Getting started with Java

I/O en Java

Introduction au système de fichiers

- **Fichiers et répertoires** : Les fichiers contiennent des données et sont organisés en hiérarchies à l'aide de répertoires.
- **Systèmes de fichiers** : Différents OS utilisent différents systèmes de fichiers (ex : NTFS, ext4).
- **Root** : Point de départ de la hiérarchie.
  - Windows : `C:\`
  - Unix/Linux : `/`
- **Java** : La JVM se connecte automatiquement au système de fichiers local pour fournir des opérations multi-plateformes.

Chemins absolus et relatifs

- **Chemin absolu** : Inclut le chemin complet depuis la racine jusqu'au fichier ou répertoire.
  - Exemple : `C:\dossier\fichier.txt`
- **Chemin relatif** : Défini par rapport au répertoire courant.
  - Exemple : `dossier\fichier.txt` (si le répertoire courant est `C:\`)

Astuce : Un chemin commençant par `/` (ou `C:\`) est absolu.

Symboles de chemin

Symbole	Description
`.`	Répertoire courant
`..`	Répertoire parent

Exemples :

- `../parent.txt` : Fichier `parent.txt` dans le répertoire parent.
- `./local.txt` : Fichier `local.txt` dans le répertoire courant.

Création d'un fichier avec `java.io.File`

- Constructeurs disponibles :

```
File fichier1 = new File("/chemin/complet/fichier.txt");
File fichier2 = new File("/chemin/complet", "fichier.txt");
File dossier = new File("/chemin/complet");
File fichier3 = new File(dossier, "fichier.txt");
System.out.println(fichier1.exists());
```

- Ces méthodes permettent de créer une référence, mais **pas encore de lire/écrire**.

Création d'un chemin avec `java.nio.file.Path`

- Utilisation de `Path.of` ou `Paths.get` :

```
Path chemin1 = Path.of("/chemin/complet/fichier.txt");
Path chemin2 = Path.of("/chemin", "complet", "fichier.txt");
Path chemin3 = Paths.get("/chemin/complet/fichier.txt");
Path chemin4 = Paths.get("/chemin", "complet", "fichier.txt");
System.out.println(Files.exists(chemin1));
```

- **Astuce** : `Path.of` (introduit en Java 11) et `Paths.get` sont interchangeables.

Conversion entre `File` et `Path`

- Conversion simplifiée :

```
File fichier = new File("document.txt");
Path chemin = fichier.toPath();
File fichierRetour = chemin.toFile();
```

- **Pourquoi ?** : Support des anciennes bibliothèques (`File`) tout en profitant des nouvelles fonctionnalités (`Path`).

Symbolic Links (Liens symboliques)

- **Définition** : Pointeur vers un autre fichier ou répertoire.
- Exemple :
  - Chemin réel : `/documents/rapport.doc`
  - Lien symbolique : `/alias/rapport.doc`
- Java NIO.2 prend en charge :
  - **Création**
  - **Détection**
  - **Navigation**

Méthodes pour créer des objets `File` et `Path`

Type	Déclaré dans	Méthode ou Constructeur
File	`File`	`public File(String pathname)`
		`public File(File parent, String child)`
		`public File(String parent, String child)`
File -> Path	`File`	`public Path toPath()`
Path -> File	`Path`	`public default File toFile()`

Exemple de hiérarchie de fichiers

```
C:\
 ├──── app
 │     ├──── animaux
 │     │     ├──── Tigre.java
 │     │     ├──── Tigre.class
 │     ├──── employes
 │     ├──── java.exe
 ├──── zoo
 │     ├──── info.txt
```

- **Chemin absolu** : `C:\app\animaux\Tigre.java`
- **Chemin relatif** (depuis `C:\app`) : `animaux\Tigre.java`

Points importants sur `I/O` et `NIO.2`

- `java.io.File` : Approche classique (ancienne).
- `java.nio.file.Path` : Approche moderne (NIO.2).
- **Différences principales** :
  - `File` : Simple mais limité.
  - `Path` : Plus complet, support des systèmes modernes et des liens symboliques.

