INSA
CENTRE VAL DE LOIRE

POO Avancée : Java

Entrées / Sorties

Salwa SOUAF

Outline

IO Simples

Un fichier peut-être lu/écrit en utilisant différentes classes de Java. Cela permet d'accéder aux données du plus bas au plus haut niveau, suivant que le développeur nécessite de comprendre les données. Au niveau le plus bas, la lecture peut se faire au niveau de l'octet. Dans ce cas, le fichier est vu comme un flux d'octets. Une lecture s'arrête quand l'opération read() renvoie -1.

```
FileInputStream in = null;
FileOutputStream out = null;
try { in = new FileInputStream("xanadu.txt");
      out = new FileOutputStream("outagain.txt");
      int c;
      while ((c = in.read()) != -1) {
          out.write(c);
      } catch ...
```

De manière très similaire, un fichier peut-être vu comme un flux de caractère. La différence réside dans la gestion de l'internationalisation: le caractère, stocké en Unicode, est projeté dans le jeu de caractère local.

```
FileReader inputStream = new FileReader("xanadu.txt");
FileWriter outputStream = new FileWriter("characteroutput.txt");
// code identique au précédent pour lecture/écriture
```

IO Simples

Entrées/sorties ligne et bufferisée

Juste après le découpage caractère par caractère vient naturellement l'accès par ligne au contenu d'un fichier. A partir des objets de type FileReader qui fournit le flux de caractères, la classe BufferedReader agrège le flux en le découpant à chaque carriage-return ("r"), ou line-feed ("n"). La taille du buffer peut être précisée à la construction.

```
BufferedReader inputStream = null;
PrintWriter outputStream = null;

try { inputStream = new BufferedReader(new FileReader("xanadu.txt"));
      outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

      String l;
      while ((l = inputStream.readLine()) != null) {
          outputStream.println(l);
    }
```

Cette implémentation est plus efficace en terme de performance grâce à l'utilisation du buffer. Le nombre d'appels à l'API native de lecture est réduit puisque la lecture s'opère depuis une zone mémoire, le buffer, qui n'est raffraichit que lorsque ce buffer est vide. Lors de l'écriture, le buffer est flushé automatiquement et peut être forcé par un appel à flush(). Dans le cas bufferisé ou non, les classes de lecture héritent de Reader et implèmentent Readable. La contrainte minimum de l'interface est très faible: il faut implémenter int read(CharBuffer cb).

IO Simples

Entrées / Sorties formattées

Les entrées sorties formattées permettent de découper le flux en mots projettés directement dans le type adéquat. Dans un premier temps, le formattage peut-être basé sur la classe String avec comme séparateur l'espace. La classe Scanner fournit l'implémentation de ce principe en travaillant par exemple sur un flux bufferisé qui se parcourt alors comme un itérateur:

```
Scanner s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));
while (s.hasNext()) {
    System.out.println(s.next());
}
```

Si le type du contenu est connu, des primitives permettent de projeter directement le mot dans la classe voulue, sauf usi le prochain mot ne peut être interprété dans le type voulu:

```
s.useLocale(Locale.FRANCE);
while (s.hasNext()) {
    if (s.hasNextDouble()) {
            sum += s.nextDouble();
        } else { s.next(); }}
```

D'autres méthodes de Scanner permettent de manipuler le flux formattée:

- String findInLine(String pattern): trouve le pattern spécifié
- public Scanner skip(Pattern pattern): positionne le scanner sur le pattern
- Scanner reset(), Scanner useDelimiter(Pattern pattern), ...

IO Simples

Interactions avec les entrées et sorties standards

Les 3 entrées/sorties standards des systèmes POSIX sont accessible depuis la machine virtuelle au travers de la classe System: • System.in: entrée standard (implémente OutputStream) • System.out: sortie standard (implémente OutputStream) • System.err: sortie d'erreur standard (impléente InputStream) On peut y accéder en utilisant les classes OutputStreamWriter et InputStreamReader. Une alternative intéressante pour intéragir avec la console est l'utilisation de la classe Console. Elle fournit notamment une implémentation plus sûre pour la lecture de mot de passe.

```
Console c = System.console();
if (c == null) {
        System.err.println("No console.");
        System.exit(1); }
char [] p = c.readPassword("Enter your password: ");
```

L'affichage des caractères est désactivé dans la console. De plus, l'implémentation de la récupération du mot de passe comme un tableau permet de désallouer plus rapidemment que s'il s'agissait d'une String. Si le programme n'est pas attaché à une console ou qu'il n'est pas en intéractivité avec une console, l'objet renvoyé est un pointeur null.

