9. TP : Comprendre les Flux d'E/S en Java

9.1 Introduction aux Flux d'E/S

En Java, un flux d'entrée/sortie (E/S) est utilisé pour faciliter l'échange de données entre un programme et des sources externes telles que des fichiers, des sockets réseau ou la mémoire du programme. Les flux en Java sont divisés en deux types principaux :

- **Flux d'Octets** : Utilisés pour manipuler les données binaires (octets de 8 bits).
- **Flux de Caractères** : Utilisés pour manipuler les données textuelles (Unicode de 16 bits).

Cette fiche d'exercices vous guidera à travers différents aspects des flux d'E/S en Java.

9.2 Les Flux d'Octets

9.2.1 Définition

Un flux d'octets en Java permet de transporter des données binaires (non textuelles), comme des fichiers audio, images, ou vidéos. Ces flux utilisent les classes abstraites `InputStream` et `OutputStream` comme base.

9.2.2 Cycle d'utilisation

Le cycle d'utilisation d'un flux d'octets suit trois étapes principales :

1. Ouverture du flux

2. Lecture ou écriture des données
3. Fermeture du flux

9.2.3 Classes principales

— **FileInputStream** : Permet de lire des données binaires à partir d'un fichier.
— **FileOutputStream** : Permet d'écrire des données binaires dans un fichier.

9.2.4 Exemple : Lecture avec FileInputStream

```
import java.io.*;

public class LectureFluxOctets {
    public static void main(String[] args) {
        try {
            FileInputStream fichier = new FileInputStream("
   monfichier.dat");
            int octet;
            while ((octet = fichier.read()) != -1) {
                System.out.print((char) octet);
            }
            fichier.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice d'application :
— Créez un fichier nommé `exemple.dat` contenant du texte.
— Modifiez le programme pour lire et afficher son contenu.

9.2.5 Exemple : Écriture avec FileOutputStream

```
import java.io.*;

public class EcritureFluxOctets {
    public static void main(String[] args) {
        try {
            FileOutputStream fichier = new FileOutputStream("
   sortie.dat");
            String texte = "Bonjour, ceci est un exemple.";
            fichier.write(texte.getBytes());
            fichier.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice d'application :

— Ajoutez une deuxième chaîne de caractères au fichier `sortie.dat` sans écraser le contenu précédent.

9.2.6 Les Classes Principales des Flux d'Octets

Les flux d'octets en Java se divisent en deux grandes catégories : les flux de communication et les flux de traitement. Voici un tableau récapitulatif des classes principales :

Classes de communication

— **FileInputStream** : lecture de fichiers octets par octets.
— **PipedInputStream** : récupère des données provenant d'un flux de sortie connecté.
— **ByteArrayInputStream** : lit des données dans un tampon structuré sous forme d'array.
— **FileOutputStream** : écriture de fichiers octets par octets.
— **PipedOutputStream** : envoie des données vers un flux d'entrée connecté.
— **ByteArrayOutputStream** : écrit des données dans un tampon structuré sous forme d'array.

9.2.7 Exemple Complet : Gestion des Classes de Communication

Voici un exemple complet qui démontre l'utilisation des classes `FileInputStream`, `FileOutputStream`, `ByteArrayInputStream`, et `ByteArrayOutputStream`.

```
1  import java.io.*;
2
3  public class CommunicationStreamsExample {
4      public static void main(String[] args) {
5          try {
6              // Utilisation de FileOutputStream pour écrire des
       données dans un fichier
7              FileOutputStream fos = new FileOutputStream("
       output.txt");
8              fos.write("Hello, world!\n".getBytes());
9              fos.close();
10
11             // Utilisation de FileInputStream pour lire les
       données du fichier
12             FileInputStream fis = new FileInputStream("output.
       txt");
13             int data;
14             while ((data = fis.read()) != -1) {
15                 System.out.print((char) data);
16             }
17             fis.close();
18
```

```
19              // Utilisation de ByteArrayOutputStream pour é
    crire dans un tampon
20              ByteArrayOutputStream baos = new
    ByteArrayOutputStream();
21              baos.write("This is a ByteArray example.".getBytes
    ());
22
23              // Utilisation de ByteArrayInputStream pour lire à
     partir du tampon
24              ByteArrayInputStream bais = new
    ByteArrayInputStream(baos.toByteArray());
25              while ((data = bais.read()) != -1) {
26                  System.out.print((char) data);
27              }
28              bais.close();
29
30          } catch (IOException e) {
31              e.printStackTrace();
32          }
33      }
34  }
```