Relations entre classes `I/O` et `NIO.2`

- `FileSystems` : Crée des instances de `FileSystem`.
- `Paths` : Fournit des méthodes pour créer des objets `Path`.
- `Files` : Utilitaire pour opérer sur des objets `Path`.

Utilisation des fonctionnalités partagées

De nombreuses opérations peuvent être effectuées à l'aide des bibliothèques I/O et NIO.2. Les tableaux ci-dessous répertorient les méthodes communes pour travailler avec des fichiers et des répertoires.

Opérations communes sur File et Path

Description	Méthode instance File (I/O)	Méthode instance Path (NIO.2)
Nom du fichier/répertoire	`getName()`	`getFileName()`
Répertoire parent ou null	`getParent()`	`getParent()`
Vérifie si le chemin est absolu	`isAbsolute()`	`isAbsolute()`

Opérations communes sur File et Files

Description	Méthode instance File (I/O)	Méthode statique Files (NIO.2)
Supprime un fichier/répertoire	`delete()`	`deleteIfExists(Path p)`
Vérifie si le fichier/répertoire existe	`exists()`	`exists(Path p, LinkOption... o)`
Chemin absolu	`getAbsolutePath()`	`toAbsolutePath()`
Est un répertoire	`isDirectory()`	`isDirectory(Path p, LinkOption... o)`

Exemple avec I/O

```
public static void io() {
    var file = new File("C:\\data\\zoo.txt");
    if (file.exists()) {
        System.out.println("Absolute Path: " + file.getAbsolutePath());
        System.out.println("Is Directory: " + file.isDirectory());
        System.out.println("Parent Path: " + file.getParent());
        if (file.isFile()) {
            System.out.println("Size: " + file.length());
            System.out.println("Last Modified: " + file.lastModified());
        } else {
            for (File subfile : file.listFiles()) {
                System.out.println("    " + subfile.getName());
            }
        }
    }
}
```

Exemple avec NIO.2

```
public static void nio() throws IOException {
    var path = Path.of("C:\\data\\zoo.txt");
    if (Files.exists(path)) {
        System.out.println("Absolute Path: " + path.toAbsolutePath());
        System.out.println("Is Directory: " + Files.isDirectory(path));
        System.out.println("Parent Path: " + path.getParent());
        if (Files.isRegularFile(path)) {
            System.out.println("Size: " + Files.size(path));
            System.out.println("Last Modified: " + Files.getLastModifiedTime(path));
        } else {
            try (Stream<Path> stream = Files.list(path)) {
                stream.forEach(p -> System.out.println("    " + p.getFileName()));
            }
        }
    }
}
```

Gestion des exceptions IOException

Les méthodes NIO.2 déclarent souvent `IOException`, qui peut être causée par :

- Perte de communication avec le système de fichiers.
- Accès ou modification impossibles d'un fichier ou répertoire.
- Fichier inexistant requis par l'opération.

Paramètres optionnels de NIO.2

Type d'énumération	Valeur d'énumération	Détails
`LinkOption`	`NOFOLLOW_LINKS`	Ne pas suivre les liens symboliques.
`StandardCopyOption`	`ATOMIC_MOVE`	Déplacement atomique du fichier.
	`COPY_ATTRIBUTES`	Copier les attributs existants.
	`REPLACE_EXISTING`	Remplacer si le fichier existe déjà.

Interagir avec les chemins NIO.2

Les chemins (`Path`) sont immuables

Comme les valeurs de type `String`, les instances de `Path` sont immuables. Dans l'exemple suivant, l'opération sur la ligne 2 est perdue, car `p` est immuable :

```
Path p = Path.of("whale");
p.resolve("krill");
System.out.println(p);  // whale
```

De nombreuses méthodes de l’interface Path transforment la valeur du chemin de manière à retourner un nouvel objet Path, permettant ainsi l’enchaînement des méthodes. Nous montrons l’enchaînement dans l’exemple suivant :

```
Path.of("/zoo/../home").getParent().normalize().toAbsolutePath();
```

Affichage du chemin

L’interface Path contient trois méthodes pour obtenir des informations de base sur la représentation du chemin. La méthode toString() retourne une représentation en String du chemin complet. C’est la seule méthode de l’interface Path qui retourne une String. Beaucoup d’autres méthodes de l’interface Path retournent des instances Path.

