Java – Les entrées / sorties

Eric Blaudez (eric.baudez@u-psud.fr) | 05 - Partie I

Description

Ce cours présente les gestions de l'information contenue dans les fichiers.

java.io

- Introduction
- Manipulation & accès au fichiers
- Fichiers textes
- Fichiers binaires
- *Pipes*

Agenda – java.io

- **Introduction**
- Manipulation & accès au fichiers
- Fichiers textes
- Fichiers binaires
- Pipes

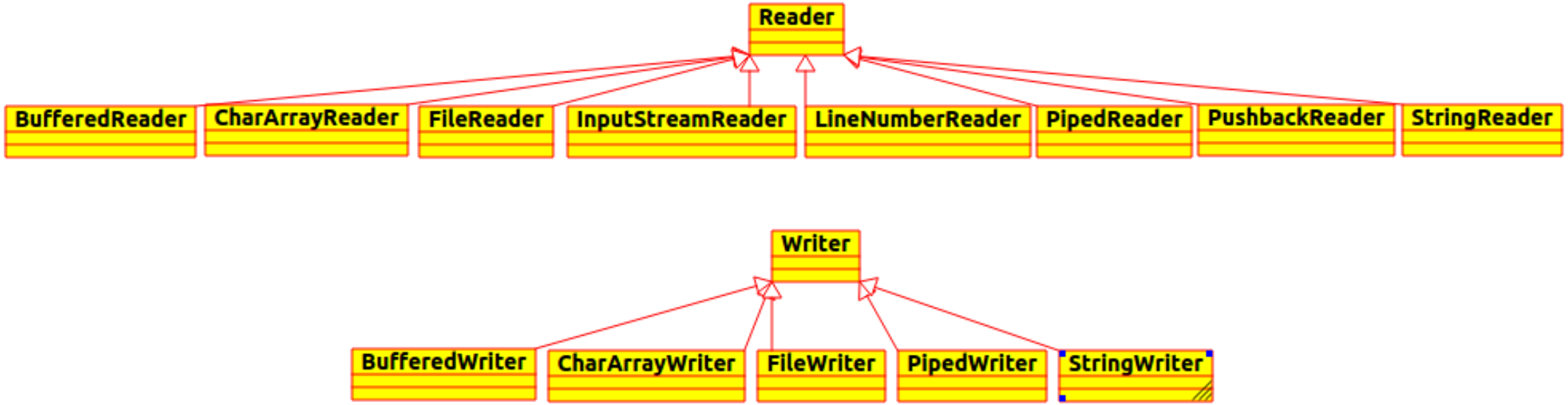
Introduction

La gestion de fichier en Java est mis en œuvre à l'aide d'un ensemble de classes spécialisées. Java scinde la gestion des fichiers en deux :

- les fichiers contenant du texte : ce sont des fichiers contenant de l'information 'lisible'
- les fichier binaires (par exemple une fichier 'zip' est un fichier binaire)

