# 1. Introduction

Lancez Eclipse. Créez un projet appelé `TP6` dans votre Workspace Java favori.

**IMPORTANT** : ne faites pas créer le source `module-info.java`. Ça compliquerait les importations.

Vérifiez qu'il sépare bien les sources `src` des binaires `bin`. Tous les sources seront mis ensemble, mais une seule des classes (`Main1`, `Main2`...) sera lancée à la fois.

Téléchargez ces deux fichiers :

- UtilXML.java,
- simple-xml-2.7.1.jar.

Il faut ajouter ces fichiers à votre projet Eclipse. Le premier est un source, donc à placer dans le dossier `src`. Il contient des méthodes pour ouvrir, lire, écrire des fichiers XML. Le second est une bibliothèque. Il faut créer un dossier appelé `lib` au même niveau que `src`, y placer ce fichier *jar*. Ensuite, dans Eclipse, taper `F5` pour rafraîchir le projet, puis déplier le dossier `lib` et cliquer droit sur le *jar* pour le menu contextuel et choisir `Build Path/Add to Build Path`. Une autre façon est d'aller dans `Build Path/Configure Build Path`, onglet `libraries` ajouter le fichier *jar* avec `Add External JARs`.

**IMPORTANT** : surtout ne rajoutez pas de package aux classes de src, cela compliquerait la correction de votre travail.

# 2. Sérialisation

Le TP va vous faire programmer ce qu'il faut pour sérialiser et dé-sérialiser des données Java sous forme d'XML. La sérialisation XML d'un objet consiste à écrire ses variables membres sous forme de sous-éléments ou d'attributs, selon ce qui est possible ou utile. Il faut penser à la dé-sérialisation qui est l'opération inverse. Elle consiste à instancier un objet à partir de ce qu'on lit dans le document : on reconstruit un objet à partir d'un document XML. Si le fichier XML est mal conçu, la récupération des informations ne sera pas facile.

La même chose pourrait être faite avec du JSON ou des données CSV.

Par exemple, soit la classe suivante :

```java
class Film
{
    private int     id;
    private String  titre;
    private int     annee;
    private float   duree;    // en nombre d'heures

    @SuppressWarnings("unused")
    private Film() {}

    public Film(int id, String titre, int annee, float duree)
    {
```

```
        this.id    = id;
        this.titre = titre;
        this.annee = annee;
        this.duree = duree;
    }

    public String toString()
    {
        StringBuilder sb = new StringBuilder();
        sb.append("Film[id=");     sb.append(id);
        sb.append(", titre=\"");   sb.append(titre);
        sb.append("\", annee=");   sb.append(annee);
        sb.append(", duree=");     sb.append(duree);
        sb.append("]");
        return sb.toString();
    }
}
```

Copiez-collez ce source dans votre projet Eclipse (utilisez le bouton en haut à gauche du source).

Soit une instance de cette classe créée par le programme suivant, on peut l'afficher avec un appel éventuellement implicite à `toString()` :

```
public class Main1
{
    public static void main(String[] args)
    {
        // instance de Film
        Film br1982 = new Film(1, "Blade Runner", 1982, 1.95f);

        // sérialisation en tant que chaîne
        System.out.println(br1982.toString());
    }
}
```

Copiez-collez dans Eclipse puis lancez-le. Ça affiche `Film[id=1, titre="Blade Runner", annee=1982, duree=1.95]` dans la console.

Vous avez remarqué comment `toString()` ajoute les `"..."` autour du titre du film ?

Cette méthode `toString()` effectue une sérialisation en mode texte, puisqu'elle écrit toutes les informations nécessaires pour reconstruire l'instance. Il suffirait d'analyser cette chaîne avec une expression régulière. L'idée est de faire la même chose que `toString()`, mais en XML.

## 2.1. Sérialisation XML

Une sérialisation XML pourrait être la suivante. Elle n'a même pas besoin d'être indentée :

```xml
<film id="1">
    <titre>Blade Runner</titre>
    <annee>1982</annee>
    <duree>1.95</duree>
</film>
```

Comme toujours avec XML, il y a plusieurs choix : tout attribut, tout élément ou mixte.

Alors il n'est pas question de faire des affichages *ad-hoc* pour produire le XML. On va utiliser l'API DOM bien proprement. Voici ce que ça donne. Ajoutez cette méthode dans la classe `Film` et complétez-la :

```
public void toXML(Document document, Node parent)
{
    // élément <film>
    Element elFilm = document.createElement("film");
    parent.appendChild(elFilm);

    // attribut <film id="n°">
    elFilm.setAttribute("id", Integer.toString(id));

    // à vous de rajouter ce qu'il faut pour créer les
    // sous-éléments <titre>, <annee> et <duree>
}
```

Les variables locales de type `Element` sont nommées en `elChose` pour les distinguer des variables membres du même nom.

