

Les entrées/sorties

En Java les entrées/sorties sont représentées par des objets de type [java.io.InputStream](#), [java.io.Reader](#), [java.io.OutputStream](#) et [java.io.Writer](#). Le package [java.io](#) définit un ensemble de classes qui vont pouvoir être utilisées conjointement avec ces quatre classes abstraites pour réaliser des traitements plus complexes.

InputStream et classes concrètes

La classe [InputStream](#) est une [classe abstraite](#) qui représente un flux d'entrée de données binaires. Elle déclare des méthodes [read](#) qui permettent de lire des données octet par octet ou bien de les copier dans un tableau. Ces méthodes retournent le nombre de caractères lus ou -1 pour signaler la fin du flux. Il existe plusieurs classes qui en fournissent une implémentation concrète.

La classe [ByteArrayInputStream](#) permet d'ouvrir un flux de lecture binaire sur un tableau de **byte**.

```
package com.cgi.udev.io;

import java.io.ByteArrayInputStream;

public class TestByteArrayInputStream {

    public static void main(String[] args) {
        byte[] tableau = "hello the world".getBytes();
        ByteArrayInputStream stream = new ByteArrayInputStream(tableau);

        int octet;
        while ((octet = stream.read()) != -1) {
            System.out.print((char) octet);
        }
    }
}
```

La classe [FileInputStream](#) permet d'ouvrir un flux de lecture binaire sur un fichier.

```

package com.cgi.udev.io;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class TestFileInputStream {

    public static void main(String[] args) throws IOException {

        try (InputStream stream = new FileInputStream("/chemin/vers/mon/fichier.bin")) {
            byte[] buffer = new byte[1024];
            int nbRead;
            while ((nbRead = stream.read(buffer)) != -1) {
                // ...
            }
        }

    }
}

```

Dans l'exemple ci-dessus, on utilise la méthode [InputStream.read](#) qui prend un tableau d'octets en paramètre. Cela est plus efficace que de lire le fichier octet par octet.

À part s'ils représentent une zone mémoire, les flux de données sont généralement attachés à des ressources système (descripteurs de fichier ou de socket). Il est donc impératif de fermer ces flux en appelant leur méthode **close** lorsqu'ils ne sont plus nécessaires pour libérer les ressources système associées. Comme toutes les méthodes d'un flux sont susceptibles de jeter une [IOException](#), on utilise généralement le bloc **finally** pour appeler la méthode **close**.

```

InputStream stream = new FileInputStream("chemin/vers/mon/fichier.bin");
try {
    byte[] buffer = new byte[1024];
    int nbRead;
    while ((nbRead = stream.read(buffer)) != -1) {
        // ...
    }
} finally {

```

```
stream.close();  
}
```

Toutes les classes qui représentent des flux d'entrée ou de sortie implémentent l'interface [Closeable](#). Cela signifie qu'elles peuvent être utilisées avec la syntaxe *try-with-resources* et ainsi faciliter leur gestion en garantissant une fermeture automatique.

```
try (InputStream stream = new FileInputStream("/chemin/vers/mon/fichier.bin")) {  
    byte[] buffer = new byte[1024];  
    int nbRead;  
    while ((nbRead = stream.read(buffer)) != -1) {  
        // ...  
    }  
}
```

Les flux [System.in](#), [System.out](#) et [System.err](#) qui permettent de lire ou d'écrire sur la console sont des cas particuliers. Ils sont ouverts au lancement de l'application et seront automatiquement fermés à la fin. Il est néanmoins possible de fermer explicitement ces flux si on veut détacher l'application du *shell* à partir duquel elle a été lancée.

OutputStream et classes concrètes

La classe [OutputStream](#) est une [classe abstraite](#) qui représente un flux de sortie de données binaires. Elle déclare des méthodes [write](#) qui permettent d'écrire des données octet par octet ou bien de les écrire depuis un tableau. La classe [OutputStream](#) fournit également la méthode [flush](#) pour forcer l'écriture de la zone tampon (s'il existe une zone tampon sinon un appel à cette méthode est sans effet).

Il existe plusieurs classes qui en fournissent une implémentation concrète.