Les méthodes getNameCount() et getName() sont souvent utilisées ensemble pour obtenir le nombre d'éléments dans le chemin et une référence à chaque élément, respectivement. Ces deux méthodes ne prennent pas en compte le répertoire racine dans le chemin.

```
Path path = Paths.get("/land/hippo/harry.happy");
System.out.println("The Path Name is: " + path);
for (int i = 0; i < path.getNameCount(); i++)
   System.out.println("   Element " + i + " is: " + path.getName(i));
```

Notez que nous n’avons pas appelé explicitement toString() sur la ligne 2. En effet, Java appelle toString() sur tout objet lors de la concaténation de chaînes. Nous utilisons cette caractéristique dans tous les Exemples.

La sortie de ce code est la suivante :

```
Element 0 is: land
Element 1 is: hippo
Element 2 is: harry.happy
```

Même si c’est un chemin absolu, l’élément racine n’est pas inclus dans la liste des noms. Ces méthodes ne considèrent pas la partie racine du chemin.

```
var p = Path.of("/");
System.out.print(p.getNameCount()); // 0
System.out.print(p.getName(0));     // IllegalArgumentException
```

Notez que si vous essayez d’appeler getName() avec un index invalide, cela lancera une exception au moment de l’exécution.

Création d’une partie du chemin

L’interface Path inclut la méthode subpath() pour sélectionner des parties du chemin. Elle prend deux paramètres : un beginIndex inclusif et un endIndex exclusif. Cela devrait vous rappeler la méthode substring() de String, que vous avez vue dans le chapitre 4, “API de base”.

Le code suivant montre comment subpath() fonctionne. Nous imprimons également les éléments du chemin à l’aide de getName() afin de voir comment les indices sont utilisés.

```
var p = Paths.get("/mammal/omnivore/raccoon.image");
System.out.println("Path is: " + p);
for (int i = 0; i < p.getNameCount(); i++) {
   System.out.println("   Element " + i + " is: " + p.getName(i));
}
System.out.println();
System.out.println("subpath(0,3): " + p.subpath(0, 3));
System.out.println("subpath(1,2): " + p.subpath(1, 2));
System.out.println("subpath(1,3): " + p.subpath(1, 3));
```

```
Path is: /mammal/omnivore/raccoon.image
   Element 0 is: mammal
   Element 1 is: omnivore
   Element 2 is: raccoon.image

subpath(0,3): mammal/omnivore/raccoon.image
subpath(1,2): omnivore
subpath(1,3): omnivore/raccoon.image
```

Comme pour getNameCount() et getName(), subpath() est indexé à partir de zéro et ne prend pas en compte la racine. Tout comme getName(), subpath() lance une exception si des indices invalides sont fournis.

```
var q = p.subpath(0, 4); // IllegalArgumentException
var x = p.subpath(1, 1); // IllegalArgumentException
```

Le premier exemple lance une exception au moment de l'exécution, car la valeur maximale de l'index autorisé est 3. Le deuxième exemple lance une exception car les indices de début et de fin sont identiques, ce qui mène à une valeur de chemin vide.

Accès aux éléments du chemin

L’interface Path contient de nombreuses méthodes pour récupérer des éléments particuliers du chemin, retournés en tant qu’objets Path eux-mêmes. La méthode getFileName() retourne l’élément de fichier ou de répertoire courant, tandis que getParent() retourne le chemin complet du répertoire parent. La méthode getParent() retourne null si elle est appelée sur le chemin racine ou au sommet d’un chemin relatif. La méthode getRoot() retourne l’élément racine du fichier dans le système de fichiers ou null si le chemin est relatif.

