TP Algorithmique Avancée : Programmation JAVA/XML
Prise en main de l' API JAXB (Java API for XML Binding)

XML est un langage de structuration et de représentation des données qui :
- d'une part est très utilisé pour la mise en place de framework et d'applications (***fichiers de configuration***)
- d'autre part est à la base d'un nouveau mode de communication entre les applications qui est appelé ***Web Service.***

… Mais XML ne comporte pas d'instructions de contrôle et ne permet pas d'exploiter directement les données.
Pour réaliser des applications XML, il faut donc avoir recours aux langages de programmation classiques.
Dans de nombreux langages (dont Java), les ***API de bas niveau*** pour manipuler et traiter des flux XML sont **SAX** et **DOM**.
- **SAX** (**S**imple **A**PI for **X**ML) est une API ***de type événementielle et incrémentale***. Elle génère différents types d'événements lorsqu'elle parcourt le fichier XML. Le développeur peut alors recevoir ces événements et adopter le comportement approprié. SAX permet donc de lire et traiter un document XML
- **DOM** (Document Object Model) permet une ***représentation objet sous forme d'arbre*** d'un document XML. DOM ne traite pas un fichier texte mais un ***arbre d'objets***.

Avec ces deux API, toute la logique de *traitement des données* doit être implémentée par le développeur.

Dans ce tutoriel, nous nous intéresserons à l'**API JAXB** (acronyme de **Java API for XML Binding**) qui permet de ***simplifier*** le traitement des processus de ***transformation d'objets Java en flux XML et vice-versa.***
En effet **JAXB** permet de générer automatiquement un ensemble de classes qui fournissent alors un niveau d'abstraction plus élevé que la simple utilisation de **SAX** ou **DOM**.
L'utilisation de **JAXB** engendre ainsi un gain de temps significatif puique **JAXB** permet de ***travailler en Java sur un flux de données XML sans se préoccuper du XML***.

JAXB est donc une ***API Java "évoluée".*** Elle est développée à l'origine par ***SUN*** .
JAXB 2.0 est inclus dans le ***Web Services Developper Pack***, disponible par défaut dans les versions standards depuis **JavaEE5** et **JavaSE6.**

Remarque : Pour information, le ***Java Web Services Developper Pack*** regroupe d'autres outils permettant de travailler avec Java, XML et les services web comme par exemple :
- **JAXP** (**J**ava **A**PI for **X**ML **P**arsing) qui permet d'***analyser*** les fichiers XML selon les normes DOM ou SAX, et de les transformer à l'aide de feuille de style XSL.
- **JAX-RPC** (**J**ava **A**PI **X**ML based **RCP**) utilise des web services

Présentation JAXB

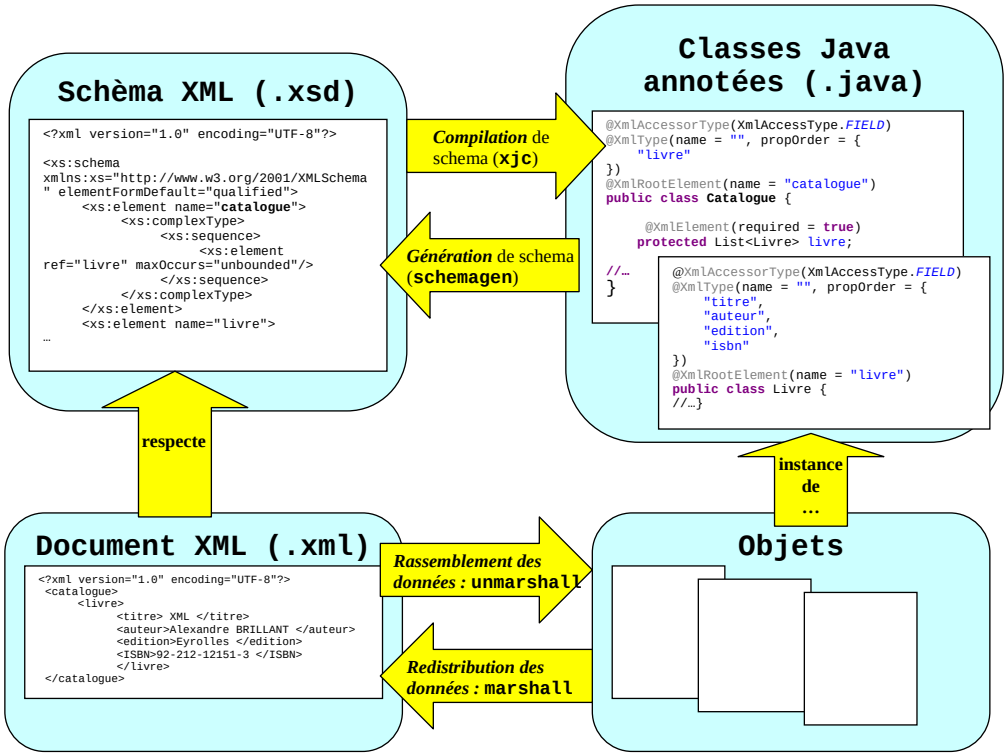
JAXB est parfois également appelé ***framework de binding Java/XML***.
Le ***Data Binding*** (*association de données ou liaison de données*) consiste à transformer des objets Java au format XML et inversement.

JAXB peut également être vue comme une boîte à outils utilisée pour la sérialisation et la désérialisation d'objets java en flux XML.
- L'opération ***sérialisation*** est appelée dans **JAXB** opération de **marshalling** (ou ***redistribution des données***) La redistribution des données consiste à récupérer le contenu de chaque instance de classe et à les envoyer dans les flux XML
- L'opération ***désérialisation*** est appelée dans **JAXB** opération de **unmarshalling** (ou ***rassemblement des données***)

JAXB *en ligne :*
http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/2.0/jaxb/
http://jaxb.java.net/
http://www.oracle.com/technetwork/articles/javase/index-140168.html

JAXB ne reconnaît le principe de **marshalling/unmarshalling** que via le **schéma XML** (fichier **.xsd**).
En effet, **JAXB** fournit dans son implémentation de référence un générateur de classes java (appelé aussi compilateur de schéma) qui va analyser un schéma XML et générer à partir de ce dernier un ensemble de classes Java qui vont encapsuler les traitements de manipulation du document.
Le compilateur de schéma fournit dans l'implémentation de référence est l'outil **xjc**.

Le dessin suivant résume le principe de fonctionnement de l'**API JAXB** :
- un document XML respecte les règles de grammaire d'un Schéma XML (fichier **.xsd**).
- Ce schéma est compilé (par l'outil **xjc**) afin de générer la (les) classe (s) Java correspondante (s).
- Cette (ces) classe (s) permette(nt) alors de créer une(des) instance(s) d'objet.
- Les opérations de marshalling/unmarshalling peuvent alors être mises en place.



Remarque:
- **JAXB** permet de créer des classes Java à partir de schémas XML grâce au compilateur de schéma **xjc** fourni dans l'implémenatation JAXB de référence.
- et inversement, **JAXB** permet créer des schémas à partir de classes Java grâce au générateur de schéma **schemagen** fourni dans l'implémentation JAXB de référence.

Autre remarque:
JAXB est simple à mettre en œuvre car il est disponible par défaut dans les dernières versions de **JavaSE** et **JavaEE**, et contrairement à d'autres framework (tel qu'Hibernate, log4J,…), il ne nécessite pas de fichier de configuration ni de fichiers de mapping pour son utilisation.

1. La génération de classes Java

La première étape pour le « ***data binding*** » est de ***générer les classes JAVA*** qui permettront d'accéder aux données contenues dans les fichiers **XML**.

1.1 A partir d'un document xml…

Le document **testJAXB.xml** sur lequel s'appuie ce tutoriel est très simple et facile à comprendre.
L'élément racine est **<catalogue>** qui est composée de deux éléments **<livre>** (dans un cas plus général, on pourrait imaginer bien plus de 2 éléments **<livre>**, puisqu'un catalogue est un ensemble de livres).
Un **livre** est composé de 4 éléments (**titre**, **auteur**, **edition**, **ISBN)** contenant chacun une chaîne de caractères.

testJAXB.xml

```
<?xml version="1.0" encoding="UTF-8"?>
 <catalogue>
   <livre>
       <titre> XML </titre>
       <auteur>Alexandre BRILLANT </auteur>
       <edition>Eyrolles </edition>
       <ISBN>92-212-12151-3 </ISBN>
   </livre>
   <livre>
       <titre>JAVA/XML </titre>
       <auteur>Renaud FLEURY </auteur>
       <edition>Cahier du Programmeur </edition>
       <ISBN>2-212-11316-1</ISBN>
   </livre>
 </catalogue>
```

Le but de la génération de classes est donc d'obtenir 2 classes Java :

↳ une première classe **Livre** contenant 4 attributs de type **String** et les méthodes permettant d'accèder et de modifier ces 4 données (getteurs/setteurs).

Livre.java

```
public class Livre {
    protected String titre;
    protected String auteur;
    protected String edition;
    protected String ISBN;
    // et getteurs/setteurs…
}
```

↳ une seconde classe **Catalogue** contenant 1 attribut correspondant à une collection de **Livre** (**List<Livre>** par exemple) et une méthode permettant de consulter cette collection de livres.

Catalogue.java

```
public class Catalogue {
    protected List<Livre> livre;

    // et getteurs/setteurs…
}
```

1.2 … et à partir de son schèma (.xsd)

C'est le compilateur **xjc** contenu dans l'API JAXB qui va nous permettre de générer les classes JAVA qui vont manipuler le fichier XML.
Le compilateur **xjc** ne s'appuie pas directement sur le document XML, mais sur son schèma (**xsd**)
Rappelons qu'un schèma XML (**.xsd**) représente la grammaire d'un document **XML**.
Avant de continuer, nous devons donc nous assurer de possèder un fichier **xsd** relatif au document XML à traiter.