La classe [ByteArrayOutputStream](#) permet d'ouvrir un flux d'écriture binaire en mémoire. Le contenu peut ensuite être récupéré sous la forme d'un tableau d'octets grâce à la méthode [toByteArray](#).

```
package com.cgi.udev.io;  
  
import java.io.ByteArrayOutputStream;  
import java.util.Arrays;  
  
public class TestByteArrayOutputStream {
```

```

public static void main(String[] args) {
    ByteArrayOutputStream stream = new ByteArrayOutputStream();

    for (byte b : "Hello the world".getBytes()) {
        stream.write(b);
    }

    byte[] byteArray = stream.toByteArray();
    System.out.println(Arrays.toString(byteArray));
}
}

```

La classe [FileOutputStream](#) permet d'ouvrir un flux d'écriture binaire sur un fichier.

```

package com.cgi.udev.io;

import java.io.FileOutputStream;
import java.io.IOException;

public class TestFileOutputStream {

    public static void main(String[] args) throws IOException {

        try (FileOutputStream stream = new
FileOutputStream("chemin/vers/mon/fichierdesortie.bin")) {
            byte[] octets = "hello the world".getBytes();
            stream.write(octets);
        }

    }

}

```

Dans l'exemple ci-dessus, on utilise la méthode [OutputStream.write](#) qui prend un tableau d'octets en paramètre. Cela est plus efficace que d'écrire dans le fichier octet par octet.

Comme cela a été signalé ci-dessus pour les [InputStream](#), les flux d'écriture qui ne correspondent pas à des zones mémoire (fichiers, sockets...) doivent impérativement être fermés lorsqu'ils ne sont plus utilisés pour libérer les ressources système associées.

Flux orientés caractères

Le package [java.io](#) contient un ensemble de classes qui permettent de manipuler des flux caractères et donc du texte. Toutes les classes qui permettent d'écrire dans un flux de caractères héritent de la classe abstraite [Writer](#) et toutes les classes qui permettent de lire un flux de caractères héritent de la classe abstraite [Reader](#).

Reader et classes concrètes

La classe [Reader](#) est une [classe abstraite](#) qui permet de lire des flux de caractères. Comme [InputStream](#), la classe [Reader](#) fournit des méthodes [read](#) mais qui acceptent en paramètre des caractères. Il existe plusieurs classes qui en fournissent une implémentation concrète.

La classe [StringReader](#) permet de parcourir une chaîne de caractères sous la forme d'un flux de caractères.

```
package com.cgi.udev.io;

import java.io.IOException;
import java.io.Reader;
import java.io.StringReader;

public class TestStringReader {

    public static void main(String[] args) throws IOException {
        Reader reader = new StringReader("hello the world");

        int caractere;
        while ((caractere = reader.read()) != -1) {
            System.out.print((char) caractere);
        }
    }
}
```

```
}
```

Il n'est pas nécessaire d'utiliser un [StringReader](#) pour parcourir une chaîne de caractères. Par contre, cette classe est très pratique si une partie d'un programme réalise des traitements en utilisant une instance de [Reader](#). Le principe de substitution peut s'appliquer en passant une instance de [StringReader](#).

La classe [FileReader](#) permet de lire le contenu d'un fichier texte.

```
package com.cgi.udev.io;

import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class TestFileReader {

    public static void main(String[] args) throws IOException {

        try (Reader reader = new FileReader("/le/chemin/du/fichier.txt")) {
            char[] buffer = new char[1024];
            int nbRead;
            while ((nbRead = reader.read(buffer)) != -1) {
                // ...
            }
        }

    }

}
```

La classe [FileReader](#) ne permet pas de positionner l'encodage de caractères (*charset*) utilisé dans le fichier. Elle utilise l'encodage par défaut de la JVM qui est dépendant du système. Dans la pratique l'usage de cette classe est donc assez limité.

Writer et classes concrètes

La classe [Writer](#) est une [classe abstraite](#) qui permet d'écrire des flux de caractères. Comme [OutputStream](#), la classe [Writer](#) fournit des méthodes [write](#) mais qui acceptent en paramètre des caractères. Elle fournit également des méthodes [append](#) qui réalisent la même type d'opérations et qui retournent l'instance du [Writer](#) afin de pouvoir chaîner les appels. Il existe plusieurs classes qui en fournissent une implémentation concrète.

