TD2 - Sérialisation et transferts de données

19 septembre 2024

Ressources :

- https://zestedesavoir.com/tutoriels/646/apprenez-a-programmer-en-java/557_java-oriente-objet/2703_les-flux-dentreesortie/
- https://www.jmdoudoux.fr/java/dej/chap-nio2.htm
- http://web.mit.edu/6.031/www/sp19/classes/23-sockets-networking/
- https://www.jmdoudoux.fr/java/dej/chap-flux.htm
- http://remy-manu.no-ip.biz/Java/Tutoriels/ExceptionsFluxFichiers/FluxFichiers.htm
- https://docs.oracle.com/cd/E56338_01/html/E53799/gnkor.html
- https://www3.ntu.edu.sg/home/ehchua/programming/java/j5b_io.html
- https://www.json.org/json-fr.html
- https://jsoneditoronline.org/
- https://github.com/FasterXML/jackson-docs

La sérialisation consiste à transformer une information dans un format permettant le transfert en série d'octets. Que ce soit pour stocker des données dans un fichier, ou les envoyer dans le réseau, c'est une opération qui revient régulièrement.

Pour illustrer les propos de ce cours, nous utiliserons Java.

1 (Ré)-introduction aux flux de données

Un flux d'entrée ou de sortie (*input/output*, abrégé en *I/O* ou *IO*) est un transfert séquentiel de données, d'un point de départ vers un point d'arrivée.

Exemples de points de départ : clavier, souris, fichier sur le disque, `socket`, etc. Exemples de points d'arrivées : sortie standard, fichier, `socket`, etc.

Ces données peuvent être transmises sous différentes formes : octets, types primitifs, caractères ou objets.

Java proposait initialement une série de classes qui encapsulent ces transferts, dont les premières sont InputStream et OutputStream. Ces classes **abstraites** proposent des méthodes permettant de lire des **octets**. Les chaînes de caractères étant encodées en Unicode, dont sur 2 octets, il existe également deux classes abstraites permettant de les manipuler : `Reader` et Writer.

Cependant, un développeur préférera travailler avec des données de plus haut niveau : des entiers, des booléens, des flottants, des images, voire des Objets. Il existe alors divers classes héritières permettant de manipuler efficacement ces données (voir fig. 1).