Dans notre exemple, le schèma XML (fichier **.xsd**) est le suivant :

catalogue.xsd

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:element name="catalogue">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="livre" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="livre">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="titre"/>
                <xs:element ref="auteur"/>
                <xs:element ref="edition"/>
                <xs:element ref="ISBN"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="titre" type="xs:string"/>
    <xs:element name="auteur" type="xs:string"/>
    <xs:element name="edition" type="xs:string"/>
    <xs:element name="ISBN" type="xs:string"/>
</xs:schema>
```

Astuce pour obtenir automatiquement un schéma à partir d'un document XML …
A partir de votre document xml, vous pouvez utiliser un générateur de schèma en ligne, comme le propose le site suivant : http://www.xmlforasp.net/CodeBank/System_Xml_Schema/BuildSchema/BuildXMLSchema.aspx

1.3 …le compilateur xjc gènère les classes java correspondantes

Le compilateur **xjc** permet d'analyser un **schéma XML** afin de générer les classes JAVA qui vont permettre la manipulation du document **xml** qui respecte ce schèma. Cette opération est appelée **binding**

Le compilateur **xjc** (appelé aussi binding compiler) peut être utilisé "directement " à partir de la ligne de commande suivante : **xjc [options] schema**

avec **schema** comme nom du fichier contenant le schéma XML.
et d'éventuelles **options** dont les principales sont :

- **-p package :** pour préciser le nom du package qui va contenir les classes générées
- **-d repertoire :** pour préciser le nom du répertoire qui va contenir les classes générées
- **-nv :** pour inhiber la validation du schèma

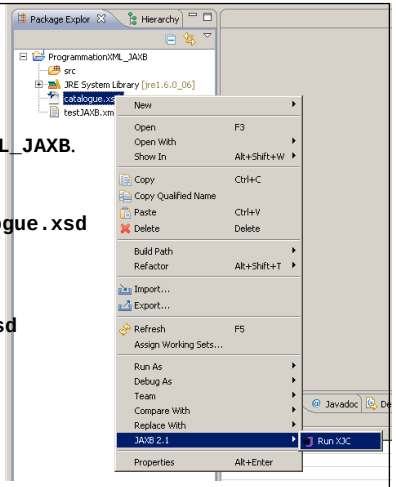
Sous **Eclipse**, il est possible d'installer le **plugin XJC** qui facilite l'utilisation du compilateur **xjc** de l'**API JAXB**
Ainsi, au lieu de générer les classes Java d'un schéma **xsd** en ligne de commande, le plugin permet en quelques clics d'obtenir le même résultat.
Pour ce turoriel, nous utiliserons ce plug-in qu'il est nécessaire de rajouter à votre installation d'Eclipse.

1.4 Installation du plugin XJC pour Eclipse…

Sur les ordinateurs de l'IUT, le **plug-in XJC** est déjà installé.
Pour votre ordinateur personnel, vous pouvez récuperer le plug-in sur la zone libre ou le ***télécharger*** depuis le site web suivant : http://jaxb.java.net/ suivant en cliquant sur le lien **JAXB Workshop, Eclipse Plugins** de la partie **Extensions/Tools** puis **IDE Plugins** (fichier **org.jvnet.jaxbw.zip** par exemple). ***Dézipper*** et ***copier*** le fichier dans le dossier **/eclipse/plugins/** d'**Eclipse** afin que le **xjc-plugin** rejoignent les autres plug-in déjà installés. ***Redémarrer*** Eclipse pour activer ce nouveau plugin.

1.5 Binding : génération des classes Java à partir du schèma xml en utilisant le xjc plug-in …

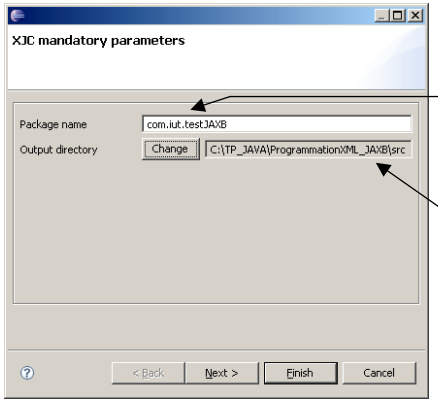
Travail à faire :



↳ Sous **Eclipse** ouvrir votre espace de travail et
créer un **nouveau projet** que vous appelerez **`ProgrammationXML_JAXB`**.

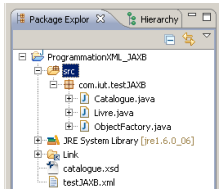
↳ Dans ce projet, importer les fichiers **`testJAXB.xml`** et **`catalogue.xsd`** disponibles sur la zone libre.

↳ Placez-vous dans la vue **`Package`** sur le fichier **`catalogue.xsd`** du projet **`programmationXML_JAXB.`**
Effectuer un clic droit avec la souris
et choisir l'option **`JAXB2.x`** puis **`Run XJC`**

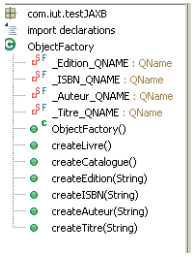


↳Une fenêtre de dialogue s'ouvre.
Remplir les champs respectivement avec :
le **nom du <u>package</u>** (**`com.iut.testJAXB`**) qui contiendra les classes annotées par le compilateur
et
le **<u>répertoire</u>** où le package sera inséré (**`src`**).

↳ Sous la vue **Package** du projet, vous trouvez maintenant le package **`com.iut.testJAXB`** "fraichement" créé.
(si vous ne voyez pas ce paquetage, rafraîchir l'arborescence en vous plaçant sur le répertoire **`src`** et faite un **F5**)



Le paquetage **`com.iut.testJAXB`** contient 3 classes Java annotées :



- **`Catalogue.java`** et **`Livre.java`** sont les 2 classes que nous attendions.
Chacune de ces classes "encapsule" un type complexe du schèma `catalogue.sxd` et possède des getters/setters. **`JAXB 2.0`** utilise de nombreuses annotations (comme vous pouvez le constater dans les classes générées **`Catalogue.java`** et **`Livre.java`**).
Elles sont définies dans le paquetage **`javax.xml.bind.annotation`** (voir javadoc et annexe 1)
Ces annotations précisent comment le mapping entre les classes Java et le document XML doit être réalisé : elles sont donc utilisées lors des transformations Java↔XML (opérations de **`marshalling/unmarshalling`**.)
- **`ObjectFactory.java`** est une fabrique. Elle permet de créer des instances de chacun des types complexes et de chaque élément du schèma (méthodes **`createXXX`**). Cette classe sera utile lors de la création d'un nouveau document XML (le graphe d'objets sera alors créé en ajoutant des instances d'objets retournées par la fabrique).

<u>Remarque :</u> Les classes générées auraient été plus nombreuses avec la première version de **`JAXB (JAXB 1.0`**).
Les classes générées sont donc dépendantes de l'implémentation **`JAXB`** utilisée (il est préférable d'utiliser les classes générées par une implémentation avec cette implémentation : nous utilisons ici l'API **`JAXB 2.x`**)

Le tableau ci-dessous illustre le ***mapping schèma XML → classe JAVA***. En annexe 2, vous trouverez une correspondance entre les types du schèma XML et les types correspondants dans les classes JAVA.

Schèma XML	JAXB Binding
`<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">`	
`<xs:element name="catalogue">`	

`<xs:complexType>`
`<xs:sequence>`
`<xs:element ref="livre" maxOccurs="unbounded"/>`
`</xs:sequence>`
`</xs:complexType>`
	`Classe Catalogue.java`
	
	`protected List<Livre> livre;`
	`public List<Livre> getLivre()`

`</xs:element>`	
`<xs:element name="livre">`	

`<xs:complexType>`
`<xs:sequence>`
`<xs:element ref="titre"/>`
`<xs:element ref="auteur"/>`
`<xs:element ref="edition"/>`
`<xs:element ref="ISBN"/>`
`</xs:sequence>`
`</xs:complexType>`
`</xs:element>`
`<xs:element name="titre" type="xs:string"/>`
`<xs:element name="auteur" type="xs:string"/>`
`<xs:element name="edition" type="xs:string"/>`
`<xs:element name="ISBN" type="xs:string"/>`
	`Classe Livre.java`
	
	`protected String titre;`
	`protected String auteur;`
	`protected String edition;`
	`protected String isbn;`
	
	`+ getteurs/setteurs`
	`public String getTitre()`
	`public void setTitre(String value)`
	`public String getAuteur()`
	`public void setAuteur(String value`
	`public String getEdition()`
	`public void setEdition(String value)`
	`public String getISBN()`
	`public void setISBN(String value)`

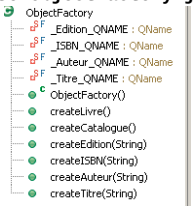
`</xs:schema>`	
	`Classe ObjectFactory.java` 

Illustration du binding :
Source : http://www.oracle.com/technetwork/articles/javase/index-140168.html

2. Utilisation de l'API JAXB 2.0

Les classes **JAXB** précédentes, générées automatiquement par le compilateur **xjc**, vont nous permettre de manipuler simplement le document **XML** via **l'API JAXB** (javadoc disponible depuis le site http://jaxb.java.net/ option **Documentation** puis *Javadoc* ou directement par http://jaxb.java.net/nonav/2.2.3u1/docs/api/)

JAXB 2.0 a été développée sous la **JSR 222** (http://jcp.org/en/jsr/detail?id=222)

Remarque :
Créé en 1998 par Sun, le **JCP** (**J**ava **C**ommunity **P**rocess) est une organisation chargée de coordonner l'évolution du langage Java et des technologies qui lui sont associées (http://www.jcp.org)
Chaque évolution est traitée sous la forme de propositions nommées **JSR** (**J**ava **S**pecification **R**equests).
Le contenu d'une JSR peut être très varié : allant d'une API, d'une spécification, de la définition d'une plate-forme et même les évolutions du JCP lui même,…

JAXB fournie une **API** composée de classes et d'interfaces regroupées dans plusieurs packages

JAXB Packages	
javax.xml.bind	Provides a runtime binding framework for client applications including unmarshalling, marshalling, and validation capabilities.
javax.xml.bind.annotation	Defines annotations for customizing Java program elements to XML Schema mapping.
javax.xml.bind.annotation.adapters	XmlAdapter and its spec-defined sub-classes to allow arbitrary Java classes to be used with JAXB.
javax.xml.bind.attachment	This package is implemented by a MIME-based package processor that enables the interpretation and creation of optimized binary data within an MIME-based package format.
javax.xml.bind.helpers	JAXB Provider Use Only: Provides partial default implementations for some of the javax.xml.bind interfaces.
javax.xml.bind.util	Useful client utility classes.

Le paqueta ge **javax.xml.bind** contient les interfaces principales et la classe **JAXBContext** qui est le point d'entrée dans l'**API**.

JAXB 2.0 permet de :

- de générer des classes Java à partir d'un schèma XML à l'aide du compilateur **xjc**
- de desérialiser un graphe d'objets Java depuis un document XML par une opération appelée **unmarshalling**, c'est-à-dire de réaliser un mapping ***flux XML → objets Java***
- de sérialiser un graphe d'objets Java dans un document XML par une opération appelée **marshalling**, c'est-à-dire de réaliser un mapping ***objets Java → flux XML***
  Lors de ces deux opérations, le document XML peut être validé.
- … mais aussi de générer un schèma XML à partir de classes Java à l'aide du générateur **schemagen**

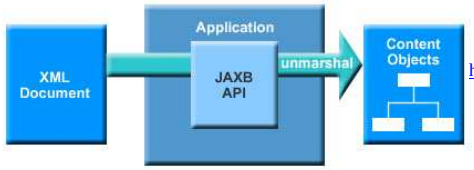
JAXB 2.0 utilise de nombreuses annotations (comme vous pouvez le constater dans les classes générées **Catalogue.java** et **Livre.java**). Elles sont définies dans le paquetage **javax.xml.bind.annotation.**

Ces annotations précisent le mapping entre les classes Java et le document xml.
Elles sont utilisées lors des opérations de **unmarshalling/marshalling** (désérialisation/sérialisation).

Nous allons maintenant détailler les opérations **unmarshalling/marshalling** , opérations de transformations de flux XML en objets Java et vice-versa.

3. Opération unmarshalling : Mapping d'un flux XML → objets JAVA (rassemblement des données)

JAXB permet de ***transformer un flux XML en graphe d'objets Java*** : cette opération de transformation est appelée **unmarshalling**.
Le rassemblement des données (***unmarshall*** en anglais) consiste à instancier les classes Java annotées JAXB et à les mettre à jour avec les données contenues dans des flux XML.



Source : http://www.oracle.com/technetwork/articles/javase/index-140168.html

Pour cela, il est nécessaire d'utiliser un rassembleur (**unmarshaller**). Ce dernier est également chargé de vérifier l'intégrité des données qu'il transporte. Si une erreur intervient, il pourra lever des exceptions
Le rassemblement des données (**unmarshalling**) s'appuie sur les méthodes du paquetage **javax.xml.bind** de l'API JAXB.

3.1 L'indispensable objet JAXBContext

L'objet **JAXBContext** du paquetage **javax.xml.bind** est le point d'entrée dans **l'API JAXB.**
Les opérations de transformations (unmarshalling/marshalling) s'appuient sur un objet JAXBContext.

Pour obtenir une instance de **JAXBContext**, il faut utiliser la ***méthode statique*** **newInstance** qui prend en paramètre le nom du paquetage contenant les classes java annotées JAXB (c-a-d les classes que nous venons de générées à partir du compilateur **xjc**).