IO Simples

Flux de données: vers la sérialisation

Les types simples supportent une projection directe de leur représentation dans un fichier. Les classes à utiliser sont DataInputStream et DataOutputStream. La classe String est elle aussi concernée car son implémentation est spéciale. L'exemple suivant montre comment écrire et relire des types simples:

```
static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = { "Java T-shirt", "Java Mug" };

DataOutputStream out = new DataOutputStream(
                  new BufferedOutputStream(new FileOutputStream(dataFile)));
for (int i = 0; i < prices.length; i ++) {
  out.writeDouble(prices[i]);
  out.writeInt(units[i]);
  out.writeUTF(descs[i]);
}
DataInputStream  in = new DataInputStream(
                  new BufferedInputStream(new FileInputStream(dataFile)));

double price; int unit; String desc; double total = 0.0;

try { while (true) {
          price = in.readDouble();
          unit = in.readInt();
          desc = in.readUTF();
          System.out.format("You ordered %d units of %s at $%.2f%n",
                  unit, desc, price);
          total += unit * price;
```

Outline

Sérialisation

Les objets peuvent être écrits directement dans des fichiers de donnée: on appelle cela le processus de sérialisation. Un objet est sérialisable s'il implémente l'interface Serializable. Aucune méthode n'est à implémenter: il s'agit d'une sorte de flag signalant que l'objet peut être sérialisé. Dans l'exemple ci-dessous, une instance de Maison peut être sérialisée:

```
 /**
 * Une classe agrégé qui est attribut de PersonneIO.
 */
package io;

import java.io.Serializable;

/**
 * Important: doit lui aussi être Serializable sous peine de:
 * java.io.NotSerializableException: io.Maison
 * lors de la serialisation de PersonneIO.
 */
public class Maison implements Serializable {
  public String adresse;
  public Maison(String adresse) {
    this.adresse = adresse;
}}
```

Sérialisation: dépendances

L'intérêt de la sérialisation est d'automatiser l'écriture des dépendances de classes. L'écriture d'une classe agrégant plusieurs objets provoque l'écriture des objets agrégés. Dans l'exemple suivant, l'instance de Maison est écrit automatiquement dans le fichier lorsque l'instance de PersonneIO est sérialisée.

```java
/** Classe qui va être sérialisée. */
package io;
import java.io.Serializable;

public class PersonneIO implements Serializable {

  private int num_compte_bancaire = 8878;
  public String nom = "none";
  public Maison m;
  public PersonneIO() {
    m = new Maison("Mehun");
  }
  protected void setNom(String n) {
    nom = n;
  }
  public void setAdresse(String a) {
    this.m.adresse = a;
  }
}
```

Sérialisation

Sérialisation: écriture

Lors du processus d'écriture, les objets sont sérialisés à l'aide de la classe ObjectOutputStream. Un identifiant unique de sérialisation est calculé à la compilation: chaque instance sérialisée posséde cet identifiant. Cela empêche le chargement d'un objet sérialisé ne correspondant plus à une nouvelle version de la classe correspondante.

```
package io;
import java.io.FileNotFoundException; import java.io.FileOutputStream;
import java.io.IOException; import java.io.ObjectOutputStream;
import java.io.Serializable;
public class PersonneIOMainWrite implements Serializable {
  public static void main(String[] args) {
    PersonneIO p = new PersonneIO();
    p.setNom("JFL");
    PersonneIO p2 = new PersonneIO();
    p2.setAdresse("?");
    FileOutputStream fos;
    try {   fos = new FileOutputStream("file.out");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(p);
      oos.writeObject(p2);
      oos.close(); }
      catch (FileNotFoundException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace();
      }}}
```

Sérialisation: lecture

A la relecture des objets sérialisés, il est impératif de connaitre l'ordre de sérialisation (on peut toutefois s'en sortir en faisant de l'introspection). L'exception particulière qu peut être levée est ClassNotFoundException dans le cas ou la JVM ne trouve pas la définition de la classe à charger.

```
package io;
import java.io.FileInputStream; import java.io.FileNotFoundException;
import java.io.IOException; import java.io.ObjectInputStream;
import java.io.Serializable;
public class PersonneIOMainRead implements Serializable {
  public static void main(String[] args) {
try { FileInputStream is = new FileInputStream("file.out");
      ObjectInputStream in = new ObjectInputStream(is);
      PersonneIO p = (PersonneIO)in.readObject();
      PersonneIO p2 = (PersonneIO)in.readObject();
      System.out.println(p.nom + " habite " + p.m.adresse);
      System.out.println(p2.nom + " habite " + p2.m.adresse);
      in.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }}}
```