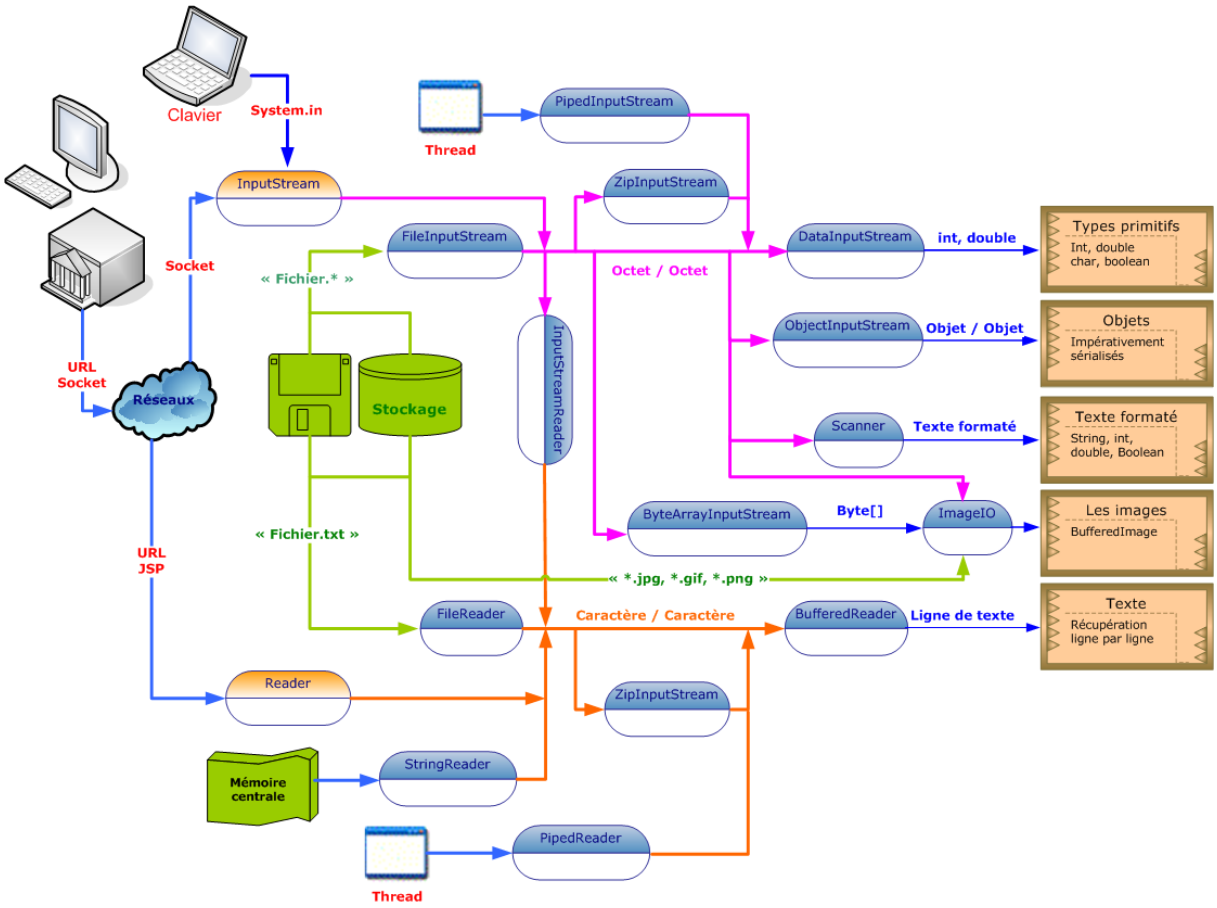

FIGURE 1 – Schéma des diverses classes permettant le flux de données avec Java I/O

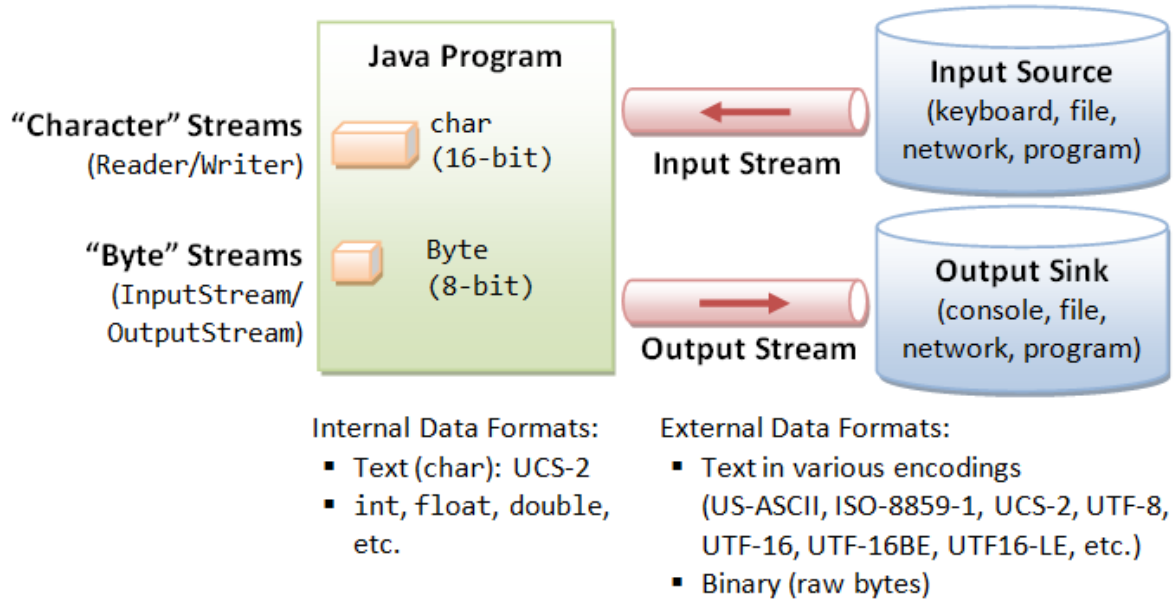

FIGURE 2 – Schéma d'un transfert de données avec Java I/O.

Source: https://www.geeksforgeeks.org/java-io-input-output-in-java-with-examples/

On remarque qu'il existe des classes labelisées Buffered. Ces classes implémentent une mémoire supplémentaire, aussi appelée **tampon**. Lorsqu'une opération de lecture est effectuée, on lit autant de données que possible que l'on stocke dans le tampon. Par exemple, si un tampon a une taille de 4 ko et qu'on souhaite lire un fichier, le programme charge 4 ko du fichier dans le tampon, puis le programme dispose de ces 4 ko. Une fois lus, ils sont évacués du tampon et le programme peut à nouveau charger 4 ko, etc.

