ISEN ALL IS DIGITAL! OUEST | yncréa

Langage Java

Manipulation de fichiers

Classes `java.io.File` et `java.nio.file.Path`

- L'interface `Path` représente un chemin sur le disque, l'implémentation réelle dépendra du système d'exploitation

  ⇒ **Unix/MacOS** : `path.getFileSystem().getSeparator() == "/"`

  ⇒ **Windows** : `path.getFileSystem().getSeparator() == "\"`

- La classe `File` représente un fichier ou un dossier

- Exemples :

```
// Chemin vers le fichier "D:\file.txt"
Path path = FileSystems.getDefault().getPath("D:", "file.txt");
File file = path.toFile(); // conversion File->Path
Path path = file.toPath(); // conversion Path->File
boolean fileExists = file.exists(); // true si le fichier existe
boolean isDir = file.isDirectory(); // false
```

Classe `java.nio.file.Files`

- La classe `Files` fournit des utilitaires de lecture/écriture pour les fichiers texte
- Plus récente que `java.io.File`
- Propose des services équivalents à la classe `File`
- Exemples :

```
// Chemin vers le fichier "D:\file.txt"
Path path = FileSystems.getDefault().getPath("D:", "file.txt");

boolean exists = Files.exists(path);
boolean isDir = Files.isDirectory(path);
boolean isReadable = Files.isReadable(path);
boolean isWritable = Files.isWritable(path);
```

Classe `java.nio.file.Files`

- Pour lire un fichier entièrement (**attention à la mémoire**) :
  - `Files.readString(Path path) : String`
  - `Files.readAllLines(Path path) : List<String>`
  - `Files.readAllBytes(Path path) : byte[]`

- Pour lire un fichier progressivement, ligne par ligne :
  - `Files.lines(Path path) : Stream<String>`

Classe java.nio.file.Files

- Pour écrire un fichier :
  - `writeString(Path path, String csq, OpenOption... options)`
  - `write(Path path, byte[] bytes, OpenOption... options)`
  - `write(Path path, Iterable<? extends CharSequence> lines, OpenOption... options)`
- Autres opérations :
  - `delete(Path path)`
  - `copy(Path from, Path to, CopyOptions... opts)`
  - `move(Path from, Path to, CopyOptions... opts)`
  - `createDirectory(Path dir, FileAttribute<?>... attrs)`
  - `createTempDirectory(String prefix, FileAttribute<?>... attrs) : Path`
  - `createTempFile(String prefix, String suffix, FileAttribute<?>... attrs) : Path`

Classe java.nio.file.Files

- Exemple : copie d'un fichier

```
public static void main(String[] args) throws IOException
{
    // Lecture d'un fichier source et recopie ligne par ligne
    // vers un fichier destination
    Path src = Paths.get("D:", "file.txt");
    Path dest = Paths.get("D:", "file_copy.txt");

    Files.lines(src).forEach(line -> { // lecture du fichier source
        try
        {
            Files.writeString(dest, line, StandardOpenOption.CREATE_NEW); // écriture
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    });
}
```

Manipulation d'images

Classe `javax.imageio.ImageIO`

- Classe fournissant un grand nombre de méthodes utilitaires pour la lecture et l'écriture des images
- Formats supportés (lien) :

Standard image format plug-ins

Image format	Reading	Writing	Notes	Metadata
BMP	yes	yes	none	BMP metadata format
GIF	yes	yes	GIF plug-in notes	GIF metadata format
JPEG	yes	yes	none	JPEG metadata format
PNG	yes	yes	none	PNG metadata format
TIFF	yes	yes	TIFF plug-in notes	TIFF metadata format
WBMP	yes	yes	none	WBMP metadata format

- Méthodes (parmi d'autres, voir javadoc) :
  - `read(File input) : BufferedImage`
  - `write(RenderedImage im, String formatName, File output)`

Classe javax.imageio.ImageIO

- Exemple : diviser la taille d'une image par 2 en hauteur/largeur

```
public static void main(String[] args) throws IOException
{
    BufferedImage originalImage = ImageIO.read(new File("myimage.jpg"));

    int width = originalImage.getWidth();
    int height = originalImage.getHeight();

    // Créer une nouvelle image avec la moitié de la largeur et de la hauteur
    BufferedImage processedImage = new BufferedImage(width / 2, height / 2, originalImage.getType());

    // Parcourir l'image originale et copier 1 pixel sur 2 en hauteur et en largeur
    for (int x = 0; x < width; x += 2)
    {
        for (int y = 0; y < height; y += 2)
        {
            int color = originalImage.getRGB(x, y); // 4 octets : Alpha Red Green Blue
            processedImage.setRGB(x / 2, y / 2, color);
        }
    }

    // Écrire le résultat dans un fichier de sortie
    ImageIO.write(processedImage, "jpg", new File("output.jpg"));
}
```