```
JAXBContext jc = JAXBContext.newInstance("com.iut.testJAXB");
```

Cette méthode est susceptible de lancer une **JAXBException**.

Travail à faire :

↳ Dans votre projet **ProgrammationXML_JAXB**, créer une classe de test que vous appelerez **EssaiXML2Java.java.**

↳ Commencer à implémenter la méthode **main** en instanciant un objet **JABXContext**

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;

public class EssaiXML2Java {

    public static void main( String[] args ) {
        try {
            // 1. Création d'un JAXBContext: point d'entrée dans l'API JAXB
            // pour mettre en place les opérations de transformation
            JAXBContext jc = JAXBContext.newInstance("com.iut.testJAXB");

            } catch (JAXBException e) {
                e.printStackTrace();
            }
    } // fin main
}
```

A retenir… : Le paramètre de la méthode *newInstance* (ici com.iut.testJAXB) est la même chaîne de caractères saisie dans le champ **Package name** du plug-in **RunXJC** (cf copie d'écran page 5 de ce tutoriel).

3.2 objet `Unmarshaller` pour le mapping flux XML vers Objets Java

javax.xml.bind
Class JAXBContext

abstract Unmarshaller	**createUnmarshaller**() Create an Unmarshaller object that can be used to convert XML data into a java content tree.

Il faut ensuite instancier un objet de type **`Unmarshaller`** qui va permettre de transformer un **flux XML** en un ensemble d'objets Java. Une telle instance est obtenue en utilisant la méthode **`createUnmarshaller()`** de la classe **`JAXBContext`** :

```
Unmarshaller unm = jc.createUnmarshaller();
```

3.3 Lecture du flux XML et construction du graphe d'objets Java :

javax.xml.bind
Interface Unmarshaller

java.lang.Object	**unmarshal**(java.io.InputStream is) Unmarshal XML data from the specified InputStream and return the resulting content tree.

La méthode **`unmarshal()`** de la classe **`Unmarshaller`** se charge de traiter un **flux XML** et retourne un objet du type complexe qui encapsule la racine du document.
Dans notre exemple (`testJAXB.xml`), l'élément racine du document **`xml`** est **`<catalogue>`** .
La méthode **`unmarshal`** renverra donc un objet de type **`Catalogue`**.

```
Catalogue unCatalogue =
        (Catalogue) unm.unmarshal(new FileInputStream("testJAXB.xml"));
```

Remarque :
La méthode **`unmarshal`** possède de nombreuses surcharges à utiliser en fonction des besoins.

L'opération d'*`unmarshalling`* peut ainsi être effectuée sur un flux XML quel que soit son support (`File`, `InputStream`, `URL`, `StringBuffer`, `org.w3c.dom.Node`, …)

Pour savoir comment utiliser ces méthodes, consulter la javadoc de l'interface **`Unmarshaller`** qui propose différents exemples de codes.

java.lang.Object	**unmarshal**(java.io.File f) Unmarshal XML data from the specified file and return the resulting content tree.
java.lang.Object	**unmarshal**(org.xml.sax.InputSource source) Unmarshal XML data from the specified SAX InputSource and return the resulting content tree.
java.lang.Object	**unmarshal**(java.io.InputStream is) Unmarshal XML data from the specified InputStream and return the resulting content tree.
java.lang.Object	**unmarshal**(org.w3c.dom.Node node) Unmarshal global XML data from the specified DOM tree and return the resulting content tree.
<T> JAXBElement<T>	**unmarshal**(org.w3c.dom.Node node, java.lang.Class<T> declaredType) Unmarshal XML data by JAXB mapped declaredType and return the resulting content tree.
java.lang.Object	**unmarshal**(java.io.Reader reader) Unmarshal XML data from the specified Reader and return the resulting content tree.
java.lang.Object	**unmarshal**(javax.xml.transform.Source source) Unmarshal XML data from the specified XML Source and return the resulting content tree.
<T> JAXBElement<T>	**unmarshal**(javax.xml.transform.Source source, java.lang.Class<T> declaredType) Unmarshal XML data from the specified XML Source by declaredType and return the resulting content tree.
java.lang.Object	**unmarshal**(java.net.URL url) Unmarshal XML data from the specified URL and return the resulting content tree.
java.lang.Object	**unmarshal**(javax.xml.stream.XMLEventReader reader) Unmarshal XML data from the specified pull parser and return the resulting content tree.
<T> JAXBElement<T>	**unmarshal**(javax.xml.stream.XMLEventReader reader, java.lang.Class<T> declaredType) Unmarshal root element to JAXB mapped declaredType and return the resulting content tree.
java.lang.Object	**unmarshal**(javax.xml.stream.XMLStreamReader reader) Unmarshal XML data from the specified pull parser and return the resulting content tree.
<T> JAXBElement<T>	**unmarshal**(javax.xml.stream.XMLStreamReader reader, java.lang.Class<T> declaredType) Unmarshal root element to JAXB mapped declaredType and return the resulting content tree.

3.4 Visualisation des données XML en consultant les objets Java.

A partir de l'objet obtenu par la méthode **`unmarshal`**, il est possible d'obtenir et de modifier les données encapsulées dans les différents objets créés à partir des classes générées. En effet, chacun de ces objets, instances des classes **`Catalogue`** ou **`Livre`** dans notre exemple, possèdent des getteurs/setteurs créés par le compilateur **`xjc`** lors de la génération des classes à partir du schéma (`xsd`).

3.5 Mise en place et test de l'opération de `unmarshalling` à partir du fichier testJAXB.xml

Travail à faire :

Compléter la méthode `main` de la classe **`EssaiXML2Java.java`** afin d'obtenir un code similaire au code suivant et exécuter-le.

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

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;


public class EssaiXML2Java {

    public static void main( String[] args ) {
        try {
            // 1. Création d'un JAXBContext: point d'entrée dans l'API JAXB
            JAXBContext jc = JAXBContext.newInstance("com.iut.testJAXB");

            // 2. Creation d'un Unmarshaller
             Unmarshaller unm = jc.createUnmarshaller();

           // 3. Lecture du flux XML et construction du graphe d'objets Java
            Catalogue unCatalogue =
                (Catalogue) unm.unmarshal(new FileInputStream("testJAXB.xml"));

          // 4. Consultation des objets : affichage du catalogue
            Collection<Livre> liste= unCatalogue.getLivre();
            for(Livre val : liste){
                  System.out.println("Titre : " + val.getTitre());
                  System.out.println("Auteur : " + val.getAuteur());
                  System.out.println("Edition : " + val.getEdition());
                  System.out.println("ISBN : " + val.getISBN());
                  System.out.println("----------------------");
              }

        } catch (JAXBException e) {
            e.printStackTrace();
        }
        catch (IOException e) { // Exception nécessaire en raison du FileInpuStream
            e.printStackTrace();
        }
    }// fin main
}
```

```
Console
<terminated> EssaiXML2Java [Java Application] C:\
Titre :  XML
Auteur : Alexandre BRILLANT
Edition : Eyrolles
ISBN : 92-212-12151-3
----------------------
Titre : JAVA/XML
Auteur : Renaud FLEURY
Edition : Cahier du Programmeur
ISBN : 2-212-11316-1
----------------------
```

Après exécution, vous devriez alors obtenir un affichage similaire à la copie d'écran ci-contre.

Remarque : Bien sûr, on aurait pu simplifier le contenu de la boucle `for` en écrivant simplement :

```
for(Livre val : liste){
   System.out.println(val);
}
```

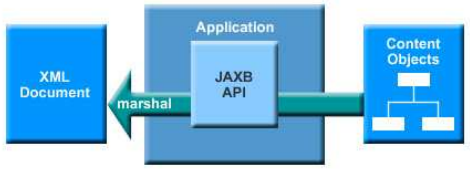
Pour visualiser le contenu de chaque livre, il est alors nécessaire de redéfinir la méthode **`toString`** dans la classe **`Livre`** générée.

Livre.java

```
public String toString() {
    return "titre : " + titre + "\n" +
    "auteur : " + auteur + "\n" +
    "edition : " + edition + "\n" +
    "isbn : " + isbn + "\n" ;
    }
```

4. Opération `marshalling` : Mapping objets JAVA → flux XML (redistribution des données)

JAXB permet ***de créer un flux XML à partir d'un graphe d'objets JAVA***: cette opération de transformation est nommée **`marshalling`**.
Une operation de `marshalling` est l'opération inverse de l'opération d'`unmarshalling`.



Source :
http://www.oracle.com/technetwork/articles/javase/index-140168.html

La redistribution des données consiste à récupérer le contenu de chaque instance de classe et à les insérer dans un flux XML.
Comme pour le rassemblement, pour la redistribution, nous disposons d'un **distributeur** (`Marshaller`). Ce dernier est chargé de vérifier l'intégrité des données qu'il transporte. Si une erreur intervient, il pourra lever des exceptions
La redistribution des données (**`marshalling`**) s'appuie sur les méthodes du paquetage **`javax.xml.bind`** de l'`API JAXB`.

4.1 L'indispensable objet `JAXBContext`

Comme dans le cas de l'`unmarshalling`, l'objet **`JAXBContext`** du paquetage **`javax.xml.bind`** est le point d'entrée dans **`l'API JAXB`** qui va donner accès aux opérations de transformation de type `marshalling` :