2 Avec une implémentation native

Java dispose de l'interface `Serializable`, qui permet de marquer une classe qui peut être sérialisée. Elle va de pair avec la classe `ObjectOutputStream`, dans laquelle on écrira les données de l'objet.

1. Dans un package nommé `testserial`, créer une classe `Entity2D` qui comme son nom l'indique, permet de représenter les informations d'une entité dans un espace à deux dimensions. Implémenter les `getter` et `setter` des attributs.

```
1  public class Entity2D implements Serializable {
2      private static final long serialVersionUID = 1L;
3      public static final int MAX_ITEMS = 10;
4      public static int nb_generated = 0;
5      private int id;
6      private String name;
7      private float x;
8      private float y;
9      private ArrayList<Integer> items;
10
11     public Entity2D(String name, float x, float y){
12         this.name = name;
```

```
13          this.x = x;
14          this.y = y;
15          this.id = nb_generated;
16          nb_generated++;
17          items = new ArrayList<Integer>();
18      }
19  }
```

2. Importer les packages nécessaires.

```
1  import java.util.ArrayList;
2  import java.io.IOException;
3  import java.io.ObjectInputStream;
4  import java.io.ObjectOutputStream;
5  import java.io.Serializable;
```

3. Et implémenter les méthodes `writeObject` et `readObject`. `readObject` instancie la classe à partir des octets contenus dans le fichier.

```
1  private void writeObject(ObjectOutputStream out) throws IOException {
2      out.defaultWriteObject();
3  }
4
5  private void readObject(ObjectInputStream in) throws IOException,
       ClassNotFoundException {
6      in.defaultReadObject();
7  }
```

A noter que c'est la JVM qui est responsable de la sérialisation dans ce cas.

4. Implémenter dans une classe `main` un test dans lequel on génère une instance `Entity2D`, qu'on sérialise puis désérialise. Dans l'exemple ci-dessous, `putItem` est une méthode qui ajoute un entier dans `items`.

```
1  public static void main(String[] args) {
2      Entity2D ent_1 = new Entity2D("test1", 0.0f, 0.0f);
3      ent_1.putItem(5);
4      ent_1.putItem(7);
5      ent_1.putItem(-1);
6      ObjectOutputStream oos = null;
7
8      // Writing into a file
9      try {
10         FileOutputStream fichier = new FileOutputStream("donnees.ser")
       ;
11         oos = new ObjectOutputStream(fichier);
12         oos.writeObject(ent_1);
13         oos.flush();
14         oos.close();
15         fichier.close();
16     }
17     catch (IOException e) {
18         e.printStackTrace();
```

```
19      }
20
21      // How long is the file [8]
22      File saved = new File("donnees.ser");
23      System.out.println("Taille du fichier : " + saved.length() + "
        octets");
24
25      // à compléter ...
```

```
Avant : Entity2D [id=0, name=test1, x=0.0, y=0.0, items=[5, 7, -1]]
Taille du fichier : 285 octets
Entity2D [id=0, name=test1, x=0.0, y=0.0, items=[5, 7, -1]]
```

FIGURE 3 – Résultat du test de sérialisation.

3 Avec une implémentation personnalisée

Java utilise l'introspection pour sérialiser automatiquement les instances, ce qui a un coût non-négligeable. Un guide de bonnes pratiques d'Oracle [1] suggère d'utiliser à la place l'interface `Externalizable`. Si une classe implémente cette interface, alors elle doit disposer d'un constructeur sans argument et des méthodes `writeExternal` et `readInternal`, et dans ces méthodes, la sérialisation est gérée intégralement par le développeur.