API historique
Reader

Classe `java.io.Reader`

- Classe abstraite qui permet de **décoder un flux d'octets** bruts et de le **transformer en un flux de données sémantiques**
- Le flux sous-jacent (`InputStream`) peut provenir de **n'importe quelle source** : fichier, socket réseau, mémoire…
- Le type des données en sortie dépend des classes concrètes : caractères, son, pixels, datagramme…
- Quelques implémentations concrètes :
  - FileReader : transforme les octets d'un fichier en texte
  - StringReader : lit simplement en mémoire une chaine de caractères déjà décodée
  - InputStreamReader : transforme les octets d'un flux sous-jacent en texte
  - BufferedReader : idem mais avec un tampon pour améliorer les performances
  - FilterReader : filtre les octets lus à partir d'une méthode fournie par l'utilisateur

Classe `java.io.FileReader`

- Classe concrète qui implémente l'interface `java.io.Reader`
- Permet de **décoder un flux d'octets** bruts à partir d'un `FileInputStream` et de le **transformer en un flux de caractères**
- À utiliser conjointement avec `BufferedReader`
- Exemple :

```
public static void main(String[] args) throws IOException
{
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader("D:\\file.txt"))) {

        String line;
        while ((line = bufferedReader.readLine()) != null) {
            // Traitement des lignes du fichier
        }
    }
}
```

Classe java.io.StringReader

- Classe à utiliser lorsque vous devez décoder non pas un fichier mais une chaîne de caractères qui est déjà en mémoire
- Cela peut être du texte dans un format particulier : CSV, JSON, XML, HTML, HTTP…etc.
- La plupart du temps, cette classe est utilisée par héritage
- On y ajoute les méthodes pertinentes pour lire les données exprimées dans le format ciblé
- Exemple pour le format CSV :

```
public class CsvReader extends StringReader {

  public String[] readLine() {...} ⇒ retourne les valeurs d'une ligne

}
```

Classe `java.io.InputStreamReader`

- Classe plus générique que `FileReader`
- Permet de gérer d'autres flux que des fichiers (ex. : sockets réseaux)
- Son but est le même que `FileReader` : décoder des octets bruts vers des données sémantiques
- Par défaut, **les octets sont décodés en caractères** provenant d'un jeu de caractères (UTF8 par défaut mais c'est modifiable dans le constructeur de `InputStreamReader`)

Classe java.io.InputStreamReader

- Exemple : décodage d'une chaîne de caractères reçue sur un socket

```
public static void main(String[] args) throws IOException
{
    Socket socket = acceptConnection();

    try (InputStream in = socket.getInputStream();
         InputStreamReader r = new InputStreamReader(in, StandardCharsets.UTF_8);
         BufferedReader reader = new BufferedReader(r)) {

        String line;
        while ((line = bufferedReader.readLine()) != null) {
            // Traitement des lignes du fichier
        }
    }
}
```

Classe `java.io.InputStreamReader`

- **Rappel** : pour lire un fichier on peut faire la même chose en **1 ligne** avec la méthode `Files.lines(Path, Charset)` donc n'utilisez pas `InputStreamReader` pour lire un fichier ;)

```
public static void main(String[] args) throws IOException
{
    Path path = Paths.get("D:", "file.txt");
    Files.lines(path, StandardCharsets.UTF_8).forEach(line -> {
        // Traitement à effectuer sur chaque ligne
    });
}
```

ISEN ALL IS DIGITAL! OUEST | yncréa

API historique
InputStream

Classe `java.io.InputStream`

- Classe permettant de simplement lire un flux d'octets bruts
- Elle est très générique et constitue la base de toute l'API de lecture de fichiers/sockets/console/mémoire...etc.
- Les méthodes principales sont :
  - `read()` ⇒ lit un octet et retourne un `int` compris entre 1 et 255
  - `read(byte[] buf)` ⇒ lit N octets et les stocke dans le buffer (N = `buf.length`)
  - `read(byte[] buf, int offset, int length)` ⇒ idem mais avec une sous-partie du buffer
- Dans les types concrets comme **`DataInputStream`** vous trouverez des méthodes permettant de lire des données de type `int`, `short`, `byte`, `char`...etc.

Classe `java.io.InputStream`

- Quelques implémentations concrètes :
  - FileInputStream,
  - DataInputStream,
  - BufferedInputStream,
  - ZipInputStream ...etc.

Classe java.io.FileInputStream

- Exemple d'utilisation pour lire un fichier texte :

```
public static void main(String[] args) throws IOException
{
    Path path = Paths.get("D:", "file.txt");

    try (FileInputStream in = new FileInputStream(path.toFile())){
        // La syntaxe "try-with-resource" permet de refermer automatiquement
        // le flux d'entrée lorsqu'on sort du bloc try/catch

        byte[] bytes = in.readAllBytes();
        String readableContent = new String(bytes);
        System.out.println(readableContent);
    }

}
```

