Technologie XML

- Concepts XML
- DTD, Shémas XML
- XSL, Xpath
- XQuery
- Application de XML :
  - SVG
  - JDOM
  - Web Services

Mohamed Youssfi

Laboratoire Signaux Systèmes Distribués et Intelligence Artificielle (SSDIA)

ENSET, Université Hassan II Casablanca, Maroc

Email : med@youssfi.net

Supports de cours : http://fr.slideshare.net/mohamedyoussfi9

Chaîne vidéo : http://youtube.com/mohamedYoussfi

Recherche : http://www.researchgate.net/profile/Youssfi_Mohamed/publications

Technologie XML

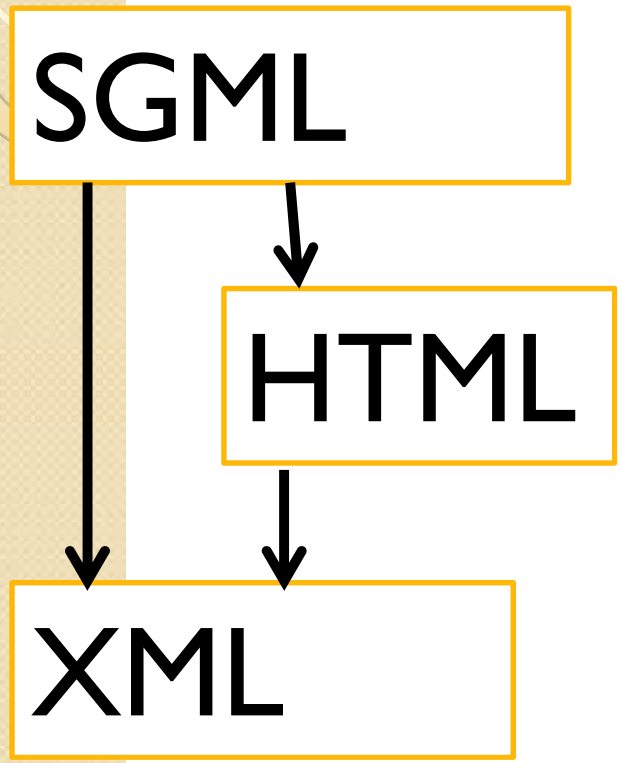
Par M.Youssfi

med@youssfi.net

Programme

- Technologie XML
  - XML,
  - DTD,
  - Schémas XML
  - XSL ,
  - XPath ,
  - Xquery
- Applications de XML
  - SVG
  - Web Services

Origine de XML

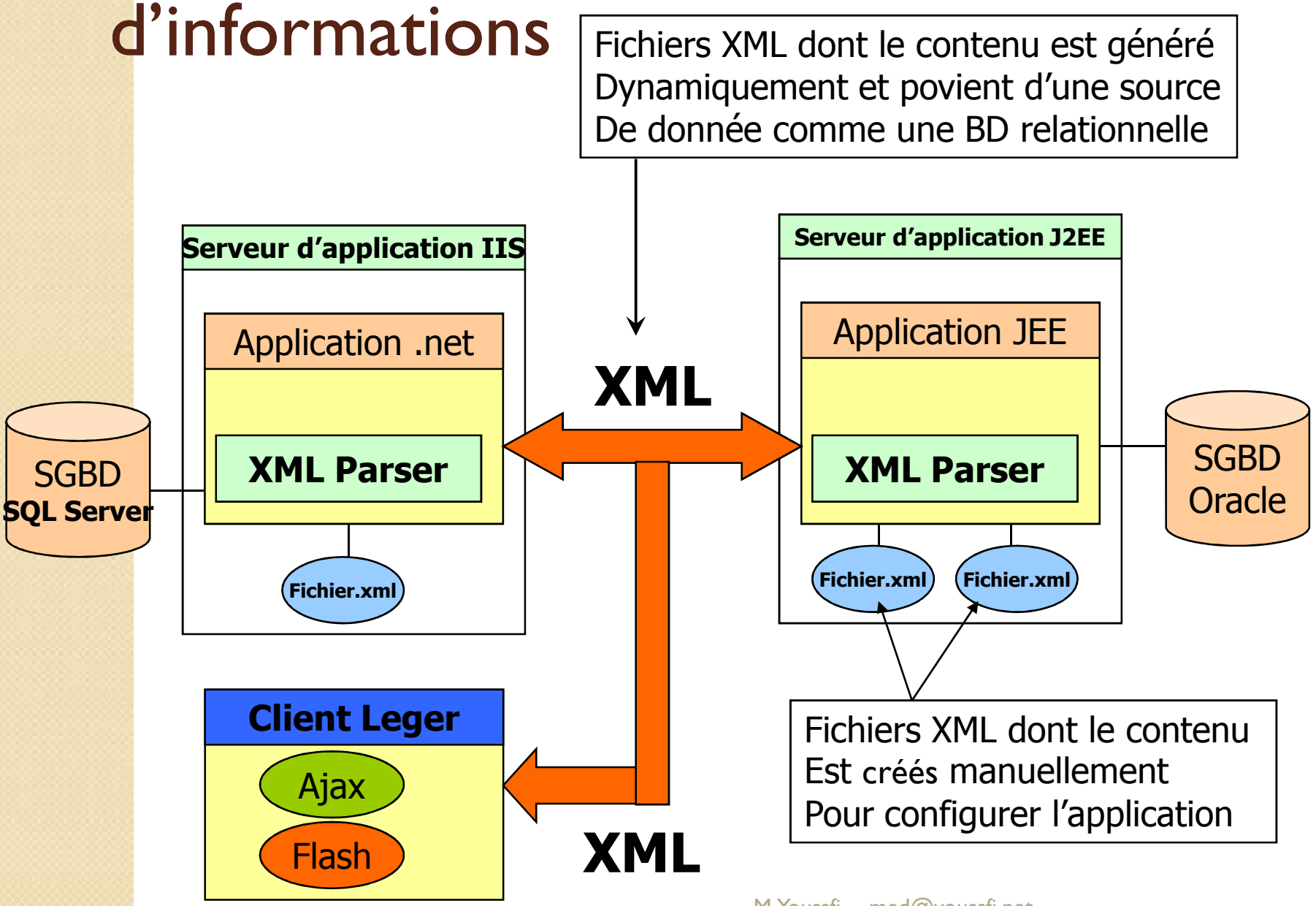


Standard Generalized Markup Language
(Sépare les données la structure des données et la mise en forme)

Hyper Text Markup Language
(Mélange les données et la mise en forme)

eXtensible Markup Language
(Sépare les données la structure des données et la mise en forme)

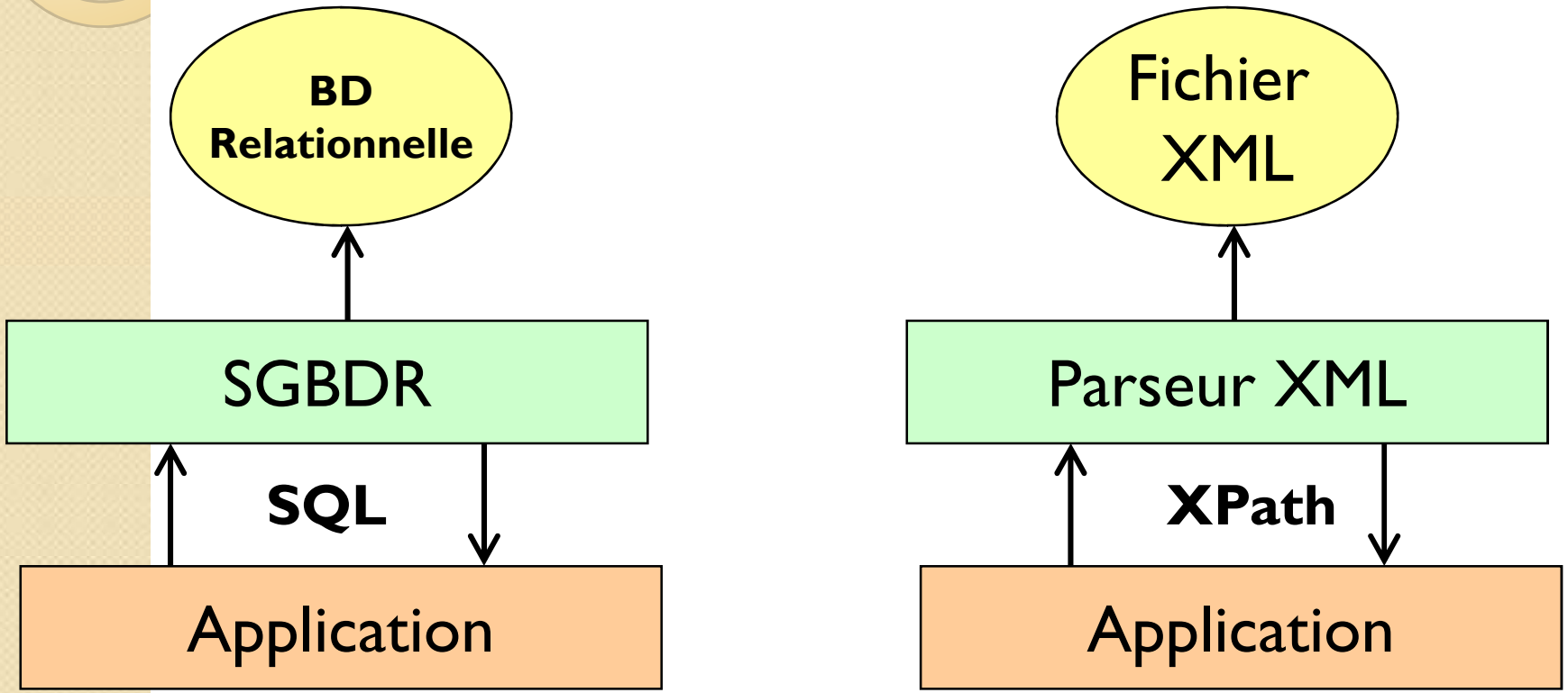
XML est au cœur des systèmes d'informations



XML ?

- XML est un langage d'échange de données structurés entre applications de différents systèmes d'informations.
- Les données d'un fichier XML sont organisée d'une manière hiérarchique
- Les données d'un fichier XML peuvent provenir des bases de données relationnelles. (Documents XML Dynamiques)
- Les fichiers XML sont également utilisés en tant que fichiers de configuration d'une application. (Documents XML Statiques)
- Pour lire un fichier XML, une application doit utiliser un parseur XML.
- Un parseur XML est une API qui permet de parcourir un fichier XML en vue d'en extraire des données précises.

Correspondance entre XML et Bases de données relationnelles

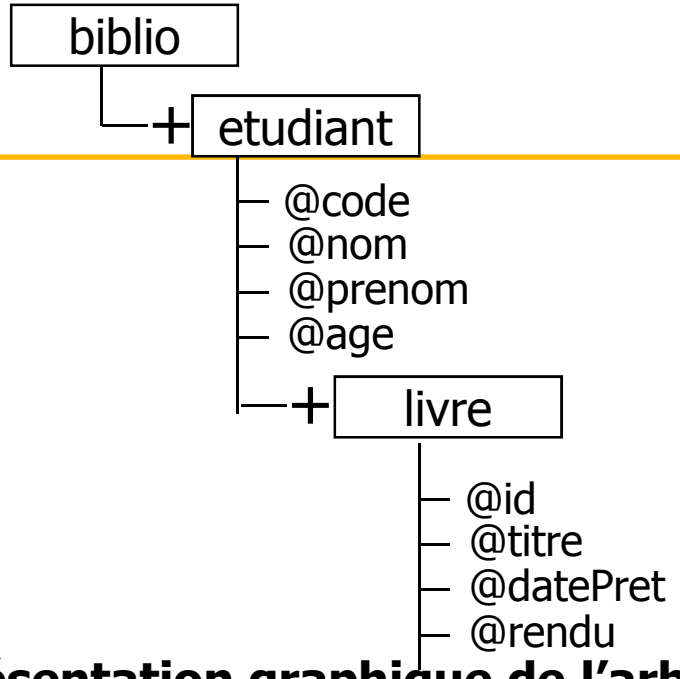


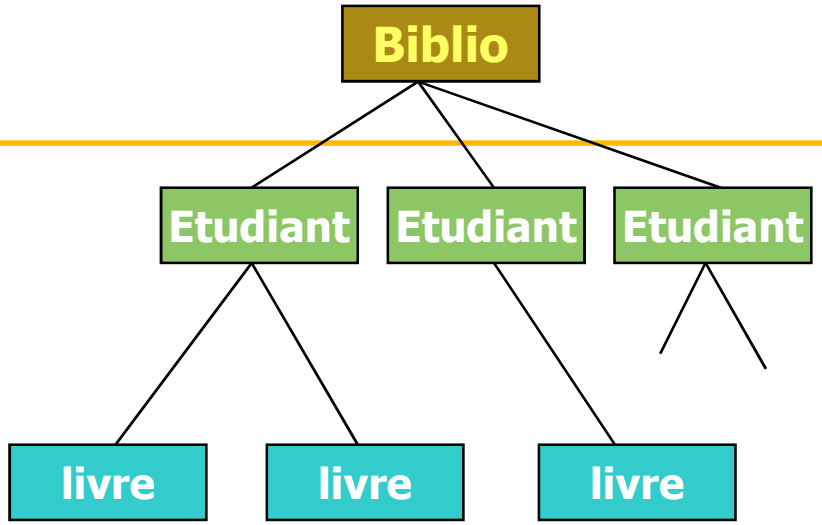
XML?

- Le parseur XML permet de créer une structure hiérarchique contenant les données contenues dans le document XML.
- Il existe deux types de parseurs XML:
  - DOM (Document Object Model) : permet d'accéder et d'agir d'une manière directe sur le contenu et la structure de l'arbre XML.
  - SAX (Simple API for XML) : permet de réagir sur le contenu et la structure d'une document XML pendant une lecture séquentielle.