D'après la documentation, ces méthodes font intervenir respectivement en paramètre un `ObjectOutput` et un `ObjectInput`. Nous détaillons le premier :

`ObjectInput` est une interface qui hérite de `DataOutput`, dont le but est d'écrire des octets. C'est à nous de le faire manuellement à partir des méthodes proposées. Il est donc possible d'écrire des `int`, `float`, `char`, `String`, mais également des objets sérialisables. Heureusement, `ArrayList` est sérialisable, donc sa problématique ne se posera pas ici. C'est au développeur de déterminer l'ordre d'écriture des attributs, mais également leur encodage.

```
1  @Override
2  public void writeExternal(ObjectOutput out) throws IOException {
3      // TODO Auto-generated method stub
4      out.writeInt(id);
5      // ...
6  }
```

```
1  @Override
2  public void readExternal(ObjectInput in) throws IOException,
       ClassNotFoundException {
3      id = in.readInt();
4      // ...
5  }
```

1. http://web.archive.org/web/20141017081821/http://docs.oracle.com/cd/E15051_01/wls/docs103/jms/design_bes

1. Remplacer l'interface `Sérializable` par `Externalizable`, et implémenter les deux méthodes `writeExternal` et `readInternal` dans `Entity2D`.
2. En suivant le même principe que précédemment, élaborer un test de sérialisation.

```
Avant : Entity2D [id=0, name=test1, x=0.0, y=0.0, items=[5, 7, -1]]

Taille du fichier : 213 octets
Entity2D [id=0, name=test1, x=0.0, y=0.0, items=[5, 7, -1]]
```

FIGURE 4 – Résultat du test avec `Externalizable`. Le fichier obtenu est moins lourd.

3. Commenter la taille du fichier ainsi généré, en comptant manuellement la taille totale des attributs de `Entity2D`. On considère que l'`ArrayList` d'entiers pèse 4 octets pour compter le nombre d'éléments plus 4 octets par élément à l'intérieur, et que la chaîne de caractères compte 10 caractères en moyenne.
   Avec ces deux possibilités, les données sérialisées sont compréhensibles par la JVM. Cependant, nous souhaiterions avoir la possibilité d'envoyer la quantité de données la plus faible possible pour un Entity2D.
4. Implémenter une méthode qui permet d'écrire dans un flux de données donné en paramètre les données de cette instance d'Entity2D. Il est cette fois impossible de sérialiser directement `items`. Trouver une méthode permettant de le faire à partir des méthodes de sérialisation de types primitifs.

```
1 public void toBytes(DataOutputStream data)
```

5. Implémenter la méthode **statique** qui permet de générer une instance d'Entity2D à partir des données récoltées dans le flux d'entrée, et de la retourner.

```
1 public static Entity2D fromBytes(DataInputStream data)
```

6. Implémenter un test de cette troisième méthode de sérialisation, et constater la taille du packet transféré.
7. Nous avons travaillé sur une seule instance d'Entity2D dans nos exemples. Nous souhaitons à présent s'approcher un peu plus d'un scénario de test, où coexistent plusieurs instances d'Entity2D, de l'ordre de grandeur allant de 10^2 à 10^3, qu'on souhaite sérialiser. Proposer une implémentation avec deux méthodes de votre choix.

```
Avant : Entity2D [id=0, name=test1, x=0.0, y=0.0, items=[5, 7, -1]]
Taille du fichier : 32 octets
Entity2D [id=0, name=test1, x=0.0, y=0.0, items=[5, 7, -1]]
```

FIGURE 5 – Résultat du test avec une implémentation de sérialisation totalement personnalisée.

4 En format texte

C'est le type de sérialisation le plus plébiscité dans le développement Web, et les API. Le format star utilisé est le JSON. Le JSON, pour *JavaScript Object Notation*, est

```
Avant : Entity2D [id=0, name=test1, x=0.0, y=0.0, items=[5, 7, -1]]
Taille du fichier : 58 octets
Entity2D [id=0, name=test1, x=0.0, y=0.0, items=[5, 7, -1]]
```

FIGURE 6 – Résultat du test avec la conversion en JSON

un format léger d'échange de données conçu pour être facile à lire et à écrire par des humains. Il est **indépendant** de tout langage de programmation, bien qu'il soit basé sur JavaScript. JSON est caractérisé par deux structures essentielles :

1. une collection de couples clef/valeur, qui peut se représenter par un objet, un dictionnaire, une table associative, etc. ;
2. une liste de valeurs ordonnées, qui peut se représenter par un tableau, un vecteur, une liste, etc.

Nous utiliserons ici la librairie la plus connue et répandue en Java : Jackson. Pour l'installer, télécharger les archives `core`, `databind` et `annotations` puis les importer avec Eclipse dans le projet. On pourra les trouver dans le lien suivant :
https://mvnrepository.com/artifact/com.fasterxml.jackson.core.

