.println("OK");

  }
}
```

Wildcards

- * : désigne toutes les chaînes de caractères, y compris la chaîne vide ;
- ? désigne un caractère quelconque
- [...] : désigne un caractère quelconque appartenant à la liste. Deux caractères séparés par un tiret (-) définissent une liste de caractère rangés par ordre alphabétique, dont le premier élément est le premier caractère et le dernier élément le dernier caractère
- [^...] : désigne une liste de caractères à exclure
- {...,...} : désigne une liste de chaînes de caractères
- \ permet d'échapper des caractères pour éviter qu'ils ne soient interprétés. Il sert notamment à échapper le caractère \ lui-même
- Exemples :
  - `*.html` : tous les fichiers ayant l'extension `.html`
  - `???` : trois caractères quelconques
  - `*[0-9]*` : tous les fichiers qui contiennent au moins un chiffre
  - `*.htm, html` : tous les fichiers dont l'extension est `htm` ou `html`
  - `I*.java` : tous les fichiers dont le nom commence par un i majuscule et possède une extension `.java`

Exercices

- un nom commençant et finissant par un *f* :
- un nom commençant par *f* et finissant par *a* ou *b* :
- un nom commençant par *f* et ne finissant ni par *a* ni par *b* :
- un nom de cinq caractères :
- un nom de trois caractères commençant par *a*,*b* ou *c* :
- un nom avec une extension de trois lettres :
- un nom commençant par un chiffre :
- un nom commençant par *abc* ou *def* :
- un nom ayant *a* en première ou deuxième position :

Exercices

- un nom commençant et finissant par un f : $f*f$
- un nom commençant par f et finissant par a ou b : $f*[ab]$
- un nom commençant par f et ne finissant ni par a ni par b : $f*[\hat{}ab]$
- un nom de cinq caractères : $?????$
- un nom de trois caractères commençant par a,b ou c : $[abc]??$
- un nom avec une extension de trois lettres : $*.???$
- un nom commençant par un chiffre : $[0-9]*$
- un nom commençant par abc ou def : $\{abc, def\}*$
- un nom ayant a en première ou deuxième position : $\{a?, ?a, aa\}*$

Table des matières

Le parcours d'un répertoire

- `java.nio.file.DirectoryStream` offre une interface permettant de parcourir un répertoire en réalisant une itération sur les éléments qu'il contient
- La méthode `newDirectoryStream()` de la classe `Files` prend un `Path` et permet d'obtenir une instance de `DirectoryStream<Path>`
- La méthode `iterator()` retourne une instance d'un itérateur sur les éléments du répertoire : fichiers, liens, sous-répertoires, ...

```
Path path = Paths.get(".");
DirectoryStream<Path> stream =
    Files.newDirectoryStream(path);
try {
  Iterator<Path> iterator = stream.iterator();
  while(iterator.hasNext()) {
    Path p = iterator.next();
    System.out.println(p);
  }
} finally {stream.close();}
```

- Attention : l'implémentation de l'interface Iterable de `DirectoryStream` ne propose pas le support de la méthode `remove()`
- Attention : il est important d'invoquer close() pour libérer les ressources

Le parcours d'un répertoire et filtres

- Il est possible de définir un **filtre** qui sera appliqué sur chacun des éléments du répertoire pour déterminer s'il doit être **retourné ou non lors du parcours**
- Pour cela, il faut une instance de `DirectoryStream.Filter<Path>` qu'il faut fournir en paramètre à la méthode `newDirectoryStream()`

```
public static void utilisationDirectoryStreamAvecFiltre() throws IOException {
  Path path = Paths.get(".");
  DirectoryStream.Filter<Path> filtre = new DirectoryStream.Filter<Path>() {
    public static final long HUIT_MEGABYTES = 8*1024*1024;

    public boolean accept(Path element) throws IOException {
      return Files.size(element) >= HUIT_MEGABYTES;
    }
  };

  try (DirectoryStream<Path> stream = Files.newDirectoryStream(path, filtre)) {
    for (Path entry : stream) {
      System.out.println(entry);
    }
  }
}
```

Le parcours d'une hiérarchie de répertoires

- La méthode `Files.walkFileTree()` permet de parcourir la hiérarchie d'un ensemble de répertoires
- Ce type de parcours peut être utilisé pour rechercher, copier, déplacer, supprimer, ... des éléments de la hiérarchie parcourue

Il faut implémenter l'interface `java.nio.file.FileVisitor<T>`.
Elle définit des méthodes qui seront appelées lors du parcours de la hiérarchie :

- `FileVisitResult postVisitDirectory(T dir, IOException exc)` : le parcours sort d'un répertoire qui vient d'être parcouru ou une exception est survenue durant le parcours
- `FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)` : le parcours rencontre un répertoire, cette méthode est invoquée avant de parcourir son contenu
- `FileVisitResult visitFile(T file, BasicFileAttributes attrs)` : le parcours rencontre un fichier
- `FileVisitResult visitFileFailed(T file, IOException exc)` : la visite d'un des fichiers durant le parcours n'est pas possible et une exception a été levée

Contrôler les traitements du parcours

- Les méthodes de l'interface FileVisitor renvoient toutes une valeur qui appartient à l'énumération FileVisitResult
- Cette valeur permet de contrôler le processus de parcours de l'arborescence :
  - **CONTINUE** : poursuite du parcours
  - **TERMINATE** : arrêt immédiat du parcours
  - **SKIP_SUBTREE** : inhibe le parcours de la sous-arborescence. Si cette valeur est rovoyée par `preVisitDirectory()`, le parcours du répertoire est ignoré
  - **SKIP_SIBLING** : inhibe le parcours des répertoires frères. Si la méthode `preVisitDirectory()` renvoie cette valeur alors le répertoire n'est pas parcouru et la méthode `postVisitDirectory()` n'est pas invoquée. Si la méthode `postVisitDirectory()` renvoie cette valeur, alors les autres répertoires frères qui n'ont pas encore été parcourus sont ignorés
- L'exemple ci-dessous parcourt l'arborescence et s'arrête dès que le fichier `test.txt` est trouvé

```
public FileVisitResult visitFile(Path file, BasicFileAttributes attr) {
  if (file.getFileName().equals("test.txt")) {
    System.out.println("Fichier trouve");
    return TERMINATE;
  }
  return CONTINUE;
}
```

Exemple d'utilisation : afficher le répertoire courant

```
import java.nio.file.*;
import java.io.*;
import java.nio.file.attribute.*;
import static java.nio.file.FileVisitResult.*;
import static java.nio.file.FileVisitOption.*;

public class PrintFichier extends SimpleFileVisitor<Path>{
  private int cpt = 0;

  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
       IOException {
    for(int i = 0 ; i<cpt ; i++) System.out.print("|\t");
    System.out.println(file.getFileName());
    return CONTINUE;
  }

  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
       throws IOException {
    for(int i = 0 ; i<cpt ; i++) System.out.print("|\t");
    System.out.println(dir.getFileName());
    cpt++;
    return CONTINUE;
  }

  public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws
       IOException {
    cpt--;
    return CONTINUE;
  }
}

// dans un autre fichier
public class Test {
  public static void main(String[] args) throws IOException{
    Path path = Paths.get(".");
    Files.walkFileTree(path, new PrintFichier());
  }
}
```

Exercice : mini Shell

- Implémentez un mini shell permettant d'exécuter les commandes suivantes : `cd`, `mv`, `ls` (avec l'option -r), `rm` (avec l'option -r) et `rmdir`