Exemple de document XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<biblio>
   <etudiant code="1" nom="A" prenom="B" age="23">
      <livre id="534" titre="java" datePret="2006-11-12" rendu="oui"/>
      <livre id="634" titre="XML" datePret="2006-11-13" rendu="non"/>
   </etudiant>
   <etudiant code="2" nom="C" prenom="D" age="22">
      <livre id="33" titre="E-Commerce" datePret="2006-1-12" rendu="non"/>
   </etudiant>
</biblio>
```

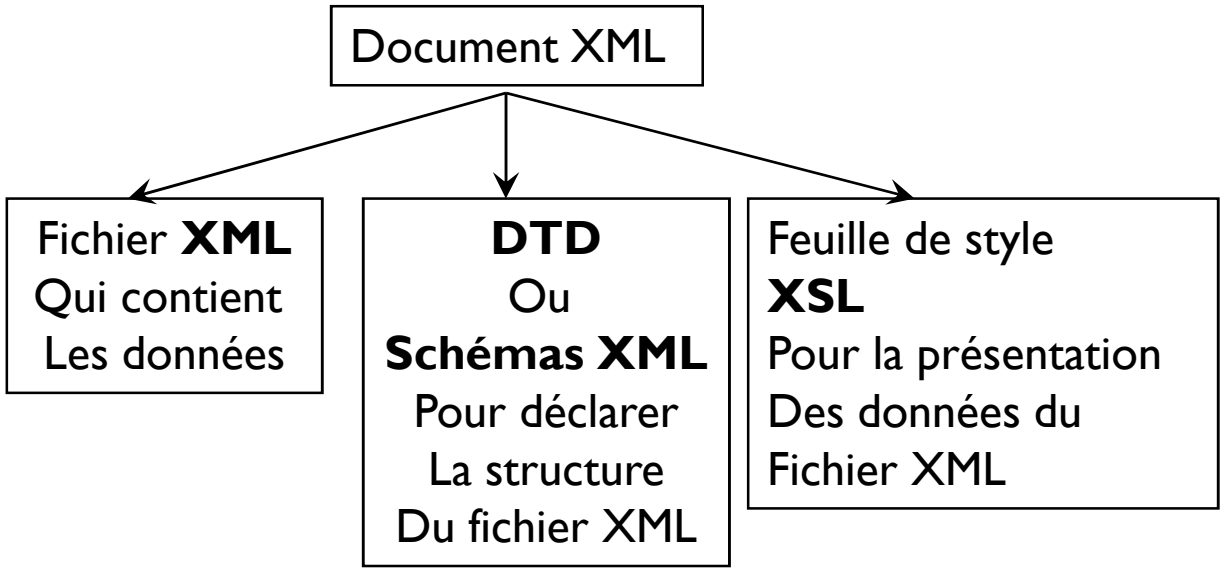




Représentation graphique de l'arbre XML

Structure d'un document XML

Un document XML se compose de 3 fichiers :



- Le fichier XML stocke les données du document sous forme d'un arbre
- DTD (Data Type Definition) ou Schémas XML définit la structure du fichier XML
- La feuille de style définit la mise en forme des données de la feuille xml

Syntaxe d'un document XML

- Un document XML est composé de deux parties :
  - **Le prologue**, lui même composés de plusieurs parties
    - Une **déclaration XML**, qui permet de définir :
      - la version de XML utilisée,
      - le codage des caractères
      - la manière dont sont stockées les informations de balisage

      ```
      <?xml version="1.0" encoding="ISO-8859-1" ?>
      ```
    - Une **déclaration de type de document** (DTD) qui permet de définir la structure logique du document et sa validité.

      ```
      <!DOCTYPE racine  SYSTEM "produit.dtd">
      ```
    - Une **transformation** définie par une feuille de style (CSS ou XSL) qui permet de définir la présentation.

      ```
      <?xml-stylesheet type="text/xsl" href="produit.xsl"?>
      ```
  - **L'instance** qui correspond au balisage du document proprement-dit (Données sous forme d'arbre).

Terminologies XML

- Balise
  - C'est un mot clef choisi par le concepteur du document qui permet de définir un élément.
    - Exemple : **<formation>**
- Elément
  - C'est un objet XML défini entre une balise de début et une balise de fin. La balise de fin porte le même nom que la balise de début, mais elle est précédée d 'un "slash".
    - **<formation>**
    - **Contenu de l 'élément**
    - **</formation>**
  - Un élément peut contenir aussi d'autre éléments
- Attribut
  - Une élément peut être qualifié par un ou plusieurs attributs. Ces attributs ont la forme clef="valeur".
    - **<formation code="T03" type="qualifiante" >**

Document XML bien formé

- Pour qu'un document soit bien formé, il doit obéir à 4 règles :
  - Un document XML ne doit posséder qu'une seule racine
  - Tous les éléments doivent être fermés
  - Les éléments contenus et contenant doivent être imbriqués.
  - La valeurs des attributs s'écrit entre guillemets

Document XML bien formé

- Un document XML ne doit posséder qu'un seul élément racine qui contient tous les autres.
  - Un document XML est un arbre.

Document mal formé :

```
<?xml version="1.0" ?>
<formation>
  TSIG Etudes
</formation>
<durée>
  1755 heures
</durée>
```

Document bien formé :

```
<?xml version="1.0" ?>
<formation>
  <nom>
    TSIG Etudes
  </nom>
  <durée>
    1755 heures
  </durée>
</formation>
```

Document XML bien formé

- Tous les éléments doivent être fermés
  - A chaque balise ouvrant doit correspondre une balise fermante.
  - A défaut, si un élément n'a pas de contenu, il est possible d'agréger la balise fermante à la balise ouvrant en terminant celle-ci par un "slash".

Document mal formé :

```
<?xml version="1.0" ?>
<formation>
  <nom>
    TSIG Etudes
  </nom>
  <durée valeur="1755">
</formation>
```

Document bien formé:

```
<?xml version="1.0" ?>
<formation>
  <nom>
    TSIG Etudes
  </nom>
  <durée valeur="1755" />
</formation>
```

Document XML bien formé

- Les éléments contenus et contenant doivent être imbriqués.
  - Tous les éléments fils doivent être contenus dans leur père.
  - Si un document XML est un arbre, un élément est une branche.

Document mal formé :

```
<?xml version="1.0" ?>
<formation>
  <nom>
    TSIG Etudes
  </nom>
  <durée valeur="1755" />
  <module id="100">
  <sequence id="525">
  </module>
  </sequence>
</formation>
```

Document bien formé :

```
<?xml version="1.0" ?>
<formation>
  <nom>
    TSIG Etudes
  </nom>
  <durée valeur="1755" />
  <module id="100">
    <sequence id="110">
    </sequence>
  </module>
</formation>
```

Document XML bien formé

- La valeurs des attributs s'écrit entre guillemets.

Document mal formé :

```
<?xml version="1.0" ?>
<formation>
  <nom>
    TSIG Etudes
  </nom>
  <durée valeur=1755 />
  <module id=100>
  </module>
  <module id=110>
  </module>
</formation>
```

Document bien formé :

```
<?xml version="1.0" ?>
<formation>
  <nom>
    TSIG Etudes
  </nom>
  <durée valeur="1755" />
  <module id="100">
  </module>
  <module id="110">
</module>
</formation>
```

Document XML valide

- Un document XML valide est un document XML bien formé qui possède une DTD (*Document Type Definition*) ou un schéma XML
- La DTD est une série de spécifications déterminant les règles structurelles des documents xml qui lui sont associés.
  - La DTD spécifie :
    - Le nom des balises associées à tous les éléments,
    - Pour chaque balise, les attributs possibles et leur type,
    - Les relations contenant-contenu entre les éléments et leur cardinalité,
    - Les entités (raccourcis) internes et externes

Déclaration des éléments

- La déclaration d 'une nouvelle balise se fait grâce à l 'instruction ELEMENT. La syntaxe est:

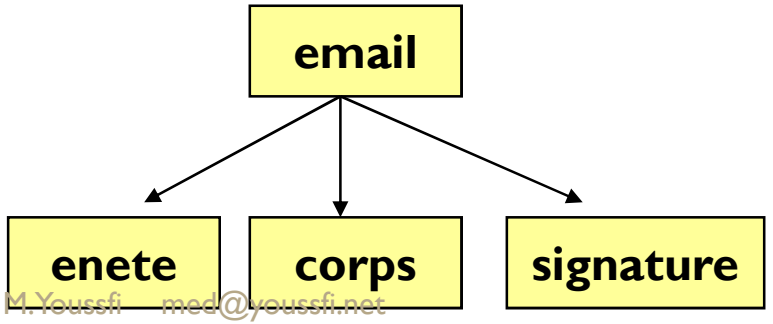
  <!ELEMENT nom contenu>

- Le Contenu peut être choisi parmi cinq modèles:
  - Modèle texte #PCDATA
    - exemple <!ELEMENT paragraphe (#PCDATA)>

    Pour déclarer une balise <paragraphe> qui reçoit du texte

    <paragraphe>Texte</paragraphe>
  - Modèle séquence d 'éléments fils :
    - Une séquence définit les éléments fils autorisés à apparaître dans une balise.
    - Exemple : **<!ELEMENT email (entete, corps, signature)>**

```
<email>
  <entete>….</entete>
  <corps>…..</corps>
  <signature>……</signature>
</email>
```



Déclaration des éléments

- Modèle séquence d 'éléments fils :
  - Chaque nom d 'élément peut être suffixé par un indicateur d 'occurrence.
  - Cet indicateur permet de préciser le nombre d 'instances de l 'élément fils que l 'élément peut contenir. Ces indicateurs sont au nombre de 3.
    - element**?** : Indique que l 'élément peut apparaître zéro ou une fois;
    - element**+** : Indique que l 'élément peut apparaître une ou plusieurs fois;
    - element***** : Indique l 'élément peut apparaître zéro, une ou plusieurs fois.
  - L 'absence de cet indicateur d 'occurrence indique que l 'élément peut apparaître une seule fois au sein de son élément parent.
  - On peut utiliser les parenthèses pour affecter un indicateur d 'occurrence à plusieurs élément

Déclaration des éléments

- Modèle séquence d 'éléments fils : (suite)
  - Exemple:

    <!ELEMNT livre (preface,introduction,(intro-chap,contenu-chap)+)>

  Nous pourrons alors écrire :

  <livre>

  <preface>…..</preface>

  <introduction> …. </introduction>

  <intro-chap> …..</intro-chap>

  <contenu-chap> …. </contenu-chap>

  <intro-chap> …..</intro-chap>

  <contenu-chap> …. </contenu-chap>

  </livre>

  - Le choix le plus libre qu 'une définition de document puisse accorder correspond à la définition suivante :

  <!ELEMENT paragraphe (a | b | c)*>

Déclaration des éléments

- Modèle mixte :
  - Le modèle mixte naît de l 'utilisation conjointe de #PCDATA et d 'éléments fils dans une séquence.
  - Un tel modèle impose d 'utiliser un ordre libre et de disposer #PCDATA en première position. De plus l 'indicateur d 'occurrence doit être *.
  - Exemple : <!ELEMENT p(#PCDATA | u | i)* >
  - En complétant ces déclarations par les deux suivantes :

    <!ELEMENT u (#PCDATA)>

    <!ELEMENT i (#PCDATA)>

  la balise p peut être alors utilisée :

  Dans son livre<u> "la huitième couleur" </u>, Tirry Pratchet nous raconte les aventures du premier touriste du Disque-Monde , le dénommé Deux Fleurs

Déclaration des éléments

- Modèle Vide.
  - Le mot EMPTY permet de définir une balise vide.
  - La balise
 de HTML reproduite en XML en constitue un parfait exemple :

<!ELEMENT livre EMPTY >

  - Une balise vide ne pourra en aucun cas contenir un élément fils.
  - Un élément vide peut avoir des attributs.

<livre id="534" titre="java" datePret="2006-11-12" rendu="oui"/>

- Modèle différent.
  - Ce dernier modèle permet de créer des balises dont le contenu est totalement libre.
  - Pour utiliser ce modèle, on utilise le mot ANY
  - Exemple : **<!ELEMENT disque ANY>**

l 'élément disque peut être utilisé librement.

Déclaration d 'attributs

- La déclaration d 'attributs dans une DTD permet de préciser quels attributs peuvent être associés à un élément donnée.
- Ces attributs reçoivent de plus une valeur par défaut.
- La syntaxe de déclaration d 'un attribut est :

  <!ATTLIST nom-balise nom-attribut type-attribut présence >
- Il est possible de déclarer plusieurs attribut au sein d 'une même balise ATTLIST.
- Exemples:

```
<!ELEMENT personne EMPTY>
<!ATTLIST personne nom CDATA #REQUIRED >
<!ATTLIST personne prenom CDATA #REQUIRED >
```

OU:

```
<!ELEMENT personne EMPTY>
<!ATTLIST personne
                nom CDATA #REQUIRED
                prenom CDATA #REQUIRED>
```

Déclaration d 'attributs

- Les types d'attributs les plus courants sont:
  - Type **CDATA** : signifie que la valeur de l 'attribut doit être une chaîne de caractères;
  - Type **NMTOKEN** : signifie que la valeur de l 'attribut doit être une chaîne de caractères ne contenant pas d'espaces et de caractères spéciaux;
  - Type **ID** : Ce type sert à indiquer que l'attribut en question peut servir d'*identifiant* dans le fichier XML. Deux éléments ne pourront pas posséder le même attribut possédant la même valeur.
  - Type énuméré : une liste de choix possible de type (A|B|C) dans laquelle A, B, C désignent des valeurs que l 'attribut pourra prendre.
    - Une valeur booléenne peut donc être représentée par en XML par les deux déclarations suivantes:

```
<!ELEMENT boolean EMPTY >
<!ATTLIST boolean value (true | false ) 'false'>
```

Déclaration d 'attributs

- Le terme Présence permet de définir comment doit être gérée la valeur de l 'attribut. Il existe 4 types de présences :
  - La valeur par défaut de l 'attribut (comme dans le cas de l 'exemple de l 'élément boolean)
  - **#REQUIRED** Indique que l 'attribut doit être présent dans la balise et que sa valeur doit être obligatoirement spécifiée.
  - **#IMPLIED** Indique que la présence de l 'attribut est facultative;
  - **#FIXED** "valeur " Fixe la valeur de l 'attribut.

Déclaration d ’entités

- Une entité définit un raccourcis vers un contenu qui peut être soit une chaîne de caractères soit le conteu d’un fichier.
- La déclaration générale est : <!ENTITY nom type>
- Exemple d ’entité caractères:

  <!ENTITY **eacute** "é">

  c ’est comme si je définie une constante eacaute qui représente le caractère « é ».
- Exemple d’entité chaîne de caractères:
  - <!ENTITY **ADN** "Acide désoxyribonucléique">
  - Pour utiliser cette entité, on écrit : **&ADN;**
- Exemples d ’entité relative au contenu d ’un fichier:

  <!ENTITY autoexec SYSTEM "file:/c:/autoexec.bat " >

  <!ENTITY notes SYSTEM "../mesnotes/notes.txt" >
- Pour utiliser une entité dans un document XML, on écrit:

  &nom-entité;

Exemple 1

- Un opérateur Télécom fournit périodiquement, à l'opérateur de réglementation des télécoms ANRT un fichier XML qui contient les clients de cet opérateur. Chaque client possède plusieurs abonnements et chaque abonnement reçoit plusieurs factures. Un exemple de fichier XML correspondant à ce problème est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<operateur>
    <client code="1" nom="hassouni">
        <abonnement num="123" type="GSM" dateAb="2006-1-11">
            <facture numFact="432" dateFact="2006-2-11" montant="350.44"
   reglee="oui"/>
            <facture numFact="5342" dateFact="2006-3-11" montant="450" reglee="oui"/>
           <facture numFact="5362" dateFact="2006-4-13" montant="800.54"
   reglee="non"/>
        </abonnement>
        <abonnement num="2345" type="FIXE" dateAb="2006-12-1">
            <facture numFact="5643" dateFact="2007-1-12" montant="299" reglee="oui"/>
            <facture numFact="6432" dateFact="2007-2-12" montant="555" reglee="non"/>
        </abonnement>
    </client>
    <client code="2" nom="abbassi">
        <abonnement num="7543" dateAb="2006-12-1" type="GSM">
            <facture numFact="7658" dateFact="2007-2-12" montant="350.44"
   reglee="oui"/>
            <facture numFact="7846" dateFact="2007-3-12" montant="770" reglee="non"/>
        </abonnement>
    </client>
</operateur>
```

Travail demandé

1. Faire une représentation graphique de l'arbre XML.
2. Ecrire une DTD qui permet de valider ce document XML
3. Créer le fichier XML
4. Ecrire une feuille de style XSL qui permet de transformer le document XML en une page HTML qui permet d'afficher pour chaque client la liste de ses abonnement en affichant le montant total des factures de chaque abonnement

Nom du client:hassouni

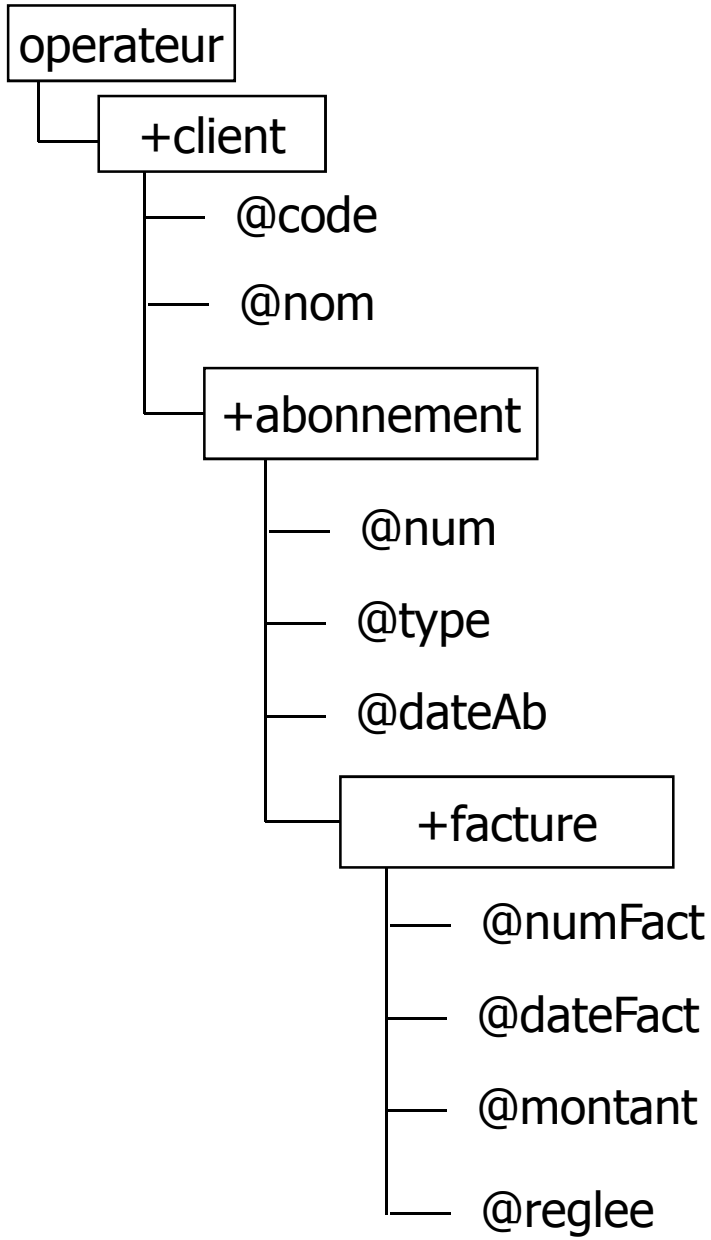
Num Abonnement	Type	Date Abonnement	Montant Total des factures
123	GSM	2006-1-11	1600.98
2345	FIXE	2006-12-1	854

Nom du client:abbassi

Num Abonnement	Type	Date Abonnement	Montant Total des factures
7543	GSM	2006-12-1	1120.44

Corrigé : DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT operateur (client+) >
<!ELEMENT client (abonnement+) >
<!ELEMENT abonnement (facture+) >
<!ELEMENT facture EMPTY >
<!ATTLIST client
        code NMTOKEN #REQUIRED
        nom CDATA #REQUIRED>
<!ATTLIST abonnement
        num NMTOKEN #REQUIRED
        type (GSM|FIXE) 'FIXE'
        dateAb CDATA #REQUIRED>
<!ATTLIST facture
        numFact NMTOKEN #REQUIRED
        dateFact CDATA #REQUIRED
        montant CDATA #REQUIRED
        reglee (oui|non) 'non'>
```



```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
 <xsl:template match="/">
   <html>
     <head></head>
     <body>
       <xsl:for-each select="operateur/client">
         <h3>
           Nom Client : <xsl:value-of select="@nom"/>
         </h3>
       <table border="1" width="80%">
         <tr>
           <th>Num</th><th>Type</th><th>Date</th><th>Total Factures</th>
         </tr>
         <xsl:for-each select="abonnement">
           <tr>
             <td><xsl:value-of select="@num"/></td>
             <td><xsl:value-of select="@type"/></td>
             <td><xsl:value-of select="@dateAb"/></td>
             <td><xsl:value-of select="sum(facture/@montant)"/></td>
           </tr>
         </xsl:for-each>
       </table>
       </xsl:for-each>
     </body>
   </html>
 </xsl:template>
</xsl:stylesheet>
```

Application

Nous souhaitons concevoir un format de fichier XML échangés entre la bourse et ses différents partenaires.
Un exemple de fichier XML manipulé est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<bourse>
   <societe type="banque">
      <codeSociete>SGMB</codeSociete>
      <nomSociete>Société Générale</nomSociete>
      <datIntroduction>2000-11-01</datIntroduction>
      <cotations>
         <cotation num="23" dateCotation="2010-01-01" valeurAction="650"/>
         <cotation num="24" dateCotation="2010-01-02" valeurAction="680"/>
      </cotations>
   </societe>
   <societe type="assurance">
      .....
      .....
   </societe>
</bourse>
```

Exemple d'aplication

Questions :

1. Faire une représentation graphique de l'arbre XML
2. Ecrire la DTD qui permet de déclarer la structure du document XML. (on suppose que le fichier XML contient deux de types de sociétés « banque» et « assurance»)
3. Ecrire un schéma XML équivalent.
4. Ecrire une feuille de style qui permet d'afficher les cotations des sociétés de type « banque».

Cotations des sociétés de type Banque :

- CODE SOCIETE : SGMB
- NOM SOCIETE : Société Générale

Cotations :

Num cotation	Date Cotation	VAL ACTION
23	2010-01-01	650
24	2010-01-02	680
25	2010-01-03	720
Moyenne des cotations		683.3333333333334

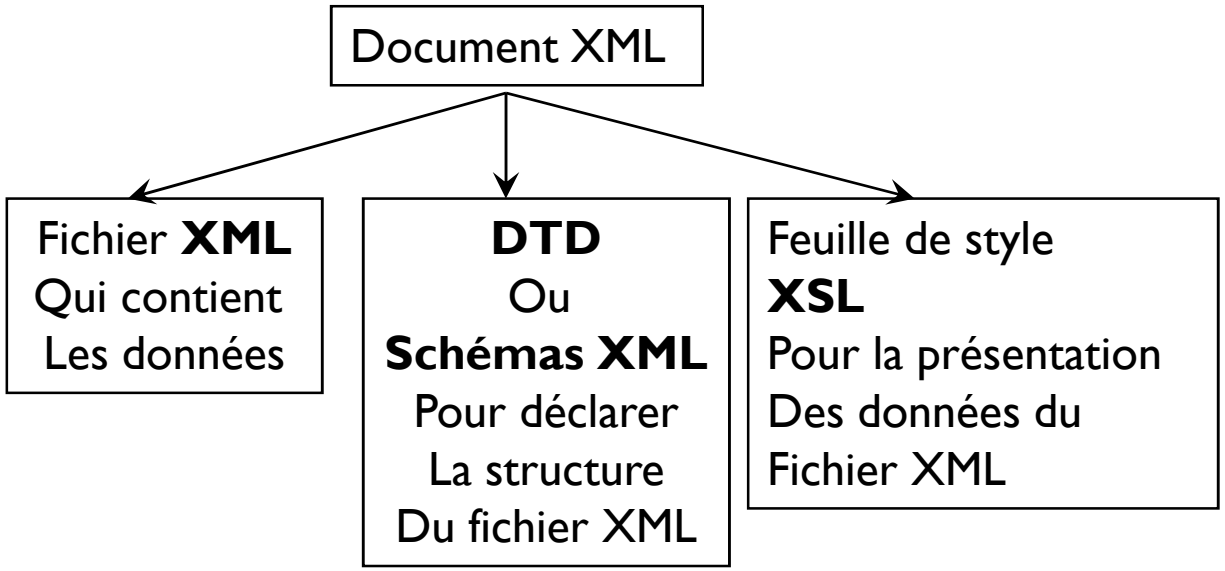
- Nombre total de sociétés :2
- Nombre de sociétés de type banque :1
- Nombre total de sociétés de type assurance :1

Schémas XML

M. Youssfi med@youssfi.net

Structure d'un document XML

Un document XML se compose de 3 fichiers :



- Le fichier XML stocke les données du document sous forme d'un arbre
- DTD (Data Type Definition) ou Schémas XML définit la structure du fichier XML
- La feuille de style définit la mise en forme des données de la feuille xml

Apports des schémas

- Conçu pour pallier aux déficiences pré citées des DTD, XML Schema propose, en plus des fonctionnalités fournies par les DTD, des nouveautés :
  - Le typage des données est introduit, ce qui permet la gestion de booléens, d'entiers, d'intervalles de temps... Il est même possible de créer de nouveaux types à partir de types existants.
  - La notion d'héritage. Les éléments peuvent hériter du contenu et des attributs d'un autre élément.
  - Les indicateurs d'occurrences des éléments peuvent être tout nombre non négatif.
  - Le support des espaces de nom.
  - Les Schémas XML sont des documents XML ce qui signifie d'un parseur XML permet de les manipuler facilement.