1. A l'aide de la classe ObjectMapper [2], ajouter à la classe Entity2D une méthode statique qui génère une instance à partir d'un String contenant un JSON, et une méthode qui permet de générer un String au format JSON des données de l'instance [3].
2. Effectuer un test de ces deux méthodes, et vérifier le fichier ainsi que sa taille en octets.

2. Voir https://www.baeldung.com/jackson-object-mapper-tutorial
3. Ces méthodes créées le sont à titre d'exemple. La librairie propose des décorateurs à insérer dans la classe, et la sérialisation et déserialisation s'effectuent plutôt en dehors, comme indiqué dans le tutoriel.

5 Les sockets

Un *socket* est un objet permettant d'ouvrir une connexion et de communiquer avec une machine locale ou distante. Il a été introduit dans les distributions de Berkeley (un système UNIX).

Ils permettent la réception et l'émission de messages à partir de plusieurs protocoles de communications, comme PF_INET, AF_INET, PF_UNIX, et bien d'autres. Dans le cas d'une communication client-serveur, il y aura deux programmes distincts.

On donne le code de base d'une application d'écho client/serveur :

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

public class EchoClient {

    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public void startConnection(String ip, int port) throws
    UnknownHostException, IOException {
        clientSocket = new Socket(ip, port);
        out = new PrintWriter(clientSocket.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(clientSocket.
    getInputStream()));
        System.out.println("Socket created! " + clientSocket);
    }

    public String sendMessage(String msg) throws IOException {
        out.println(msg);
        String resp = in.readLine();
        return resp;
    }

    public void stopConnection() throws IOException {
        in.close();
        out.close();
        clientSocket.close();
    }
}
```

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

public class EchoServer {

    private ServerSocket serverSocket;
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public void start(int port) throws IOException {
        serverSocket = new ServerSocket(port);
        System.out.println("Server opened on port " + port + ", waiting for
```