Considérons la méthode suivante, qui imprime différents éléments du chemin :

```
public void printPathInformation(Path path) {
   System.out.println("Filename is: " + path.getFileName());
   System.out.println("   Root is: " + path.getRoot());
   Path currentParent = path;
   while ((currentParent = currentParent.getParent()) != null)
      System.out.println("   Current parent is: " + currentParent);
   System.out.println();
}
```

```
printPathInformation(Path.of("zoo"));
printPathInformation(Path.of("/zoo/armadillo/shells.txt"));
printPathInformation(Path.of("./armadillo/../shells.txt"));
```

Cet exemple produit la sortie suivante :

```
Filename is: zoo
   Root is: null

Filename is: shells.txt
   Root is: /
   Current parent is: /zoo/armadillo
   Current parent is: /zoo
   Current parent is: /

Filename is: shells.txt
   Root is: null
   Current parent is: ./armadillo/..
   Current parent is: ./armadillo
   Current parent is: .
```

En examinant la sortie de l’exemple, vous pouvez voir la différence de comportement de getRoot() sur les chemins absolus et relatifs. Vous pouvez également voir que les méthodes ne résolvent pas les symboles de chemin et les traitent comme une partie distincte du chemin.

Résolution des chemins

Si vous souhaitez concaténer des chemins de manière similaire à la concaténation de chaînes, la méthode resolve() offre des versions surchargées qui vous permettent de passer un paramètre Path ou String. L'objet sur lequel la méthode resolve() est invoquée devient la base du nouvel objet Path, avec l'argument d'entrée ajouté à la fin du chemin.

Voyons ce qui se passe si nous appliquons resolve() à un chemin absolu et à un chemin relatif :

```
Path path1 = Path.of("/cats/../panther");
Path path2 = Path.of("food");
System.out.println(path1.resolve(path2));
```

Le code génère la sortie suivante :

```
/cats/../panther/food
```

Comme les autres méthodes que nous avons vues, resolve() ne nettoie pas les symboles de chemin. Si le paramètre d'entrée à la méthode resolve() est un chemin absolu, la sortie serait la suivante :

```
Path path3 = Path.of("/turkey/food");
System.out.println(path3.resolve("/tiger/cage"));
```

La sortie serait :

```
/tiger/cage
```

En résumé, si un chemin absolu est fourni en entrée à la méthode, c'est la valeur retournée. Il n'est pas possible de combiner deux chemins absolus avec resolve().

Création d’un chemin relatif

L’interface Path inclut la méthode relativize() pour construire le chemin relatif entre deux chemins, souvent en utilisant des symboles de chemin. Que pensez-vous que les exemples suivants afficheront ?

```
var path1 = Path.of("fish.txt");
var path2 = Path.of("friendly/birds.txt");
System.out.println(path1.relativize(path2));
System.out.println(path2.relativize(path1));
```

Création de répertoires

Méthodes pour créer des répertoires :
- Files.createDirectory(Path dir) : crée un répertoire, lance une exception si déjà existant ou si les chemins parents n'existent pas.
- Files.createDirectories(Path dir) : crée le répertoire et tous les parents manquants.

Exemple :

```
Files.createDirectory(Path.of("/bison/field"));
Files.createDirectories(Path.of("/bison/field/pasture/green"));
```

Copie de fichiers

Méthode principale :

- Files.copy(Path source, Path target, CopyOption... options)

Exemple de copie de fichier :

```
Files.copy(Paths.get("/panda/bamboo.txt"), Paths.get("/panda-save/bamboo.txt"));
```

Remplacement lors de la copie

Option REPLACE_EXISTING :

- Remplace le fichier cible s’il existe déjà.
  Exemple :

```
Files.copy(Paths.get("book.txt"), Paths.get("movie.txt"), StandardCopyOption.REPLACE_EXISTING);
```

Déplacement ou renommage avec move()

Méthode :

- Files.move(Path source, Path target, CopyOption... options)
  Exemple:

```
Files.move(Path.of("C:\\zoo"), Path.of("C:\\zoo-new"));
Files.move(Path.of("C:\\user\\addresses.txt"), Path.of("C:\\zoo-new\\addresses2.txt"));
```