Structure de base d'un schéma xml

- Comme tout document XML, un Schema XML commence par un prologue, et a un élément racine.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

<!-- déclarations d'éléments, d'attributs et de types ici -->

</xsd:schema>
```

- L'élément racine est l'élément **xsd:schema**.
- Les éléments du XML schéma sont définis dans l'espace nom **http://www.w3.org/2000/10/XMLSchema** qui est préfixé par xsd.
- Cela signifie que tous le éléments de XML schéma commencent par le préfix xsd (<xsd:element>)

Déclarations d'éléments

- Un élément, dans un schéma, se déclare avec la balise <xsd:element>.
- Par exemple,

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
<xsd:element name="remarque" type="xsd:string">
</xsd:element>
<xsd:element name="contacts" type="typeContacts">
</xsd:element>
  <!-- déclarations de types ici -->
</xsd:schema>
```

- remarque est un élément de type simple
- contacts est un élément de type complexe

Déclarations d'attributs

- Un attribut, dans un schéma, se déclare avec la balise **<xsd:attribute>**.
- Un attribut ne peut être que de type simple
- Exemple:

```
<xsd:element name="contacts" type="typeContacts"/>
<xsd:element name="remarque" type="xsd:string"/>
    <!-- déclarations de types ici -->
<xsd:complexType name="typeContacts">
     <!-- déclarations du modèle de contenu ici -->
   <xsd:attribute name="maj" type="xsd:date" />
</xsd:complexType>
```

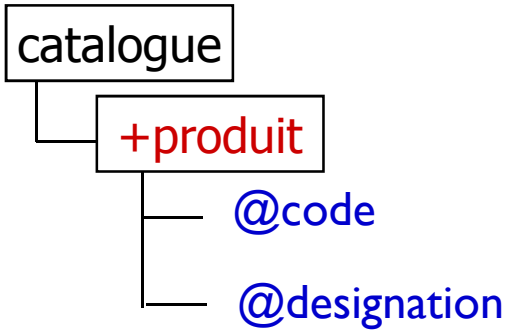
- Ici on déclare que contacts est un élément de type complexe et qu'il possède un attribut maj de type date

Contraintes d'occurrences pour les attribus

- L'élément **attribute** d'un Schema XML peut avoir trois attributs optionnels :
  - **use** : indique le présence , il peut prendre pour valeur required(obligatoire), optional(facultatif) ou prohibited (ne doit pas apparaitre)
  - **default** : pour indiquer la valeur par défaut
  - **fixed** : indique l'attribut est renseigné, la seule valeur que peut prendre l'attribut déclaré est celle de l'attribut **fixed**. Cet attribut permet de "réserver" des noms d'attributs pour une utilisation future, dans le cadre d'une mise à jour du schéma.
- Exemple :

```
<xsd:attribute name="maj" type="xsd:date" use="optional"
    default="2003-10-11" />
```

Premier Exemple schéma xml

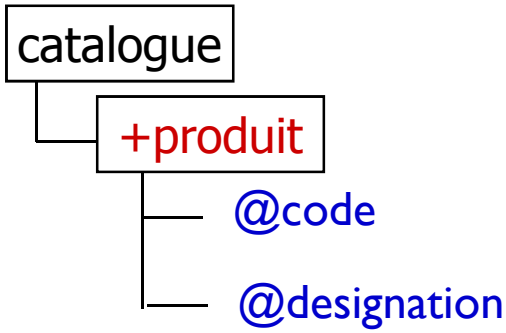



```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:element name="catalogue">
    <xsd:complexType>
       <xsd:sequence>
          <xsd:element name="produit" type="T_PRODUIT" maxOccurs="unbounded"/>
       </xsd:sequence>
    </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="T_PRODUIT">
      <xsd:attribute name="code" type="xsd:int" use="required"/>>
      <xsd:attribute name="designation" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

Fichier XML respectant le schéma précédent




```
<?xml version="1.0" encoding="UTF-8"?>
<catalogue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="file:/C:/m/catalogue.xsd">
   <produit code="1" designation="PC HP 650"/>
   <produit code="2" designation="Imprimante HP 690"/>
</catalogue>
```

Déclaration d'élément ne contenant que du texte avec un (ou plusieurs) attribut(s)

- Un tel élément est de type complexe, car il contient au moins un attribut.
- Afin de spécifier qu'il peut contenir également du texte, on utilise l'attribut **mixed** de l'élément **<xsd:complexType>**.
- Par défaut, **mixed="false"**; il faut dans ce cas forcer **mixed="true"**.
- Par exemple:

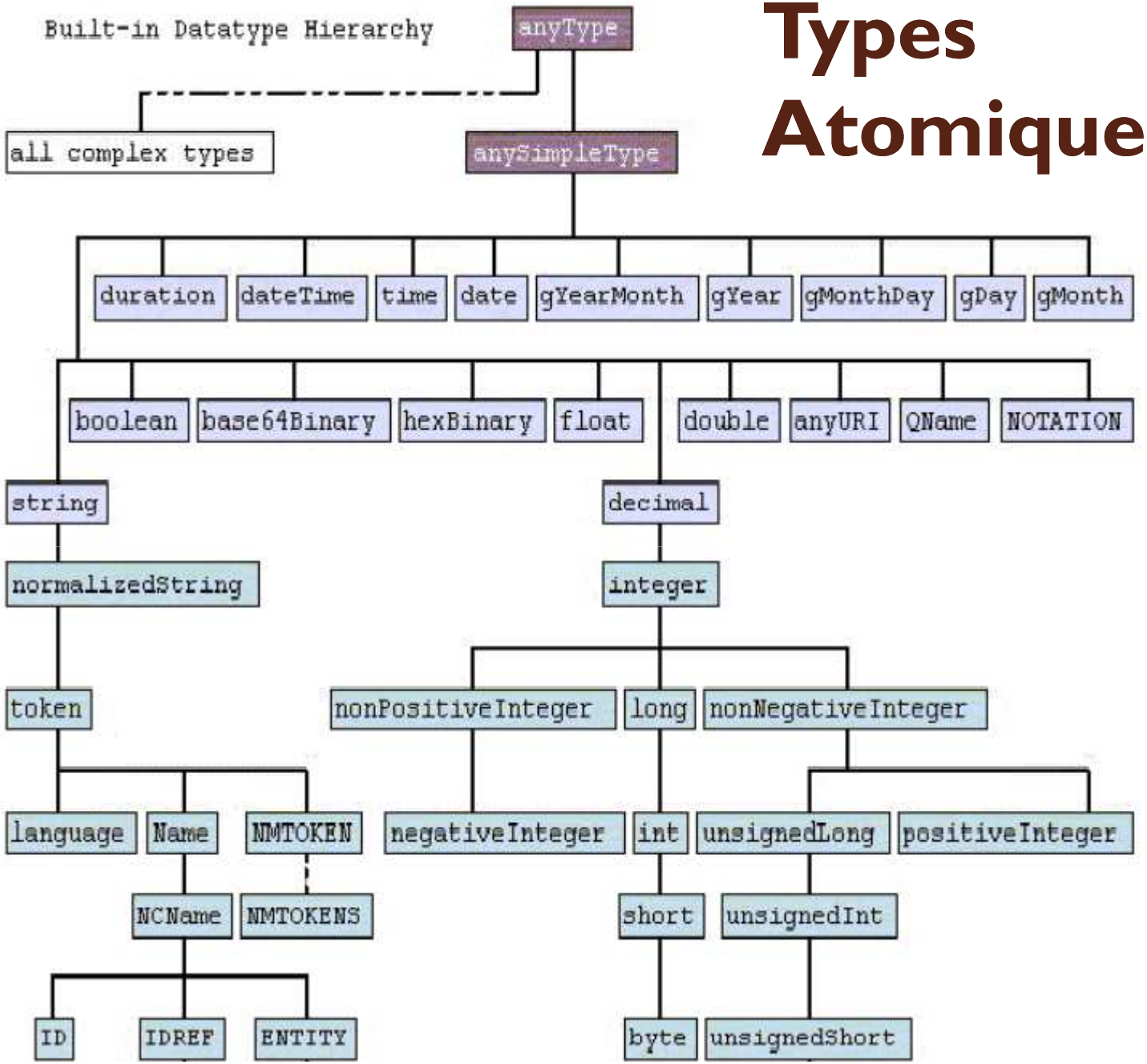
```
<xsd:element name="elt">
 <xsd:complexType mixed="true">
  <xsd:attribute name="attr" type="xsd:string" use="optional" />
 </xsd:complexType>
</xsd:element>
```

Déclaration et référencement

- Il est beaucoup plus avantageux, pour des raisons de clarté, d'ordonner ces déclarations, ainsi qu'on peut le voir sur cet exemple:

```
<xsd:element name="pages" type="xsd:positiveInteger"/>
<xsd:element name="auteur" type="xsd:string"/>
<xsd:element name="livre">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="auteur" />
        <xsd:element ref="pages" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

Types Atomiques



Types simples : Listes

- Les types listes sont des suites de types simples (ou *atomiques*).
- Il est possible de créer une liste personnalisée, par "dérivation" de types existants. Par exemple,

```
<xsd:simpleType name="numeroDeTelephone">
  <xsd:list itemType="xsd:unsignedByte" />
</xsd:simpleType>
```

- Un élément conforme à cette déclaration serait

<telephone>01 44 27 60 11</telephone>.

- Il est également possible d'indiquer des contraintes plus fortes sur les types simples ; ces contraintes s'appellent des "facettes".
- Elles permettent par exemple de limiter la longueur de notre numéro de téléphone à 10 nombres.

Types simples: Unions

- Les listes et les types simples intégrés ne permettent pas de choisir le type de contenu d'un élément.
- On peut désirer, par exemple, qu'un type autorise soit un nombre, soit une chaîne de caractères particuliers.
- Il est possible de le faire à l'aide d'une déclaration d'union.
- Par exemple, sous réserve que le type simple numéroDeTéléphone ait été préalablement défini (voir précédemment), on peut déclarer...

```
<xsd:simpleType name="numeroTelTechnique">
 <xsd:union memberTypes="xsd:string numeroDeTelephone" />
</xsd:simpleType>
```

- Les éléments suivants sont alors des "instances" valides de cette déclaration :

<téléphone>18**</téléphone>**
<téléphone>Pompiers**</téléphone>**

Types complexes

- Un élément de type simple ne peut contenir de sous-élément ou des attributs.
- Un type complexe est un type qui contient soit des éléments fils, ou des attributs ou les deux.
- On peut alors déclarer,
  - Des séquences d'éléments,
  - Des types de choix

Types complexes: séquences d'éléments:

- Nous savons déjà comment, dans une DTD, nous pouvons déclarer un élément comme pouvant contenir une suite de sous-éléments, dans un ordre déterminé.
- Il est bien sûr possible de faire de même avec un schéma.
- On utilise pour ce faire l'élément **xsd:sequence**, qui reproduit l'opérateur, du langage DTD. Exemple:

```
<xsd:complexType>
 <xsd:sequence>
  <xsd:element name="nom" type="xsd:string" />
  <xsd:element name="prénom" type="xsd:string" />
  <xsd:element name="dateDeNaissance" type="xsd:date"/>
  <xsd:element name="adresse" type="xsd:string" />
  <xsd:element name="email" type="xsd:string" />
  <xsd:element name="tel" type="numéroDeTéléphone" />
 </xsd:sequence>
</xsd:complexType>
```

Types Complexes : Choix d'élément

- On peut vouloir modifier la déclaration de type précédente en stipulant qu'on doive indiquer soit l'adresse d'une personne, soit son adresse électronique. Pour cela, il suffit d'utiliser un élément **xsd:choice** :

```
<xsd:complexType name="typePersonne">
  <sequence>
   <xsd:element name="nom" type="xsd:string" />
   <xsd:element name="prénom" type="xsd:string" />
   <xsd:element name="dateDeNaissance" type="xsd:date" />
   <xsd:choice>
     <xsd:element name="adresse" type="xsd:string" />
     <xsd:element name="email" type="xsd:string" />
   </xsd:choice>
   <xsd:element name="tel" type="numéroDeTéléphone" />
  </sequence>
</xsd:complexType>
```

Types Complexe : Elément All

- L'élément All indique que les éléments enfants doivent apparaître une fois (ou pas du tout), et dans n'importe quel ordre.
- Cet élément **xsd:all** doit être un enfant direct de l'élément **xsd:complexType**.
- Par exemple

```
<xsd:complexType>
  <xsd:all>
    <xsd:element name="nom" type="xsd:string" />
    <xsd:element name="prénom" type="xsd:string" />
    <xsd:element name="dateDeNaissance" type="xsd:date" />
    <xsd:element name="adresse" type="xsd:string" />
    <xsd:element name="adresseElectronique" type="xsd:string" />
    <xsd:element name="téléphone" type="numéroDeTéléphone" />
  </xsd:all>
</xsd:complexType>
```

Indicateur d'occurrences

- Pendant la déclaration d'un élément, on peut indiquer le nombre minimum et le nombre maximum de fois qu'il doit apparaître
- On utilise pour cela les attributs **minOccurs** et **maxOccurs**:
- **minOccurs** prend par défaut la valeur 1 et peut prendre les autres valeurs positives
- **maxOccurs** prend par défaut la valeur 1 et peut prendre les autres valeurs positive ou unbounded (infini).

Dérivation

- Les types simples et complexes permettent déjà de faire plus de choses que les déclarations dans le langage DTD.
- Il est possible de raffiner leur déclaration de telle manière qu'ils soient
  - une "restriction"
  - ou une extension d'un type déjà existant, en vue de préciser un peu plus leur forme.
- Nous allons nous limiter dans ce cours d'initiation à la restriction des types simples.

Dérivation : Restriction de types

- Une "facette" permet de placer une contrainte sur l'ensemble des valeurs que peut prendre un type de base.
- Par exemple, on peut souhaiter créer un type simple, appelé MonEntier, limité aux valeurs comprises entre 0 et 99 inclus.
- On dérive ce type à partir du type simple prédéfini **nonNegativeInteger**, en utilisant la facette **maxExclusive**.

```
<xsd:simpleType name="monEntier">
    <xsd:restriction base="nonNegativeInteger
  ">
      <xsd:maxExclusive value="100" />
    </xsd:restriction>
</xsd:simpleType>
```

Dérivation : Restriction de types

- Il existe un nombre important de facettes qui permettent de :
  - fixer, restreindre ou augmenter la longueur minimale ou maximale d'un type simple
  - énumérer toutes les valeurs possibles d'un type
  - prendre en compte des expressions régulières
  - fixer la valeur minimale ou maximale d'un type (voir l'exemple ci-dessus)
  - fixer la précision du type...