Exercice :

— Créez un fichier nommé `data.txt` contenant une phrase.
— Modifiez le programme pour lire le contenu de `data.txt` avec `FileInputStream` et écrire ce contenu dans un tableau de bytes à l'aide de `ByteArrayOutputStream`.
— Utilisez `ByteArrayInputStream` pour afficher le contenu du tableau dans la console.

Classes de traitement

— **SequenceInputStream** : concaténation de plusieurs flux d'entrée en un seul.
— **ObjectInputStream** : lecture d'objets Java sérialisés.
— **FilterInputStream** : ajoute des fonctionnalités aux flux d'entrée.
— **DataInputStream** : lecture des types primitifs directement.
— **BufferedInputStream** : améliore les performances grâce au tamponnage.
— **PushbackInputStream** : permet de remettre un octet dans le flux entrant.
— **ObjectOutputStream** : écrit des objets Java sérialisés.
— **FilterOutputStream** : ajoute des fonctionnalités aux flux de sortie.
— **DataOutputStream** : écriture des types primitifs directement.
— **BufferedOutputStream** : améliore les performances grâce au tamponnage.

9.2.8 Exemple : FileInputStream

```
1  import java.io.*;
2
3  public class ExempleFileInputStream {
4      public static void main(String[] args) {
5          try {
```

```
            FileInputStream fis = new FileInputStream("exemple.txt");
            int data;
            while ((data = fis.read()) != -1) {
                System.out.print((char) data);
            }
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice : Modifiez le programme pour compter le nombre de caractères lus.

9.2.9 Exemple : SequenceInputStream

```
import java.io.*;
import java.util.*;

public class ExempleSequenceInputStream {
    public static void main(String[] args) {
        try {
            FileInputStream fis1 = new FileInputStream("fichier1.txt");
            FileInputStream fis2 = new FileInputStream("fichier2.txt");
            SequenceInputStream sis = new SequenceInputStream(fis1, fis2);
            int data;
            while ((data = sis.read()) != -1) {
                System.out.print((char) data);
            }
            sis.close();
            fis1.close();
            fis2.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice : Créez deux fichiers et concaténez leur contenu en un seul flux à l'aide de `SequenceInputStream`.

9.2.10 Exemple : ObjectOutputStream et ObjectInputStream

```
import java.io.*;

class Person implements Serializable {
```

```
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name + ", " + age + " ans";
    }
}

public class ExempleObjectStream {
    public static void main(String[] args) {
        try {
            ObjectOutputStream oos = new ObjectOutputStream(
   new FileOutputStream("personnes.dat"));
            oos.writeObject(new Person("Alice", 30));
            oos.writeObject(new Person("Bob", 25));
            oos.close();

            ObjectInputStream ois = new ObjectInputStream(new
   FileInputStream("personnes.dat"));
            Person p1 = (Person) ois.readObject();
            Person p2 = (Person) ois.readObject();
            ois.close();

            System.out.println(p1);
            System.out.println(p2);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice : Ajoutez une nouvelle propriété `email` à la classe `Person` et sérialisez un nouvel objet.

9.2.11 Exemple : PushbackInputStream

```
import java.io.*;

public class ExemplePushbackInputStream {
    public static void main(String[] args) {
        try {
            PushbackInputStream pbis = new PushbackInputStream
   (new FileInputStream("exemple.txt"));
            int data = pbis.read();
            if (data == 'a') {
```

```
                pbis.unread(data);
            }
            System.out.println((char) pbis.read());
            pbis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice : Modifiez le programme pour détecter une séquence spécifique au début du fichier et la remettre dans le flux.