Cette fonction est appelée, par exemple dans la fonction `main`, de cette manière :

```
import org.w3c.dom.Document;

public class Main2
{
    public static void main(String[] args) throws Exception
    {
        // instance à sérialiser
        Film br2049 = new Film(2, "Blade Runner 2049", 2017, 2.733f);

        // créer le document XML et sa racine
        Document document = UtilXML.creerDocumentDOM();

        // sérialiser le film directement au niveau du document
        br2049.toXML(document, document);

        // écrire le document dans un fichier
        UtilXML.ecrireDocumentDOM(document, "film.xml");
    }
}
```

Donc maintenant, la problématique, c'est faire l'inverse : initialiser une instance de `Film` à partir

d'un document XML.

## 2.2. Dé-sérialisation XML

On part d'un document XML valide (conforme à la fonction de sérialisation) :

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<film id="3"><titre>Alien</titre><annee>1979</annee><duree>1.933</duree></film>
```

Voici un extrait de la fonction principale :

```java
import org.w3c.dom.Document;
import org.w3c.dom.Node;

public class Main3
{
    public static void main(String[] args) throws Exception
    {
        // ouverture du fichier XML
        Document document = UtilXML.lireDocumentDOM("film.xml", false);

        // lire et afficher la racine
        Node racine = document.getDocumentElement();
        Film film = Film.fromXML(racine);
        if (film != null) {
            System.out.println(film);
        } else {
            System.err.println("film.xml n'a pas pu être lu");
        }
    }
}
```

La méthode `fromXML(Node node)` est statique dans la classe `Film`. C'est une *fabrique* qui retourne une instance de `Film` construite à partir du document XML ou `null` s'il est incorrect. Voici un extrait à placer dans la classe `Film` et à compléter :

```java
public static Film fromXML(Node node) throws Exception
{
    // TODO vérifier que node est un élément <film>, sinon retourner null

    // conversion en Element
    Element elFilm = (Element) node;

    // valeurs du film à initialiser et retourner
    Integer id = null;
    String titre = null;
    Integer annee = null;
    Float duree = null;
```

```
    // lire l'attribut id
    id = Integer.parseInt(elFilm.getAttribute("id"));

    // lire le titre
    Node elTitre = elFilm.getFirstChild();
    titre = elTitre.getTextContent();

    // FIXME lire l'année
    Node elAnnee = elFilm.getNextSibling();     // <- FIXME
    annee = 0;

    // TODO lire la durée
    duree = 0.0f;

    // TODO vérifier que toutes les informations sont présentes

    // créer et retourner l'instance
    return new Film(id, titre, annee, duree);
}
```

Cette manière de programmer la fonction `main` et la méthode `toXML` sans faire aucun test et en attendant exactement les éléments prévus présente plusieurs problèmes potentiels :

- Aucun test n'est fait pour être sûr qu'on est bien sur un élément `<film>`. On pourrait être sur un texte, ça ferait tout planter : `ClassCastException`. Utilisez la méthode `UtilXML.isElement(node, nom)` pour cette vérification (étudiez son source).
- Aucun test qu'il y a ou non l'attribut `id`.
- Aucun test de l'élément `<titre>` : présent/absent, vide ?
- Aucun test des éléments `<annee>` et `<duree>` : présents/absents, contenant un nombre correct (`NumberFormatException`) ?
- Et si les sous-éléments n'étaient pas dans cet ordre ?

Également, c'est à vous de gérer les exceptions. Avec la manière montrée ci-dessus, c'est dans la méthode `main` que ça plantera s'il y a un problème avec le fichier XML. L'autre manière serait d'intercepter les exceptions dans `fromXML` et de retourner `null` le cas échéant.

Vérifier si ça plante si on indente le fichier xml. Rajoutez les tests et les boucles pour sauter les `Nodes` non attendus, afin de tolérer les variantes, ou même les documents invalides.

## 3. Sérialisation de données liées

On continue avec une autre classe : 

```
import java.util.List;
import java.util.ArrayList;

class Collection
{
```

```
    private String     genre;
    private List<Film> films = new ArrayList<>();

    public Collection() {}

    public Collection(String genre)
    {
        this.genre = genre;
    }

    public Collection(String genre, List<Film> films2)
    {
        this.genre = genre;
        this.films.addAll(films2);
    }

    public void add(Film film)
    {
        if (film != null) {
            films.add(film);
        }
    }

    public String toString()
    {
        StringBuilder sb = new StringBuilder();
        sb.append("Collection[genre=");
        sb.append(genre);
        for (Film film: films) {
            sb.append(", ");
            sb.append(film.toString());
        }
        sb.append("]");
        return sb.toString();
    }
}
```

Programmez les méthodes `toXML` et `fromXML` comme pour `Film`.

Voici le programme qui permet de mettre en œuvre `toXML` :