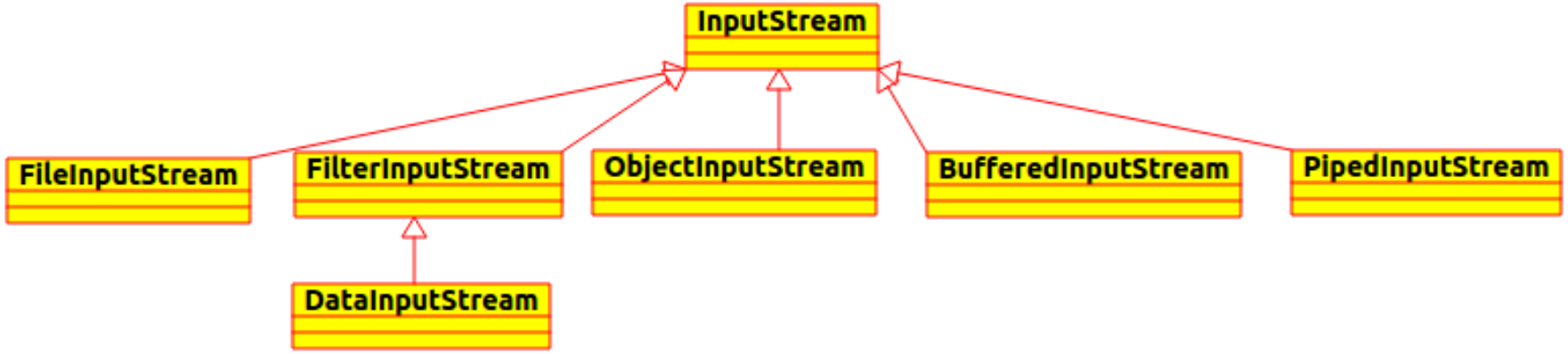
Introduction

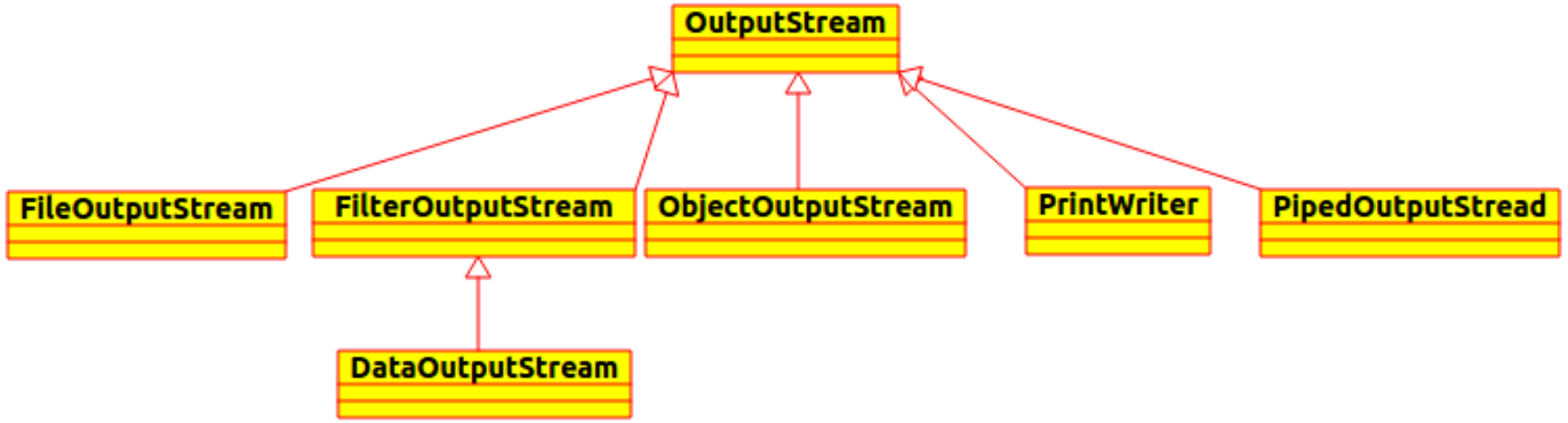
Les classes importantes pour les entrées/sorties de type texte



Introduction

Les classes importantes pour les entrées/sorties de type binaire





Agenda – java.io

- Introduction
- **Manipulation & accès au fichiers**
- Fichiers textes
- Fichiers binaires
- Pipes

Manipulation & accès aux fichiers
Système de fichier

La classe ***java.io.File*** offre un ensemble de méthodes pour la manipulation et les accès des fichiers.

Cette classe est à la base de la gestion de fichiers, elle permet de modifier les droits d'accès d'un fichier d'obtenir des informations sur le chemin d'un fichier, d'en connaître la taille ...