La classe [StringWriter](#) permet d'écrire dans un flux caractères pour ensuite produire une chaîne de caractères.

```
package com.cgi.udev.io;

import java.io.IOException;
import java.io.StringWriter;

public class TestStringWriter {

    public static void main(String[] args) throws IOException {
        StringWriter writer = new StringWriter();

        writer.append("Hello")
            .append(' ')
            .append("the")
            .append(' ')
            .append("world");

        String resultat = writer.toString();

        System.out.println(resultat);
    }
}
```

La classe [FileWriter](#) permet d'écrire un flux de caractères dans un fichier.

```
package com.cgi.udev.io;

import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
```

```
public class TestFileWriter {

    public static void main(String[] args) throws IOException {

        try (Writer writer = new FileWriter("/chemin/vers/mon/fichier.txt", true)) {
            writer.append("Hello world!\n");
        }

    }

}
```

Le booléen passé en second paramètre du constructeur de [FileWriter](#) permet de spécifier si le fichier doit être ouvert en ajout (*append*).

La classe [FileWriter](#) ne permet pas de positionner l'encodage de caractères (*charset*) utilisé pour écrire dans le fichier. Elle utilise l'encodage par défaut de la JVM qui est dépendant du système. Dans la pratique l'usage de cette classe est donc assez limité.

Les décorateurs de flux

Le package [java.io](#) fournit un ensemble de classes qui agissent comme des [décorateurs](#) pour des instances de type [InputStream](#), [Reader](#), [OutputStream](#) ou [Writer](#). Ces [décorateurs](#) permettent d'ajouter des fonctionnalités tout en présentant les mêmes méthodes. Il est donc très simple d'utiliser ces décorateurs dans du code initialement implémenté pour manipuler des instances des types décorés.

Les classes [BufferedInputStream](#), [BufferedReader](#), [BufferedOutputStream](#) et [BufferedWriter](#) permettent de créer un décorateur qui gère une zone tampon dont il est possible d'indiquer la taille à la construction de l'objet. Ces classes sont très utiles lorsque l'on veut lire ou écrire des données sur un disque ou sur un réseau afin de limiter les accès système et améliorer les performances.

```
package com.cgi.udev.io;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
```



```

public class TestFileWriter {

    public static void main(String[] args) throws IOException {

        try (Writer writer = new BufferedWriter(new FileWriter("monfichier.txt", true), 1024)) {
            writer.append("Hello world!\n");
        }

    }

}

```

Dans l'exemple ci-dessus, on crée un [BufferedWriter](#) avec une zone tampon de 1 Ko.

La classe [LineNumberReader](#) permet quant à elle, de compter les lignes lors de la lecture d'un flux caractères. Elle fournit également la méthode [readLine](#) pour lire une ligne complète.

```

package com.cgi.udev.io;

import java.io.IOException;
import java.io.LineNumberReader;
import java.io.StringReader;

public class TestStringReader {

    public static void main(String[] args) throws IOException {
        StringReader stringReader = new StringReader("hello the world\nhello the world");

        LineNumberReader reader = new LineNumberReader(stringReader);

        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }

        System.out.println("Nombre de lignes lues : " + reader.getLineNumber());
    }

}

```

Les classes [InputStreamReader](#) et [OutputStreamWriter](#) permettent de manipuler un flux binaire sous la forme d'un flux caractères. La classe [InputStreamReader](#) hérite de [Reader](#) et prend comme paramètre de constructeur une instance de [InputStream](#). La classe [OutputStreamWriter](#) hérite de [Writer](#) et prend comme paramètre de constructeur une instance de [OutputStream](#). Ces classes sont particulièrement utiles car elles permettent de préciser l'encodage des caractères (*charset*) qui doit être utilisé pour passer d'un flux binaire au flux caractères et vice-versa.

```
package com.cgi.udev.io;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;

public class TestFileReader {

    public static void main(String[] args) throws IOException {

        String fichier = "/le/chemin/du/fichier.txt";
        try (Reader reader = new InputStreamReader(new FileInputStream(fichier), "UTF-8")) {
            char[] buffer = new char[1024];
            int nbRead;
            while ((nbRead = reader.read(buffer)) != -1) {
                // ...
            }
        }

    }

}
```

Dans l'exemple ci-dessus, le fichier est ouvert grâce à une instance de [FileInputStream](#) qui est passée à une instance de [InputStreamReader](#) qui lit les caractères au format UTF-8.