Suppression de fichiers et répertoires

Méthodes :

- Files.delete(Path path) : supprime un fichier ou répertoire vide, lance une exception si non vide.
- Files.deleteIfExists(Path path) : supprime s'il existe, retourne true si réussi.
  Exemples :

```
Files.delete(Paths.get("/vulture/feathers.txt"));
Files.deleteIfExists(Paths.get("/pigeon"));
```

Comparaison de fichiers

Méthodes :

- Files.isSameFile(Path path1, Path path2) : vérifie si deux chemins pointent vers le même fichier, y compris les liens symboliques.
- Files.mismatch(Path path1, Path path2) : retourne l'indice de la première différence entre deux fichiers.
  Exemple de mismatch() :

```
System.out.println(Files.mismatch(Path.of("/animals/monkey.txt"), Path.of("/animals/wolf.txt")));
```

Introduction à la Nomenclature des Flux I/O

Introduction aux Flux I/O

- **java.io API** : Nombreuses classes pour créer, accéder et manipuler les flux I/O.
- **Importance** : Comprendre les différences majeures entre chaque classe de flux et comment les distinguer.

Stockage des Données en Octets

- **Bit** : Unité de base (0 ou 1).
- **Octet (byte)** : Groupe de 8 bits.

Flux d'Octets vs Flux de Caractères

- **Flux d'Octets** : Lisent/écrivent des données binaires. Noms de classes finissant par `InputStream` ou `OutputStream`.
- **Flux de Caractères** : Lisent/écrivent des données textuelles. Noms de classes finissant par `Reader` ou `Writer`.

Exemples :

- `FileInputStream` (octets) vs `FileReader` (caractères).

Codage des Caractères en Java

- **Charset** : Spécifie le codage des caractères.
- **Exemples de codages** :
  - `Charset.forName("US-ASCII")`
  - `Charset.forName("UTF-8")`
  - `Charset.forName("UTF-16")`

Différences entre InputStream et OutputStream

- **InputStream** : Classe de base pour lire des octets.
- **OutputStream** : Classe de base pour écrire des octets.
- **Correspondance** : Chaque `InputStream` a une classe `OutputStream` correspondante (ex. `FileInputStream` et `FileOutputStream`).

Flux de Bas Niveau vs Flux de Haut Niveau

- **Flux de bas niveau** : Accès direct aux données (ex. `FileInputStream`).
- **Flux de haut niveau** : Utilisent le "wrapping" pour améliorer les performances et ajouter des fonctionnalités (ex. `BufferedReader`).

Exemple :

```
try (var br = new BufferedReader(new FileReader("zoo-data.txt"))) {
    System.out.println(br.readLine());
}
```

Classes de Base des Flux

Nom de la classe	Description
`InputStream`	Classe abstraite pour les flux d'entrée en octets
`OutputStream`	Classe abstraite pour les flux de sortie en octets
`Reader`	Classe abstraite pour les flux d'entrée en caractères
`Writer`	Classe abstraite pour les flux de sortie en caractères

Classes de Flux Concrets

Nom de la classe	Niveau	Description
`FileInputStream`	Bas	Lit des données de fichier en octets
`FileOutputStream`	Bas	Écrit des données de fichier en octets
`FileReader`	Bas	Lit des données de fichier en caractères
`FileWriter`	Bas	Écrit des données de fichier en caractères
`BufferedReader`	Haut	Lit des données de caractère en mode tamponné
		Écrit des données de caractère en

Lecture et Écriture de Fichiers

- **Méthodes principales** :
  - `read()` pour lire (InputStream/Reader).
  - `write()` pour écrire (OutputStream/Writer).