Outline

SAX

Il existe de nombreuses librairies de manipulation de fichiers XML. Les deux grandes familles de parseurs sont les parseurs évenementiels et les parseurs à parcours d'arbres. Les parseurs évenementiels déclenchent des callbacks lorsque les balises sont rencontrées. Le code de ces callbacks sont dans un handler, c'est à dire une classe à part qui surcharge le handler par défaut. Le parseur le plus connu est SAX qui signifie Simple API for XML. Les parseurs à parcours d'arbres sont basés sur un parseur SAX. Ils instancient une représentation objet du document, ce qui est plus long en temps et plus couteux en mémoire. Néanmoins, le parcours d'un tel arbre se révèle très pratique à l'utilisation. En Java, un parseur SAX s'instancie de la sorte:

```
public class ParserXML {
  public static void main(String[] args)
    throws ParserConfigurationException, SAXException, IOException {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser parser = factory.newSAXParser();

    DefaultHandler handler = new XMLHandler(); // mon handler
    parser.parse(new File("io/exemple.xml"),handler);
    }}
```

SAX

Handler pour SAX

Un handler recoit les évenements liés au parsing du document. Il ne s'agit pas réellement d'évenements mais tout simplement du fait que les méthodes du handler son appellées par le parser lorsqu'un évenement du type "je rencontre une balise" se produit. Plusieurs méthodes de DefaultHandler peuvent être surchargées, notamment à la rencontre d'une balise, fermeture d'une balise ou lors de la détection d'une erreur. Au moment de la récupération d'une balise, ses attributs et leurs valuers peuvent être récupérés dans un objet de type Attributes.

```
package io;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class XMLHandler extends DefaultHandler {

    public void startDocument() throws SAXException {
      System.out.println("Ca démarre !");
    }
    public void startElement(String uri, String localName,
                             String qName, Attributes attributes)
                             throws SAXException {
      System.out.println("qname = " + qname);
        System.out.println("Attribut: " + attributes.getValue(0));
}}
```

Outline

Java NIO 2

Deux paquets ont été spécialement développés pour intéragir avec le filesystem: java.nio.file et java.nio.file.attribute. Ce sont deux sous packages du package java.nio, nio signifiant New Input/Output. La classe la plus importante dans ces packages est la classe Path qui fournit des services de manipulation d'arborescence de répertoires et de fichiers. Ils sont disponibles à partir de Java 1.7. La difficulté de l'implémentation de ces classes réside dans la portabilité de la machine virtuelle: il faut être capable de gérer des systèmes de fichiers différents sur des operating systems différents.

Path

La classe Path permet de manipuler un fichier ou une arborescence sous la forme d'un objet. La classe Paths fournit une méthode statique pour construire des objets Path à partir d'une String:

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get("file:///Users/joe/FileTest.java");
```

C'est en fait un raccourci vers l'objet FileSystems:

