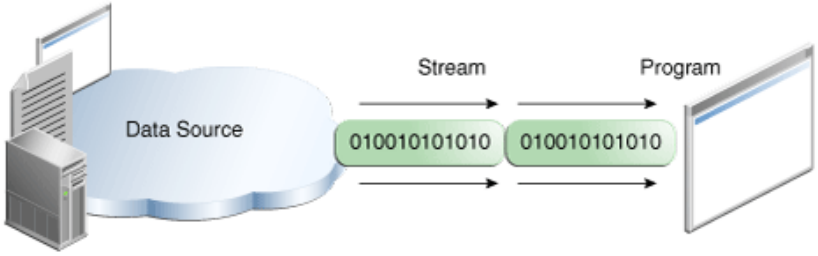
Plan

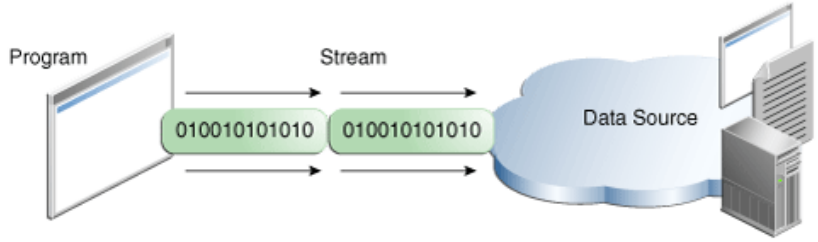
- Introduction au Java
  - Généralités
  - Syntaxe de base
- Concepts et modélisation orientée objets
  - Objet, classe et modélisation UML
  - Les principes fondamentaux: encapsulation, abstraction, héritage et polymorphisme
- Programmation Orientée Objets en Java
  - Classes, objets et bonnes pratiques
  - Héritage, interfaces, agrégation, composition et association
  - Packages
  - Généricité
  - Exceptions
  - **Flux d'Entrée/Sortie et fichiers**
  - Empaqueter et déployer son programme

Le concept de flux E/S (*I/O Stream*)

- Flux d'Entrée/Sortie (*I/O Stream*) = un "tuyau" entre une source de données et un consommateur
  - Données = séquence d'octets (*bytes*), de types primitifs, de caractères, ou d'objets
  - Entrée/Sorties = fichiers, périphériques, autres programmes, mémoire, …
  - Deux types de flux:
    - ceux qui transfèrent la donnée de la source (flux de bas niveau)
    - ceux qui manipulent et transforment la donnée (flux de haut niveau)



lecture de données (*input stream*)



écriture de données (*output stream*)

- Possibilité de chainer les flux (*chaining*)
  - P.ex. flux lisant des octets dans un fichier suivi d'un flux les transformant en entiers, réels, etc.

https://docs.oracle.com/javase/tutorial/essential/io/streams.html

L'API Java sur les flux E/S

- Grande variété de classes sur les flux dans le package **java.io.***

Flux d'octets (*Byte Streams*)	**Flux de caractères (*Character Streams*)**	
InputStream FilterInputStream BufferedInputStream DataInputStream ObjectInputStream FileInputStream …	Reader BufferedReader LineNumberReader FilterReader InputStreamReader **FileReader** …	**Entrée**
OutputStream FilterOutputStream BufferedOutputStream PrintStream DataOutputStream ObjetOutputStream FileOutputStream …	Writer BufferedWriter FilterWriter OutputStreamWriter **FileWriter** PrintWriter StringWriter …	**Sortie**

