Cours de Java Avancé

Nancy Awad

November 2021

1 Les flux d'entrée/sortie

1.1 Présentation des flux:

Un programme a souvent besoin d'échanger des informations, soit pour recevoir des données d'une source, soit pour envoyer des données à un destinataire.
La source et la destination de ces échanges peuvent être de plusieurs natures:

- un fichier
- une socket réseau
- un autre programme

De même, la nature des données échangées peut être diverse: du texte, des images, du son, des objets...

Le flux de données est traité avec le package **java.io**
Les flux sont toujours traités de façon séquentielle.
Les flux peuvent être des flux d'entrée (lecture) ou des flux de sortie (ecriture).
Les flux peuvent contenir soit des caractères soit des octets.

1.2 Les classes basiques de gestion des flux de java.io

Le paquet **java.io** contient un nombre assez important de classes qui traitent les entrées et sorties Java. Ce qui déroute dans l'utilisation de ces classes, c'est leur nombre et la difficulté de choisir celle qui convient le mieux en fonction des besoins. Les classes traitent respectivement deux caracteristiques: le type du flux (octets ou cractèrs) et le sens du flux (entrée ou sortie). Le nom des classes se compose d'un préfixe et d'un suffixe.

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

Figure 1: Les quatre suffixes possibles (classes abstraites)

Aux classes abstraites {*Reader, Writer, InputStream, OutputStream*}, on rajoute un préfixe qui exprime le type du traitement qu'il effectue. Le type de traitement peut être:

- **Buffered** : ce type de filtre permet de mettre les données du flux dans un tampon. Il peut être utilisé en entrée et en sortie
- **Sequence** : ce filtre permet de fusionner plusieurs flux

- **Data** : ce type de flux permet de traiter les octets sous forme de type de données
- **File** : ce filtre est utilisé pour lire ou ercrire dans un fichier
- **LineNumber** : ce filtre permet de numéroter les lignes contenues dans le flux
- **PushBack** : ce filtre permet de remettre des données lues dans le flux
- **Print** : ce filtre permet de réaliser des impressions formatées
- **Object** : ce filtre est utilisé par la sérialisation
- **InputStream / OuputStream** : ce filtre permet de convertir des octets en caractères

Ainsi plusieurs classes sont definis dans le package java.io héritant d'une des 4 classes abstraites {Reader, Writer, InputStream, OutputStream}

	Flux en lecture	Flux en sortie
Flux de caractères	BufferedReader	BufferedWriter
	CharArrayReader	CharArrayWriter
	FileReader	FileWriter
	InputStreamReader	OutputStreamWriter
	LineNumberReader	
	PipedReader	PipedWriter
	PushbackReader	
	StringReader	StringWriter
Flux d'octets	BufferedInputStream	BufferedOutputStream
	ByteArrayInputStream	ByteArrayOutputStream
	DataInputStream	DataOuputStream
	FileInputStream	FileOutputStream
	ObjectInputStream	ObjetOutputStream
	PipedInputStream	PipedOutputStream
		PrintStream
	PushbackInputStream	
	SequenceInputStream	

Figure 2: Les differentes classes de java.io

Toutes ces classes contiennent des constructeurs et des méthodes specifiques pour manipuler precisement le flux en entrée et en sortie.

Quelques exemples:

```java
import java.io.*;

public class TestBufferedReader {
  protected String source;

  public TestBufferedReader(String source) {
    this.source = source;
    lecture();
  }

  public static void main(String args[]) {
    new TestBufferedReader("source.txt");
  }

  private void lecture() {
    try {
      String ligne ;
      BufferedReader fichier = new BufferedReader(new FileReader(source));

      while ((ligne = fichier.readLine()) != null) {
          System.out.println(ligne);
      }

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 3: Lecture bufférisée

Dans l'exemple de la Figure 3:
La méthode readLine() permet de lire dans le flux une ligne, une suite de caractères qui se termine par un retour chariot '\r' ou un saut de ligne '\n' ou les deux.
Le constructeur BufferedReader(Reader): prend en paramètre un flux à lire.
Il faut noter qu'une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

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

public class TestBufferedWriter {
  protected String destination;

  public TestBufferedWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestBufferedWriter("print.txt");
  }

  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      BufferedWriter fichier = new BufferedWriter(new FileWriter(destination));

      fichier.write("bonjour tout le monde");
      fichier.newLine();
      fichier.write("Nous sommes le "+ new Date());
      fichier.write(", le nombre magique est " + nombre);

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 4: L'écriture bufférisée

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

public class CopieFichier {

  public static void main(String args[]) {
    try {
      copierFichier("source.txt", "copie.txt");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public static void copierFichier(String source, String destination) throws IOException {
    FileInputStream fis = null;
    FileOutputStream fos = null;

    try {
      byte buffer[] = new byte[1024];
      int taille = 0;

      fis = new FileInputStream(source);
      fos = new FileOutputStream(destination);
      while ((taille = fis.read(buffer)) != -1) {
        System.out.println(taille);
        fos.write(buffer, 0, taille);
      }
    } finally {
      if (fis != null) {
        try {
          fis.close();
        } catch (IOException e) {
        }
      }
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
        }
      }
    }
  }
}
```

Figure 5: Lecture et écriture de flux en octets

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble d'octets plutôt que de traiter les données octet par octet. Le nombre d'opérations est ainsi réduit.