Il est possible de créer très facilement des chaînes de décorateurs.

```
package com.cgi.udev.io;

import java.io.FileInputStream;
import java.io.IOException;
```

```

import java.io.InputStreamReader;
import java.io.LineNumberReader;
import java.io.Reader;

public class TestFileReader {

    public static void main(String[] args) throws IOException {

        String fichier = "/le/chemin/du/fichier.txt";
        Reader inputStreamReader = new InputStreamReader(new FileInputStream(fichier), "UTF-8");
        try (LineNumberReader reader = new LineNumberReader(inputStreamReader)) {
            String ligne;
            while ((ligne = reader.readLine()) != null) {
                // ...
            }
        }

    }
}

```

Dans l'exemple ci-dessus, on utilise la syntaxe **try-with-resources** pour appeler automatiquement la méthode close à la fin du bloc **try**. Les décorateurs de flux implémentent la méthode close de manière à appeler la méthode close de l'objet qu'il décore. Ainsi quand on crée une chaîne de flux avec des décorateurs, un appel à la méthode close du décorateur le plus englobant appelle automatiquement toutes les méthodes close de la chaîne de flux.

Les objets statiques System.in, System.out et System.err qui représentent respectivement le flux d'entrée de la console, le flux de sortie de la console et le flux de sortie d'erreur de la console sont des instances de InputStream ou de PrintStream. PrintStream est un décorateur qui offre notamment les méthodes print, println et printf.

Pour manipuler les flux de la console, il est également possible de récupérer une instance de Console en appelant la méthode System.console().

La classe Scanner

La classe [java.util.Scanner](#) agit comme un décorateur pour différents types d'instance qui représentent une entrée. Elle permet de réaliser des opérations de lecture et de validation de données plus complexes que les classes du packages [java.io](#).

```
package com.cgi.udev.io;

import java.io.IOException;
import java.util.Scanner;

public class TestScanner {

    public static void main(String[] args) throws IOException {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Saisissez une chaîne de caractères : ");
        String chaine = scanner.nextLine();

        System.out.print("Saisissez un nombre : ");
        int nombre = scanner.nextInt();

        System.out.print("Saisissez les 8 caractères de votre identifiant : ");
        String identifiant = scanner.next("{8}");

        System.out.println("Vous avez saisi :");
        System.out.println(chaine);
        System.out.println(nombre);
        System.out.println(identifiant);
    }
}
```

On peut compléter l'implémentation précédente en effectuant une validation sur les données saisies par l'utilisateur :

```
package com.cgi.udev.io;

import java.io.IOException;
import java.util.InputMismatchException;
```

```
import java.util.Scanner;

public class TestScanner {

    public static void main(String[] args) throws IOException {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Saisissez une chaîne de caractères : ");
        String chaine = scanner.nextLine();

        Integer nombre = null;
        do {
            try {
                System.out.print("Saisissez un nombre : ");
                nombre = scanner.nextInt();
            } catch (InputMismatchException e) {
                scanner.next();
                System.err.println("Ceci n'est pas un nombre valide");
            }
        } while (nombre == null);

        String identifiant = null;
        do {
            System.out.print("Saisissez les 8 caractères de votre identifiant : ");
            // On utilise une expression régulière pour vérifier le prochain token
            if (scanner.hasNext(".{8}")) {
                identifiant = scanner.next();
            } else {
                scanner.next();
                System.err.println("Ceci n'est pas un identifiant valide");
            }
        } while (identifiant == null);

        System.out.println("Vous avez saisi :");
        System.out.println(chaine);
        System.out.println(nombre);
        System.out.println(identifiant);
    }
}
```

Fichiers et chemins

En plus des flux de type fichier, le package [java.io](#) fournit la classe [File](#) qui représente un fichier. À travers, cette classe, il est possible de savoir si le fichier existe, s'il s'agit d'un répertoire... On peut également créer le fichier ou le supprimer.

```
package com.cgi.udev.io;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class TestFile {

    public static void main(String[] args) throws IOException {
        File fichier = new File("unfichier.txt");

        if (!fichier.exists()) {
            fichier.createNewFile();
        }

        if (fichier.canWrite()) {
            try (BufferedWriter writer = new BufferedWriter(new FileWriter(fichier))) {
                writer.write("Hello world!");
            }
        }

        fichier.delete();
    }
}
```