Utiliser des flux E/S

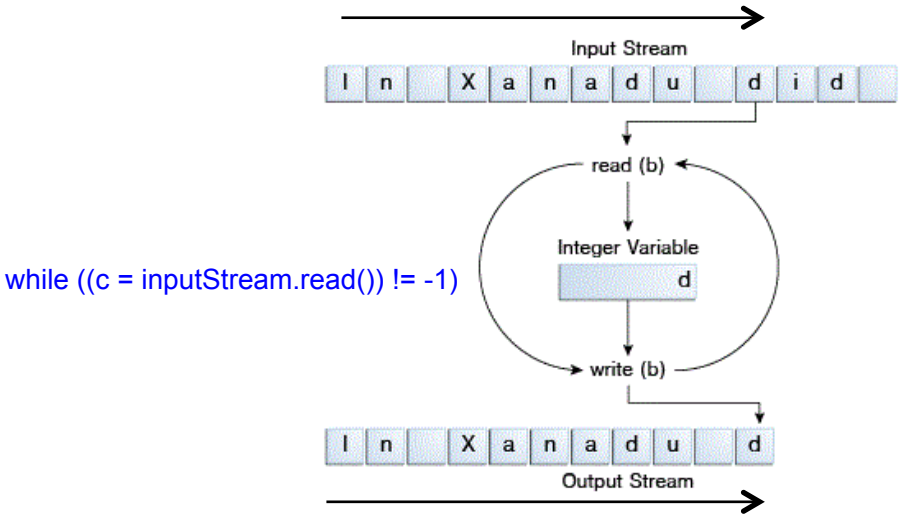
```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

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

      FileReader inputStream = null;
      FileWriter outputStream = null;

      try {
         inputStream = new FileReader("myfile.txt");
         outputStream = new FileWriter("copy.txt");

         int c;
         while ((c = inputStream.read()) != -1) {
            outputStream.write(c);
         }
      } finally {
         if (inputStream != null) {
            inputStream.close();
         }
         if (outputStream != null) {
            outputStream.close();
         }
      }
   }
}
```



while ((c = inputStream.read()) != -1)

1. Créer un objet de flux et lui associer une source de données (ou une destination)
2. Tant que(il reste des données à traiter)
   - lire (ou écrire) la donnée suivante dans (ou vers) le flux
3. Fermer le flux

https://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html

Chaîner des flux E/S

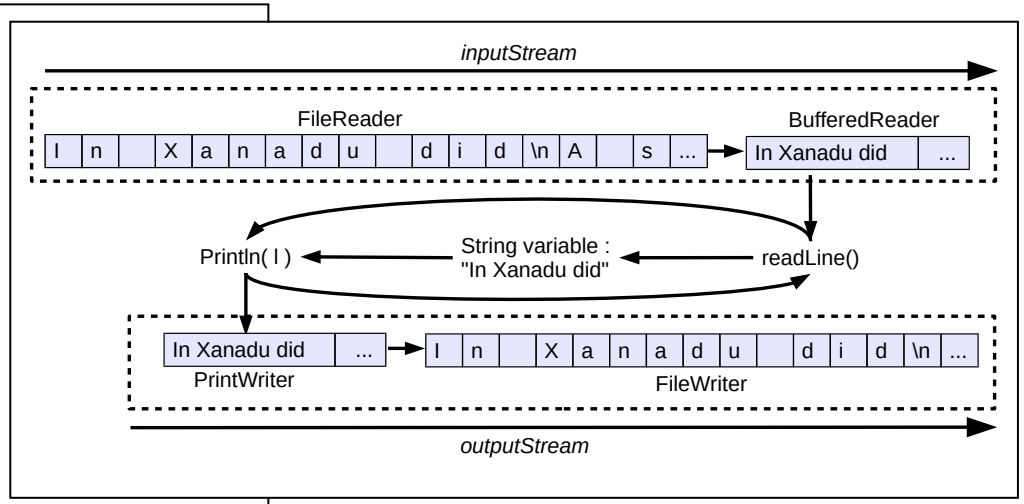
```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

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

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```



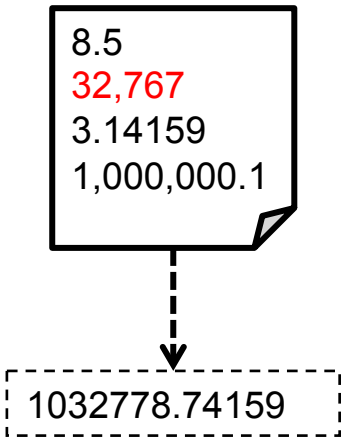
- Flux de bas niveau (*FileReader* et *FileWriter*):
  - Lit/écrit un caractère à la fois dans/vers le fichier
- Flux de haut niveau (*BufferedReader* et *PrintWriter*):
  - Lit/écrit une chaîne de caractères (une ligne) à la fois dans/vers le flux de bas niveau

Les flux avec buffer (*buffered stream*)

- Entrée/Sortie classique = requêtes de lecture ou d'écriture traitées directement par le système d'exploitation
  - Peut être très coûteux car implique de multiples accès disque, connexions réseau, ...
- Mise en place de flux avec buffer pour limiter ces coûts
  - Lecture/écriture des données d'une zone en mémoire appelée un buffer
  - Lecture/écriture des données faite par l'API native du système uniquement quand le buffer est vide/plein
- Comment créer des flux avec buffer ?
  - En utilisant des flux de haut niveau avec buffer
    - classes **BufferedInputStream** et **BufferedOutputStream** pour des flux d'octets
    - classes **BufferedReader** et **BufferedWriter** pour des flux de caractères
  - Et en les chaînant avec des flux de bas niveau
- Le "*flushing*" = écrire (vider) le contenu d'un buffer avant que celui-ci soit plein
  - Souvent automatique (*autoflush*)
    - p.ex. **PrintWriter** "flush" le buffer à chaque invocation de la méthode **println**
  - Manuellement en invoquant la méthode **flush**

Scanner un flux

- Classe **Scanner** : découper ("parser") une entrée formatée en parties (*tokens*), les extraire et les convertir
  - Entrée décomposée en fonction d'un séparateur
    - par défaut: espace, tabulation et fin de ligne
    - changer de séparateur avec la méthode **useDelimiter(** *myStr* **)**
  - Des fonctionnalités aussi pour s'adapter aux spécificités "locales"
    - p.ex. "32767" écrit "32,767" aux USA