Classe `java.io.BufferedInputStream`

- Identique à `InputStream` mais s'utilise "par-dessus" pour ajouter une capacité de "bufferisation"
- Les données sont lues par **paquets de 8192 octets** par défaut (cette taille est configurable via le constructeur `BufferedInputStream(InputStream in, int bufSize)`) et mises dans un *buffer*
- `read()` va ensuite lire dans le *buffer* au lieu de lire sur le disque
- Si le *buffer* est entièrement lu, lecture d'un nouveau paquet de données au prochain `read()`
- **Remarque** : pas très intéressant si vous lisez tout le fichier d'un seul coup mais permet d'économiser la mémoire pour lire des fichiers de plusieurs Giga octets

ISEN ALL IS DIGITAL! OUEST | yncréa

API historique
Writer + OutputStream

Classe `java.io.Writer`

- Classe complémentaire à `Reader`, basée sur les mêmes principes
- Un `Writer` transforme un flux de **caractères** encodés dans un certain jeu de caractères, en un flux d'**octets bruts**
- Les différentes implémentations d'un `Writer` permettent de traiter la particularité de chaque type de flux (fichier, socket…)
- Quelques implémentations :
  - `PrintWriter` et `FileWriter` (souvent utilisées ensemble)
  - `BufferedWriter` ⇒ améliore les perfs
  - `StringWriter` ⇒ simplement réécrire une chaine vers une autre

Classe `java.io.OutputStream`

- Classe complémentaire à `InputStream`, basée sur les mêmes principes
- Un `OutputStream` permet d'écrire un flux d'**octets bruts**
- Les différentes implémentations permettent de traiter la particularité de chaque type de flux (fichier, socket…)
- Quelques implémentations :
  - FileOutputStream + BufferedOutputStream ⇒ écrit dans un fichier
  - `DataOutputStream` ⇒ écriture de données numériques
  - `ObjectOutputStream` ⇒ pour la sérialisation d'objets Java
  - `ZipOutputStream` ⇒ écrit dans un fichier et compresse les données

Quelques remarques

- Le nom des classes est parfois trompeur ; certaines classes sont plus des Writers que des OutputStreams…
- Cette confusion rend la lecture de la documentation plus compliquée
- Pas de panique : si vous cherchez à écrire des données dans un format Xxxxx, recherchez d'abord un XxxxxWriter qui prend en charge ce format, si vous ne trouvez rien recherchez une classe XxxxxOutputStream

Exemples

- `PrintWriter` : écriture de texte

```
public static void main(String[] args) throws IOException
{
    String fileName = "output.txt";

    try (PrintWriter printWriter = new PrintWriter(new FileWriter(fileName))) {

        printWriter.println("Bonjour, le monde!");
        printWriter.println("C'est un exemple de PrintWriter.");
    }
}
```

Exemples

- `DataOutputStream` : écriture de données numériques binaires

```
public static void main(String[] args) throws IOException
{
    String fileName = "data.bin";

    try (FileOutputStream fos = new FileOutputStream(fileName);
         DataOutputStream dataOutputStream = new DataOutputStream(fos)) {

        int intValue = 42;
        double doubleValue = 3.14;
        char charValue = 'A';

        dataOutputStream.writeInt(intValue);
        dataOutputStream.writeDouble(doubleValue);
        dataOutputStream.writeChar(charValue);
    }
}
```

Exemples

- `ObjectOutputStream` : sérialisation d'un objet

```
public static void main(String[] args) throws IOException
{
    String fileName = "person.ser";

    // Crée un objet Person
    Person person = new Person("Alice", 30);

    try (FileOutputStream fis = new FileOutputStream(fileName);
         ObjectOutputStream objectOutputStream = new ObjectOutputStream(fis)) {

        // Ecrit l'objet dans le fichier
        objectOutputStream.writeObject(person);
    }
}
```

- **Note** : sérialiser = transformer un objet java résident en mémoire en un flux d'octets bruts

- Les objets doivent implémenter l'interface `Serializable` et redéfinir les méthodes `readObject()` et `writeObject()` (cf. javadoc)

Exemples

- `ZipOutputStream` : création d'une archive ZIP

```
public class ZipUtil {

    public static void zipFileOrDirectory(File fileOrDirectoryToZip) throws IOException {

        String zipFileName = "output.zip"; // Nom de l'archive zip de sortie

        try (FileOutputStream fos = new FileOutputStream(zipFileName);
             ZipOutputStream zipOut = new ZipOutputStream(fos)) {

            addFilesOrDirToZip(fileOrDirectoryToZip, fileOrDirectoryToZip.getName(), zipOut);
        }
    }

    private static void addFileOrDirToZip(File sourceFile, String entryName, ZipOutputStream zipOut) throws IOException {
        if (sourceFile.isDirectory()) {
            File[] files = sourceFile.listFiles();
            if (files != null) {
                for (File file : files) {
                    addFilesToZip(file, entryName + File.separator + file.getName(), zipOut);
                }
            }
        } else {
            FileInputStream fis = new FileInputStream(sourceFile);
            ZipEntry zipEntry = new ZipEntry(entryName);
            zipOut.putNextEntry(zipEntry);

            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = fis.read(buffer)) != -1) {
                zipOut.write(buffer, 0, bytesRead);
            }
            fis.close();
        }
    }
}
```

Questions ?