Exemple de méthode `copyStream()` :

```
void copyStream(InputStream in, OutputStream out) throws IOException {
    int b;
    while ((b = in.read()) != -1) {
        out.write(b);
    }
}
```

Utilisation de Flux de Haut Niveau

Exemple de copie de fichier texte :

```
void copyTextFile(File src, File dest) throws IOException {
    try (var reader = new BufferedReader(new FileReader(src));
         var writer = new BufferedWriter(new FileWriter(dest))) {
        String line;
        while ((line = reader.readLine()) != null) {
            writer.write(line);
            writer.newLine();
        }
    }
}
```

- Flux de bas niveau : Accès brut aux données.
- Flux de haut niveau : Amélioration des performances, ajout de fonctionnalités (ex. BufferedReader).

Sérialisation des Données

Introduction

Vous pouvez utiliser les classes de flux d'E/S que vous avez apprises pour stocker des données textuelles et binaires, mais il faut encore savoir comment insérer ces données dans le flux d'E/S et les décoder plus tard. Bien qu'il existe des formats de fichier comme XML et CSV que vous pouvez normaliser, vous devez souvent construire vous-même la traduction.

La sérialisation permet de convertir un objet en mémoire en un flux d'octets. La désérialisation est le processus inverse, qui consiste à convertir un flux d'octets en un objet.

Interface `Serializable`

Pour sérialiser un objet en utilisant l'API d'E/S, la classe de l'objet doit implémenter l'interface `java.io.Serializable`, qui est une interface marqueur (sans méthodes). Toute classe peut l'implémenter, car elle ne contient aucun membre abstrait.

Pourquoi utiliser l'interface `Serializable` ?
Elle informe tout processus de sérialisation que l'objet peut être sérialisé. Toutes les classes Java de base et de nombreux types primitifs sont sérialisables.

Exemple :

```
import java.io.Serializable;

public class Gorilla implements Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private int age;
   private Boolean friendly;
   private transient String favoriteFood;

   // Constructeurs, getters, setters, toString() omis
}
```

Remarque

Le champ `favoriteFood` est marqué `transient` et ne sera pas sauvegardé lors de la sérialisation.

Interface `Serializable`

Pour sérialiser un objet en utilisant l'API I/O, l'objet doit implémenter l'interface `java.io.Serializable`. Il s'agit d'une interface de marque, ce qui signifie qu'elle n'a pas de méthodes. N'importe quelle classe peut implémenter cette interface, car il n'y a pas de méthodes obligatoires à implémenter.

Exemple de classe sérialisable

champ `transient`

Le mot-clé `transient` peut être utilisé pour marquer des données sensibles de la classe, comme un mot de passe. Les autres objets qui n'ont pas de sens à être sérialisés, tels que l'état d'un `Thread` en mémoire, doivent aussi être marqués `transient`. Lors de la désérialisation, les champs marqués `transient` retrouvent leurs valeurs par défaut Java (comme `null` pour `String`, ou `0.0` pour `double`).

Pratique recommandée : `serialVersionUID`

Il est conseillé de déclarer une variable statique `serialVersionUID` dans chaque classe implémentant `Serializable`. Cette version est

Exemple de sérialisation avec ObjectOutputStream

```
void saveToFile(List<Gorilla> gorillas, File dataFile) throws IOException {
    try (var out = new ObjectOutputStream(
            new BufferedOutputStream(
                new FileOutputStream(dataFile)))) {
        for (Gorilla gorilla : gorillas)
            out.writeObject(gorilla);
    }
}
```

Exemple de désérialisation avec ObjectInputStream

```
    List<Gorilla> readFromFile(File dataFile) throws IOException, ClassNotFoundException {
    var gorillas = new ArrayList<Gorilla>();
    try (var in = new ObjectInputStream(
            new BufferedInputStream(
                new FileInputStream(dataFile)))) {
        while (true) {
            var object = in.readObject();
            if (object instanceof Gorilla g)
                gorillas.add(g);
        }
    } catch (EOFException e) {
        // Fin de fichier atteinte
    }
    return gorillas;
}
```

Création d’un objet désérialisé

Lors de la désérialisation, le constructeur de la classe sérialisée, ainsi que tout initialiseur d’instance, n’est pas appelé. Java appelle le constructeur sans argument de la première classe parente non sérialisable trouvée dans la hiérarchie des classes. Si la classe contient des champs statiques ou transient, ils sont ignorés lors de la désérialisation.