```
import org.w3c.dom.Document;

public class Main4
{
    public static void main(String[] args) throws Exception
    {
        // une collection
```

```
        Collection collection = new Collection("Fantastique");

        // quelques films
        collection.add(new Film(1, "Blade Runner", 1982, 1.95f));
        collection.add(new Film(2, "Blade Runner 2049", 2017, 2.733f));
        collection.add(new Film(3, "Alien", 1979, 1.933f));

        // créer le document XML et sa racine
        Document document = UtilXML.creerDocumentDOM();

        // sérialiser la collection
        collection.toXML(document, document);

        // écriture du document dans un fichier
        UtilXML.ecrireDocumentDOM(document, "collection.xml");
    }
}
```

Pour relire les données, on pourrait faire ceci :

```
import org.w3c.dom.Document;
import org.w3c.dom.Node;

public class Main5
{
    public static void main(String[] args) throws Exception
    {
        // ouverture du fichier XML
        Document document = UtilXML.lireDocumentDOM("collection.xml", false);

        // lire et afficher la racine
        Node racine = document.getDocumentElement();
        Collection collection = Collection.fromXML(racine);
        if (collection != null) {
            System.out.println(collection);
        } else {
            System.err.println("collection.xml n'a pas pu être lu");
        }
    }
}
```

# 4. Sérialisation avec l'API Simple

Cette API, incluse en standard dans le SDK Java, rend tout le travail incroyablement plus facile. Elle demande seulement de coller des annotations sur les classes et variables membres. Relire le cours.

## 4.1. Sérialisation d'une classe

Dupliquez la classe `Film` sous le nom `Livre` et remplacez les méthodes `toXML` et `fromXML` par celles du cours, en changeant le nom de la classe.

Remplacez les import W3C par ceux de Simple. Rajoutez les annotations `@Root`, `@Attribute` et `@Element` pour la sérialisation. Choisissez si les variables doivent devenir des attributs ou des éléments.

Voici maintenant comment lancer la sérialisation : 

```
public class Main6
{
    public static void main(String[] args) throws Exception
    {
        // un livre
        Livre livre = new Livre(1, "Le Vagabond des étoiles", 1915, 7.50f);

        // sérialisation
        livre.toXML("livre.xml");
    }
}
```

La désérialisation est tout aussi facile : 

```
public class Main7
{
    public static void main(final String[] args) throws Exception
    {
        // désérialisation
        Livre livre = Livre.fromXML("livre.xml");

        // utilisation des données
        System.out.println(livre.toString());
    }
}
```

## 4.2. Sérialisation d'une collection

Copiez la classe `Collection` et nommez-la `Librairie` (vous pouvez changer le champ `duree` en `prix`). Renommez `films` en `livres` et écrasez les méthodes `toXML` et `fromXML` par celles du cours en changeant le nom de la classe. Notez que toutes les classes sérialisables auront ces deux méthodes quasiment identiques.

Il reste une petite particularité à découvrir. Ne pas faire correctement entraîne une exception `TransformException`. C'est parce qu'on ne peut pas sérialiser une collection en tant que simple élément. Mettez donc la bonne annotation, voir le CM.

Voici maintenant comment lancer la sérialisation : 

```java
public class Main8
{
    public static void main(String[] args) throws Exception
    {
        // données
        Librairie librairie = new Librairie("classiques américains");
        librairie.add(new Livre(1, "Le Vagabond des étoiles", 1915, 7.50f));
        librairie.add(new Livre(2, "Les Aventures de Tom Sawyer", 1876, 6.90f));
        librairie.add(new Livre(3, "Histoires extraordinaires", 1856, 7.20f));

        // sérialisation
        librairie.toXML("librairie.xml");
    }
}
```

Si vous regardez le fichier `librairie.xml`, vous verrez que les livres sont sérialisés chacun dans un élément `<livres>`. Voir le cours pour mettre à plat cette liste, sans cet élément inutile.

Écrivez la classe `Main9` qui permet de désérialiser `librairie.xml` et l'afficher sur l'écran : adaptez `Main7`.

## 5. Travail à rendre

Vous avez programmé dans un projet `TP6` dans le workspace Java. Il faut seulement rendre son dossier `src` (pas les fichiers XML ni les binaires).

Cliquez droit sur le dossier `src` du projet, choisissez `Compresser...`, cliquez sur `Créer`. Ça va créer une archive `src.tar.gz`. Déposez cette archive sur Moodle, dans la page de cours dédiée L4IN121T Formats et traitements de données internet.