Manipulation & accès aux fichiers
Système de fichier – droits d'accès

Méthode	Description
boolean canExecute()	retourne vrai si le fichier est exécutable
boolean canRead()	retourne vrai si le fichier est lisible (accès en lecture)
boolean canWrite()	retourne vrai si le fichier est accessible en écriture
boolean isHidden()	retourne vrai si le fichier est caché
boolean setExecutable(boolean executable) ***boolean setExecutable(boolean executable, boolean ownerOnly)***	donne (ou enlève) la permission d'exécution sur un fichier, il est possible de préciser si c'est le propriétaire du fichier ou tous le monde qui obtient cette permission (***ownerOnly***)
boolean setReadable(boolean readable) ***boolean setReadable(boolean readable, boolean ownerOnly)***	donne (ou enlève) la permission en lecture sur un fichier, il est possible de préciser si c'est le propriétaire du fichier ou tous le monde qui obtient cette permission (***ownerOnly***)
boolean setWritable(boolean writable) ***boolean setWritable(boolean writable, boolean ownerOnly)***	donne (ou enlève) la permission d'écriture sur un fichier, il est possible de préciser si c'est le propriétaire du fichier ou tous le monde qui obtient cette permission (***ownerOnly***)
boolean setReadOnly()	marque le fichier comme étant en lecture seule

Manipulation & accès aux fichiers

Système de fichier – droits d'accès

```
public class RightsExample {
  public static void main(String[] args)
  {
    try {
      File tempFile = File.createTempFile("course_iut", ".txt");
      System.err.println("File : "+tempFile.getAbsolutePath());
      System.err.println("Executable : "+tempFile.canExecute());
      System.err.println("Read : "+tempFile.canRead());
      System.err.println("Write : "+tempFile.canWrite());
      System.err.println("Hidden : "+tempFile.isHidden());
      tempFile.setExecutable(true);
      tempFile.setReadable(false);
      tempFile.setWritable(false);
      System.err.println("Executable : "+tempFile.canExecute());
      System.err.println("Read : "+tempFile.canRead());
      System.err.println("Write : "+tempFile.canWrite());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

```
File : /tmp/course_iut5669059861343009960.txt
Executable : false
Read : true
Write : true
Hidden : false
Executable : true
Read : false
Write : false
```

Manipulation & accès aux fichiers

Informations associées aux fichiers

Méthode	Description
boolean exists()	retourne vrai si le fichier existe
String getAbsolutePath() ***File getAbsoluteFile()***	retourne chemin absolu du fichier
File getCanonicalFile() ***String getCanonicalPath()***	retourne le chemin canonique (résolution de liens)
String getName()	retourne le nom du fichier (sans le chemin)
String getParent() ***File getParentFile()***	retourne le chemin du dossier parent
String getPath()	retourne chemin du fichier
long lastModified()	retourne la date de la dernière modification
boolean isAbsolute()	retourne vrai si chemin du fichier est absolu
boolean isFile()	retourne vrai si le chemin et celui d'un fichier
URI toURI()	retourne l'URI du chemin
long length()	retourne taille du fichier

Manipulation & accès aux fichiers

Informations associées aux fichiers

```
public class FileInfoExample {
  public static void main(String[] args)
  {
    try {
      File tempFile = File.createTempFile("course_iut", ".txt");
      System.err.println("exists : "+tempFile.exists());
      System.err.println("File : "+tempFile.getAbsolutePath());
      System.err.println("canonical path : "+tempFile.getCanonicalPath());
      System.err.println("name : "+tempFile.getName());
      System.err.println("parent : "+tempFile.getParent());
      System.err.println("path : "+tempFile.getPath());
      System.err.println("absolute file : "+tempFile.getAbsoluteFile());
      System.err.println("canonical file : "+tempFile.getCanonicalFile());
      System.err.println("last modified : "+tempFile.lastModified());
      System.err.println("is absolute : "+tempFile.isAbsolute());
      System.err.println("is directory : "+tempFile.isDirectory());
      System.err.println("is file : "+tempFile.isFile());
      System.err.println("uri : "+tempFile.toURI().toString());
      System.err.println("length : "+tempFile.length());
    } catch (IOException e) {
       e.printStackTrace();
    }
  }
}
```