Exemple avec une classe Chimpanzee:

```
public class Chimpanzee implements Serializable {
    private static final long serialVersionUID = 2L;
    private transient String name;
    private transient int age = 10;
    private static char type = 'C';

    { this.age = 14; }

    public Chimpanzee() {
        this.name = "Unknown";
        this.age = 12;
        this.type = 'Q';
    }

    public Chimpanzee(String name, int age, char type) {
        this.name = name;
        this.age = age;
        this.type = type;
    }

    // Getters/Setters/toString() omis
}
```

Lors de la désérialisation :

- Les champs name et age sont initialisés à null et 0, respectivement, car ils sont transient.
- Le champ type reste la dernière valeur définie, soit C si la désérialisation se fait dans la même exécution du programme.

Exemple de sous-classe BabyChimpanzee

```
public class BabyChimpanzee extends Chimpanzee {
    private static final long serialVersionUID = 3L;
    private String mother = "Mom";

    public BabyChimpanzee() { super(); }

    public BabyChimpanzee(String name, char type) {
        super(name, 0, type);
    }

    // Getters/Setters/toString() omis
}
```

La sous-classe BabyChimpanzee est sérialisable car la classe parente Chimpanzee l'est également. La sérialisation et la désérialisation fonctionnent de la même manière, et il est possible de caster un objet BabyChimpanzee en Chimpanzee.

Interaction avec l'utilisateur en Java

Affichage de données à l'utilisateur

Java propose deux instances de `PrintStream` pour afficher des informations : `System.out` et `System.err`. Tandis que `System.out` est couramment utilisé pour afficher des messages de sortie, `System.err` est destiné à rapporter des erreurs dans un flux de sortie distinct. L'utilisation de `System.err` permet de différencier les erreurs des messages réguliers, notamment lorsqu'on travaille dans des environnements comme des IDE où ces flux peuvent être colorés différemment ou redirigés vers des fichiers de logs sur un serveur.

Exemple :

Utilisation des APIs de logging

Pour le développement professionnel, `System.out` et `System.err` sont rarement utilisés. Les applications se tournent plutôt vers des services de logging ou des APIs comme `Log4j`, `SLF4J`, et autres, permettant de logger des messages à différents niveaux : `debug()`, `info()`, `warn()`, `error()`.

Création d'un objet de logging

Pour utiliser une API de logging, il est courant de créer un objet de logging statique dans chaque classe. Cela permet de logger des messages avec différents niveaux de gravité selon le besoin de l'application.

Exemple d'utilisation :

```
private static final Logger logger = LoggerFactory.getLogger(NomDeLaClasse.class);

logger.debug("Message de débogage");
logger.info("Message d'information");
logger.warn("Avertissement");
logger.error("Erreur");
```

Avantages des APIs de logging

- **Flexibilité** : Possibilité de configurer la sortie du log (fichier, console, etc.).
- **Niveaux de journalisation** : Permet de filtrer les messages en fonction de leur importance.
- **Performance** : Les messages de niveau `debug` peuvent être désactivés en production pour améliorer les performances.

Les APIs de logging offrent une approche plus robuste pour gérer la sortie des logs dans des applications complexes ou professionnelles.

Fermeture des flux System

Les flux System.out, System.err, et System.in sont des objets statiques partagés par toute l'application, créés par la JVM. Bien qu'ils puissent être utilisés dans un bloc try-with-resources, leur fermeture n'est pas recommandée car cela les rendra inaccessibles pour le reste du programme.

Utilisation de la classe Console

La classe java.io.Console est conçue pour gérer les interactions avec l'utilisateur. Contrairement aux flux bruts de System.in et System.out, Console propose des méthodes adaptées à la lecture et à l'écriture de données.