Pour représenter un chemin d'accès à un fichier, on peut utiliser une URL avec le schéma *file* :

```
file:///home/david/monfichier.txt
```

ou bien une chaîne de caractère représentant directement le chemin. L'inconvénient de cette dernière méthode est qu'elle n'est pas portable suivant les différents systèmes de fichiers et les différents systèmes d'exploitation. En Java, on utilise l'interface [Path](#) pour représenter un chemin

de fichier de manière générique. Les classes [Paths](#) et [FileSystem](#) servent à construire des instances de type [Path](#). La classe [FileSystem](#) fournit également des méthodes pour obtenir des informations à propos du ou des systèmes de fichiers présents sur la machine. On peut accéder à une instance de [FileSystem](#) grâce à la méthode [FileSystems.getDefault\(\)](#).

```
package com.cgi.udev.io;

import java.io.File;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.nio.file.Paths;

public class TestPath {

    public static void main(String[] args) throws IOException {
        Path cheminFichier = Paths.get("home", "david", "fichier.txt");

        System.out.println(cheminFichier); // home/david/fichier.txt
        System.out.println(cheminFichier.getNameCount()); // 3
        System.out.println(cheminFichier.getParent()); // home/david
        System.out.println(cheminFichier.getFileName()); // fichier.txt

        cheminFichier = FileSystems.getDefault().getPath("home", "david", "fichier.txt");
        File fichier = cheminFichier.toFile();

        // maintenant on peut utiliser le fichier
    }

}
```

Pour les opérations les plus courantes sur les fichiers, la classe outil [Files](#) fournit un ensemble de méthodes statiques qui permettent de créer, de consulter, de modifier ou de supprimer des fichiers et des répertoires en utilisant un minimum d'appel.

```
package com.cgi.udev.io;

import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
```

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.List;

public class TestFiles {

    public static void main(String[] args) throws IOException {
        Path fichier1 = Paths.get("fichier.txt");

        // création du fichier
        fichier1 = Files.createFile(fichier1);

        System.out.println("Taille du fichier : " + Files.size(fichier1));

        try (BufferedWriter writer = Files.newBufferedWriter(fichier1, StandardOpenOption.WRITE))
        {
            writer.append("Hello !\n");
            writer.append("Hello !\n");
            writer.append("Hello !\n");
        }

        System.out.println("Taille du fichier : " + Files.size(fichier1));

        // Copie vers un nouveau fichier
        Path fichier2 = Paths.get("fichier2.txt");
        Files.copy(fichier1, fichier2);

        // Lecture de l'intégralité du fichier
        List<String> lignes = Files.readAllLines(fichier2);

        // Suppression des fichiers créés
        Files.deleteIfExists(fichier1);
        Files.deleteIfExists(fichier2);

        System.out.println(lignes);
    }
}
```


La classe [Files](#) se révèle très pratique d'utilisation notamment pour lire l'intégralité d'un fichier. Elle ne rend pas pour autant obsolète l'utilisation de [Reader](#) ou de [OutputStream](#). En effet, travailler à partir d'un flux peut avoir un impact important sur l'empreinte mémoire d'une application. Si une application doit parcourir un fichier pour trouver une information précise alors, si le fichier peut être de taille importante, l'utilisation de flux sera plus optimale car l'empreinte mémoire d'un flux est généralement celle de la taille de la zone tampon allouée pour la lecture ou l'écriture.

Accès au réseau

La classe [URL](#), comme son nom l'indique, représente une URL. Elle déclare la méthode [openConnection](#) qui retourne une instance de [URLConnection](#). Une instance de [URLConnection](#) ouvre une connexion distante avec le serveur et permet de récupérer des informations du serveur distant. Elle permet surtout d'obtenir une instance de [OutputStream](#) si on désire envoyer des informations au serveur et une instance de [InputStream](#) si on désire récupérer les informations retournées par le serveur.

```
package com.cgi.udev.io;

import java.io.IOException;
import java.io.InputStreamReader;
import java.io.LineNumberReader;
import java.io.Reader;
import java.net.URL;
import java.net.URLConnection;
import java.util.Objects;

public class HttpClient {

    public static void main(String[] args) throws IOException {
        URL url = new URL("https://www.ietf.org/rfc/rfc1738.txt");
        URLConnection connection = url.openConnection();

        String encodage = Objects.toString(connection.getContentEncoding(), "ISO-8859-1");
        Reader reader = new InputStreamReader(connection.getInputStream(), encodage);

        try (LineNumberReader lineNumberReader = new LineNumberReader(reader)) {
            String line;
```

```

while ((line = lineNumberReader.readLine()) != null) {
    System.out.println(line);
}