```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;

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

      Scanner s = null;
      double sum = 0;

      try {
         s = new Scanner(new BufferedReader(
                  new FileReader("usnumbers.txt") ) );
         s.useLocale(Locale.US);

         while (s.hasNext()) {
            if (s.hasNextDouble()) {
               sum += s.nextDouble();
            } else {
               s.next();
            }
         }
      } finally {
         s.close();
      }

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

Les flux de données et d'objets

- Flux de données (*data streams*): entrées/sorties binaires de tout type primitif (**boolean, char, byte, int, long, float et double**), ainsi que des **String**, dans un seul et même flux
  - Classes implémentant les interfaces **DataInput** et **DataOuput**
    - p.ex. **DataInputStream** et **DataOutputStream**

```
double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
int[] units = { 12, 8, 13, 29, 50 };
String[] descs = {"Java T-shirt", "Java Mug","Duke Juggling Dolls","Java Pin","Java Key Chain"};

DataOutputStream out = new DataOutputStream(new BufferedOutputStream( new FileOutputStream("save.txt")));

for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeInt(units[i]);
    out.writeUTF(descs[i]);
}
```

- Flux d'objets (**sérialisation** d'objets): entrées/sorties binaires d'objets
  - Objet à écrire/lire implémente l'interface **Serializable** et flux implémente les interfaces **ObjectInput** ou **ObjectOuput**
    - p.ex. **ObjectInputStream** et **ObjectOutputStream**
  - Utilisation de méthodes **readObject** et **writeObject**

```
Book myBook = new Book( 1, "Fondation", "Hachette") ;
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("/tmp/livre1.ser") );
out.writeObject(myBook);
out.close();
```

```
public class Product
        implements Serializable {
// …
}
```

Exercice: Pérenniser les données de la bibliothèque

- Ajouter dans la classe *Document* deux méthodes permettant d'enregistrer, et de lire, dans un flux les informations d'un document
  - la méthode *write(out: BufferedWriterStream)* écrira les données dans le flux passé en paramètre (avec buffer)
  - la méthode *read(in: BufferedReaderStream)* lira les données du flux passé en paramètre (avec buffer) et modifiera les attributs de l'objet en question