1.3 La classe File

La classe **File**, est une classe importante dans le package java.io. Cette classe représente un nom de fichier ou de répertoire d'une manière indépendante du système. Elle fournit des méthodes pour :

- Ouvrir des fichiers sur le disque pour l'ecriture ou la lecture (avec FileReader, FileWriter, FileInputStream et FileOutputStream)
- Créer des fichiers temporaires (createNewFile, createTempFile, deleteOnExit)
- Gére des attributs "caché" et "lecture seule" (isHidden, isReadOnly)

- Lister les répertoires
- Consulter les attributs des fichiers
- Renommer et supprimer des fichiers

Il n'existe pas de classe pour traiter les répertoires car ils sont considérés comme des fichiers. Une instance de la classe File est une représentation logique d'un fichier ou d'un répertoire qui peut ne pas exister physiquement sur le disque.

```java
import java.io.*;

public class TestFile {
  protected String nomFichier;
  protected File fichier;

  public TestFile(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+" n'existe pas");
      System.exit(1);
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
    System.out.println(" Droit d'ecriture  : "+fichier.canWrite());

    if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      String fichiers[] = fichier.list();
      for(int i = 0; i > fichiers.length; i++) System.out.println("  "+fichiers[i]);
    }
  }
}
```

Figure 6: Gestion des fichiers

1.4 Le filtrage

Le filtrage est utilisé pour lire ou écrire un sous-ensemble de données à partir d'un flux d'entrée ou de sortie beaucoup plus important. Le filtrage peut être indépendant du format des données, (compter le nombre d'éléments dans la liste) ou peut être directement lié au format des données (obtenir toutes les données d'une certaine ligne d'un tableau). Vous attachez un flux de filtre à un flux d'entrée ou de sortie pour filtrer les données.
Les Classes les plus importantes pour le filtrage sont:

- PushbackInputStream
- PushbackReader
- LineNumberReader
- PrintWriter, PrintStream

1.5 La serialisation

La sérialisation est un procédé qui permet de rendre un objet ou un graphe d'objets de la JVM persistant pour stockage ou échange et vice versa. Cet objet est mis sous une forme sous laquelle il pourra être reconstitué à l'identique. Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau.

1.5.1 L'interface Serializable

L'interface **Serializable**, ne définit aucune méthode mais permet simplement de marquer une classe comme pouvant être sérialisée.
Tout objet qui doit être sérialisé grâce au mécanisme par défaut doit implémenter cette interface ou une de ses classes mères doit l'implémenter.

```java
public class Personne implements java.io.Serializable {
  private String nom    = "";
  private String prenom = "";
  private int    taille = 0;

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

  public String getNom() {
    return this.nom;
  }

  public void setNom(final String nom) {
    this.nom = nom;
  }

  public int getTaille() {
    return this.taille;
  }

  public void setTaille(final int taille) {
    this.taille = taille;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(final String prenom) {
    this.prenom = prenom;
  }
}
```

Figure 7: Une classe sérialisable

1.5.2 Le mot clé Transient

Les champs statiques, ou attributs de classe, ne sont pas enregistrés car ils ne font pas partie d'un objet. Si vous ne voulez pas qu'un attribut de données soit sérialisé, vous pouvez le rendre "transient" (transitoire). Cela permet d'économiser sur la quantité de stockage ou de transmission nécessaire pour transmettre un objet.

```java
public class Foo implements Serializable
 {
   private String saveMe;
   private transient String dontSaveMe;
   private transient String password;
   //...
 }
```

Figure 8: Attributs transitoires en sérialisation

1.5.3 Ecriture d'un objet

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

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

public class SerializerDonnees {

  public static void main(final String argv[]) {
    final Personne personne = new Personne("Dupond", "Jean", 175, "1234");
    ObjectOutputStream oos = null;

    try {
      final FileOutputStream fichier = new FileOutputStream("donnees.ser");
      oos = new ObjectOutputStream(fichier);
      oos.writeObject(new java.util.Date());
      oos.writeObject(personne);
      final int[] tableau = { 1, 2, 3 };
      oos.writeObject(tableau);
      oos.writeUTF("ma chaine en UTF8");
      oos.writeLong(123456789);
      oos.writeObject("ma chaine de caracteres");

      oos.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      try {
        if (oos != null) {
          oos.flush();
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Figure 9: L'ecriture d'un objet en sérialisation

La classe ObjectOutputStream contient aussi plusieurs méthodes qui permettent de sérialiser des types élémentaires : writeInt(), writeDouble(), writeFloat(), ... Il est possible dans un même flux d'écrire plusieurs objets les uns à la suite des autres. Ainsi plusieurs objets peuvent être sérialisés. Dans ce cas, il faut faire

attention de relire les objets dans leur ordre d'écriture.