System.out.println("Ce fichier contient " + lineNumberReader.getLineNumber() + "
lignes.");
}

}

}

```

Le programme ci-dessus récupère, affiche sur la sortie standard et donne le nombre de lignes du document accessible à l'adresse <https://www.ietf.org/rfc/rfc1738.txt> (il s'agit du document de l'IETF qui décrit ce qu'est une URL).

L'API d'entrée/sortie de Java fournit une bonne abstraction. Généralement, une méthode qui manipule des flux fonctionnera pour des fichiers, des flux mémoire et des flux réseaux.

La sérialisation d'objets

Les classes [ObjectOutputStream](#) et [ObjectInputStream](#) permettent de réaliser la sérialisation/désérialisation d'objets : un objet (et tous les objets qu'il référence) peut être écrit dans un flux ou lu depuis un flux. Cela peut permettre de sauvegarder dans un fichier un état de l'application ou bien d'échanger des données entre deux programmes Java à travers un réseau. La sérialisation d'objets a des limites :

- Seul l'état des objets est écrit ou lu, cela signifie que les fichiers *class* ne font pas partie de la sérialisation et doivent être disponibles pour la JVM au moment de la lecture (opération de désérialisation réalisée avec la classe [ObjectInputStream](#)).
- Le format des données sérialisées est propre à Java, ce mécanisme n'est donc pas adapté pour échanger des informations avec des applications qui ne seraient pas écrites en Java.
- Les données sérialisées sont très dépendantes de la structure des classes. Si des modifications sont apportées à ces dernières, une grappe d'objets préalablement sérialisée dans un fichier ne sera sans doute plus lisible.

Pour qu'un objet puisse être sérialisé, il faut que sa classe implémente [l'interface marqueur Serializable](#). Si un objet référence d'autres objets dans ses attributs alors il faut également que les classes de ces objets implémentent l'interface [Serializable](#). Beaucoup de classes de l'API standard de Java implémentent l'interface

[Serializable](#), à commencer par la classe [String](#).

Tenter de sérialiser un objet dont la classe n'implémente pas [Serializable](#) produit une exception de type [java.io.NotSerializableException](#).

Prenons comme exemple une classe *Personne* qui contient la liste de ses enfants (eux-mêmes de type *Personne*). Cette classe implémente l'interface [Serializable](#) :

```
package com.cgi.udev;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Personne implements Serializable {

    private String prenom;
    private String nom;
    private List<Personne> enfants = new ArrayList<>();

    public Personne(String prenom, String nom) {
        this.prenom = prenom;
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void ajouterEnfants(Personne... enfants) {
        Collections.addAll(this.enfants, enfants);
    }

    public List<Personne> getEnfants() {
```

```

        return enfants;
    }

    @Override
    public String toString() {
        return this.prenom + " " + this.nom;
    }
}

```

Le code ci-dessous sérialise les données dans le fichier *arbre_genialogique.bin*

```

package com.cgi.udev.io;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Paths;

import com.cgi.udev.Personne;

public class TestSerialisation {

    public static void main(String[] args) throws IOException {

        Personne personne = new Personne("Donald", "Duck");
        personne.ajouterEnfants(new Personne("Riri", "Duck"),
                                new Personne("Fifi", "Duck"),
                                new Personne("Loulou", "Duck"));

        OutputStream outputStream = Files.newOutputStream(Paths.get("arbre_genialogique.bin"));
        try(ObjectOutputStream objectStream = new ObjectOutputStream(outputStream);) {
            objectStream.writeObject(personne);
        }
    }
}

```

Un autre code qui a accès à la même classe *Personne* peut ensuite lire le fichier *arbre_genialogique.bin* pour retrouver les objets dans l'état attendu.