- Ajouter dans la classe *Bibliotheque* deux méthodes permettant d'enregistrer, et de lire, dans un fichier la liste des documents d'une bibliothèque.
  - la méthode *writeDocuments(nomFichier: String)* enregistrera la liste des documents dans le fichier dont le nom est passé en paramètre
    - utiliser une combinaison de *FileOutputStream*, *BufferedWriter* et de *DataInputStream*
  - la méthode *readDocuments(nomFichier: String)* lira la liste des documents dans le fichier dont le nom est passé en paramètre
    - utiliser une combinaison de *FileInputStream*, *BufferedReader* et de *DataInputStream*

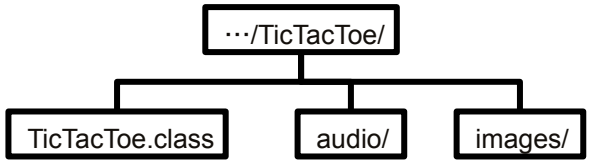
- Faire de même pour la liste des clients et la liste des fiches emprunts, mais en faisant de la sérialisation dans ces cas

Plan

- Introduction au Java
  - Généralités
  - Syntaxe de base
- Concepts et modélisation orientée objets
  - Objet, classe et modélisation UML
  - Les principes fondamentaux: encapsulation, abstraction, héritage et polymorphisme
- Programmation Orientée Objets en Java
  - Classes, objets et bonnes pratiques
  - Héritage, interfaces, agrégation, composition et association
  - Packages
  - Généricité
  - Exceptions
  - Flux d'Entrée/Sortie et fichiers
  - **Empaqueter et déployer son programme**

Empaqueter ses programmes dans des fichiers JAR

- Regrouper tous les fichiers de son programme dans un seul fichier archive (JAR, i.e. *Java Archive*)
  - Améliorer la sécurité (p.ex. en associant une signature électronique avec *jarsigner*)
  - Diminuer la taille du programme (compression au format ZIP)
  - Transformer son code en librairie utilisable dans d'autres programmes
  - Stocker et exploiter des informations de versions
- Un programme du JDK utilisable en ligne de commande



Opération	Commande
Créer un fichier JAR	**jar cf** *myapp.jar input-file(s)* ex. : jar cf TicTacToe.jar TicTacToe.class audio images jar cf TicTacToe.jar *
Afficher le contenu d'un fichier JAR	**jar tf** *myapp.jar*
Extraire le contenu d'un fichier JAR	**jar xf** *myapp.jar*
Exécuter une application empaquetée sous forme de fichier JAR *	**java -jar** *myapp.jar*

MANIFEST.MF

```
Main-Class: TicTacToe
```

* nécessité d'avoir dans le JAR un fichier *MANIFEST.MF* indiquant la classe à exécuter

Le fichier *Manifest*

- Fichier texte contenant des méta-informations (signature électronique, versions, etc) sur le JAR
  - Fichier encodé en UTF-8
  - Un seul par archive avec toujours le même chemin:

    META-INF/MANIFEST.MF
  - Dernière ligne nécessairement un retour à la ligne
  - Lignes au format "header: value"

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
```

fichier par défaut

Point d'entrée de l'application	**Main-Class:** *MyPackage.MyClassName*
Lien avec des librairies externes	**Class-Path:** *MyUtils.jar dirName/MyUtils2.jar …*
Informations sur l'application	**Name:** *MyApplicationName* **Specification-title:** *My detailed name* **Specification-Version:** *…*
…	…

- Créer un fichier JAR avec un point d'entrée pour l'exécution (génération automatique du fichier *manifest*)

```
jar cfe myApp.jar MainClassName *
```

- Intégrer un fichier *manifest* personnalisé (*Manifest.txt*) dans une archive:

```
jar cfm myApp.jar Manifest.txt MyPackage/*.class
```