Dérivation : Exemples de facettes

- Limiter les différentes valeurs possible d'un type:

```
<xsd:attribute name="jour" type="jourSemaine" use="req
  uired" />
<xsd:simpleType name="jourSemaine">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="lundi" />
      <xsd:enumeration value="mardi" />
      <xsd:enumeration value="mercredi" />
      <xsd:enumeration value="jeudi" />
      <xsd:enumeration value="vendredi" />
      <xsd:enumeration value="samedi" />
      <xsd:enumeration value="dimanche" />
    </xsd:restriction>
</xsd:simpleType>
```

Dérivation : Exemples de facettes

- Limiter la longueur d'une chaîne de caractères:

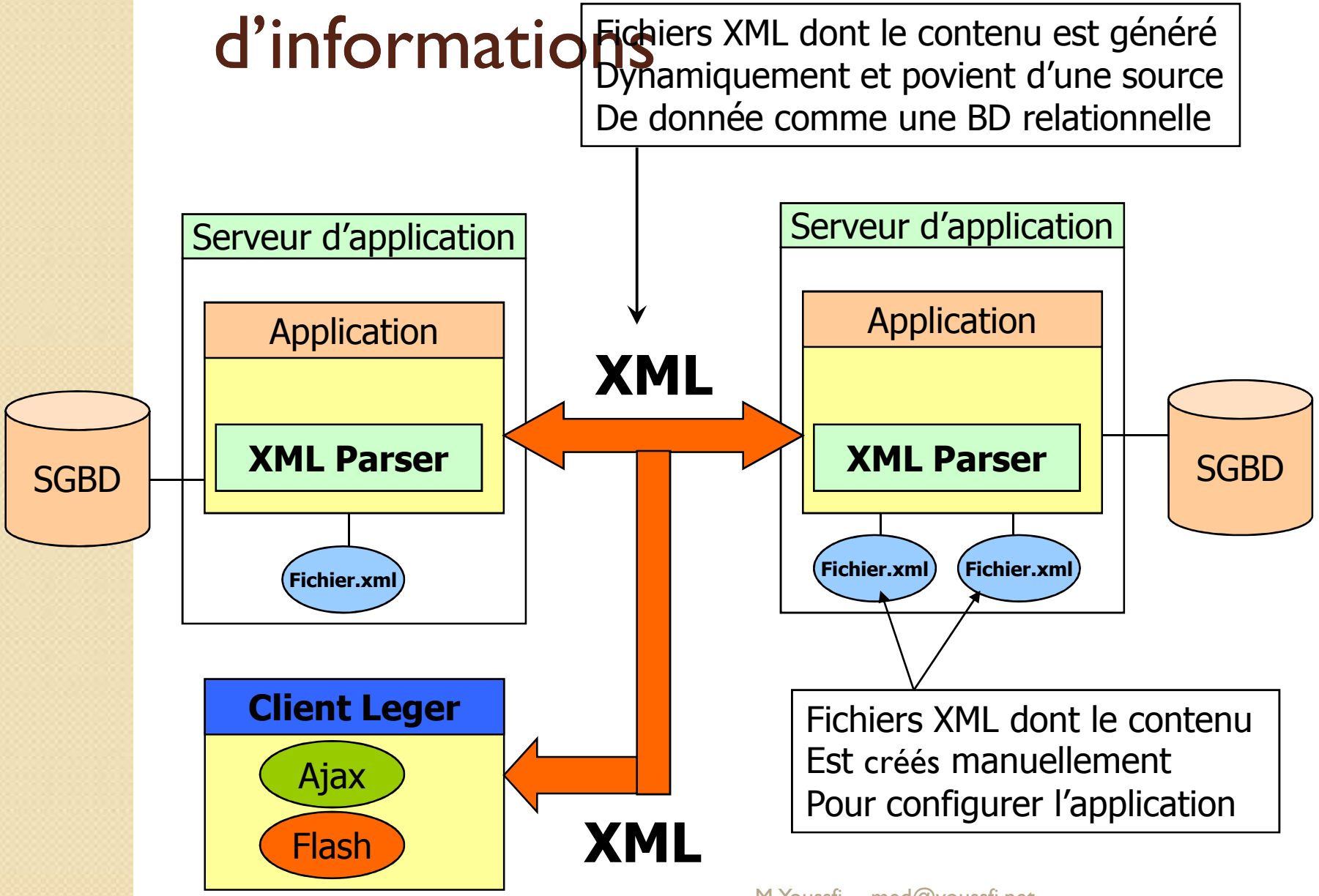
```
<xsd:simpleType name="monType">
  <xsd:restriction base="xsd:string">
    <xsd:length value="21" />
  </xsd:restriction>
</xsd:simpleType>
```

Dérivation : Exemples de facettes

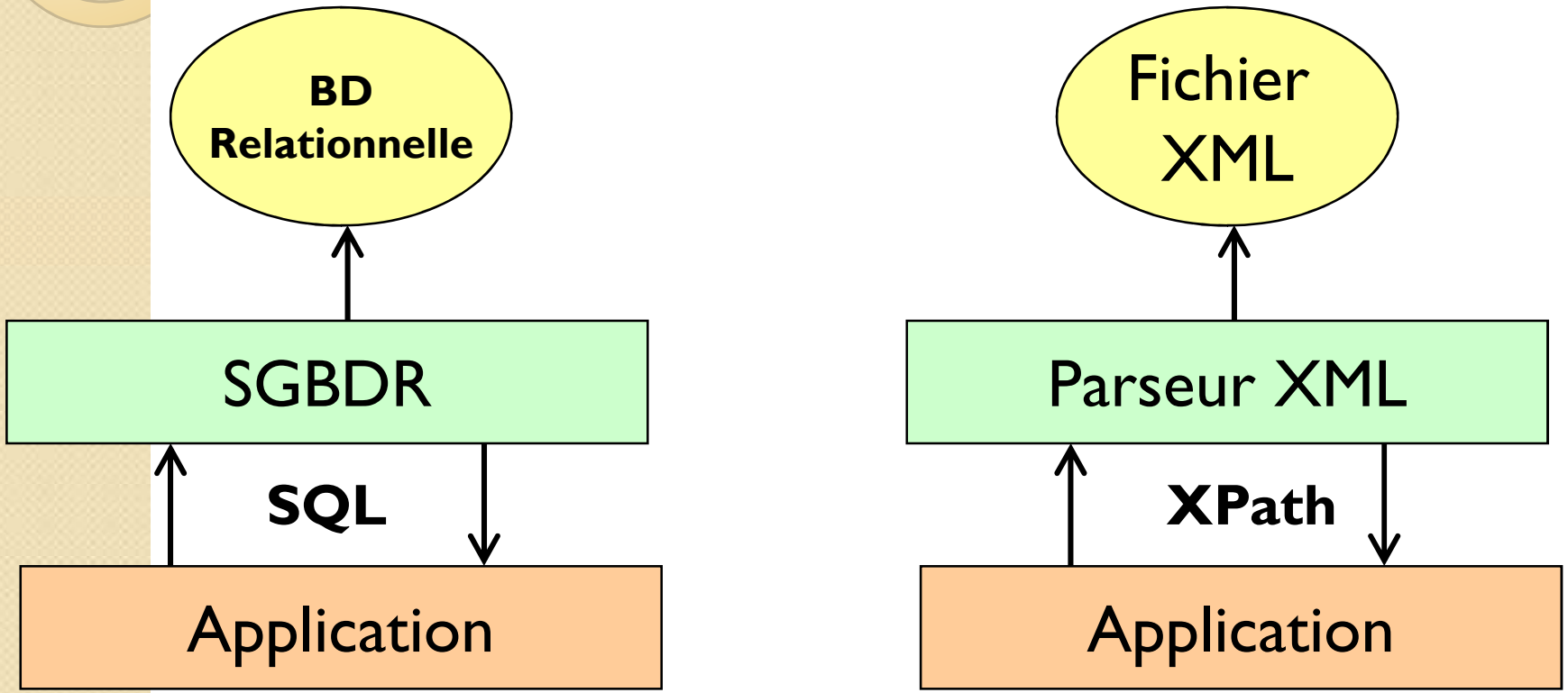
- Expressions régulières pour une adresse email:

```
<xsd:simpleType name="typeAdresseElectronique">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="(.)+@(.)+" />
    </xsd:restriction>
</xsd:simpleType>
```

XML est au cœur des systèmes d'informations

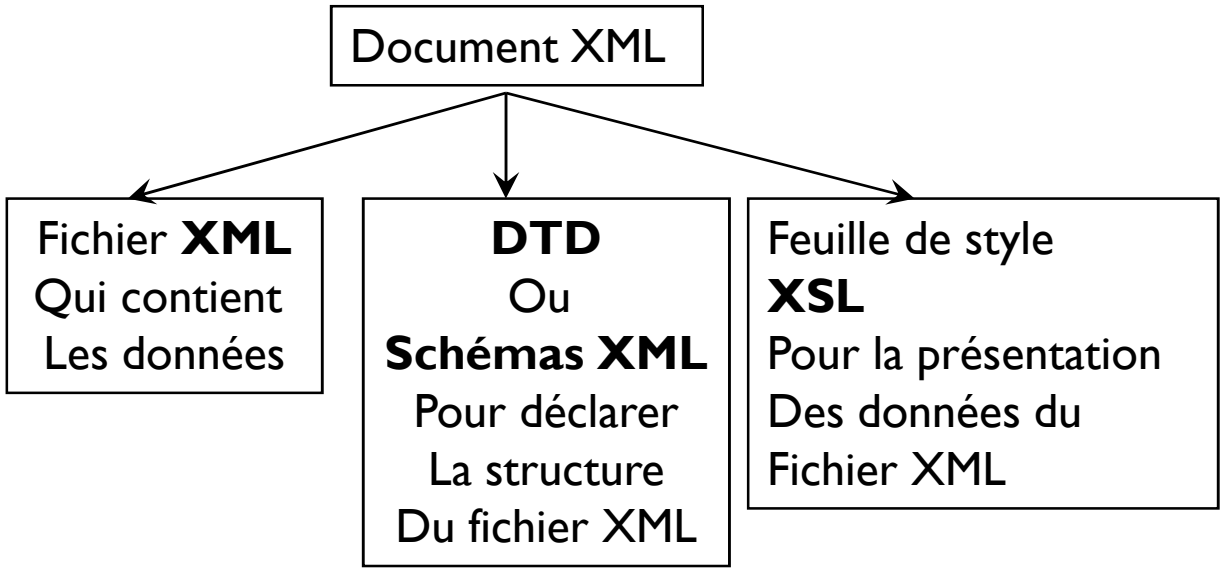


Correspondance entre XML et Bases de données relationnelles



Structure d'un document XML

Un document XML se compose de 3 fichiers :



- Le fichier XML stocke les données du document sous forme d'un arbre
- DTD (Data Type Definition) ou Schémas XML définit la structure du fichier XML
- La feuille de style définit la mise en forme des données de la feuille xml

Application

- Un service d'envoi des mandats établit mensuellement un rapport qui contient les mandats envoyés. Chaque mandat concerne un expéditeur et un destinataire. L'expéditeur est défini par son cin, son nom, son prénom et sa ville. Le destinataire est défini également par son cin, son nom, son prénom et sa ville.

Application

- Un exemple de fichier XML correspondant à ce problème est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<rapport>
   <mandat num="7124536" date="2007-1-1" montant="1000" etat="reçu">
      <expediteur cin="A123245" nom="slimani" prenom="youssef" ville="casa"/>
      <destinataire cin="P98654" nom="hassouni" prenom="laila" ville="fes"/>
   </mandat>
   <mandat num="7124537" date="2007-1-1" montant="3200" etat="non reçu">
      <expediteur cin="M123245" nom="alaoui" prenom="mohamed" ville="marrakech"/>
      <destinataire cin="M92654" nom="alaoui" prenom="imane" ville="casa"/>
   </mandat>
   <mandat num="7124538" date="2007-1-2" montant="500" etat="reçu">
      <expediteur cin="H123222" nom="qasmi" prenom="slim" ville="oujda"/>
      <destinataire cin="B91154" nom="qasmi" prenom="hassan" ville="rabat"/>
   </mandat>
</rapport>
```

Travail à faire

- Faire une représentation graphique de l'arbre XML
- Ecrire une DTD qui permet de déclarer la structure du document XML
- Ecrire le schéma XML qui permet de déclarer la structure du document XML
- Créer le fichier XML valide respectant ce schéma.
- Ecrire une feuille de style XSL qui permet de transformer le document XML en document HTML suivant :

Num Mandat	Date	Expéditeur	Destinataire	Etat	Montant
7124536	2007-1-1	• CIN: A123245 • Nom:slimani	• CIN: P98654 • Nom:hassouni	reçu	1000
7124537	2007-1-1	• CIN: M123245 • Nom:alaoui	• CIN: M92654 • Nom:alaoui	non reçu	3200
7124538	2007-1-2	• CIN: H123222 • Nom:qasmi	• CIN: B91154 • Nom:qasmi	reçu	500
				Total des mandats	4700
				Total des mandats reçus	1500
				Total des mandats non reçus	3200

XSL-XPATH

M.Youssfi med@youssfi.net

Introduction à XSL

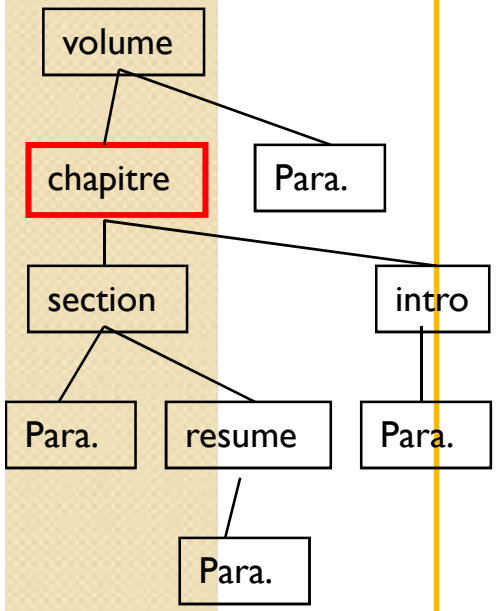
- XSL signifie e*Xtensive Stylesheet Langage*, ou langage extensible de feuille de style.
- XSL est un langage de feuille de style. Il est aussi un très puissant manipulateur d'éléments. Il permet de transformer un document XML source en un autre, permettant ainsi, à l'extrême, d'en bouleverser la structure.
- XSL est un outil privilégié de production de fichiers HTML à partir de sources XML.
- Un fichier XSL étant un fichier XML, il doit respecter les normes de syntaxe de ce format.

Structure d'un document XSL

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
 <xsl:template match="/">


 </xsl:template>
</xsl:stylesheet>
```

Sélection d'éléments et d'attributs



- Sélection d'un élément:
  - Syntaxe: **<xsl:value-of select="nom_element" />**
  - L'opérateur / permet de définir le chemin d'accès aux éléments à sélectionner
    - Par exemple, **section/paragraphe** sélectionne les éléments **section** du noeud courant et pour chaque élément **section**, sélectionne les éléments **paragraphe** qu'il contient.
    - En d'autres termes, cette expression sélectionne les petits-fils **paragraphe** du noeud courant qui ont pour père un noeud **section**.
  - Un nom d'élément peut être remplacé par * dans une expression.
    - Par exemple, ***/paragraphe** sélectionne tous les petits-fils **paragraphe** quel que soit leur père.
  - L'utilisation de // permet d'appliquer la recherche aux descendants et non pas seulement aux fils directs. Par exemple, **section//paragraphe** sélectionne tous les éléments **paragraphe** descendant d'un élément **section** fils direct du noeud courant.
  - Le caractère . sélectionne le noeud courant. Par exemple, **.//paragraphe** sélectionne tous les descendants **paragraphe** du noeud courant.
  - La chaîne .. sélectionne le père du noeud courant. Par exemple, **../paragraphe** sélectionne tous les noeuds **paragraphe** frères du noeud courant.