```
exists : true File : /tmp/course_iut6699692281849897767.txt
canonical path : /tmp/course_iut6699692281849897767.txt
name : course_iut6699692281849897767.txt
parent : /tmp
path : /tmp/course_iut6699692281849897767.txt
absolute file : /tmp/course_iut6699692281849897767.txt
canonical file : /tmp/course_iut6699692281849897767.txt
last modified : 1436247561000
is absolute : true
is directory : false
is file : true
uri : file:/tmp/course_iut6699692281849897767.txt
length : 0
```

Manipulation & accès aux fichiers
Création / Suppression de fichier

Méthodes	Description
boolean delete()	efface un fichier/dossier
boolean mkdir() ***boolean mkdirs()***	créé un dossier ou un chemin complet (mkdirs)
boolean renameTo(File dest)	renome/déplace un dossier/fichier

Manipulation & accès aux fichiers

Création / Suppression de fichier

```
String tempDir = System.getProperty("java.io.tmpdir");
System.err.println("Temp dir:"+tempDir);
File tempFile = File.createTempFile("course_iut", ".txt");
String testDir = tempDir+File.separator+"testDir";
File testCreateDir = new File(testDir);
testCreateDir.mkdir();
System.err.println("test create dir exists : ["+testDir+"]:"+testCreateDir.exists());
System.err.println("["+testDir+"] exists:"+testCreateDir.exists());
testDir = tempDir+File.separator+"testDir2"+File.separator+"testDir";
File testCreateDir2 = new File(testDir);
testCreateDir2.mkdirs();
System.err.println("["+testDir+"] exists:"+testCreateDir2.exists());
testDir = tempDir+File.separator+"testDir3";
File testCreateDir3 = new File(testDir);
 testCreateDir2.renameTo(new File(testDir));
System.err.println("["+testDir+"] exists:"+testCreateDir3.exists());
tempFile.delete();
testCreateDir.delete();
testCreateDir2.delete();
testCreateDir3.delete();
System.err.println("["+tempFile.getAbsolutePath()+"] exists:"+tempFile.exists());
System.err.println("["+testCreateDir.getAbsolutePath()+"] exists:"+testCreateDir.exists());
System.err.println("["+testCreateDir2.getAbsolutePath()+"] exists:"+testCreateDir2.exists());
System.err.println("["+testCreateDir3.getAbsolutePath()+"] exists:"+testCreateDir3.exists());
```

```
Temp dir:/tmp
test create dir exists : [/tmp/testDir]:true
[/tmp/testDir] exists:true
[/tmp/testDir2/testDir] exists:true
[/tmp/testDir3] exists:true
[/tmp/course_iut7795147377861021513.txt] exists:false
[/tmp/testDir] exists:false
[/tmp/testDir2/testDir] exists:false
[/tmp/testDir3] exists:false
```

Manipulation & accès aux fichiers
Système de fichier

Méthodes	Description
long getTotalSpace()	retourne la taille totale du système de fichiers
long getFreeSpace()	retourne la taille encore disponible sur le système de fichiers
static File[] listRoots()	liste les racines du système de fichiers
String[] list() ***String[] list(FilenameFilter filter)*** ***File[] listFiles()*** ***File[] listFiles(FileFilter filter)*** ***File[] listFiles(FilenameFilter filter)***	liste l'ensemble des fichiers et dossiers. ***FileFilter*** et ***FilenameFilter*** sont des interfaces permettant de filtrer la liste (dans l'exemple du *slide* suivant nous filtrons toutes les entrées se terminant par 't'

Manipulation & accès aux fichiers
Système de fichier