	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
00	ACED	0005	7372	000E	6A61	7661	2E75	7469	¬í..sr..java.uti
10	6C2E	4461	7465	686A	8101	4B59	7419	0300	l.DatehjÐ.KYt...
20	0078	7077	0800	0001	409B	99B1	7278	7372	.xpw....@›™±rxsr
30	0008	5065	7273	6F6E	6E65	6F32	A2F0	796B	..Personneo2¢ðyk
40	7022	0200	0349	0006	7461	696C	6C65	4C00	p"...I..tailleL.
50	036E	6F6D	7400	124C	6A61	7661	2F6C	616E	.nomt..Ljava/lan
60	672F	5374	7269	6E67	3B4C	0006	7072	656E	g/String;L..pren
70	6F6D	7100	7E00	0378	7000	0000	AF74	0006	omq.~..xp...¯t..
80	4475	706F	6E64	7400	044A	6561	6E75	7200	Dupondt..Jeanur.
90	025B	494D	BA60	2676	EAB2	A502	0000	7870	.[IM°`&vê²¥...xp
A0	0000	0003	0000	0001	0000	0002	0000	0003	....
B0	771B	0011	6D61	2063	6861	696E	6520	656E	w...ma chaine en
C0	2055	5446	3800	0000	0007	5BCD	1574	0017	UTF8.....[Í.t..
D0	6D61	2063	6861	696E	6520	6465	2063	6172	ma chaine de car
E0	6163	7465	7265	73					acteres

Figure 10: Fichier contenat des données serialisées

1.5.4 Lecture d'un objet sérialisé

A la relecture des objets sérialisés, il est impératif de connaitre l'ordre de sérialisation (on peut toutefois s'en sortir en faisant de l'introspection). L'exception particulière qui peut être levée est ClassNotFoundException dans le cas ou la JVM ne trouve pas la définition de la classe à charger. La classe **ObjectInputStream** a pour but de désérialiser un objet précédemment sérialisé.

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

public class DeSerializerPersonne {
  public static void main(final String argv[]) {

    ObjectInputStream ois = null;

    try {
      final FileInputStream fichier = new FileInputStream("personne.ser");
      ois = new ObjectInputStream(fichier);
      final Personne personne = (Personne) ois.readObject();
      System.out.println("Personne : ");
      System.out.println("nom : " + personne.getNom());
      System.out.println("prenom : " + personne.getPrenom());
      System.out.println("taille : " + personne.getTaille());
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Figure 11: Deserialisation des données

La méthode readObject() renvoie une instance de type Object qu'il est nécessaire de caster vers le type présumé. Pour être sûr du type, il est possible d'effectuer un test sur le type de l'objet retourné en utilisant l'opérateur instanceof.

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

public class DeSerializerPersonne {
  public static void main(final String argv[]) {

    ObjectInputStream ois = null;

    try {
      final FileInputStream fichier = new FileInputStream("personne.ser");
      ois = new ObjectInputStream(fichier);
      final Personne personne = (Personne) ois.readObject();
      System.out.println("Personne : ");
      System.out.println("nom : " + personne.getNom());
      System.out.println("prenom : " + personne.getPrenom());
      System.out.println("taille : " + personne.getTaille());
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Figure 12: Résultat de la desérialisation des données

1.5.5 La sérialisation et la sécurité

La sérialisation n'est pas sécurisée : même si le format par défaut est un format binaire, ce format est connu et documenté. Le contenu des données binaires peut assez facilement permettre de définir une classe qui permettra de lire le contenu du résultat de la sérialisation.

Un simple éditeur hexadécimal permet d'obtenir les valeurs des différents champs sérialisés même ceux qui sont déclarés privés.

Il ne faut pas sérialiser de données sensibles par le processus de sérialisation standard car cela rend public ces données. Une fois un objet sérialisé, les mécanismes d'encapsulation de Java ne sont plus mis en oeuvre : il est possible d'accéder aux champs private par exemple. Il faut soit :

- exclure ces champs de la sérialisation
- encrypter/décrypter ces données avec une personnalisation de la sérialisation

- encrypter tous les résultats de la sérialisation en utilisant la classe javax.crypto.SealedObject ou la classe java.security.SignedObject qui implémente l'interface Serializable.
  Elles permettent d'encrypter et de signer un objet