Les entrées / sorties en Java

Licence DANT

PITTON Olivier

Les entrées / sorties

- Les entrées sorties sont un ensemble d’actions qui servent à recevoir et produire des données quelque soit le type d’interaction

Les entrées / sorties

- Entrer / consulter une valeur
- Lire / écrire dans un fichier
- Envoyer / recevoir un message

Les entrées / sorties

- Les flux encapsulent l'envoi et la réception de données
- Le support sous-jacent est masqué
- Les données sont traitées séquentiellement

Les entrées / sorties

Java gère les I/O selon deux types de classes :

- Le sens du flux et son type de données

- La fonctionnalité (compression, serialization …)

Les types de flux

- Il existe deux types de flux :
- Flux d’octets : InputStream / OutputStream
- Flux de caractères : Reader / Writer

Les types de flux : InputStream

- int read() □ Lit 1 byte
- int read(byte[] b) □ Remplit le tableau d'octet
- long skip(long n) □ Saute n octets
- int available() □Le nombre d'octets disponibles
- void close() □ Ferme le flux et libère les ressources

Les types de flux : OutputStream

- void write(byte b) Ecrit 1 octet
- void write(byte[] b) Ecrit le tableau d'octet
- void flush() Vide les buffers et force l'écriture
- void close() Ferme le flux et libère les ressources

Exemple : InputStream

```
public void readFully(InputStream is) throws Exception {
 int b;
 while((b = is.read()) != -1) {
  System.out.println(b);
 }
}
```

Exemple : OutputStream

```
public void writeString(OutputStream os, String val)
  throws Exception {
 os.write(val.getBytes());
 os.flush();
}
```

Comment trouver des I/O ?

- En Java, la nomenclature est : <Type><I/O>Stream

- FileInputStream / FileOutputStream
- SocketInputStream / SocketOutputStream
- ByteArrayInputStream / ByteArrayOutputStream
- GzipInputStream / GzipOutputStream

Ecrire dans un fichier

```
public void writeFile(String filename, String content)
  throws Exception {
 FileOutputStream fos = new FileOutputStream(filename);
 fos.write(content.getBytes());
 fos.close();
}
```

Lire un fichier

```
public byte[] readFile(String filename) throws Exception {
 FileInputStream fis = new FileInputStream(filename);
 byte[] b = new byte[fis.available()];
 fis.read(b);
 fis.close();
 return b;
}
```

Les I/O par défaut

- System.out Ecrit dans la sortie standard
- System.err Ecrit dans la sortie d’erreur
- System.in Lit depuis l’entrée du process
- Utilisation du PrintStream

Les filtres

- Java utilise des filtres en tant que décorateurs
- Un décorateur nécessite un autre flux I/O
- Doit se lire / écrire dans le même sens

GZIP un fichier

```
public void writeFile(String filename, String content)
  throws Exception {

 GZIPOutputStream fos = new GZIPOutputStream(
    new FileOutputStream(filename));

fos.write(content.getBytes());
 fos.close();
}
```

Lire un fichier GZIP

```
public byte[] readFile(String filename) throws Exception {

 GZIPInputStream fis = new GZIPInputStream(
   new FileInputStream(filename));

 byte[] b = new byte[fis.available()];
 fis.read(b);
 fis.close();
 return b;
}
```

Accélérer les entrées / sorties

- Buffer pour ne pas lire octet par octet
- Utilisation des classes BufferedInputStream / BufferedOutputStream
- Définition d'une taille de buffer dans le constructeur

La sérialisation

- La sérialisation consiste à lire / écrire un objet Java qui implémente l'interface Serializable
- Utilisation de ObjectInputStream / ObjectOutputStream
- Que se passe t'il en cas de récursivité ? A □ B □ A

Ecrire un objet Java

```
public void writeFile(String filename,
                      ArrayList<String> content)
   throws Exception {
 ObjectOutputStream fos = new ObjectOutputStream(
   new FileOutputStream(filename));

 fos.writeObject(content);
 fos.close();
}
```

Lire un objet Java

```
public List<String> readFile(String filename)
  throws Exception {
 ObjectInputStream fis = new ObjectInputStream(
   new FileInputStream(filename));

 List<String> list = (ArrayList<String>) fis.readObject();
 fis.close();
 return list;
}
```

Créer son propre filtre

- Créer deux classes héritant de :

- FilterInputStream

- FilterOutputStream

SumInputStream

```
public class SumInputStream extends FilterInputStream {

 private final int sum;

 public SumInputStream(InputStream in, int sum) {
  super(in);
  this.sum = sum;
 }

 @Override
 public int read() throws IOException {
  int val = super.read();
  return sum + val;
 }
}
```

NIO2

- Depuis Java 7, des nouvelles API I/O ont été introduites

- Support des liens symboliques / physiques
- Parcours de répertoire avec filtre
- Gestion des attributs RWX POSIX
- WatchService
- Copie / déplacement de fichiers

Classes de NIO2

- Path : encapsule un chemin dans le système de fichiers
- Files : contient des méthodes statiques pour manipuler les éléments du système de fichiers
- FileSystemProvider : service provider qui interagit avec le système de fichiers sous-jacent
- FileSystem : encapsule un système de fichiers
- FileSystems : fabrique qui permet de créer une instance de FileSystem

L’interface Path

- Représente un chemin (fichier, repertoire, lien …)
- Est Immuable, Comparable, Watchable, Iterable
- Utilisation de Paths pour créer un chemin

Création de Path

```
Path chemin1 = Paths.get("app/monfichier.txt");

Path chemin2 = Paths.get(
  URI.create("file:///app/monfichier.txt"));

Path chemin3 = Paths.get(System.getProperty("java.io.tmpdir"),
  "monfichier.txt");
```

Classe Files

- Possède plein de méthodes utilitaires static
- Création (repertoire, fichier …)
- Manipulation (delete, copy, move …)
- Type (est un fichier, est un repertoire, …)
- Métadonnées (permission …)

Quelques méthodes de Files

```
Path path = Paths.get("source.txt");
byte[] content = Files.readAllBytes(path);
List<String> lines = Files.readAllLines(path);
Files.write(path, "salut".getBytes());
Files.copy(path, Paths.get("destination.txt"));
Files.move(path, Paths.get("destination.txt"));
Files.createDirectory(Paths.get("dir"));
Files.createFile(Paths.get("newfile.txt"));
```

Parcourir un répertoire

```
Path directory = Paths.get("directory");
try (DirectoryStream<Path> stream =
    Files.newDirectoryStream(directory)) {

Iterator<Path> iterator = stream.iterator();
 while(iterator.hasNext()) {
  Path p = iterator.next();
  System.out.println(p);
 }
}
```

Conclusion

- Les flux Java ont été fait de manière simple et élégante
- Les nouvelles API facilitent les I/O
- Encore plein d'autres à découvrir (asynchrone, channel, …)