```
JAXBContext jc = JAXBContext.newInstance("com.iut.testJAXB");
```

4.2 Objet `Marshaller` pour le mapping Objets Java vers document XML

Il faut ensuite instancier un objet de type **`Marshaller`** qui va permettre de transformer un ensemble d'objets java en un **flux XML**. Une telle instance est obtenue en utilisant la méthode **`createMarshaller()`** de la classe **`JAXBContext`** :

```
Marshaller marshaller = jc.createMarshaller();
```

4.3 Ecriture du flux XML à partir du graphe d'objets Java :

↳ La méthode **`marshal()`** de la classe **`Marshaller`** se charge de créer un **flux XML** à partir d'un graphe d'objets java dont l'objet racine (de nom `jaxbElement` dans la javadoc) est fourni en tant que premier paramètre

javax.xml.bind
Interface Marshaller

`void` **`marshal`**`(java.lang.Object jaxbElement, java.io.OutputStream os)`
Marshal the content tree rooted at `jaxbElement` into an output stream.

Comme la méthode `unmarshall`, la méthode **`marshal`** possède de nombreuses surcharges au niveau du second paramètre. Ce paramètre permet de préciser la forme du flux XML généré (`File`, `OutputStream`, arbre DOM, … etc… : voir javadoc …)

↳ Un objet **`Marshaller`** possède des propriétés qu'il est possible d'activer en utilisant la méthode **`setProperty`**.

javax.xml.bind
Interface Marshaller

`void` **`setProperty`**`(java.lang.String name, java.lang.Object value)`
Set the particular property in the underlying implementation of `Marshaller`.

Les spécifications de **`JAXB`** proposent les propriétés suivantes :

Supported Properties

All JAXB Providers are required to support the following set of properties. Some providers may support additional properties.

`jaxb.encoding` - value must be a java.lang.String
The output encoding to use when marshalling the XML data. The Marshaller will use "UTF-8" by default if this property is not specified.

`jaxb.formatted.output` - value must be a java.lang.Boolean
This property controls whether or not the Marshaller will format the resulting XML data with line breaks and indentation. A true value for this property indicates human readable indented xml data, while a false value indicates unformatted xml data. The Marshaller will default to false (unformatted) if this property is not specified.

`jaxb.schemaLocation` - value must be a java.lang.String
This property allows the client application to specify an xsi:schemaLocation attribute in the generated XML data. The format of the schemaLocation attribute value is discussed in an easy to understand, non-normative form in Section 5.6 of the W3C XML Schema Part 0: Primer and specified in Section 2.6 of the W3C XML Schema Part 1: Structures.

`jaxb.noNamespaceSchemaLocation` - value must be a java.lang.String
This property allows the client application to specify an xsi:noNamespaceSchemaLocation attribute in the generated XML data. The format of the schemaLocation attribute value is discussed in an easy to understand, non-normative form in Section 5.6 of the W3C XML Schema Part 0: Primer and specified in Section 2.6 of the W3C XML Schema Part 1: Structures.

`jaxb.fragment` - value must be a java.lang.Boolean
This property determines whether or not document level events will be generated by the Marshaller. If the property is not specified, the default is `false`. This property has different implications depending on which marshal api you are using - when this property is set to true:

Nous utiliserons pour notre exemple la propriété **`jaxb.formatted.output`** qui permet l'indentation du document (par défaut, cette propriété est à `false`).
La propriété **`jaxb.encoding`** peut également être intéressante si on souhaite changer l'encodage du document (par défaut un encodage UTF-8 est utilisé)

4.4 Création du graphe d'objets en Java compatible JAXB:

La question que l'on doit se poser maintenant est : ***comment doit-on construire le graphe d'objets dans notre application java pour permettre une opération de*** **`marshalling`** ***?***

javax.xml.bind
Interface Marshaller

`void` **`marshal`**`(java.lang.Object jaxbElement, java.io.OutputStream os)`
Marshal the content tree rooted at `jaxbElement` into an output stream.

Le premier paramètre de la méthode **`marshal`** doit être une instance d'une classe manipulable par l'`API JAXB`.
Dans notre application Java, nous avons constaté que les classes générées par l'`API JAXB` sont des classes annotées. Le premier paramètre de la méthode `marshal` ne se limite pas qu'aux classes générées par le compilateur `xjc`.
Vous pouvez très bien utiliser une classe de votre application : la condition à respecter est que cette classe soit correctement annotée avec des annotations `JAXB` sous peine de **`MarshalException`**.

4.4.1 Création du graphe d'objets Java à partir d'une classe générée par la compilateur `xjc` :

4.4.1.1 Flux XML contenant un <`livre`>:

La classe **`Livre.java`** a été créée à partir du compilateur `xjc` de `JAXB`.
Pour commencer simplement, nous allons nous contenter de créer un flux XML composé d'un seul livre.

Travail à faire :

↳ Dans votre projet **`ProgrammationXML_JAXB`**, importer la classe **`EssaiJava2XML.java`** disponible sur la zone libre dont le code est le suivant :

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

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;


public class EssaiJava2XML {

   public static void main( String[] args ) {
      try {
         // Création d'un JAXBContext: point d'entrée dans l'API JAXB
         JAXBContext jc = JAXBContext.newInstance("com.iut.testJAXB");

         // Creation d'un Marshaller
          Marshaller marshaller = jc.createMarshaller();
          marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

          // Creation d'un graphe d'objets compatible JAXB
          Livre livre1 = new Livre();
          livre1.setTitre("XML");
          livre1.setAuteur("Alexandre BRILLANT");
          livre1.setEdition("Eyrolles");
          livre1.setISBN("92-212-12151-3");

          // Ecriture du flux XML à partir
          // de l'objet racine du graphe d'objets
         marshaller.marshal(livre1, new FileOutputStream("JAXBOutput.xml"));

      } catch (JAXBException e) {
         e.printStackTrace();
      }
      catch (IOException e) {// Exception nécessaire en raison du FileInpuStream
         e.printStackTrace();
      }
   }// fin main
}
```

↳ Ouvrir cette classe, consulter le code et lancer l'exécution.
Vérifier qu'un fichier `JAXBOutput.xml` a bien été créé et l'ouvrir pour le consulter. (peut-être sera-t-il nécessaire de rafraîchir la vue **`Package`** avec un **F5**).

Pour illustrer ce qui a été dit précédemment :

- Vous pouvez relancer ce programme et consulter le fichier `xml` en prenant soin de commenter l'instruction:
  `// marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);`

- Vous pouvez également relancer ce programme en commentant l'appel à la méthode `marshal` et en le remplaçant par l'instruction suivante :
  `marshaller.marshal(livre1, System.out);`
  Vous constaterez alors que le flux XML s'affiche désormais dans la console (`System.out`)

Afin de continuer le tutoriel, remettez-vous dans la configuration initiale : envoi du flux XML dans le **fichier (`JAXBOutput.xml`) correctement indenté** (c-a-d code identique au code récupéré sur la zone libre).

4.4.1.2 Comment enregistrer une succession de <livre> dans le flux xml ?

Nous venons d'enregistrer **un seul `livre`** dans un document **`xml`**.
Est-il possible d'enregistrer une succession de livres en essayant d'envoyer dans le flux XML une `Collection<Livre>` comme l'indique le code suivant ?

```
// …

// Creation d'un graphe d'objets
    Collection<Livre> liste = new ArrayList<Livre>();
    Livre livre1 = new Livre();
    livre1.setTitre("XML");
    livre1.setAuteur("Alexandre BRILLANT");
    livre1.setEdition("Eyrolles");
    livre1.setISBN("92-212-12151-3");
    liste.add(livre1);
// Ecriture du flux XML
    marshaller.marshal(liste, new FileOutputStream("JAXBOutput.xml"));

// …
```

La réponse est **NON**.
En effet, si vous procédiez à l'implémentation de ce code, vous constateriez qu'à l'exécution une `javax.xml.bind.JAXBException` est déclenchée. En effet, la classe `ArrayList` du paquetage `java.util.` n'est pas une classe annotée avec les annotations JAXB. Elle ne peut donc pas être passée en tant que premier paramètre de la méthode `marshal`.

```
javax.xml.bind.JAXBException: class java.util.ArrayList nor any of its super class is known to this context.
        at com.sun.xml.internal.bind.v2.runtime.JAXBContextImpl.getBeanInfo(Unknown Source)
        at com.sun.xml.internal.bind.v2.runtime.XMLSerializer.childAsRoot(Unknown Source)
        at com.sun.xml.internal.bind.v2.runtime.MarshallerImpl.write(Unknown Source)
        at com.sun.xml.internal.bind.v2.runtime.MarshallerImpl.marshal(Unknown Source)
        at javax.xml.bind.helpers.AbstractMarshallerImpl.marshal(Unknown Source)
        at com.iut.testJAXB.EssaiJava2XML.main(EssaiJava2XML.java:53)
```

Dans notre application Java, pour enregistrer une succession d'éléments **`<livre>`** dans un flux **`XML`**, il faut regrouper tous les éléments **`<livre>`** à l'intérieur d'un même élément racine, puisque rappelons-le un flux XML n'est composé que d'un seul élément racine. Pour respecter notre schèma, l'élément racine doit être un élément **`<catalogue>.`** Pour enregistrer une succession de **`<livre>`**, il ne nous reste plus qu'à créer un flux XML contenant un **`<catalogue>.`**

4.4.1.3 Flux XML contenant un <catalogue>:

L'enregistrement d'un flux **XML** dont l'élément racine est **`<catalogue>`** ne sera possible que si dans l'application Java un objet de type **`Catalogue`** compatible JAXB est créé.