```
public class FileSystemExample {
  public static void main(String[] args) {
    System.err.println("System space:"+(new File(".")).getTotalSpace());
    System.err.println("System usable space:"+(new File(".")).getUsableSpace());
    System.err.println("Root files:"+Arrays.asList(File.listRoots()));
    File rootDir = new File(File.listRoots()[0].toString());
    System.err.println("List:"+Arrays.asList(rootDir.list()));
    System.err.println("List file:"+Arrays.asList(rootDir.listFiles()));
    FilenameFilter filter = new FilenameFilter() {
        @Override
        public boolean accept(File file, String str) {
          return str.endsWith("t");
        }
    };
    System.err.println("Filtered list:"+Arrays.asList(rootDir.listFiles(filter)));
  }
}
```

```
System space:1472389771264

System usable space:363233787904

Root files:[/]

List:[media, vmlinuz.old, cdrom, sys, boot,
      initrd.img, vmlinuz, lib64, etc, mnt,
      lib, usr, proc, sbin, home, initrd.img.old,
      opt, lost+found, share, bin, dev,
      lib32, tmp, run, var, srv, root]

List file:[/media, /vmlinuz.old, /cdrom, /sys,
           /boot, /initrd.img, /vmlinuz, /lib64,
           /etc, /mnt, /lib, /usr, /proc, /sbin,
           /home, /initrd.img.old, /opt,
           /lost+found, /share, /bin, /dev,
          /lib32, /tmp, /run, /var, /srv, /root]

Filtered list:[/boot, /mnt, /opt, /root]
```

Agenda – java.io

- Introduction
- Manipulation & accès au fichiers
- **Fichiers textes**
- Fichiers binaires
- Pipes

Fichiers textes
Introduction

Les fichiers textes sont lisibles et compréhensibles (par exemple des fichiers html), Java propose une API de lecture et une d'écriture.

Les classes implémentant ces API ajoutent simplement des fonctionnalités ou des traitements particuliers.

Fichiers textes
API - *Reader*

Méthodes	**Description**
abstract void close()	ferme le flux de données
int read() ***int read(char[] cbuf)*** ***abstract int read(char[] cbuf, int off, int len)*** ***int read(CharBuffer target)***	selon la signature de la méthode, il est possible de lire un seul caractère, un tableau de caractères, un tableau de caractère partir d'un position prédéfinie
boolean ready()	le flux et prêt (ou pas)
void reset()	reset le flux
long skip(int n)	'saute' n caractères

Fichiers textes
API - *Writer*

Méthode	Description
abstract void close()	ferme le flux de données
void write(int c) ***void write(char[] cbuf, int off, int len)*** ***void write(String str, int off, int len)***	selon la signature de la méthode, il est possible de d'écrire un seul caractère, un tableau de caractères, une chaîne de caractère à partir d'un position prédéfinie
void flush()	force l'écriture
String getEncoding()	retourne l'encodage
void close()	ferme le flux

Fichiers textes
Exemple – *FileWriter/FileReader*

```
public class FileReaderExample {
  public static void main(String[] args) {
    try {
      File tempFile = File.createTempFile("course_iut", ".txt");
      FileWriter fileWriter = new FileWriter(tempFile);
      fileWriter.write("test string 1\n");
      fileWriter.write("test string 2\n");
      fileWriter.close();
      FileReader fileReader = new FileReader(tempFile);
      fileReader.skip(5);
      char[] charBuffer = new char[6];
      fileReader.read(charBuffer);
      System.err.println(charBuffer);
      int readC;
      while ((readC = fileReader.read()) != -1) {
        System.err.println((char)readC);
      }
      fileReader.close();
      tempFile.deleteOnExit();
    }
    catch (IOException e) { e.printStackTrace(); }
  }
}
```