9.2.12 Exercice 1 : Lecture depuis un Fichier avec les Flux d'Octets

Dans cet exercice, vous allez lire des données à partir d'un fichier en utilisant un flux d'octets.

Instructions :

1. Créez un fichier nommé `"input.txt"` avec du contenu texte (par exemple, "Bonjour, Flux Java !").
2. Utilisez `FileInputStream` pour lire les données du fichier, octet par octet.
3. Affichez chaque octet en tant que caractère pour montrer le contenu du fichier dans la console.

Exemple de Code :

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

public class ByteReadExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("input.
    txt")) {
            int byteData;
            while ((byteData = fis.read()) != -1) {
                System.out.print((char) byteData);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9.2.13 Exercice 2 : Écriture dans un Fichier avec les Flux d'Octets

Écrivez des données dans un fichier en utilisant `FileOutputStream`.

Instructions :

1. Créez un fichier nommé `"output.txt"`.
2. Utilisez `FileOutputStream` pour écrire une chaîne de caractères (par exemple, "Les flux Java sont puissants !") dans le fichier.

3. Fermez le flux après l'écriture.

Exemple de Code :

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

public class ByteWriteExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("
    output.txt")) {
            String data = "Les flux Java sont puissants!";
            fos.write(data.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9.2.14 Exercice 3 : Utilisation des Flux Tamponnés pour l'Efficacité

Utilisez `BufferedInputStream` et `BufferedOutputStream` pour lire et écrire des données plus efficacement.

Instructions :

1. Lisez le contenu de `"input.txt"` en utilisant `BufferedInputStream` et affichez-le.
2. Écrivez le même contenu dans `"bufferedOutput.txt"` en utilisant `BufferedOutputSt`

Exemple de Code :

```
import java.io.*;

public class BufferedStreamExample {
    public static void main(String[] args) {
        try (BufferedInputStream bis = new BufferedInputStream
    (new FileInputStream("input.txt"));
             BufferedOutputStream bos = new
    BufferedOutputStream(new FileOutputStream("bufferedOutput.
    txt"))) {
            int byteData;
            while ((byteData = bis.read()) != -1) {
                bos.write(byteData);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9.2.15 Exercice 4 : Lecture et Écriture de Caractères avec FileReader et FileWriter

Utilisez des flux de caractères pour lire et écrire des données textuelles.

Instructions :

1. Lisez le contenu de `"input.txt"` en utilisant `FileReader`.
2. Écrivez le contenu dans `"characterOutput.txt"` en utilisant `FileWriter`.

Exemple de Code :

```
import java.io.*;

public class CharacterStreamExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("input.txt");
             FileWriter fw = new FileWriter("characterOutput.
   txt")) {
            int charData;
            while ((charData = fr.read()) != -1) {
                fw.write(charData);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9.3 Introduction au Cumul des Filtres

En Java, il est possible de **cumuler des filtres** dans les flux d'E/S pour ajouter plusieurs couches de traitement sur les données lues ou écrites. Cette technique consiste à enchaîner plusieurs flux pour ajouter des fonctionnalités spécifiques, comme la lecture de types de données de haut niveau, le tamponnage pour optimiser la performance, etc.

9.3.1 Qu'est-ce qu'un Filtre ?

Un filtre est un flux qui modifie ou améliore le traitement des données d'un flux existant. Par exemple :

— `BufferedInputStream` améliore les performances de lecture en ajoutant un tampon.

— `DataInputStream` permet de lire des types de données Java (entiers, chaînes, etc.) à partir d'un flux d'octets.

En combinant plusieurs de ces filtres, on peut cumuler leurs fonctionnalités, ce qui permet de traiter les données de manière plus flexible et efficace.

9.4 Exemples de Cumul de Filtres

9.4.1 Exemple 1 : Combinaison de `FileInputStream` et `BufferedInputStream`

Dans cet exemple, nous allons lire un fichier en ajoutant un tampon pour optimiser la lecture.