Sélection d'un élément : Utilisation du DOM

- Il est également possible de "naviguer" dans les branches de l'arborescence du document XML, en utilisant les ressources du DOM.
- Différents types de syntaxes sont possibles, fondées sur une expression de la forme **Element[Expression]**.
- Par exemple :
  - **elt[i]** : où **i** est un nombre entier désigne le **i**-ème descendant direct d'un même parent ayant le nom indiqué.
    - Par exemple, **paragraphe[3]** désigne le *3*ème enfant de l'élément courant, portant le nom **paragraphe**. *Attention*, la numérotation commence à 1 et non à 0.
  - **elt[position()>i]** : où **i** est un nombre entier sélectionne tous les éléments précédés d'au moins **i** éléments de même nom comme descendants du même parent.
    - Par exemple, **paragraphe[position()>5]** sélectionne tous les éléments **paragraphe** dont le numéro d'ordre est strictement supérieur à 5.
  - **elt[position() mod 2 =1]** : Sélectionne tout élément qui est un descendant impair.

Sélection d'un élément : Utilisation du DOM

- **elt[souselt]** : Sélectionne tout élément **elt** qui a au moins un descendant **souselt** (à ne pas confondre avec **elt/souselt**, qui sélectionne tout élément **souselt** ayant pour parent **elt**...).
- **elt[.="valeur"]** : Sélectionne tout élément **elt** dont la valeur est valeur.
- **elt[first-of-any()]** : Sélectionne le premier élément **elt** fils de l'élément courant.
- **elt[last-of-any()]** : Sélectionne le dernier élément **elt** fils de l'élément courant.

Sélection d'un Attribut

- Les attributs d'un élément sont sélectionnés en faisant précéder leur nom par le caractère @. Les règles relatives à la sélection des éléments s'appliquent également aux attributs :
  - section[@titre] : sélectionne les éléments section qui ont un attribut titre.
  - section[@titre='Introduction'] : sélectionne les éléments section dont l'attribut titre a pour valeur Introduction.
- Si l'on veut *afficher* le contenu de l'attribut, on le fait précéder du caractère @.
  - Par exemple, <xsl:value-of select="paragraphe/@titre"/> permet l'affichage du titre de l'élément paragraphe fils de l'élément courant.

Opérateurs logiques

- Les opérateurs logiques **not()**, **and** et **or** peuvent être utilisés, comme par exemple
  - **section[not(@titre)]** sélectionne les éléments **section** qui n'ont pas d'attribut **titre**.
  - Attention : lorsque, dans la DTD par exemple, l'attribut est défini comme ayant une valeur par défaut, même s'il n'est pas explicité dans le document XML, il est considéré comme existant.

Eléments de XSLT

- **<xsl:value-of>**
  - **Cet élément permet d'insérer la valeur d'un nœud dans la transformation.**
    - **Ce nœud est évalué en fonction d'une expression.**
    - **Cette expression peut correspondre à un élément, à un attribut ou à tout autre nœud contenant une valeur.**
    - **L'utilisation de cet élément est de la forme :**
  - **<xsl:value-of select="expression" disable-output-escaping="yes | no" />**
    - **La valeur de select est évaluée et c'est cette évaluation qui sera insérée dans la transformation.**
    - **disable-output-escaping agit sur la transformation des caractères.**
      - **Dans le cas où sa valeur est yes, la chaîne < est insérée dans la transformation en tant que signe <, ce qui peut entraîner des erreurs.**
      - **Dans le cas contraire, la chaîne < est insérée telle quelle dans la transformation.**

Eléments de XSLT

- **<xsl:element>**
  - **Cet élément insère un nouvel élément dans la transformation. Le nom de l'élément est donné par l'attribut name. L'utilisation de cet élément est de la forme :**
  - `<xsl:element name="nomelement" use-attribute-sets="jeuattr"> </xsl:element>`
    - **name correspond au nom de l'élément à créer.**
    - **use-attribute-sets correspond au jeu d'attributs à associer à l'élément créé. Par exemple :**
      - `<xsl:element name="p">`
        - `<xsl:value-of select="texte" />`
      - `</xsl:element>`
    - **permet de créer dans le fichier HTML un élément de paragraphe renfermant le contenu de l'élément texte du document XML. texte **

Eléments de XSLT

- **<xsl:attribute>**
  - Cet élément définit un attribut et l'ajoute à l'élément résultat de la transformation. L'utilisation de cet élément est de la forme :
  - <xsl:attribute name="nom">valeur</xsl:attribute>
    - name correspond au nom de l'attribut à ajouter dans le contexte courant. valeur correspond à la valeur à lui donner. Par exemple :
    - `<image>`
      - `<xsl:attribute name="src">test.gif</xsl:attribute>`
    - `</image>`
    - permet d'ajouter à l'élément image l'attribut src et de lui affecter la valeur test.gif, ce qui a pour effet de produire en sortie l'élément suivant : <image src="test.gif"></image>
    - On aurait pu, de manière tout à fait équivalente, écrire
    - <xsl:element name="image">
      - <xsl:attribute name="src">test.gif</xsl:attribute>
    - </xsl:element>

Eléments de XSLT

- **<xsl:for-each>**
  - **Crée une boucle dans laquelle sont appliquées des transformations.**
  - **Son utilisation est de la forme :**
  - **<xsl:for-each select="jeunoeud">**
    - **…..**
  - **</xsl:for-each>**
    - **select correspond au jeu de nœuds devant être parcouru par la boucle. Exemple d'utilisation :**

```
<ul>
  <xsl:for-each select="item">
    <li><xsl:value-of select="." /></li>
  </xsl:for-each>
</ul>
```

Eléments de XSLT

- **<xsl:apply-templates>**
  - **Permet d'appliquer les transformation à un jeu d'éléments en utilisant les templates définis.**
  - **Exemple:**

```
<xsl:template match="/">
   <xsl:apply-templates select="//personne">
   </xsl:apply-templates>
</xsl:template>
 <xsl:template match="//personne">
   .......
 </xsl:template>
```

Eléments de XSLT

- **<xsl:sort>**
  - Cet élément permet d'effectuer un tri sur un jeu de nœuds.
  - Il doit être placé soit dans un élément <xsl:for-each> soit dans un élément <xsl:apply-templates>.
  - L'utilisation de cet élément est de la forme :
    - <xsl:sort select="noeud" data-type="text | number | elt" order="ascending | descending" lang="nmtoken" case-order="upper-first | lower-first" />
      - select permet de spécifier un nœud comme clé de tri.
      - data-type correspond au type des données à trier. Dans le cas où le type est number, les données sont converties puis triés.
      - order correspond à l'ordre de tri. Cet attribut vaut ascending ou descending.
      - lang spécifie quel jeu de caractères utiliser pour le tri ; par défaut, il est déterminé en fonction des paramètres système.
      - case-order indique si le tri a lieu sur les majuscules ou minuscules en premier.
  - Par exemple :
  -
    <xsl:for-each select="livre">
    <xsl:sort select="auteur" order="descending" />
    <xsl:value-of select="auteur" />
  - <xsl:value-of select="titre" />
    </xsl:for-each>

  - Dans cet exemple, la liste des livres est classée dans l'ordre alphabétique décroissant des noms d'auteur.y

Eléments de XSLT

- **<xsl:if>**
  - Cet élément permet la fragmentation du modèle dans certaines conditions. Il est possible de tester
    - La présence d'un attribut,
    - La présence d'un élément,
    - De savoir si un élément est bien le fils d'un autre,
    - De tester les valeurs des éléments et attributs.
  - L'utilisation de cet élément est de la forme :
    - <xsl:if test="condition">
      - ……Action
    - </xsl:if>
  - Exemple d'utilisation:

```
<ul>
    <xsl:for-each select="livre">
      <li>
        <b><xsl:value-of select="auteur" /><br /></b>
        <xsl:value-of select="titre" />
      </li>
      <xsl:if test="@langue='fr'">
        <li>Ce Livre est en français</li>
      </xsl:if>

    </xsl:for-each>
</ul>
```

Eléments de XSLT

- **<xsl:choose>**
  - Cet élément permet de définir une liste de choix et d'affecter à chaque choix une transformation différente.
  - Chaque choix est défini par un élément **<xsl:when>**
  - et un traitement par défaut peut être spécifié grâce à l'élément **<xsl:otherwise>.**
  - Exemple d'utilisation :

```
<ul>
    <xsl:for-each select="livre">
     <li>
      <b><xsl:value-of select="auteur" /><br /></b>
      <xsl:value-of select="titre" />
     <xsl:choose>
       <xsl:when test="@langue='francais'">
         Ce livre est en français.
       </xsl:when>
      <xsl:when test="@langue='anglais'">
         Ce livre est en anglais.
       </xsl:when>
      <xsl:otherwise>
       Ce livre est dans une langue non répertoriée.
      </xsl:otherwise>
     </xsl:choose>
     </li>
    </xsl:for-each>
  </ul>
```

Eléments de XSLT

- **<xsl:variable>**
  - **L'élément <xsl:variable> sert à créer les variables dans XSLT. Il possède les attributs suivants :**
    - **name : cet attribut est obligatoire. Il spécifie le nom de la variable.**
    - **select : expression XPath qui spécifie la valeur de la variable.**
  - **Par exemple :**
    - ```
      <xsl:variable name="nombre_livres" select="255" />
      <xsl:variable name="auteur" select="'Victor Hugo'" />
      <xsl:variable name="nb" select="livre/tome/@page" />
      ```
    - **On notera la présence des guillemets imbriqués quand il s'agit d'affecter une chaîne de caractères à une variable.**
    - **La portée d'une variable est limitée aux éléments-frères et à leurs descendants. Par conséquent, si une variable est déclarée dans une boucle xsl:for-each ou un élément xsl:choose ou xsl:if, on ne peut s'en servir en-dehors de cet élément.**
    - **Une variable est appelée en étant précédée du caractère $:**
      - **<xsl:value-of select="$nb" />.**
  - **On peut utiliser une variable pour éviter la frappe répétitive d'une chaîne de caractères, et pour faciliter la mise à jour de la feuille de style.**
      ```

Eléments de XSLT

- **<xsl:call-template>**
  - L'élément <xsl:template> peut être appelé indépendamment d'une sélection d'un nœud.
  - Pour cela, il faut renseigner l'attribut name, et l'appeler à l'aide de l'élément <xsl:call-template>.
  - Par exemple
    - <xsl:template name="separateur">
      <hr />
      <hr />
      </xsl:template>
    - Il suffit alors de l'appeler avec
      - <xsl:call-template name="separateur"/>

Eléments de XSLT

- **<xsl:param> et <xsl:with-param>**
  - Un paramètre est créé avec l'élément <xsl:param>, et passé à un modèle avec l'élément <xsl:with-param>.
  - Les deux ont deux attributs :
    - name, obligatoire, qui donne un nom au paramètre ;
    - select, une expression XPath facultative permettant de donner une valeur par défaut au paramètre.
  - Par exemple, on peut imaginer un template permettant d'évaluer le résultat d'une expression polynômiale :
  - <xsl:template name="polynome">
    <xsl:param name="x"></xsl:param>
    <xsl:value-of select="2*$x*$x+(-5)*$x+2" />
    </xsl:template>
  - Ce template peut être appelé comme suit:
  - <xsl:call-template name="polynome">
    <xsl:with-param name="x" select="3.4"></xsl:with-param>
    </xsl:call-template>
  - permet d'afficher le résultat de 2*3.4^2-5*3.4+2.

XPATH

- Comme son nom l'indique, XPath est une spécification fondée sur l'utilisation de chemin d'accès permettant de se déplacer au sein du document XML.
- Le XPath établit un arbre de noeuds correspondant au document XML.
- La syntaxe de base du XPath est fondée sur l'utilisation d'expressions.
- Chaque évaluation d'expression dépend du contexte courant.
- Une des expressions les plus importantes dans le standard XPath est **le *chemin de localisation***.
- Cette expression sélectionne un ensemble de nœuds à partir d'un nœud contextuel.

XPATH : Chemin de localisation

- La syntaxe de composition d'un chemin de localisation peut être de type abrégé ou non abrégé.
- Un chemin de localisation est composé de trois parties :
  - un *axe*, définissant le sens de la relation entre le nœud courant et le jeu de nœuds à localiser;
  - un nœud spécifiant le type de nœud à localiser;
  - 0 à n *prédicats* permettant d'affiner la recherche sur le jeu de nœuds à récupérer.
- Par exemple, dans le chemin **child::section[position()=1]**,
  - **child** est le nom de l'axe,
  - **section** le type de nœud à localiser (élément ou attribut)
  - **[position()=1]** est un prédicat. Les doubles **::** sont obligatoires.
- Dans cet exemple, on cherche dans le nœud courant, un nœud section qui est le premier nœud de son type.