Exemple de code :

```
Console console = System.console();
if (console != null) {
   String userInput = console.readLine();
   console.writer().println("Vous avez saisi : " + userInput);
} else {
   System.err.println("Console non disponible");
}
```

Accès aux flux sous-jacents de Console

La classe `Console` donne accès à deux flux pour lire et écrire des données :

- `public Reader reader()`
- `public PrintWriter writer()`

L'accès à ces classes est similaire à l'utilisation de `System.in` et `System.out` directement, mais elles utilisent des flux de caractères plutôt que des flux d'octets. Cela les rend plus appropriées pour la gestion des données textuelles.

Formatage des données de Console

Dans le chapitre 4, vous avez appris à utiliser la méthode `format()` sur `String` ; et dans le chapitre 11, "Exceptions et localisation", vous avez travaillé avec le formatage en utilisant des locales. De manière pratique, chaque classe de flux de sortie inclut une méthode `format()` qui dispose d'une version surchargée acceptant une `Locale` , permettant de combiner ces deux concepts :

- **PrintStream**

```
public PrintStream format(String format, Object... args)
public PrintStream format(Locale loc, String format, Object... args)
```

- **PrintWriter**

```
public PrintWriter format(String format, Object... args)
public PrintWriter format(Locale loc, String format, Object... args)
```

Pour plus de commodité et pour s'assurer que les développeurs venant de C se sentent à l'aise, Java inclut également des méthodes printf() qui fonctionnent de manière identique aux méthodes format(). Ces méthodes sont interchangeables avec format().

Exemple d’utilisation pour afficher des informations à l’utilisateur :

```
Console console = System.console();
if (console == null) {
    throw new RuntimeException("Console not available");
} else {
    console.writer().println("Bienvenue dans notre zoo !");
    console.format("Le zoo possède %d animaux et emploie %d personnes", 391, 25);
    console.writer().println();
    console.printf("Le zoo s'étend sur %5.1f acres", 128.91);
}
```

Résultat attendu (si la Console est disponible au moment de l’exécution) :

```
Bienvenue dans notre zoo !
Le zoo possède 391 animaux et emploie 25 personnes
Le zoo s'étend sur 128.9 acres.
```

Lecture de données de la Console

La classe `Console` propose quatre méthodes permettant de lire des données textuelles courantes de l'utilisateur :

- `public String readLine()`
- `public String readLine(String fmt, Object... args)`
- `public char[] readPassword()`
- `public char[] readPassword(String fmt, Object... args)`

La méthode `readLine()` lit l'entrée jusqu'à ce que l'utilisateur appuie sur la touche Entrée. La version surchargée de `readLine()` affiche un message de demande formaté avant de solliciter l'entrée.

Les méthodes `readPassword()` sont similaires à `readLine()`, mais avec deux différences importantes :

- Le texte saisi par l'utilisateur n'est pas affiché à l'écran pendant la saisie, améliorant ainsi la sécurité (utile pour la saisie de mots de passe).
- Les données sont retournées sous forme de `char[]` au lieu de `String`, ce qui empêche l'entrée de se retrouver dans le pool de chaînes.

Résumé des méthodes Console

Le dernier exemple de code montre comment poser une série de questions à l'utilisateur et afficher des résultats basés sur les réponses en utilisant diverses méthodes apprises dans cette section :

```
Console console = System.console();
if (console == null) {
    throw new RuntimeException("Console not available");
} else {
    String name = console.readLine("Veuillez entrer votre nom : ");
    console.writer().format("Bonjour %s", name);
    console.writer().println();

    console.format("Quelle est votre adresse ? ");
    String address = console.readLine();

    char[] password = console.readPassword("Entrez un mot de passe entre %d et %d caractères : ", 5, 10);
    char[] verify = console.readPassword("Entrez de nouveau le mot de passe : ");
    console.printf("Les mots de passe %s", (Arrays.equals(password, verify) ? "correspondent" : "ne correspondent pas"));
}
```

Sortie attendue (si la Console est disponible) :

```
Veuillez entrer votre nom : Max
Bonjour Max
Quelle est votre adresse ? Spoonerville
Entrez un mot de passe entre 5 et 10 caractères :
Entrez de nouveau le mot de passe :
Les mots de passe correspondent
```