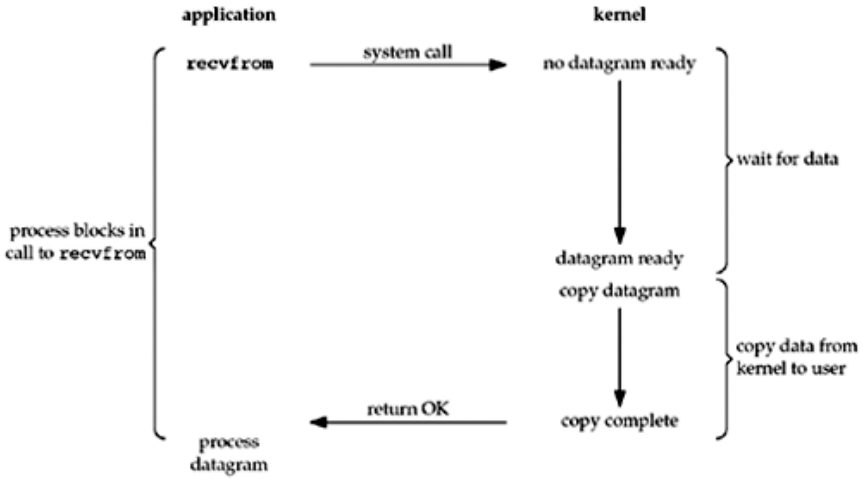

FIGURE 7 – Schéma d'un socket bloquant

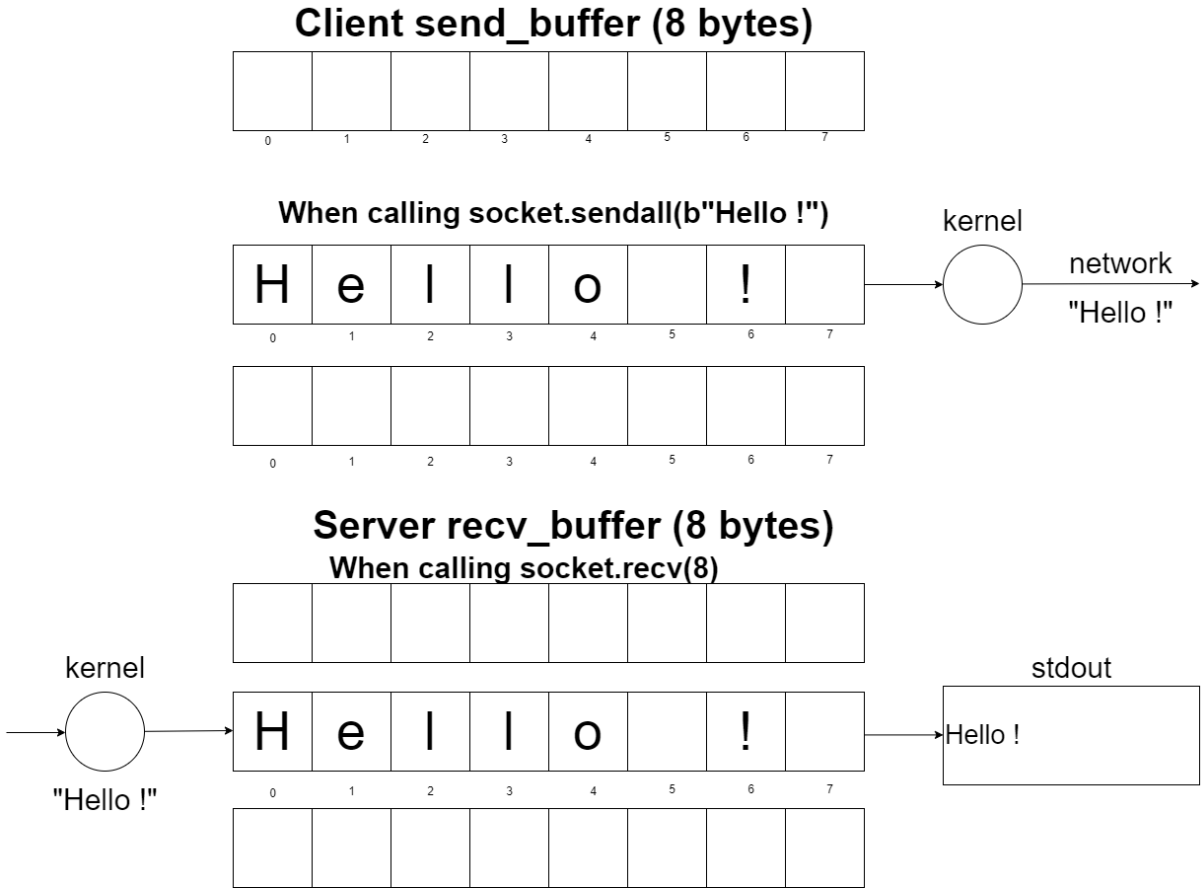

FIGURE 8 – Le fonctionnement derrière les coulisses des sockets . Le buffer d'envoi se remplit au moment de l'appel, puis se vide lorsque l'envoi est terminé. Idem du côté du receveur.

```
    a client...");
14          // Will block until a client connects.
15          clientSocket = serverSocket.accept();
16          System.out.println("Client connected: "+clientSocket);
17          System.out.println("InetAddress: " + clientSocket.getInetAddress()
        + "\nPort: " + clientSocket.getPort());
18          out = new PrintWriter(clientSocket.getOutputStream(), true);
19          in = new BufferedReader(new InputStreamReader(clientSocket.
        getInputStream()));
20          String text = in.readLine();
21          out.println(text);
22      }
23
24      public void stop() throws IOException{
25          in.close();
26          out.close();
27          clientSocket.close();
28          serverSocket.close();
29      }
30  }
```

1. Réaliser les deux classes `TestClient` et `TestServeur`, qui créent chacun une instance de `EchoClient` et `EchoServeur`, et effectuent la transmission de la chaîne *Hello World*.
2. En réutilisant une partie du code de `EchoClient` et `EchoServer`, implémenter une mini application client-serveur où lorsqu'un client se connecte au serveur, ce dernier lui transmet une collection de `Entity2D` sous forme **binaire**, que le client va ensuite décoder, et afficher dans la console.

Nous utilisons ici un `PrintWriter` pour gérer les **flux d'octets** en émission en et un `InputStreamReader` en lecture, tous les deux pour travailler à partir de **caractères**.

6 A retenir

Définition d'un flux de données avec exemples ; différence entre byte et char streams ; définition du buffer ; sérialisation en Java automatique avec `Serializable`, et manuelle.

Réalisation d'un protocole de stockage et transmission de données en binaire, en choisissant judicieusement les types de données.

Notions de socket.