La classe **`Catalogue`** générée par **`JAXB`** est composé d'un attribut **`livre`** dont le rôle est de contenir une liste de livres et d'une méthode **`getLivre`**.
Travail à faire : Procéder à l'implémentation du code suivant et constater qu'à l'exécution le fichier XML généré est bien conforme à nos attentes. Pour vous en assurer. Ajouter un nouveau livre dans le catalogue et relancer l'exécution.

```
// …
// Creation d'un graphe d'objets à partir de la classe Catalogue obtenue par xjc
    Catalogue unCatalogue = new Catalogue();
    Livre livre1 = new Livre();
    livre1.setTitre("XML3");
    livre1.setAuteur("Alexandre BRILLANT");
    livre1.setEdition("Eyrolles");
    livre1.setISBN("92-212-12151-3");
    unCatalogue.getLivre().add(livre1);
// Ecriture du flux XML
    marshaller.marshal(unCatalogue, new FileOutputStream("JAXBOutput.xml"));
// …
```

4.4.1.4 Création de graphes d'objets à partir de la classe `ObjetFactory` :

Cependant, le code précédent n'est pas le code utilisé habituellement pour créer un objet d'une classe générée par le compilateur **JAXB** comme **`Catalogue`** dans notre application.

En effet dans ce cas-là, ***les bonnes pratiques de programmation recommandent d'utiliser la classe*** ***`ObjectFactory`*** pour créer un graphe d'objets Java avant de le sérialiser dans un flux XML (opération de `marshalling`). Rappellez-vous de la classe ***`ObjectFactory`*** : c'était la troisième classe générée par le compilateur `xjc` en même temps que les classes correspondantes aux types complexes du schèma.
La classe **`ObjectFactory`** est une fabrique d'objets qui encapsule des données d'un document en respectant son schèma : la classe **`ObjectFactory`** permet d'instancier les différents objets.

Travail à faire : Procéder à l'implémentation du code ci-dessous, et constatez qu'à l'exécution le flux xml généré est bien conforme à nos attentes. Comme précédemment, vous pouvez ajouter un nouveau livre dans le catalogue et relancer l'exécution.

```
// …
// Creation d'un graphe d'objets à partir de la fabrique
    ObjectFactory fabrique = new ObjectFactory();
    Catalogue unCatalogue=fabrique.createCatalogue();

    Livre livre1 = fabrique.createLivre();
    livre1.setTitre("XML4");
    livre1.setAuteur("Alexandre BRILLANT");
    livre1.setEdition("Eyrolles");
    livre1.setISBN("92-212-12151-3");

    unCatalogue.getLivre().add(livre1);
// Ecriture du flux XML
    marshaller.marshal(unCatalogue, new FileOutputStream("JAXBOutput.xml"));
// …
```

4.4.2 Création du graphe d'objets Java à partir d'une classe annotée JAXB "manuellement"…

JAXB permet de transformer un document XML vers une ou plusieurs classes Java sans être obligé d'utiliser un schèma. En effet, il n'est pas absolument nécessaire d'utiliser les classes générées par le compilateur JAXB **`xjc`** pour effectuer des opérations de **`marshalling/umarshalling`**. Par contre, les classes Java que vous allez manipuler pour réaliser le binding doivent obligatoirement comporter des **annotations JAXB.**
Les annotations **JAXB** sont définies dans le paquetage **`javax.xml.bind.annotation`**

4.4.2.1 Essai de marshalling sur une classe Java simple : la classe `Personne`

Dans notre application Java, nous allons travailler avec la classe simplifiée **`Personne`** suivante composée de :

- ***3 attributs*** : **`nom`** et **`prenom`** de type **`String`**
  `taille` de type **`Integer`**
- ***getteurs/setteurs*** associés ainsi que d'une méthode **`toString`**.

Pour commencer, nous souhaitons sérialiser une seule `Personne` dans le fichier XML c-a-d que pour le moment nous souhaitons obtenir un flux XML ayant un élémént racine de type **`<personne>.`**

Travail à faire :

↳ Dans votre projet **`ProgrammationXML_JAXB`**, importer la classe **`Personne.java`** et la classe **`EssaiJ2X.java`** disponibles sur la zone libre.

↳ Exécuter la classe **`EssaiJ2X.java`** . Cette classe permet de réaliser une opération de `marshalling` (sérialisation d'un objet **J**ava dans un fichier **X**ML). Constatez que l'exécution déclenche une `javax.xml.bind.JAXBException` puisque la classe **`Personne`** ne possède actuellement aucune annotation **JAXB**, et notamment pas l'annotation `@XmlRootElement` .

Remarque : Dans la classe **`EssaiJ2X.java`**, le paramètre de la méthode **`newInstance`** doit être le nom de la classe, et non plus le nom du paquetage passé en paramètre au compilateur JAXB, puisque nous n'avons pas utilisé le compilateur JAXB pour générer cette classe…

```
JAXBContext jc = JAXBContext.newInstance(Personne.class);
```

4.4.2.2 Transformation de la classe Java simple en classe Java annotée (c-a-d en `JAXBElement`)

Pour qu'une classe Java puisse être compatible avec une opération de binding, il faut lui rajouter des annotations JAXB.

↳annotation `@XmlRootElement` : Indispensable
L'annotation **`@XmlRootElement`** est doit être utilisée en début de chaque classe JAVA qui décrit un élément XML qui est susceptible d'être l'élément racine d'un flux XML.
C'est pourquoi, la classe **`Personne`** qui va être sérialisée comme l'élément racine **`<personne>`** du flux XML doit être annotée avec l'annotation **`@XmlRootElement.`**
Comme nous venons de la voir, une exception de type `javax.xml.bind.JAXBException` est levée par **JAXB** si la racine ne possède pas cette annotation.

Travail à faire : Opération de **`marshalling`** (***sérialisation*** d'un objet **J**ava dans un fichier **X**ML).

↳ Rajouter l'annotation **`@XmlRootElement`** dans la classe **`Personne`**
(et son import correspondant : `import javax.xml.bind.annotation.XmlRootElement`)

```
@XmlRootElement
public class Personne {
…}
```

↳ Exécuter le fichier **`EssaiJ2X.java`** et ouvrir le document **`Personne.xml`** généré.

Remarques :

- ***L'annotation `@XmlRootElement` est indispensable pour réaliser le binding*** sinon une exception de type `javax.xml.bind.JAXBException` est levée par **JAXB** si la racine ne possède pas cette annotation.
- L'annotation **`@XmlRootElement`** peut être paramétrée. Si rien n'est précisé la balise du document XML aura le même nom que la classe Java. Mais on peut indiquer un autre nom pour la balise. Essayer par exemple :

```
@XmlRootElement(name = "toto")
public class Personne {…}
```

Puis revenir à la configuration initiale :

```
@XmlRootElement(name = "personne")
public class Personne {…}
```

Travail à faire : Opération de **`unmarshalling`** (***désérialisation*** d'un objet **J**ava depuis un fichier **X**ML).

↳ Afin de tester l'opération de **`unmarshalling`** sur votre classe `Personne` annotée **JAXB**, importer la classe **`EssaiX2J.java`** disponible sur la zone libre et exécuter.

4.4.2.3 Utilisation d'autres annotations JAXB

Nous venons de voir que l'annotation JAXB indispensable pour mettre en place les opérations de binding est l'annotation : `@XmlRootElement`
Le paquetage **`javax.xml.bind.annotation`** propose de nombreuses annotations que vous pouvez consulter en ligne en ouvrant la javadoc à l'adresse suivante : http://jaxb.java.net/nonav/2.2.3u1/docs/api/ Pour chaque annotation des exemples sont donnés. L'annexe 1 de ce tutoriel vous propose également un extrait de la javadoc.

Nous allons maintenant manipuler quatre annotations, libre à vous de tester les autres avec l'aide de la javadoc !

↳annotation @XmlAttribute :

L'annotation **@XmlAttribute** permet de sérialiser une propriété de la classe JAVA en tant qu'***attribut*** dans le flux XML.

Travail à faire : Test de **@XmlAttribute**

↳ Modifier le code de la classe `Personne.java` en appliquant l'annotation **@XmlAttribute** dans un premier temps au-dessus de la méthode `getTaille` :

```
@XmlAttribute
public Integer getTaille() {…}
```

↳Enregistrer et exécuter le fichier **EssaiJ2X.java.** Ouvrir le fichier `Personne.xml` et constater que la taille apparaît désormais en tant qu'attribut de l'élément **<personne>**…

↳Supprimer l'annotation `@XmlAttribute` de votre code et essayer de la placer au-dessus de la déclaration de l'attribut `taille` :

```
@XmlAttribute
private Integer taille;
```

Enregistrer la classe **Personne** et exécuter la classe **EssaiJ2X.java.**
Une exception survient. En effet, par défaut les opérations de `marshalling/umarshalling` de JAXB s'appuient sur les propriétés getteurs/setteurs. Pour que l'annotation `@XmlAttribute` soit prise en compte, il faudrait que les opérations de binding s'appuient sur les attributs au lieu des propriétés. Il est possible de configurer cela à l'aide de l'annotation `@XmlAccessorType.`

↳ Pour que notre annotation soit prise en compte, vous devez donc rajouter en début de classe une annotation `@XmlAccessorType` paramétrer avec `XmlAccessType.FIELD` (et les deux **import** correspondants !)

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "personne")
public class Personne {…}
```

Enregistrer la classe **Personne**, exécuter la classe **EssaiJ2X.java** et consulter le document XML généré.

↳annotation @XmlElement :

L'annotation **@XmlElement** permet de sérialiser une propriété de la classe JAVA en tant qu' ***élément*** dans le document XML . L'annotation **@XmlElement** s'applique au-dessus de la déclaration des attributs. Elle prend souvent en paramètre l'attribut **required**.

Travail à faire : Test de **@XmlElement**

↳ Remplacer l'annotation **@XmlAttribute** au-dessus de `taille` par l'annotation **@XmlElement :**

```
@XmlElement
public Integer taille;
```

↳Enregistrer et exécuter le fichier **EssaiJ2X.java.** Ouvrir le fichier `Personne.xml` et constater que la taille apparaît désormais en tant qu'élément de **<personne>**…

↳ annotation @XmlTransient

L'annotation **@XmlTransient** permet d'ignorer une entité dans le mapping.
L'attribut marqué transient ne sera pas sérialisé dans le flux XML.

Travail à faire : Test de **@XmlTransient**

↳ Modifier le code de la classe `Personne.java` en applicant l'annotation **@XmlTransient** sur l'attribut `taille` par exemple.