```
Path p4 = FileSystems.getDefault().getPath("/home/jf/java.rst");
```

Le constructeur de Path même s'il existe, est assez inutile puisqu'il ne raccroche pas l'objet instancié au système de fichier. L'objet instancié contient ensuite une représentation d'un chemin, de sa racine jusqu'au fichier ou au répertoire finaux. Cette représentation permet alors d'apeller les méthodes suivantes qui retournent toutes des objets de type Path:

- La méthode Path getName(n) permet de récupérer le n-ème élément du chemin, e.g. "jf" pour p4 avec n=1.
- La méthode Path getParent() renvoie le répertoire parent du chemin, e.g. /home pour p4..
- La méthode Path getRoot() renvoie la racine du chemin.

Dépendances avec l'OS

Certaines méthodes peuvent dépendre du système d'exploitation et de l'encodage de certaines informations pour le système de fichier. Par exemple, la méthode isHidden() se base sur la présence d'un . au début du nom de fichier pour les systèmes GNU/Linux alors que Microsoft Windows stocke cette information dans un fichier. Dans l'exemple suivant montre les résultats obtenus sur Path "sally/bar" sous Solaris et Windows:

Method Invoked	Returns in the Solaris OS	Returns in Microsoft Windows
toString	sally/bar	sally\bar
getName	bar	bar
getName(0)	sally	sally
getNameCount	2	2
subpath(0,1)	sally	sally
getParent	sally	sally
getRoot	null	null
isHidden	false	false

Java NIO 2

Normalisation, conversion, concatenation

Un chemin peut contenir des parties comportant de "." ou des "..", notamment si on les construit dynamiquement par concaténation. On peut donc par exemple se retrouver avec un chemin instancié sous la forme "/home/jf/.", ce qui n'est pas très élégant. La méthode normalize() permet de simplifier la représentation du chemin, sans s'occuper de l'existence réelle du fichier dénommé. Une autre méthode permet de récupérer un chemin absolu: toAbsolutePath(). Cette méthode est particulièrement utile lorsque la méthode getRoot() renvoie un pointeur null (le chemin est relatif). La méthode toRealPath(b) combine les effets précédents: il est simplifié, puis si le chemin est relatif, il est converti, et si b est true les liens symboliques sont résolus. Deux chemins peuvent être concaténés à l'aide de la méthode resolve():

```
Path p1 = Paths.get("/home/joe/foo");          // Solaris
System.out.format("%s%n", p1.resolve("bar"));  // Result is /home/joe/foo/bar
```

A l'inverse, un chemin relatif peut être construit entre 2 chemins:

```
Path p1 = Paths.get("home");
Path p3 = Paths.get("home/sally/bar");
Path p1_to_p3 = p1.relativize(p3);  // Result is sally/bar
Path p3_to_p1 = p3.relativize(p1);  // Result is ../..
```

Java NIO 2

Parcourt, comparaison

Un chemin se parcourt en utilisant l'interface Iterable. L'itérateur parcourt le chemin à partir de la racine. De plus, un chemin implémente l'interface Comparable ce qui permet de trier des ensembles de chemins. Souvent, on ne compare par l'égalité mais si un chemin est un sous chemin d'un autre. Deux méthodes sont proposées: startsWith(beginning) et endsWith(ending).

```
Path path = ...;
for (Path name: path) {
  System.out.println(name);
}

Path path = ...;
Path otherPath = ...;
Path beginning = Paths.get("/home");
Path ending = Paths.get("foo");

if (path.equals(otherPath)) {
//equality logic here
} else if (path.startsWith(beginning)) {
//path begins with "/home"
} else if (path.endsWith(ending)) {
//path ends with "foo"
}
```

Vérification

La primitive checkAccess(AccessMode... modes) permet de vérifier si un certains nombres de droits sont présent sur le fichier dénommé par l'objet de type Path:

```
Path file = ...;
try {
  file.checkAccess(READ, EXECUTE);
  ...
} catch (IOException x) {
//Logic for error condition...
return;
}
```

dénomment le même fichier sur le système de fichier (incluant la prise en compte des liens symboliques le cas échéant):

```
Path p1 = ...;
Path p2 = ...;
try {
if (p1.isSameFile(p2)) {
//Logic when the paths locate the same file
}} catch (IOException x) {
//Logic for error condition...
}
```

Opérations sur les fichiers

Un fichier se déplace à l'aide de moveTo(Path target, CopyOption... options), la copie est réalisée par un méthode similaire: copyTo(Path target, CopyOption... options), la suppression avec delete():

```
Path path = ...;
Path newPath = ...;
path.moveTo(newPath, REPLACE_EXISTING);
Path newPath2 = ...;
newPath.copyTo(newPath2, REPLACE_EXISTING, COPY_ATTRIBUTES);
newPath2.delete();
```

Une nouvelle primitive permet de créer un fichier: createFile(). Pour les entrées/sorties, on utilise les classes vues précédemment à partir de l'InputStream récupéré sur le Path:

```
Path file = ...;
try {
file.createFile();
in = file.newInputStream();
BufferedReader reader = new BufferedReader(
                        new InputStreamReader(in));
OutputStream out = new BufferedOutputStream(
                    file.newOutputStream(CREATE, APPEND));
```

Attributs d'un fichier

Les attributs d'un fichier peuvent être lus à travers la classe Attributes du package java.nio.file.attribute. Un appel statique à la méthode readBasicFileAttributes ou les méthodes spécifiques readDosFileAttributes et readPosixFileAttributes.