Code :

```
import java.io.*;

public class CumulFiltresExample1 {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("exemple.txt");
            BufferedInputStream bis = new BufferedInputStream(fis);

            int byteData;
            while ((byteData = bis.read()) != -1) {
                System.out.print((char) byteData);
            }

            bis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explication :

- `FileInputStream` est utilisé pour accéder aux octets bruts du fichier.
- `BufferedInputStream` ajoute une couche de tampon, ce qui réduit le nombre d'accès disque et optimise la lecture.

9.4.2 Exemple 2 : Combinaison de `FileInputStream`, `DataInputStream`, et `BufferedInputStream`

Nous allons maintenant lire des données de haut niveau en ajoutant les filtres 'DataInputStream' et 'BufferedInputStream' pour lire et optimiser la lecture.

Code :

```
import java.io.*;

public class CumulFiltresExample2 {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("data.bin");
            DataInputStream dis = new DataInputStream(new BufferedInputStream(fis));
```

```
8
9               int entier = dis.readInt(); // Lecture d'un entier
10              double reel = dis.readDouble(); // Lecture d'un
    nombre réel
11              dis.close();
12
13              System.out.println("Entier lu : " + entier);
14              System.out.println("Réel lu : " + reel);
15          } catch (IOException e) {
16              e.printStackTrace();
17          }
18      }
19  }
```

Explication :

- `FileInputStream` accède aux données brutes du fichier.
- `BufferedInputStream` optimise la lecture en ajoutant un tampon.
- `DataInputStream` permet de lire des types de données Java de haut niveau (comme des entiers et des réels).

9.4.3 Exercice 1 : Lecture d'un Fichier Texte avec des Filtres

Dans cet exercice, vous allez lire le contenu d'un fichier texte en utilisant un cumul de filtres pour améliorer les performances.

Instructions :

1. Créez un fichier nommé `"exemple.txt"` contenant plusieurs lignes de texte.
2. Utilisez un `FileInputStream` pour accéder au fichier.
3. Ajoutez un `BufferedInputStream` pour optimiser la lecture.
4. Lisez et affichez le contenu du fichier ligne par ligne.

Solution suggérée :

```
1   import java.io.*;
2
3   public class Exercice1 {
4       public static void main(String[] args) {
5           try (BufferedInputStream bis = new BufferedInputStream
    (new FileInputStream("exemple.txt"))) {
6               int byteData;
7               while ((byteData = bis.read()) != -1) {
8                   System.out.print((char) byteData);
9               }
10          } catch (IOException e) {
11              e.printStackTrace();
12          }
13      }
14  }
```

9.4.4 Exercice 2 : Lecture de Données Binaires avec des Filtres

Dans cet exercice, vous allez lire des données binaires depuis un fichier en utilisant des filtres pour traiter des types de données de haut niveau.

Instructions :

1. Créez un fichier binaire `"data.bin"` qui contient un entier et un réel (utilisez un autre programme Java pour écrire ces données).
2. Utilisez un `FileInputStream` pour lire le fichier.
3. Ajoutez un `BufferedInputStream` pour optimiser la lecture.
4. Utilisez `DataInputStream` pour lire l'entier et le réel du fichier et les afficher dans la console.

Solution suggérée :

```
import java.io.*;

public class Exercice2 {
    public static void main(String[] args) {
        try (DataInputStream dis = new DataInputStream(new
   BufferedInputStream(new FileInputStream("data.bin")))) {
            int entier = dis.readInt();
            double reel = dis.readDouble();

            System.out.println("Entier lu : " + entier);
            System.out.println("Réel lu : " + reel);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9.4.5 Les Fichiers à Accès Direct

La classe `RandomAccessFile` permet de lire et d'écrire dans un fichier à des positions spécifiques, offrant ainsi un accès direct aux données.

Caractéristiques de RandomAccessFile

- Encapsule les opérations de lecture et écriture sur un fichier.
- Implémente les interfaces `DataInput` et `DataOutput`.
- Deux modes d'accès :
  - `r` : Lecture seule.
  - `rw` : Lecture et écriture.