```
@XmlTransient
private Integer taille;
```

Enregistrer et exécuter le fichier **EssaiJ2X.java.**
Ouvrir le fichier `Personne.xml` et constater que l'élément **taille** a disparu.

↳annotation @XmlType :

L'annotation **@XmlType** permet de configurer l'ordre des éléments dans le document XML.

Travail à faire : Test de **@XmlType**

↳ Après avoir supprimer l'annotation `@XmlTransient` au dessus de l'attribut `taille`, modifier le code de la classe `Personne.java` en appliquant en début de classe l'annotation **@XmlType** :

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "taille",
    "nom",
    "prenom",
})
@XmlRootElement(name = "personne")
public class Personne {…}
```

↳Enregistrer et exécuter le fichier **EssaiJ2X.java.** Ouvrir le fichier `Personne.xml` et constater **<taille>** apparaît désormais en tant que premier élément de **<personne>**…

… Il existe encore de nombreuses autres annotations….
Si vous êtes intéressés vous pouvez consulter la javadoc, ou le tutoriel en anglais à l'adresse suivante:
http://jaxb.java.net/tutorial/section_6_1-JAXB-Annotations.html#JAXB%20Annotations

Remarque : *Il n'est pas trivial d'annoter soi-même les classes **JAXB**.*
*… Mieux vaut générer les classes Java avec le compilateur **xjc** en travaillant à partir d'un schèma…*

4.5 Validation d'un flux XML

La validation n'est pas intégrée à l'opération de `marshalling` mais elle peut être effectuée à la demande séparement.
Il est donc tout à fait possible de demander la validation du graphe d'objets.
La validation s'effectue en utilisant la classe **Validator**.
Nous n'effectuerons pas de validation dans ce tutoriel, si vous êtes intéressés vous pourrez, en fin de tutoriel, effectuer quelques recherches sur le web…

5. Génération d'un schèma à partir de classes annotées compilées Java : outil schemagen de l'API JAXB

Dans l'API JAXB est fourni par défaut un outil qui permet de générer un **schéma XML** à partir de classes annotées compilées. Cet outil est appelé **schemagen** (**Schema Gen**erator).
Le générateur **schemagen** est un utilitaire en ligne de commande :

schemagen [options] <java files>

avec comme **options** :

-d <path>	: specify where to place processor and javac generated class files
-cp <path>	: specify where to find user specified files
-classpath <path>	: specify where to find user specified files
-episode <file>	: generate episode file for separate compilation
-version	: display version information

SchemaGen est un utilitaire en ligne de commande permettant de créer des fichiers **.xsd** pour une classe Java
Contrairement à l'outil **xjc**, il n'existe pas à ma connnaissance à l'heure actuelle de *plugin* sous Eclipse pour schemagen. Pour exécuter une commande depuis Eclipse, il faut donc créer un ***external tool***.

Reportez-vous à l'annexe 3 et suivre les instructions qui expliquent comment créer dans Eclipse **un External Tool** pour l'utilisation de **schemagen** . Utiliser ensuite l'outil **schemagen** sur la classe **Personne.**

6. Exercice : Mise en pratique de JAXB dans le cadre de l'application CabinetMedical :

Dans l'application CabinetMedical, nous avons déjà utilisé 2 supports de persistance : les fichiers et la base de données. Nous allons maintenant nous intéresser à la persistance des données dans un flux XML.
Dans cet exercice, nous travaillerons dans le cas simple où les patients n'ont **pas d'ascendant**.

6.1 Documents Fichier XML et Schéma XSD: listePatients.xml et listePatients.xsd

Pour commencer, nous avons décidé d'écrire un document XML **listePatients.xml** (disponible sur la zone libre) qui correspond au fichier XML tel que nous souhaitons l'utiliser dans notre application.
Pour pouvoir utiliser le compilateur xjc de **JAXB**, nous devons obligatoirement disposer d'un schéma.
Nous avons donc créé le correspondant schèma XML correspondant dans le fichier **listePatients.xsd**
(si vous voulez savoir comment nous avons créé rapidement ce schèma, consulter l'annexe 4)

Travail à faire : Dans votre projet **CabinetMedical,** importer les fichiers **listePatients.xml** et **listePatients.xsd** disponibles sur la zone libre.

6.2 Génération automatique de classes Java annotées JAXB à l'aide du *compilateur* xjc :

Travail à faire :

Placez-vous dans la vue **Package** sur le fichier **listePatients.xsd** du projet **CabinetMedical.**
Effectuer un clic droit avec la souris et choisir l'option **JAXB2.x** puis **Run XJC**

Une fenêtre de dialogue s'ouvre.
Remplir les champs respectivement avec :
- le **nom du package** qui contiendra les classes annotées par le compilateur
  com.iut.cabinet.metier.JAXB
- et le **répertoire** oú le package sera inséré (chemin pour arriver au répertoire **src** du projet cabinetMedical).
  ……\cabinetMedical\src

Un petit clic sur **Finish…** et 4 classes se créeent dans le paquetage **com.iut.cabinet.metier.JAXB**
AdresseType.java - ListePatientsType.java - ObjectFactory.java - PatientType.java

Attention ! Si vous ouvrez **ListePatientsType**, vous constaterez l'annotation **@XmlRootElement** est absente de cette classe. Or nous avons vu précédemment que l'élément racine doit obligatoirement être marqué **@XmlRootElement :** Rajoutez cette annotation !!! C'est une limite de **JAXB** qui n'arrive pas à déterminer à partir du schèma donné quel est l'élément racine. Donc un bon réflexe avec **JAXB**, vérifier que votre élément racine XML est bien annoté **@XmlRootElement** en Java !!!

6.3 Mise en place de Helper pour convertir les classes annotées JAXB en classe métier :

Lorsque nous avions mis en place les classes DTO, nous avions eu besoin d'écrire des méthodes pour assurer la conversion des objets métiers vers les objets DTO et vice-versa
De la même manière, il sera nécessaire d'écrire des méthodes pour assurer la conversion des classes annotées JAXB vers les objets métier et vice-versa.

Les méthodes de conversion **objet DTO↔objet métier** ont été implémentées de manière statique dans la classe **HelperPatient** du paquetage **com.iut.cabinet.application**

```
public static PatientDTO toPatientDTO (Patient unPatient) throws HelperException
public static Patient toPatient (PatientDTO unPatientDTO) throws
                                        CabinetMedicalException, HelperException
```

Les méthodes de conversion **objet JAXB↔objet métier** seront implémentées de manière statique dans une classe **HelperPatient** et **HelperAdresse** dans le paquetage **com.iut.cabinet.metier.JAXB.**
Dans ce paquetage, on réécrira également une classe **HelperException**.

Travail à faire :

Importer dans le paquetage **com.iut.cabinet.metier.JAXB**, les deux classes **Helper** disponibles sur la zone libre à savoir, ainsi que la classe déclenchant des exceptions, c-a-d :

- **HelperPatient** composée des deux méthodes suivantes :

```
public static PatientType toPatientType (Patient unPatient) throws HelperException
public static Patient toPatient (PatientType unPatientTypeJAXB) throws
                                        CabinetMedicalException, HelperException
```

- **HelperAdresse** composée des deux méthodes suivantes :

```
public static AdresseType toAdresseType (Adresse uneAdresse) throws HelperException
public static Adresse toAdresse (AdresseType uneAdresseTypeJAXB) throws
                                                        HelperException
```

- **HelperException**

6.4 Mise de l'opération de `unmarshalling` : désérialisation d'un fichier XML en objets JAVA :

Grâce à la mise en place des couches logicielles IHM, contrôleur, metier, le changement de support de persistance c-a-d le passage à un flux XML doit pouvoir s'effectuer rapidement comme cela avait été le cas lors du passage des fichiers à la base de donnée.

***➢ Modifications à apporter au niveau de la Vue : au niveau des classes* `PanelCreerPatient`, `PanelListerPatients` *du paquetage* `com.iut.cabinet.presentation` …**

… Vous l'aviez deviné, il n'y a aucune modification à apporter dans Vue : c'est tout l'intérêt de la programmation en couches….

***➢ Modifications à apporter au niveau du Contrôleur : au niveau de la classe* `GererPatientCtrl` *du paquetage* `com.iut.cabinet.application` …**

Il faut bien sûr modifier le code des méthodes de la classe `GererPatientCtrl` puisque cette fois-ci le contrôleur va faire en sorte que le support de persistance soit un **fichier XML**.
Intéressons-nous pour commencer à la méthode : **`listerPatients`** de la classe `GererPatientCtrl.`

Le rôle du Contrôleur reste le même quel que soit le support de persistance.
Ainsi que l'on travaille avec des fichiers, une base de données ou un fichier XML, on retrouve pour toutes les méthodes **`listerPatients`** les étapes suivantes :

→1.Récupération de la liste de Patient(ou Personne) provenant du support de persistance :
Pour récupérer une liste de `Patient` (ou `Personne`) en mémoire, le Contrôleur fait appel à un `DAO` (c'est le `DAO` qui assure les accès vers la couche de persistance)

→ 2. Création de la liste de PatientDTO à partir de la liste d'objets métier récupérée :
Le Contrôleur transforme ensuite les objets métiers en objets `DTO` qui seront renvoyés dans la Vue.

Le seul code dans la méthode **`listerPatients`** que nous devrons modifier pour changer de support de persistance est celui de la partie 1, c-a-d la récupération de la liste de `Personne`, puisque cette fois-ci nous devrons faire appel à une classe **`PersonneDAOXML`**.

Travail à faire : ⇘ Dans la méthode **`listerPatients`** existante, nous allons réécrire la première partie de cette méthode qui concerne la récupération de la liste de `Personne.`
Pour travailler avec des **fichiers XML**, il faut désormais faire appel à une méthode **`findAllPersonne`** d'une classe **`PersonneDAOXML.`**
Votre méthode **`listerPatients`** de la classe **`GererPatientCtrl`** doit ressembler au code suivant :

```
public Collection<PatientDTO>listerPatients()throws CabinetTechniqueException,
                                                     HelperException
    {
    ////////////////////////////////////////////////////////////
    // 1. Récupération de la liste de Personne provenant du support de persistance :
    // fichier XML …
    ////////////////////////////////////////////////////////////
    Collection<Personne> maListe= PersonneDAOXML.findAllPersonne();

    ////////////////////////////////////////////////////////////
    // 2. Création de la liste de PatientDTO
    ////////////////////////////////////////////////////////////
    // ce code ne change pas par rapport à ce que vous aviez écrit avec les fichiers
    // et les bases de données
    //……

        return maListeDTO;
    } // fin listerPatients
```


***➢ Modifications à apporter au niveau du Modèle : création de la classe* `PersonneDAOXML` *du paquetage* `com.iut.cabinet.metier…`**

Dans la classe `PersonneDAOXML` du paquetage `com.iut.cabinet.metier`, nous devons écrire la méthode **`findAllPersonne`** suivante :

```
public static Collection<Personne> findAllPersonne() throws
                                                    CabinetTechniqueException
```

La méthode `findAllPersonne` va se décomposer en deux étapes :

→1.Mise en place d'une opération de `unmarshalling` (Désérialisation XML vers JAVA) en vue de la récupération d'un graphe d'objets JAVA (instancié à partir de classes annotées JAXB) :
Pour écrire cette partie, vous vous inspirerez de ce qui a été fait dans la partie 3 de ce tutoriel : "Opération `unmarshalling`" afin de retrouver les instructions suivantes :

→ `Création d'un JAXBContext depuis` **`com.iut.cabinet.metier.JAXB`**
→ `Creation d'un Unmarshaller`
→ `Lecture du flux XML (depuis le fichier` **`listePatients.xml`**`) dans un graphe d'objets Java de type` **`ListePatientsType`**

→2.Conversion du graphe d'objets JAVA instanciés à partir de classes annotées JAXB en graphe d'objets JAVA instanciés à partir de classes metier
Cette partie de code consiste à créer une `Collection<Personne>` à partir de la *liste de Patients* "JAXB" issus de l'opération de `unmarshalling` (qui correspond dans notre application au champ `patient` de la classe `ListePatientsType`)

Travail à faire :

⇘ Dans le paquetage `com.iut.cabinet.metier`, créer la classe `PersonneDAOXML` et implémenter la méthode **`listerPatients`** suivante :

```
public static Collection<Personne> findAllPersonne() throws
                                                    CabinetTechniqueException
```

Remarque : Relancer en tant que `CabinetTechniqueException` toutes les exceptions que vous capturez.