```
import java.nio.file.attribute.*;
public class WindowsAttributePrinter {
public static void main(String args) throws IOException {
  for (String name : args) {
    Path p = Path.get(name);
    DosFileAttributes attrs =
              Attributes.readDosFileAttributes(path, false);
    if (attrs.isArchive()) {
      System.out.println(name + " is backed up.");
    }
    if (attrs.isReadOnly()) {
      System.out.println(name + " is read-only.");
    }
    if (attrs.isHidden()) {
      System.out.println(name + " is hidden.");
    }
    if (attrs.isSystem()) {
      System.out.println(name + " is a system file.");
  }}
```

Java NIO 2

La classe FileVisitor

L'interface FileVisitor<T> permet de parcourir une arborescence de répertoire ainsi que tous les fichiers contenus dans cette arborescence. Pensé comme un parseur SAX, le programmeur doit surcharger les méthodes suivantes qui sont déclenchées au cours de la visite:

- preVisitDirectory(T): appellé avant chaque visite d'un répertoire
- preVisitDirectoryFailed(T, IOException): invoqué en cas de visite impossible
- postVisitDirectory(T,IOException): appellé après chaque visite d'un répertoire
- visitFile et visitFileFailed*: appellé après chaque visite de fichier

```
import static java.nio.file.FileVisitResult.*;
public static class PrintFiles extends SimpleFileVisitor<Path> {
  //Print information about each type of file.
  public FileVisitResult visitFile(Path file, BasicFileAttributes attr) {
      if (attr.isSymbolicLink()) {
          System.out.format("Symbolic link: %s ", file);
      } else if (attr.isRegularFile()) {
          System.out.format("Regular file: %s ", file);
      }
      return CONTINUE;
  }
  public FileVisitResult preVisitDirectoryFailed(Path d, IOException e) {
      System.err.println(e);
      return CONTINUE;
  }}
```

Java NIO 2

Démarrage de la visite

Le lancement du parcourt de l'arborescence s'effectue à l'appel de Files.walkFileTree. Deux signatures différentes sont disponibles:

```
walkFileTree(Path start, FileVisitor<? super Path> visitor)
walkFileTree(Path start, Set<FileVisitOption> options, int maxDepth,
             FileVisitor<? super Path> visitor)
```

Dans les deux cas, le chemin de départ est donné ainsi que l'objet dérivant de FileVisitor. Dans la deuxième signature des options parmi FOLLOW_LINKS et DETECT_CYCLES et la profondeur maximum peuvent être précisés. La valeur retournée est particulièrement importante: elle est choisie parmi CONTINUE, TERMINATE, SKIP_SUBTREE , SKIP_SIBLINGS (abandon du répertoire courant et de ses frères).

```
Path startingDir = ...;
PrintFiles pf = new PrintFiles();
Files.walkFileTree(startingDir, pf);
EnumSet<FileVisitOption> opts = EnumSet.of(FOLLOW_LINKS);
Finder finder = new Finder(pattern);
Files.walkFileTree(startingDir, opts, Integer.MAX_VALUE, finder);
```

Le comportement du FileVisitor n'est pas spécifié. Dans la machine virtuelle de Sun, le parcourt est en profondeur. Il est donc particulièrement important d'effectuer la bonne action dans la bonne méthode, par exemple si l'on encode une suppression

Java NIO 2

Surveiller un répertoire

Surveiller un répertoire permet de réagir à la modification du système de fichier. Un service de surveillance WatchService doit être créé et associé à un Path:

```
WatchService watcher = FileSystems.getDefault().newWatchService();
Path dir = ...;
try {
  WatchKey key = dir.register(watcher, ENTRY_CREATE, ENTRY_DELETE,
                              ENTRY_MODIFY);
} catch (IOException x) {
  System.err.println(x);
}
```

A partir de cette clef, les évenements sont récupérés à l'aide de la méthode pollEvents(). Ils peuvent ensuite être filtrés par types ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY, OVERFLOW.

```
for (WatchEvent<?> event: key.pollEvents()) {
  WatchEvent.Kind<?> kind = event.kind();
  if (kind == ENTRY_CREATE) {
```

```
    WatchEvent<Path> ev = (WatchEvent<Path>)event;
    Path filename = ev.context();
  }
}
```