Exemple : Écriture dans un fichier avec RandomAccessFile

```
import java.io.*;

public class TestRandomAccessFile {
    public static void main(String[] args) {
        try {
```

```
            RandomAccessFile fichier = new RandomAccessFile("monfichier.dat", "rw");
            for (int i = 0; i < 10; i++) {
                fichier.writeInt(i * 100); // Écriture de nombres entiers
            }
            fichier.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice :

— Modifiez le programme pour écrire également des chaînes de caractères après chaque entier.

— Testez la lecture des données pour vérifier leur cohérence.

Exemple : Lecture à une position spécifique

```
import java.io.*;

public class LecturePositionSpecifique {
    public static void main(String[] args) {
        try {
            RandomAccessFile fichier = new RandomAccessFile("monfichier.dat", "r");
            fichier.seek(5 * 4); // Déplacement à la 6e donnée (indice 5)
            int valeur = fichier.readInt();
            System.out.println("Valeur à la position 6 : " + valeur);
            fichier.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice :

— Créez un fichier avec des données de types variés (entiers et chaînes).

— Modifiez le programme pour lire des données à plusieurs positions précises.

9.5 Les Flux de Caractères

9.5.1 Définition

Un flux de caractères en Java permet de transporter des données textuelles (caractères). Ces flux sont généralement utilisés pour traiter des fichiers texte ou des

flux réseau contenant du texte. Les classes abstraites `Reader` et `Writer` servent de base pour ces flux.

9.5.2 Cycle d'utilisation

Le cycle d'utilisation d'un flux de caractères suit trois étapes principales :

1. Ouverture du flux
2. Lecture ou écriture des caractères
3. Fermeture du flux

9.5.3 Les Classes Principales des Flux de Caractères

Les flux de caractères en Java se divisent en deux grandes catégories : les flux de communication et les flux de traitement. Voici un tableau récapitulatif des classes principales :

Classes de communication

— **FileReader** : Permet de lire des fichiers texte caractère par caractère.
— **FileWriter** : Permet d'écrire des données textuelles dans un fichier.

9.5.4 Exemple : Lecture avec FileReader

Dans cet exercice, vous allez lire le contenu d'un fichier texte en utilisant `FileReader`.

Instructions :

1. Créez un fichier texte nommé `"monfichier.txt"` contenant du texte de votre choix.
2. Utilisez la classe `FileReader` pour lire le contenu du fichier.
3. Affichez le contenu du fichier dans la console, caractère par caractère.

```
1  import java.io.*;
2
3  public class LectureFluxCaracteres {
4      public static void main(String[] args) {
5          try {
6              FileReader fichier = new FileReader("monfichier.
       txt");
7              int caractere;
8              while ((caractere = fichier.read()) != -1) {
9                  System.out.print((char) caractere);
10             }
11             fichier.close();
12         } catch (IOException e) {
13             e.printStackTrace();
14         }
15     }
16 }
```

Exercice d'application :

— Créez un fichier nommé `texte.txt` contenant plusieurs lignes de texte.
— Modifiez le programme pour lire et afficher le contenu du fichier ligne par ligne.

9.5.5 Exemple : Écriture avec FileWriter

Cet exercice consiste à écrire du texte dans un fichier en utilisant `FileWriter`.

Instructions :

1. Créez un fichier nommé `"sortie.txt"`.
2. Utilisez la classe `FileWriter` pour y écrire une chaîne de caractères.
3. Fermez le flux après l'écriture pour libérer les ressources.

```
import java.io.*;

public class EcritureFluxCaracteres {
    public static void main(String[] args) {
        try {
            FileWriter fichier = new FileWriter("sortie.txt");
            fichier.write("Bonjour, ceci est un exemple de
   flux de caractères.\n");
            fichier.write("Ajoutons une deuxième ligne au
   fichier.");
            fichier.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice d'application :

— Modifiez le programme pour ajouter une nouvelle ligne au fichier `sortie.txt` sans écraser le contenu existant.

Classes de traitement

— **BufferedReader** : Améliore les performances de lecture grâce au tamponnage et permet de lire une ligne complète.
— **BufferedWriter** : Améliore les performances d'écriture grâce au tamponnage.
— **PrintWriter** : Permet d'écrire des données textuelles avec des fonctionnalités supplémentaires comme `printf()`.

9.5.6 Exemple : Lecture avec BufferedReader

```
import java.io.*;

public class ExempleBufferedReader {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new
   FileReader("monfichier.txt"));
```

```
            String ligne;
            while ((ligne = reader.readLine()) != null) {
                System.out.println(ligne);
            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice :

— Créez un fichier contenant au moins cinq lignes de texte.