```
package com.cgi.udev.io;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.nio.file.Files;
import java.nio.file.Paths;

import com.cgi.udev.Personne;

public class TestDeserialisation {

    public static void main(String[] args) throws IOException, ClassNotFoundException {

        InputStream outputStream = Files.newInputStream(Paths.get("arbre_genialogique.bin"));
        try(ObjectInputStream objectStream = new ObjectInputStream(outputStream);) {
            Personne personne = (Personne) objectStream.readObject();

            System.out.println(personne);
            for (Personne enfant : personne.getEnfants()) {
                System.out.println(enfant);
            }
        }
    }
}
```

L'exécution du programme ci-dessus affichera :

```
Donald Duck
Riri Duck
Fifi Duck
Loulou Duck
```

Donnée transient

Parfois une classe contient des informations que l'on ne souhaite pas sérialiser. Cela peut être dû à des limitations techniques (par exemple la classe associée n'implémente pas l'interface [Serializable](#)). Mais il peut aussi s'agir de données sensibles ou volatiles qui n'ont pas à être sérialisées. Pour que les processus de sérialisation/désérialisation ignorent ces attributs, il faut leur

ajouter le mot-clé **transient**.

Pour la classe *Personne*, si on veut exclure la liste des enfants de la sérialisation/désérialisation, on peut modifier les attributs comme suit :

```
package com.cgi.udev;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Personne implements Serializable {

    private String prenom;
    private String nom;
    private transient List<Personne> enfants = new ArrayList<>();

    public Personne(String prenom, String nom) {
        this.prenom = prenom;
        this.nom = nom;
    }

    // ...
}
```

Si nous exécutons à nouveau les programmes de sérialisation et de désérialisation du paragraphe précédent, la sortie standard affichera alors :

```
Donald Duck
```

Car l'état de la liste des enfants ne sera plus écrit dans le fichier *arbre_genalogique.bin*.

Identifiant de version de sérialisation

La principale difficulté dans la mise en pratique des mécanismes de sérialisation/désérialisation provient de leur extrême dépendance au format des classes.

Si la sérialisation est utilisée pour sauvegarder dans un fichier l'état des objets entre deux exécutions, alors il n'est pas possible de modifier significativement puis de recompiler les classes sérialisables (sinon l'opération de désérialisation échouera avec une erreur [InvalidClassException](#)). Si la sérialisation est utilisée pour échanger des informations entre deux applications sur un

réseau, alors les deux applications doivent disposer dans leur *classpath* des mêmes définitions de classes.

En fait les classes qui implémentent l'interface [Serializable](#) possèdent un numéro de version interne qui change à la compilation si des modifications substantielles ont été apportées (ajout ou suppression d'attributs ou de méthodes par exemple). Lorsqu'un objet est sérialisé, le numéro de version de sa classe est également sérialisé. Ainsi, lors de la désérialisation, il est facile de comparer ce numéro avec celui de la classe disponible. Si ces numéros ne correspondent pas, alors le processus de désérialisation échoue en considérant que la classe disponible n'est pas compatible avec la classe qui a été utilisée pour créer l'objet sérialisé.

Si on ne souhaite pas utiliser ce mécanisme implicite de version, il est possible de spécifier un numéro de version de sérialisation pour ses classes. À charge du développeur de changer ce numéro lorsque les modifications de la classe sont trop importantes pour ne plus garantir la compatibilité ascendante avec des versions antérieures de cette classe. Le numéro de version est une constante de classe de type **long** qui doit s'appeler *serialVersionUID*.

```
package com.cgi.udev;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Personne implements Serializable {

    private static final long serialVersionUID = 1775245980933452908L;

    // ...
}
```

Eclipse produit un avertissement si une classe qui implémente [Serializable](#) ne déclare pas une constante *serialVersionUID*.

Pour contourner le problème de dépendance entre le format de sérialisation et la déclaration de la classe, il est possible d'implémenter soi-même l'écriture et la lecture des données. Pour cela, il faut déclarer deux méthodes privées dans la classe : *writeObject* et *readObject*. Ces méthodes seront appelées (même si elles sont privées) en lieu et place de l'algorithme par défaut de sérialisation/désérialisation.

```
package com.cgi.udev;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Personne implements Serializable {

    private static final long serialVersionUID = 1775245980933452908L;

    private String prenom;
    private String nom;
    private List<Personne> enfants = new ArrayList<>();

    private void writeObject(ObjectOutputStream s) throws IOException {
        // on ne s rialise que le pr nom et le nom
        s.writeObject(prenom);
        s.writeObject(nom);
    }

    private void readObject(ObjectInputStream s) throws ClassNotFoundException, IOException {
        // on lit les donn es dans le m me ordre qu'elles ont  t   crites
        this.prenom = (String) s.readObject();
        this.nom = (String) s.readObject();
        this.enfants = new ArrayList<>();
    }

    // ...

}
```

Revision #1

Created 10 February 2025 23:21:05 by Nicolas

Updated 10 February 2025 23:24:39 by Nicolas