XPATH : Axes

- **child** : contient les enfants directs du nœud contextuel.
- **descendant** : contient les descendants du nœud contextuel. Un descendant peut être un enfant, un petit-enfant...
- **parent** : contient le parent du nœud contextuel, s'il y en a un.
- **ancestor** : contient les ancêtres du nœud contextuel. Cela comprend son père, le père de son père...
- **following** : contient tous les nœuds du même nom que le nœud contextuel situés après le nœud contextuel dans l'ordre du document,
- **preceding** : contient tous les nœuds du même nom que le nœud contextuel situés avant lui dans l'ordre du document

XPATH : Axes

- **attribute** : contient les attributs du nœud contextuel ;
- **self** : contient seulement le nœud contextuel.
- **descendant-or-self** : contient le nœud contextuel et ses descendants.
- **ancestor-or-self** : contient le nœud contextuel et ses ancêtres. Cet axe contiendra toujours le nœud racine.

XPATH : Prédicats

- **Chaque expression est évaluée et le résultat est un booléen.**
- **Par exemple, section[3] est équivalent à section[position()=3].**
  - **Ces deux expressions sont équivalentes : chacune d'entre elles produit un booléen.**
  - **Dans le premier cas, il n'y a pas de test, on sélectionne simplement le troisième élément, l'expression est obligatoirement vraie.**
  - **Dans le second cas, un test est effectué par rapport à la position de l'élément section ; lorsque la position sera égale à 3, l'expression sera vraie.**

XPATH : Exemples

- child::*para* : sélectionne l'élément *para* enfant du nœud contextuel. (abréviation : para)
- child::* : sélectionne tous les éléments enfants du nœud contextuel. (abréviation : *)
- child::text() : sélectionne tous les nœuds de type texte du nœud contextuel. (abréviation : text())
- child::node() : sélectionne tous les enfants du nœud contextuel, quel que soit leur type.
- attribute::*name* : sélectionne tous les attributs *name* du nœud contextuel. (abréviation : @name)
- attribute::* : sélectionne tous les attributs du nœud contextuel. (abréviation : @*)
- descendant::*para* : sélectionne tous les descendants *para* du nœud contextuel.
- ancestor::*div* : sélectionne tous les ancêtres *div* du nœud contextuel.
- ancestor-or-self::*div* : sélectionne tous les ancêtres *div* du nœud contextuel et le nœud contextuel lui-même si c'est un *div*.
- //equipe[not(.=preceding::equipe)] : sélectionne tous les éléments equipe dont la valeur est différente des équipes précédentes.

Xpath: Fonctions de base

- **Manipulation de nœuds**
  - *Fonctions retournant un nombre :*
    - **last()** : retourne un nombre égal à l'index du dernier nœud dans le contexte courant.
    - **position()** : retourne un nombre égal à la position du nœud dans le contexte courant.
    - **id(objet)**, permet de sélectionner les éléments par leur identifiant.

Xpath: Fonctions de base

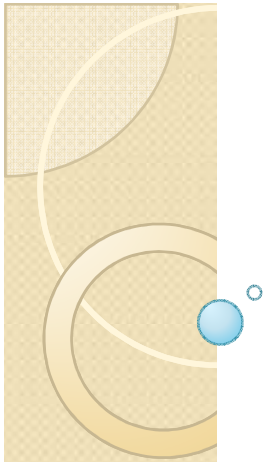
- **Manipulation de chaînes de caractères**
  - **string(noeud?)** : cette fonction convertit un objet en chaîne de caractères.
- **concat(chaine1, chaine2, chaine*)** : retourne une chaîne résultant de la concaténation des arguments
- **string-length(chaine?)** : cette fonction retourne le nombre de caractères de la chaîne. Dans le cas où l'argument est omis, la valeur retournée est égale à la longueur de la valeur textuelle du nœud courant

Xpath: Fonctions de base

- **Manipulation de booléens**
  - Outre la fonction logique **not()**, ainsi que les fonctions **true()** et **false()**, une fonction utile est
  - **lang(chaine)**. Elle teste l'argument **chaine** par rapport à l'attribut **xml:lang** du nœud contextuel ou de son plus proche ancêtre dans le cas où le nœud contextuel ne contient pas d'attribut de ce type.
  - La fonction retourne **true** si l'argument est bien la langue utilisée ou si la langue utilisée est un sous-langage de l'argument (par exemple, **en//us**). Sinon elle retourne **false**.
- **Manipulation de nombres**
  - **floor(nombre)** : retourne le plus grand entier inférieur à l'argument passé à la fonction.
  - **ceiling(nombre)** : retourne le plus petit entier supérieur à l'argument passé à la fonction.
  - **round(nombre)** : retourne l'entier le plus proche de l'argument passé à la fonction.

APPLICATIONS DE XML

M.Youssfi med@youssfi.net



XQuery

Par M.Youssfi

med@youssfi.net

XQuery, un langage de requête

- XQuery, langage non XML, permet de traiter des ressources XML (fichier ou SGBD-XML) pour obtenir des structures XML. C'est un langage fonctionnel typé et basé sur la manipulation de liste de noeuds XML.

Requête XQuery

- XQuery peut être vu comme un langage fonctionnel manipulant des séquences d'items.
- Une requête XQuery est donc une expression portant sur des items (séquences d'arbres XML ou de valeurs atomiques).
- Chaque expression retourne une valeur
- Une expression peut être :
  - une valeur atomique, un noeud XML ou une séquence ;
  - une variable;
  - une expression arithmétique ou logique (les opérateurs de base sont ceux d'XPath 2.0) ;
  - une union, intersection ou différence de séquences ;
  - une requête XPath 2.0 ;
  - une fonction prédéfinie ou définie en local;
  - un des opérateurs spécifiques (alternative "if-then-else", boucle, etc.).

Exemple de fichier XML : factures.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<factures>
  <facture num="1" dateFacture="2006-1-1">
    <client codeClient="c1" societe="AGFA"/>
    <detail>
      <produit designation="PC212" quantite="12" prixUnitaire="6700" montant="80400"/>
      <produit designation="Imp11" quantite="3" prixUnitaire="1700" montant="5100"/>
      <produit designation="Papier" quantite="22" prixUnitaire="40" montant="880"/>
    </detail>
  </facture>
  <facture num="2" dateFacture="2006-11-1">
    <client codeClient="c2" societe="MITAL"/>
    <detail>
      <produit designation="Inf33" quantite="11" prixUnitaire="3400" montant="37400"/>
      <produit designation="TVLCD52" quantite="6" prixUnitaire="9000" montant="54000"/>
    </detail>
  </facture>
  <facture num="3" dateFacture="2006-12-11">
    <client codeClient="c2" societe="MITAL"/>
    <detail>
      <produit designation="Inf33" quantite="11" prixUnitaire="3400" montant="37400"/>
      <produit designation="TVLCD52" quantite="6" prixUnitaire="9000" montant="54000"/>
    </detail>
  </facture>
</factures>
```

Exemple 1 de XQuery

```
<resultat>
 {
   doc("factures.xml")/factures/facture[@num=1]
 }
</resultat>
```

- Retourne la facture numéro 1 du documents factures.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>
  <facture num="1" dateFacture="2006-1-1">
      <client codeClient="c1" societe="AGFA"/>
      <detail>
         <produit designation="PC212" quantite="12" prixUnitaire="6700" montant="80400"/>
         <produit designation="Imp11" quantite="3" prixUnitaire="1700" montant="5100"/>
         <produit designation="Papier" quantite="22" prixUnitaire="40" montant="880"/>
      </detail>
   </facture>
</resultat>
```

Exemple 2 de XQuery

```
<resultat>
 {
   sum(doc("factures.xml")/factures/facture[@num=1]/detail/produit/@montant)
 }
</resultat>
```

- Retourne la somme des montant des produits de la facture numéro 1

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>86380</resultat>
```

Exemple 3 XQuery

```
<resultat>
 {
   for $f in doc("factures.xml")/factures/facture
   return
   <totalFacture num="{$f/@num}">{ sum($f/detail/produit/@montant)}
    </totalFacture>
}
</resultat>
```

- Retourne la somme des montant des produits de chaque facture.

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>
  <totalFacture num="1">86380</totalFacture>
  <totalFacture num="2">91400</totalFacture>
  <totalFacture num="3">91400</totalFacture>
</resultat>
```

Exemple 4 XQuery

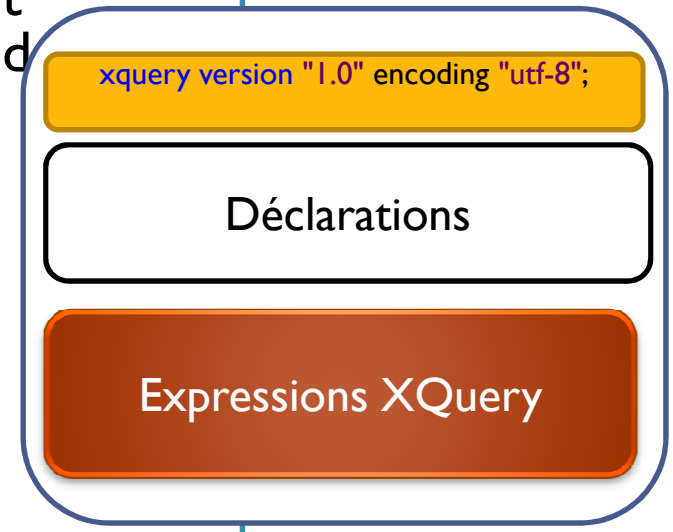
```
<resultat> {
   for $f in doc("factures.xml")/factures/facture
     let $total:=sum($f/detail/produit/@montant)
   where $total>90000
   return
    <totalFacture num="{$f/@num}">{$total}</totalFacture>
}</resultat>
```

- Retourne la somme des montant des produits des factures dont le total est supérieur à 90000

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>
  <totalFacture num="2">91400</totalFacture>
  <totalFacture num="3">91400</totalFacture>
</resultat>
```

Requête XQuery

- Une requête XQuery est composée de trois parties :
  - une ligne d'entête commencée par "xquery" et contenant la version et, éventuellement l'encod document ;
    - xquery version "1.0" encoding "utf-8";
  - un ensemble de déclarations :
    - déclarations de variables ou constantes,
    - déclarations de fonctions utilisateur locales,
    - importation de modules (bibliothèques XQuery),
    - détermination du schéma du résultat,
    - détermination des espaces de nom et de leur utilisation,
    - détermination du format du résultat et autres paramètres ;
  - l'expression XQuery à exécuter.
- La ligne d'entête et les déclarations sont toutes terminées par ";".



Opérateurs de XQuery

- Les opérateurs arithmétiques et logiques sont les mêmes que ceux de XPath 2.0.
  - Opérateurs de comparaison sur les valeurs atomiques ("**eq**", "**ne**", "**gt**", "**ge**", "**lt**", "**le**") et sur les séquences ("**=**", "**!=**", "**>**", "**>=**", "**<**", "**<=**").
  - Des opérateurs existent aussi sur les noeuds ("**is**", "**isnot**", "**<<**", "**>>**").
- XQuery exploite aussi les opérateurs déjà proposés par XPath 2.0 comme :
  - l'alternative avec "**if** (condition) **then** Seq_1 **else** Seq_2" où, contrairement à beaucoup de langages, le "**else**" est **obligatoire** ;
  - la quantification existentielle avec "**some** $v **in** Seq **satisfies** exp($v)" qui sera satisfaite si au moins un item de "Seq" satisfait l'expression booléenne "exp" ;
  - la quantification universelle avec "**every** $v **in** Seq **satisfies** exp($v)" qui sera satisfaite si tous les items de "Seq" satisfont l'expression booléenne "exp".
- XQuery propose un opérateur fondamental pour combiner des informations : l'opérateur FLWOR . Il peut être mis en comparaison avec le célèbre "select" de SQL.

Opérateur FLWOER

- L'opérateur FLWOR est un opérateur complexe qui permet de parcourir une ou plusieurs séquences (et d'effectuer ainsi des opérations de type "jointure").
- La forme de cet opérateur comprend un certain nombre de clauses dans l'ensemble suivant (la première lettre de chacune des clauses forme le sigle FLWOR) :
  - **f**or
  - **l**et
  - **w**here
  - **o**rder by
  - **r**eturn

La clause for

- permet de parcourir une ou plusieurs séquences en positionnant la valeur d'autant de variables.
- Une variable XQuery commence par un "$".
- Si plusieurs séquences sont présentes, la clause parcourra le produit cartésien des différentes séquences.
- Cette clause est de la formes :
  - for v_1 in Seq_{f1}, v_2 in Seq_{f2}, … v_n in Seq_{fn}'.
- Exemple :
  - for $i in (1 to 3), $j in (4 to 6)

La clause let

- La clause "let" permet de positionner des variables "locales" au n-uplet courant.
- Cette clause permet souvent de simplifier grandement les filtres, voire la génération du résultat.
- Elle est de la forme :
  - let $\$loc_1$:= $\$Seq_{l1}$, $\$loc_2$:= $\$Seq_{l2}$, …, $\$loc_m$:= $\$Seq_{lm}$

La clause where

- La clause "where" est une clause de filtre.
- Elle permet de ne sélectionner que les n-uplets qui satisfont à l'expression booléenne qu'elle décrit.
- Cette expression comporte donc nécessairement une ou plusieurs variables d'une clause "for" ou d'une clause "let" précédente.
- Elle est de la forme :
  - where cond(v_1, v_2, ..., v_n, loc_1, loc_2, ..., loc_m)

La clause order by

- La clause "order by" décrit un critère d'ordre pour la séquence de valeurs qui sera générée, fonction des variables des clauses "for" ou "let".
- Elle est de la forme :
  - order by $critère$_1$ dir$_1$, …, $critère$_p$ dir$_p$
- où les "dir$_i$" sont pris dans { ascending, descending}.
- Ce dernier peut être suivit de
  - "empty greatest" ou "empty least".

La clause return

- La clause "return" permet de construire un résultat pour chacun des n-uplets.
- Elle est de la forme
  - return exp
  - où "exp" est une expression XQuery.

Exemple 1 de FLOWER