```
string

1


t
e
s
t

s
t
r
i
n
g

2
```

Fichiers textes
Quelques types de flux 'textes'

Java propose différents types de flux, dérivant de ***FileReader*** (pour la lecture) ou de ***FileWriter*** (pour l'écriture).

CharArrayReader : permet la lecture de tableau de caractères

CharArrayWriter : permet l'écriture de tableau de caractères

BufferedReader : utilise un tampon mémoire pour stocker le contenu (ou une partie) du fichier. En limitant les accès disque, la lecture est ainsi plus efficace.

BufferedWriter : utilise un tampon mémoire pour stocker le contenu (ou une partie) des données à écrire. En limitant les accès disques l'écriture est efficace.

Agenda – java.io

- Introduction
- Manipulation & accès au fichiers
- Fichiers textes
- **Fichiers binaires**
- Pipes

Fichiers binaires
Introduction

Les fichiers binaires sont des fichiers plus 'génériques' que les fichiers textes : ce sont généralement des fichiers qui ne sont pas compréhensibles (par exemple une image).

Les fichiers binaires sont généralement des implémentations des classes abstraites ***InputStream*** et ***OutputStream***.

Fichiers binaires
API - *InputStream*

Méthodes	Description
abstract int read() ***int read(byte[] cbuf)*** ***abstract int read(byte[] cbuf, int off, int len)***	selon la signature de la méthode, il est possible de lire un seul caractère, un tableau de caractères, un tableau de caractère partir d'un position prédéfinie. Les retournent le nombre de bytes lus, ou -1 si rien n'est lu.
int available()	retourne le nombre de bytes lisibles en une seule fois sans blocage
void reset()	reset le flux
long skip(int n)	'saute' n bytes
void close()	ferme le flux

Fichiers binaires
API - *OutputStream*

Méthodes	Description
void write(int c) ***void write(byte[])*** ***void write(byte[] cbuf, int off, int len)***	selon la signature de la méthode, il est possible de d'écrire un seul entier, un tableau de bytes, un tableau de bytes à partir d'un position prédéfinie
void flush()	force l'écriture
void close()	ferme le flux

Fichiers binaires
Quelques types de flux binaires

BufferedInputStream : utilise un tampon mémoire pour stocker le contenu (ou une partie) du fichier. En limitant les accès disque, la lecture est ainsi plus efficace.

BufferedOutputStream : utilise un tampon mémoire pour stocker le contenu (ou une partie) des données à écrire. En limitant les accès disques l'écriture est efficace.

FileInputStream : permet la lecture de fichiers binaires comme des images

FileOutputStream : permet l'écriture de fichiers binaires comme des images

ObjectInputStream : lecture d'objet java (sérialisés), implémentant l'interface ***Serializable***

ObjectOutputStream : sérialisation d'objet java, implémentant l'interface ***Serializable***

Fichiers binaires
Exemple

```
public class SerializableData implements Serializable {
  protected int age;
  protected String name;