⇘ Tester votre code en exécutant la classe **`PanelListerPatients`** du paquetage `com.iut.cabinet.presentation.`

Remarques:

- Une implémentation possible de la méthode `findAllPersonne` est proposée dans la classe `PersonneDAOXML` disponible sur la zone libre.
- Pour tester votre code, vous pouvez également exécuter la méthode **`main`** de la classe `PersonneDAOXML` disponible sur la zone libre.

6.4 Mise de l'opération de `marshalling` : sérialisation d'objets JAVA dans un fichier XML:

En suivant la même démarche, nous allons maintenant mettre en place dans notre application l'opération de marshalling en nous intéressant à la méthode **`creerPatient`** du Contrôleur.

➢ Modifications à apporter au niveau de la Vue … toujours aucune modification au niveau de la Vue grâce à la programmation en couches….

➢ *Modifications à apporter au niveau du Contrôleur : classe* `GererPatientCtrl`

Pour enregistrer un patient dans le fichier XML, nous pouvons garder la même démarche que celle que nous avions adoptée avec les fichiers : nous avions alors choisi de toujours travailler à partir de la liste de Personne (Patients) contenue dans le fichier de persistance.

Rappel de l'énoncé du TP lors de la gestion de la création d'un Patient dans un simple fichier :

La création d'un `Patient` revient donc à ajouter un `Patient` dans la liste de persistance grâce au processus suivant :

a. charger dans une `Collection` la liste de `Personne` initialement contenue dans le fichier
b. créer un nouvel objet de type **`Patient`**
c. ajouter cet objet à la liste
d. sauvegarder la nouvelle liste dans le fichier (nouvelle liste incluant bien sûr le nouveau Patient)

Travail à faire :

Dans le paquetage `com.iut.cabinet.application`, implémenter la méthode **`creerPatient`** de la classe `GererPatientCtrl` de la manière suivante :

```
public void creerPatient(PatientDTO unPatientDTO) throws
            CabinetMedicalException, HelperException, CabinetTechniqueException
   {
      Collection<Personne> maListe=PersonneDAOXML.findAllPersonne();

      ////////////////////////////////////////////////////
      // Etape 1 : Créer un objet métier à partir du DTO
      ////////////////////////////////////////////////////
      Patient unPat;
      unPat = HelperPatient.toPatient(unPatientDTO);
      maListe.add(unPat);


      ////////////////////////////////////////////////////////////////////////
      // Etape 2 : Assurer la persistance des objets métiers à l'aide du DAO
      ////////////////////////////////////////////////////////////////////////
       PersonneDAOXML.storeAllPersonne(maListe);

   }// fin creerPatient
```

➢ *Modifications à apporter au niveau du Modèle : classe* `PersonneDAOXML`

Dans la classe `PersonneDAOXML` du paquetage `com.iut.cabinet.metier`, nous devons écrire la méthode **`storeAllPersonne`** suivante :

```
   public static void storeAllPersonne (Collection<Personne> uneListe)
                                                  throws CabinetTechniqueException
```


La méthode `storeAllPersonne` va se décomposer en deux étapes :

→ 1. Conversion du graphe d'objets JAVA instanciés à partir de classes métier en graphe d'objets JAVA instanciés à partir de classes annotées JAXB
Cette partie de code consiste à créer une `ListePatientsType` à partir `Collection<Personne>` passée en paramètre. Pour cela, on utilisera la classe `ObjectFactory` (voir partie 4.4.1.4 du tutoriel).

→2. Mise en place d'une opération de `marshalling` (Sérialisation JAVA vers XML)
Pour écrire cette partie, vous vous inspirerez de ce qui a été fait dans la partie 4.Opération `marshalling` de ce tutoriel.

Travail à faire :

Dans le paquetage `com.iut.cabinet.metier`, créer la classe `PersonneDAOXML` et implémenter la méthode **`storeAllPersonne`** suivante :

```
   public static void storeAllPersonne (Collection<Personne> uneListe)
                                                  throws CabinetTechniqueException
```

Remarque : Relancer en tant que `CabinetTechniqueException` toutes les exceptions que vous capturez.

Pour tester votre code, commencez par utiliser la méthode **`main`** de la classe `PersonneDAOXML` (disponible sur la zone libre).
- Commencer par supprimer le fichier **`listerPatients.xml`** de votre projet **`cabinetMedical`**.
- Décommenter la partie ***`Test storeAllPersonne`*** de la méthode **`main`** de la classe **`PersonneDAOXML`** .

Il ne reste plus qu'à exécuter ce main pour voir si la méthode ***`storeAllPersonne`*** fonctionne…

Tester ensuite votre code à partir de l'IHM proposée dans la classe **PanelCreerPatient** du paquetage `com.iut.cabinet.presentation`

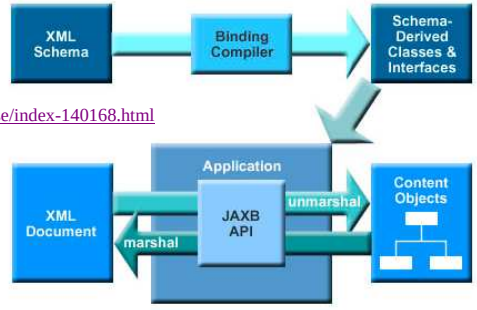
6.6 Pour aller plus loin …

… Libre à vous d'envisager maintenant d'améliorer votre application, en complétant le CRUD, en vous intéressant au cas de l'ascendance d'un **`Patient`**, en rendant persistant dans un fichier XML la liste de Professionnel, etc …

Rappel des principaux avantages de JAXB…

L'**`API JAXB`** permet de faciliter la gestion de la persistance de données stockées dans un flux XML :
- en automatisant la génération des classes à partir des schèmas XML (gain de temps pour le développeur)
- en assurant une bonne intégrité des données (exceptions levées dès qu'une erreur intervient)
- en étant simple d'utilisation (pas de fichier de configuration)

Source :
http://www.oracle.com/technetwork/articles/javase/index-140168.html



Annexe 1 : Liste des Annotations JAXB 2.0 du paquetage `javax.xml.bind.annotation`

Annotation Types Summary	
XmlAccessorOrder	Controls the ordering of fields and properties in a class.
XmlAccessorType	Controls whether fields or Javabean properties are serialized by default.
XmlAnyAttribute	Maps a JavaBean property to a map of wildcard attributes.
XmlAnyElement	Maps a JavaBean property to XML infoset representation and/or JAXB element.
XmlAttachmentRef	Marks a field/property that its XML form is a uri reference to mime content.
XmlAttribute	Maps a JavaBean property to a XML attribute.
XmlElement	Maps a JavaBean property to a XML element derived from property name.
XmlElementDecl	Maps a factory method to a XML element.
XmlElementRef	Maps a JavaBean property to a XML element derived from property's type.
XmlElementRefs	Marks a property that refers to classes with `XmlElement` or JAXBElement.
XmlElements	A container for multiple @`XmlElement` annotations.
XmlElementWrapper	Generates a wrapper element around XML representation.
XmlEnum	Maps an enum type `Enum` to XML representation.
XmlEnumValue	Maps an enum constant in `Enum` type to XML representation.
XmlID	Maps a JavaBean property to XML ID.
XmlIDREF	Maps a JavaBean property to XML IDREF.
XmlInlineBinaryData	Disable consideration of XOP encoding for datatypes that are bound to base64-encoded binary data in XML.
XmlList	Used to map a property to a list simple type.
XmlMimeType	Associates the MIME type that controls the XML representation of the property.
XmlMixed	Annotate a JavaBean multi-valued property to support mixed content.
XmlNs	Associates a namespace prefix with a XML namespace URI.
XmlRegistry	Marks a class that has `XmlElementDecl`s.
XmlRootElement	Maps a class or an enum type to an XML element.
XmlSchema	Maps a package name to a XML namespace.
XmlSchemaType	Maps a Java type to a simple schema built-in type.
XmlSchemaTypes	A container for multiple @`XmlSchemaType` annotations.
XmlSeeAlso	Instructs JAXB to also bind other classes when binding this class.
XmlTransient	Prevents the mapping of a JavaBean property/type to XML representation.
XmlType	Maps a class or an enum type to a XML Schema type.
XmlValue	Enables mapping a class to a XML Schema complex type with a simpleContent or a XML Schema simple type.

Javadoc extraite de : https://jaxb.dev.java.net/nonav/2.2-ea/docs/api/

Package javax.xml.bind.annotation Description

Defines annotations for customizing Java program elements to XML Schema mapping.

Package Specification

The following table shows the JAXB mapping annotations that can be associated with each program element.