— Modifiez le programme pour compter le nombre total de lignes dans le fichier.

9.5.7 Exemple : Écriture avec BufferedWriter

```
import java.io.*;

public class ExempleBufferedWriter {
    public static void main(String[] args) {
        try {
            BufferedWriter writer = new BufferedWriter(new
   FileWriter("sortie.txt", true));
            writer.write("Cette ligne est ajoutée avec
   BufferedWriter.\n");
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exercice :

— Modifiez le programme pour écrire plusieurs lignes dans le fichier `sortie.txt` en une seule opération.

9.5.8 Exercice : Utilisation de `BufferedReader` et `BufferedWriter`

Dans cet exercice, vous allez lire et écrire du texte de manière plus efficace en utilisant un tampon.

Instructions :

1. Utilisez `BufferedReader` pour lire le contenu de `"texteExemple.txt"` ligne par ligne.
2. Utilisez `BufferedWriter` pour écrire le contenu dans un nouveau fichier, `"texteTamponne.txt"`.

Exemple de Code :

```
import java.io.*;
```

```
2
3  public class BufferedCharacterExample {
4      public static void main(String[] args) {
5          try (BufferedReader br = new BufferedReader(new
       FileReader("texteExemple.txt"));
6               BufferedWriter bw = new BufferedWriter(new
       FileWriter("texteTamponne.txt"))) {
7              String line;
8              while ((line = br.readLine()) != null) {
9                  bw.write(line);
10                 bw.newLine(); // Ajoute un saut de ligne après
       chaque ligne
11             }
12         } catch (IOException e) {
13             e.printStackTrace();
14         }
15     }
16 }
```

9.5.9 Conclusion

Les flux de caractères, grâce à des classes comme `FileReader`, `FileWriter`, `BufferedReader`, et `BufferedWriter`, facilitent la manipulation de données textuelles. Leur utilisation dépend des besoins spécifiques en termes de performance et de fonctionnalités, mais ils constituent une base essentielle pour le traitement de fichiers texte en Java.

9.6 Exercices Non Guidés

Exercice 1 : Copier le Contenu d'un Fichier avec des Flux Tamponnés

Créez un programme qui lit le contenu de `"source.txt"` et l'écrit dans `"destination.txt"` en utilisant des flux d'octets tamponnés.

Exercice 2 : Compter les Caractères dans un Fichier

Écrivez un programme qui lit un fichier avec `FileReader` et compte le nombre d'occurrences d'un caractère spécifique (par exemple, 'e'). Affichez le nombre total à la fin.

Exercice 3 : Lecture de l'Entrée Utilisateur et Écriture dans un Fichier

Demandez à l'utilisateur d'entrer du texte depuis la console et enregistrez cette entrée dans un fichier nommé `"userInput.txt"`. Continuez de lire jusqu'à ce que l'utilisateur entre "exit".

Exercice 4 : Gestion des Exceptions avec les Opérations d'E/S

Améliorez le programme de l'Exercice 5 pour gérer les exceptions de manière élégante. Affichez un message convivial si le fichier n'est pas trouvé ou si une erreur

se produit lors de la lecture ou de l'écriture.

Exercice 5 : Utilisation du Fichier d'Accès Aléatoire

Utilisez `RandomAccessFile` pour lire et écrire des données à une position spécifique dans un fichier. Écrivez un programme pour :

— Écrire "Bonjour, le monde !" au début de `"randomFile.txt"`.
— Lire à partir d'une position spécifique dans le fichier et afficher le contenu.