  public SerializableData() { age = -1; name = null; }
  public SerializableData(int age, String name) {
    this.age = age; this.name = name;
  }
  public int getAge() { return age; }
  public String getName() { return name; }
  public void setAge(int age) { this.age = age; }
  public void setName(String name) { this.name = name; } }
```

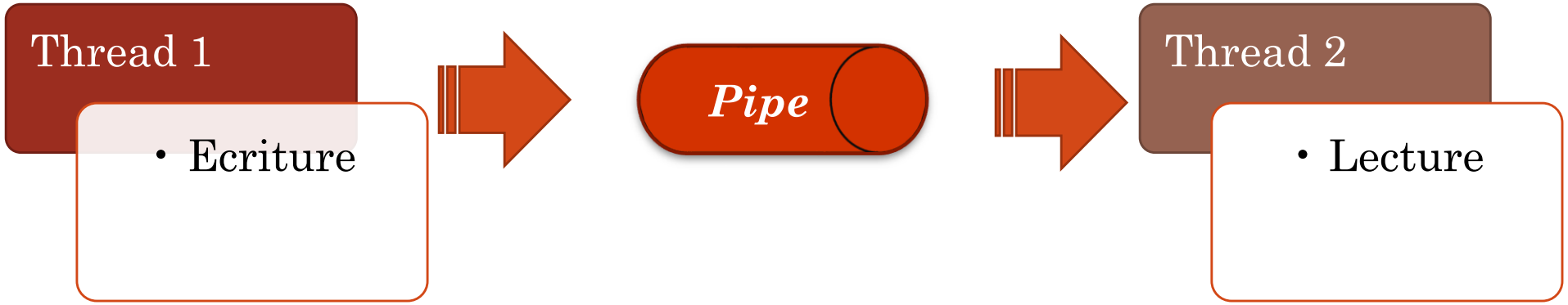
```
public class FileStreamExample {
  public static void main(String[] args) {
    String tempDir = System.getProperty("java.io.tmpdir");
    SerializableData data = new SerializableData(25,"Lili");
    File ofile=new File(tempDir+File.separator+"data.ser");
    FileOutputStream ofStream;
    FileInputStream ifStream;
    try {
      ofStream = new FileOutputStream(ofile);
      ObjectOutputStream oStream = new ObjectOutputStream(ofStream);
      oStream.writeObject(data);
      oStream.close();
      data = new SerializableData();
      ifStream = new FileInputStream(ofile);
      ObjectInputStream ioStream = new ObjectInputStream(ifStream);
     data = (SerializableData)ioStream.readObject();
     System.err.println("age:"+data.age+" / name:"+data.name);
   }
  catch (FileNotFoundException e) { e.printStackTrace(); }
  catch (IOException e) { e.printStackTrace(); }
  catch (ClassNotFoundException e) { e.printStackTrace(); } } }
```

```
age:25 / name:Lili
```

Agenda – java.io

- Introduction
- Manipulation & accès au fichiers
- Fichiers textes
- Fichiers binaires
- **Pipes**

Pipes
Introduction



En java, les pipes sont souvent utilisés pour la communication entre threads : ils permettent de faire transiter des données d'un thread vers un autre.

Pipes
Piped*Reader & PipedWriter*

Dans l'exemple suivant, deux threads communiquent par l'intermédiaire d'un *Pipe*.

- Un *thread* écrit des données ***(PipeWriter)***.
- Un *thread* lit des données ***(PripeReader)***.
- Le *thread* écrivant les données est 'connecté' au thread les lisant avec l'instruction ***connect***

```
public static void main(String[] args) {
  PipedReader pipeReader = new PipedReader();
  PipedWriter pipeWriter = new PipedWriter();
  try {
    pipeWriter.connect(pipeReader);
    new Thread(new PipeStreamExample().new PSWriter(pipeWriter)).start();
    new Thread(new PipeStreamExample().new PSReader(pipeReader)).start();
  } catch (IOException e)
  { e.printStackTrace(); } }
```

Pipes
Piped*Reader & PipedWriter*

```
public class PSWriter implements Runnable {
  private PipedWriter writer;
  public PSWriter(PipedWriter writer) {
    this.writer = writer;
  }
  @Override
  public void run() {
    int i = 0;
    try { while (true) {
      writer.write("I:"+i+"\n"); i++; writer.flush();
     Thread.sleep(1000); }
    } catch (Exception e)
    { System.out.println(" PipeThread Exception: " + e); }
  }
}
```

Dans le *thread* écrivant les données, nous faisons un boucle infinie pour écrire un compteur qui s'incrémente à chaque tour.

La méthode ***write*** écrit les données dans le ***pipe***.

Pipes
Piped*Reader & PipedWriter*

```
public class PSReader implements Runnable {
  private PipedReader reader;
  public PSReader(PipedReader reader) {
    this.reader = reader;
  }
  @Override
  public void run() {
    try { while (true) {
      char c = (char) reader.read();
      if (c != -1) {
        System.out.print(c);
       }
     }
    } catch (Exception e)
    { System.out.println(" PipeThread Exception: " + e); }
  }
}
```

Dans le *thread* lisant les données, nous faisons un boucle infinie pour lire les données provenant du *thread* écrivant les donnée

La méthode ***read*** lit un caractère dans le ***pipe***.