Program Element	JAXB annotation
Package	XmlAccessorOrder XmlAccessorType XmlSchema XmlSchemaType XmlSchemaTypes XmlJavaTypeAdapter XmlJavaTypeAdapters
Class	XmlAccessorOrder XmlAccessorType XmlInlineBinaryData XmlRootElement XmlType XmlJavaTypeAdapter
Enum type	XmlEnum XmlEnumValue (enum constant only) XmlRootElement XmlType XmlJavaTypeAdapter
JavaBean Property/field	XmlElement XmlElements XmlElementRef XmlElementRefs XmlElementWrapper XmlAnyElement XmlAttribute XmlAnyAttribute XmlTransient XmlValue XmlID XmlIDREF XmlList XmlMixed XmlMimeType XmlAttachmentRef XmlInlineBinaryData XmlElementDecl (only on method) XmlJavaTypeAdapter
Parameter	XmlList XmlAttachmentRef XmlMimeType XmlJavaTypeAdapter

Javadoc extraite de : https://jaxb.dev.java.net/nonav/2.2-ea/docs/api/

Annexe 2 : Correspondance des types XML↔ JAVA

Correspondance des types utilisée lors du mapping schéma XML → JAVA

XML Schema Type	Java Data Type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd.long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

Correspondance des types utilisée lors du mapping JAVA →schéma XML

Java Class	XML Data Type
java.lang.String	xs:string
java.math.BigInteger	xs:integer
java.math.BigDecimal	xs:decimal
java.util.Calendar	xs:dateTime
java.util.Date	xs:dateTime
javax.xml.namespace.QName	xs:QName
java.net.URI	xs:string
javax.xml.datatype.XMLGregorianCalendar	xs:anySimpleType
javax.xml.datatype.Duration	xs:duration
java.lang.Object	xs:anyType
java.awt.Image	xs:base64Binary
javax.activation.DataHandler	xs:base64Binary
javax.xml.transform.Source	xs:base64Binary
java.util.UUID	xs:string

Annexe 3 : Manipulation de l'outil `schemagen` dans Eclipse à partir de la création d'un `External Tool`

L'outil **`schemagen`** fourni dans l'API JAXB permet de générer un **schéma XML** à partir de classes Java annotées compilées.

Le générateur **`schemagen`** est un utilitaire en ligne de commande :

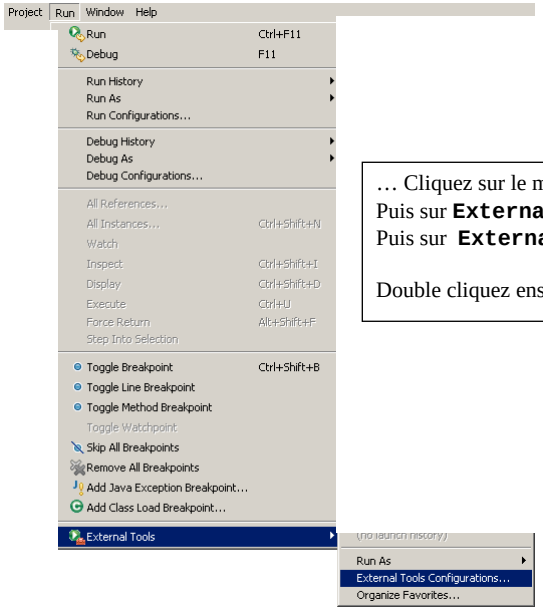
`schemagen [options]` `<java files>`

avec comme **`options`** :

- **`-d <path>`** : specify where to place processor and javac generated class files
- **`-cp <path>`** : specify where to find user specified files
- **`-classpath <path>`** : specify where to find user specified files
- **`-episode <file>`** : generate episode file for separate compilation
- **`-version`** : display version information

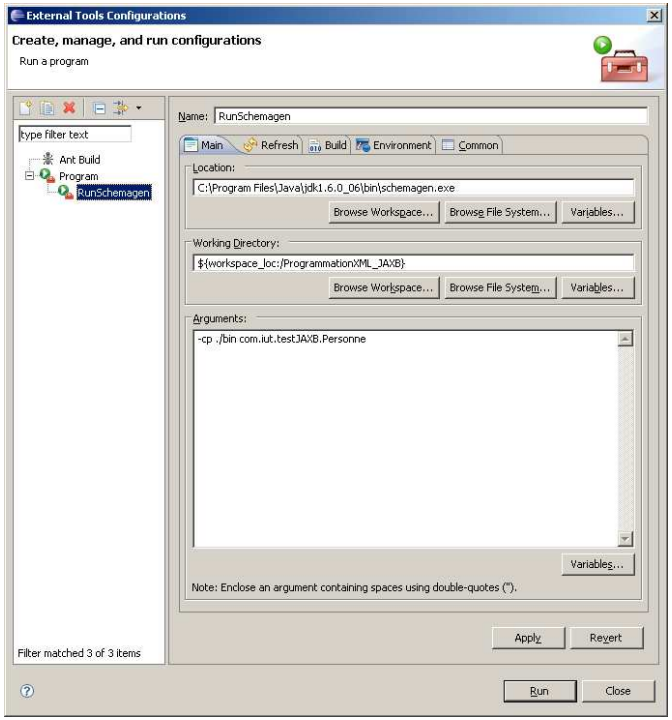
*Comment faire pour utiliser **`schemagen`** sous **Eclipse** ?*

Contrairement à l'outil **`xjc`**, il n'existe pas à ma connnaissance à l'heure actuelle de *plugin* sous Eclipse pour `schemagen`. Pour exécuter une commande depuis Eclipse, il faut passer par un ***external tool***.
Pour créer un external tool pour **`schemagen`**, suivez les étapes suivantes :



… Cliquez sur le menu **`Run`**
Puis sur **`External Tools`**
Puis sur **`External Tools Configurations...`**

Double cliquez ensuite sur **`Program`**



↳ Dans le premier onglet **`Main`** :
→ champ **`Name`**: vous devez donner le nom que vous souhaitez à la nouvelle configuration que vous êtes en train de créer et qui permettra de lancer **`schemagen`** par exemple : **`RunSchemagen`**

→ champ **`Location :`** vous devez indiquer le *chemin de l'exécutable* qui sera lancé (c-a-d retrouver à l'aide du bouton **`Browse File System`**… le fichier **`schemagen.exe`**) **`C:\Program Files\Java\jdk1.6.0_06\bin\schemagen.exe`**

→ champ **`Working Directory :`** vous devez indiquer le chemin vers votre projet. Cliquez sur **`Browse Workspace…`** et recherchez le répertoire du projet. Vous devez obtenir : **`${workspace_loc:/ProgrammationXML_JAXB}`**

→ champ **`Arguments :`** vous devez indiquer les arguments à passer à votre exécutable. D'après la document l'option **`-cp`** permet de spécifier où trouver les classes Java : comme schemagen attend des classes compilées, il faut lui indiquer le répertoire **`./bin`**
On précise ensuite le nom de(s) classe(s) : 1 seule pour notre projet qui est **`com.iut.testJAXB.Personne`**
Au final, le champ **`Arguments`** contient : **`-cp ./bin com.iut.testJAXB.Personne`**

↳ Passer à l'onglet "**`Refresh`**", cochez les cases "**`Refresh ressources upon completion`**" et "**`The project containing the selected resource`**" afin que votre projet soit rafraîchi dès la fin de l'exécution de la commande

↳ Passer à l'onglet "**`Build`**", cochez la case "**`The project containing the selected resource`**"

↳ Passer à l'onglet "**`Common`**", cochez sur **`External Tool`**

↳ Appuyer sur "**`Apply`**" pour sauvegarder la configuration saisie

Une fois la configuration finie, pour lancer **`RunSchemagen`**, il suffit de cliquer sélectionner :
`Run→External Tools→` **`RunSchemagen.`**

Si tout se passe bien un message qui commence par `Note: Writing…` s'affiche dans la console (cela peut parfois prendre un peu de temps …)

Depuis la vue **`Package`**, placez-vous sur le projet **`ProgrammationXML_JAXB`** et effectuer un **`F5`** pour effectuer un rafraîchissement des fichiers. Dans votre arborescence, le fichier **`schema1.xsd`** doit apparaître.
Remarque : on ne peut pas choisir le nom du fichier généré : ce sera toujours **`schema1.xsd`**

Annexe 4 : Comment nous avons créer le schèma XML `listePatients.xsd` pour notre application `CabinetMedical`

Pour commencer, nous avons décidé d'écrire un document XML **`listePatients.xml`** qui respecte le format XML (enchaînement des éléments) tel que nous souhaitons l'utiliser dans notre application.

Pour pouvoir utiliser le compilateur **`xjc`** de **`JAXB`**, nous devons obligatoirement disposer d'un schéma.
Nous avons utilisé le site mentionné précédemment dans ce tutoriel qui propose un générateur de schèma en ligne (http://www.xmlforasp.net/CodeBank/System_Xml_Schema/BuildSchema/BuildXMLSchema.aspx).

A partir du document **`listePatients.xml`**, et après avoir coché l'option **`Separate Complex Type`**[1] nous avons obtenu un schèma XML .

Nous avons enregistré ce schéma dans le fichier **`listePatients.xsd.`**

Nous avons cependant changé certains types du schéma XML qui ne nous convenait pas :

- **la date de naissance** que nous préférions en **`date`** plutôt qu'en `string` …

```
<xsd:element name="dateNaissance" type="xsd:dateTime" />
                    … a été remplacé par …
<xsd:element name="dateNaissance" type="xsd:date" />
```

- **le téléphone, le portable** et le **codepostal** que nous préférions en **`string`** plutôt qu'en `int` …

```
<xsd:element name="portable" type="xsd:int" />
<xsd:element name="email" type="xsd:int" />
<xsd:element name="codePostal" type="xsd:int" />
                    … ont été remplacés par …
<xsd:element name="portable" type="xsd:string" />
<xsd:element name="email" type="xsd:string" />
<xsd:element name="codePostal" type="xsd:string" />
```

- **le nir** que nous préférions **`string`** plutôt qu'en `decimal` …

```
<xsd:element name="nir" type="xsd:string" />
```

- pour tous les champs de l'**`adresse`** (`numero,rue,voie,batiment,codePostal,ville,pays`) nous avons supprimé la partie `maxOccurs="unbounded"` dans la déclaration du champ, sinon lors du mapping nous aurions eu un champ Java `List<String>` au lieu d'un simple `String`.

```
xsd:element maxOccurs="unbounded" name="numero" type="xsd:string" />
```

Remarque : Nous avons volontairement laissé le type **`string`** pour l'élément de nom **`sexe`** :

```
<xsd:element name="sexe" type="xsd:string" />
```

Nous avons ensuite ré-enregistré ce schéma dans le fichier **`listePatients.xsd.`**

[1] ***Remarque :*** Choix de l'option ***Separate Complex Types*** sur le site du générateur de shcèma :

- ***Russian Doll Style*** : lors du binding (XML→ JAVA), les classes `Patient` et `Adresse` générées seront **des classes internes statiques** à la `classe ListePatients` ⇒ ce qui n'est pas vraiment exploitable pour notre application

- ***Separate Complex Types*** : lors du binding (XML→ JAVA), les classes `Patient` et `Adresse` générées seront des **classes externes totalement indépendantes** de la `classe ListePatients`

… A vous d'essayer les deux options si vous n'êtes pas convaincu !!! …

Sur la zone libre, les deux fichiers sont disponibles, il ne reste plus qu'à les compiler sous Eclipse avec ***`xjc`****…*

- le fichier **`listePatients.xsd`** : schèma généré à partir de l'option ***Separate Complex Types***
- le fichier **`listePatientsRussianDole.xsd`** : schèma généré à partir de l'option ***Russian Doll Style***