```
for $i in (1 to 3), $j in (4 to 6)
 let $k:= ($i to $ j)
 return <res>{$i}/{$j}/{$k}</res>
```

- Produit le résultat suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<res>1/4/1 2 3 4</res>
<res>1/5/1 2 3 4 5</res>
<res>1/6/1 2 3 4 5 6</res>
<res>2/4/2 3 4</res>
<res>2/5/2 3 4 5</res>
<res>2/6/2 3 4 5 6</res>
<res>3/4/3 4</res>
<res>3/5/3 4 5</res>
<res>3/6/3 4 5 6</res>
```

Exemple 2 de FLOWER

```
xquery version "1.0" encoding "utf-8";
<resultat> {
   for $f in doc("factures.xml")/factures/facture
     let $total:=sum($f/detail/produit/@montant)
   where $total>90000
   order by $f/@num  descending
   return
    <totalFacture num="{$f/@num}">{$total}</totalFacture>
}</resultat>
```

- Retourne les factures dont le total est supérieur à 90000 classées par ordre décroissant des numéros de factutue

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>
  <totalFacture num="3">91400</totalFacture>
  <totalFacture num="2">91400</totalFacture>
</resultat>
```

Exemple 2 de FLOWER

```
<resultat> {
   for $f in doc("factures.xml")/factures/facture
     let $total:=sum($f/detail/produit/@montant)
   where some $p in $f/detail/produit satisfies $p/@designation eq "PC212"
   order by $f/@num  descending
   return
    <totalFacture num="{$f/@num}">{$total}</totalFacture>
}</resultat>
```

- Retourne le total des factures dont le total dont la désignation de l'un des produits est « **PC212** », classées par ordre décroissant des numéros de factutue

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>
  <totalFacture num="1">86380</totalFacture>
</resultat>
```

Exemple 3 de FLOWER

```
<resultat> {
  let $xmlDoc:=doc("factures.xml")
  for $c in $xmlDoc//client[not(@codeClient=preceding::client/@codeClient)]
  let  $nbFactures:=count($xmlDoc//facture[client/@codeClient=$c/@codeClient])
  let  $totalFactures:=sum(
   $xmlDoc//facture[client/@codeClient=$c/@codeClient]/detail/produit/@montant
   )
  return
   <client societe="{$c/@societe}" nbFact="{$nbFactures}"
   totalFact="{$totalFactures}"/>
 }</resultat>
```

- Retourne pour chaque client le nombre et le total de ses factures

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>
  <client totalFact="86380" societe="AGFA" nbFact="1"/>
  <client totalFact="182800" societe="MITAL" nbFact="2"/>
</resultat>
```

XQuery : Langage de programmation

- XQuery n'est pas seulement un langage de requête "à la SQL".
- C'est aussi un vrai langage de programmation fonctionnel.
- Il possède un certain nombre de caractéristiques qui, couplées à quelques extensions, en font un outil intéressant pour du développement d'applications Web.
- Nous allons évoquer ici quelques caractéristiques de bases.

Les variables

- Nous avons déjà abordé un peu la question des variables puisqu'elles sont utilisées dans les structures "some" et "every", mais aussi dans la structure "FLWOR".
- Cependant, XQuery propose aussi un mécanisme de variables globales internes ou externes.
- Les variables internes sont déclarées avant la requête selon la forme
  - declare variable $v [as type] := expression_XQuery ;
  - Le type peut être omis, il sera alors estimé en fonction de la valeur retournée par l'expression.

Les types de variables

- Les types, aussi bien pour les variables que pour les signatures des fonctions, sont ceux de XSD avec, en plus, des types spécifiques.
- La figure (http://www.w3.org/TR/xpath-functions/type-hierarchy.png). les rappelle

Déclaration de variables externes

- Les variables externes sont déclarées de la manière suivante :
  - **declare variable** $v [**as type**] **external**;
  - Exemple :
  - declare variable $v as xs:double external;

Exemple de déclaration de variables globales

```
declare variable $xmlDoc:=doc("factures.xml");
declare variable $max:=100000;
declare variable $clients:=
 for $c in $xmlDoc//client[not(@codeClient=preceding::client/@codeClient)]
   let  $nbFactures:=count($xmlDoc//facture[client/@codeClient=$c/@codeClient])
   let  $totalFactures:=sum(
     $xmlDoc//facture[client/@codeClient=$c/@codeClient]/detail/produit/@montant)
   return
    <client societe="{$c/@societe}" nbFact="{$nbFactures}" totalFact="{$totalFactures}"/>;
for $c in $clients
where $c/@totalFact>$max
return $c
```

- Retourne pour chaque client le nombre et le total de ses factures dont le total est supérieur à la variable globale max

```
<?xml version="1.0" encoding="UTF-8"?>
<client totalFact="182800" societe="MITAL" nbFact="2"/>
```

Fonctions et modules

- De très nombreuses fonctions prédéfinies (http://www.w3.org/TR/xpath-functions/) existent.
- Les fonctions sont présentées avec une signature sur les paramètres et le retour de fonction.
- Pour les comprendre, il faut juste savoir que les opérateurs sur les types signifient :
  - "*" : une séquence qui peut être vide ;
  - "+" : une séquence non vide ;
  - "?" : éventuellement la séquence vide.

Fonctions et modules

- Pour illustrer ces opérateurs, voici quelques exemples parmi les très nombreuses fonctions disponibles :
  - fn:count($arg as item()*) as xs:integer : la fonction "count" prend en argument une séquence, éventuellement vide, d'items et retourne un entier ;
  - fn:string-length($arg as xs:string?) as xs:integer : la fonction "string-length" prend en paramètre une chaîne de caractères et retourne un entier (elle peut ne pas avoir de paramètre);
  - fn:sum($arg as xs:anyAtomicType*) as xs:anyAtomicType" : elle prend en argument une séquence de type de base et retourne un type de base ;

Fonctions et modules

- fn:tokenize($input as xs:string?, $pattern as xs:string) as xs:string*" : elle prend une chaîne de caractères et un motif et retourne une séquence de chaînes de caractères.
- Quelques exemples supplémentaires classés par thème :
  - Numériques : min, max, sum, avg, count, abs, ceiling, floor, round...
  - Chaînes de caractères : compare, concat, substring, string-length,starts-with, ends-width...
  - Dates : date, current-date …
  - Séquences : zero-or-one, one-or-more, exactly-one, empty, exists, distinct-values, reverse,...

Fonctions locales

- Il est possible de proposer ses propres fonctions.
- Pour cela, dans la partie des déclarations, il suffit de décrire sa fonction de la manière suivante :
  - "declare function nom signature {code} ;"
  - où :
    - "nom" : soit dans un espace de noms déclaré soit préfixé par "local" (espace de noms des fonctions pas défaut) ;
    - "signature" : (liste des paramètres, éventuellement typés) ["as" type de retour] ;
    - "code" : n'importe quelle expression XQuery.
- Comme premier exemple simple, proposons une fonction permettant de construire une séquence de nombres pairs entre 2 et une valeur donnée en paramètre. Nous proposons ici deux versions : une avec les types explicites et une sans expliciter les types.

Exemple de fonction locales

```
xquery version "1.0" encoding "utf-8";
declare function local:liste_entier_pair($max as xs:integer)  as xs:integer* {
    (2 to $max)[. mod 2 = 0]
} ;
<pair>{local:liste_entier_pair(10)}</pair>
```

Fonction avec les types explicites

```
xquery version "1.0" encoding "utf-8";
declare function local:liste_entier_pair($max ) {
    (2 to $max)[. mod 2 = 0]
} ;
<pair>{local:liste_entier_pair(10)}</pair>
```

Fonction non typées

- Retourne les nombre paires entre 0 et 10 en utilisant la fonction locale.

```
<?xml version="1.0" encoding="UTF-8"?>
<pair>2 4 6 8 10</pair>
```

Exemple2 de fonction locales

```
declare function local:facturesClient($client as element(client)) as element(facture)*{
  doc("factures.xml")//facture[client/@codeClient=$client/@codeClient]
};
<resultat>{
 let $c:=doc("factures.xml")//client[@codeClient="c1"]
 return local:facturesClient($c)
}</resultat>
```

- Factures du client dont le code est « c1»

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>
   <facture num="1" dateFacture="2006-1-1">
        <client codeClient="c1" societe="AGFA"/>
        <detail>
            <produit designation="PC212" quantite="12" prixUnitaire="6700" montant="80400"/>
            <produit designation="Imp11" quantite="3" prixUnitaire="1700" montant="5100"/>
            <produit designation="Papier" quantite="22" prixUnitaire="40" montant="880"/>
        </detail>
    </facture>
</resultat>
```

Les modules

- Comme pour tout langage de programmation, il est possible de mettre en place des bibliothèques de fonctions.
- En XQuery, elles s'appellent des modules.
- Un module consiste en un programme XQuery sans requête principale.
- On y trouve donc toutes les déclarations autorisées dans un programme XQuery.
- Pour caractériser un module, il suffit, en tout début, d'indiquer son espace de noms :
  - module namespace mon-module = 'ma:uri';
- Pour utiliser un module, il suffit, dans l'entête, d'indiquer l'espace de noms et le(s) fichier(s) qui le décri(ven)t :
  - import module namespace mon-module = 'ma:uri' at 'monModule1.xquery', 'monModule2.xquery';"

Exemple de module

```
module namespace facturation="ma:enset.bp";
(:Déclaration d'une variable globale :)
declare variable $facturation:code as xs:string:="c1";

(:Déclaration d'une fonction qui retourne les nombres paires :)
declare function facturation:liste_entier_pair($max ) {
    (2 to $max)[. mod 2 = 0]
} ;

(:Déclaration d'une fonction qui retournes les factures d'un client :)
declare function facturation:facturesClient($client as element(client)) as
    element(facture)*{
  doc("factures.xml")//facture[client/@codeClient=$client/@codeClient]
};
```

- Module qui déclare une variable globale max et deux fonctions.

Utilisation d'un module

```
import module namespace fact="ma:enset.bp" at "module1.xquery";
<resultat>
 {
  let $c:=doc("factures.xml")//client[@codeClient=$fact:code]
  return fact:facturesClient($c)
 }
</resultat>
```

- Retourne les factures des clients « c1 ».

```
<?xml version="1.0" encoding="UTF-8"?>
<resultat>
  <facture num="1" dateFacture="2006-1-1">
      <client codeClient="c1" societe="AGFA"/>
      <detail>
         <produit designation="PC212" quantite="12" prixUnitaire="6700" montant="80400"/>
         <produit designation="Imp11" quantite="3" prixUnitaire="1700" montant="5100"/>
         <produit designation="Papier" quantite="22" prixUnitaire="40" montant="880"/>
      </detail>
   </facture>
</resultat>
```

JDOM

Mohamed Youssfi

med@youssfi.net

JDOM ?

- JDOM est une API du langage Java développée indépendamment de Sun Microsystems.
- Elle permet de manipuler des donnés XML plus simplement qu'avec les API classiques.
- Son utilisation est pratique pour tout développeur Java et repose sur les API XML de Sun.
- JDOM permet donc de construire des documents, XML, de naviguer dans leur structure, s'ajouter, de modifier, ou de supprimer leur contenu.

Origines de JDOM

- Les Parseurs SAX
  - SAX est l'acronyme de *Simple API for XML*.
  - Ce type de parseur utilise des événements pour piloter le traitement d'un fichier XML.
  - JDOM utilise des collections SAX pour parser les fichiers XML.
- Les Parseurs DOM :
  - DOM est l'acronyme de *Document Object Model*. C'est une spécification du W3C pour proposer une API qui permet de modéliser, de parcourir et de manipuler un document XML.
  - Le principal rôle de DOM est de fournir une représentation mémoire d'un document XML sous la forme d'un arbre d'objets et d'en permettre la manipulation (parcours, recherche et mise à jour).
  - A partir de cette représentation (le modèle), DOM propose de parcourir le document mais aussi de pouvoir le modifier.
  - Ce dernier aspect est l'un des aspect les plus intéressant de DOM.
  - DOM est défini pour être indépendant du langage dans lequel il sera implémenté.
  - JDOM utilise DOM pour manipuler les éléments d'un Document Object Model spécifique (créé grâce à un constructeur basé sur SAX).

Créer un document XML avec JDOM

- L'API JDOM :
  - Il vous faut dans un premier temps télécharger la dernière version de JDOM disponible à cette adresse : http://www.jdom.org/dist/binary/.
  - Il suffit ensuite de rendre accessible le fichier */build/jdom.jar*, en le plaçant dans votre classpath.
- Création d'un document XML avec JDOM:
  - La création d'un fichier XML en partant de zéro est des plus simple.
  - Il suffit de construire chaque élément puis de les ajouter les uns aux autres de façon logique.
  - Un noeud est une instance de *org.jdom.Element*.
  - Nous commençons donc par créer une classe JDOM1 qui va se charger de créer l'arborescence suivante :

```
<personnes>
    <etudiant classe="P2">
        <nom>Katib</nom>
    </etudiant>
</personnes>
```

Exemple d'application pour la création d'un document XML avec JDOM

```
import java.io.*;import org.jdom.*;
import org.jdom.output.*;
public class App1 {
 public static void main(String[] args) throws IOException {
 /*Créer l'élément racine */
 Element racine = new Element("personnes");
/*Créer un document JDOM basé sur l'élément racine*/
    Document document = new Document(racine);
  /*Créer un nouveau Element xml etudiant */
 Element etudiant = new Element("etudiant");
/* Ajouter cet élément à la racine */
 racine.addContent(etudiant);
   /*Créer un nouveau attribut classe dont la valeur est P2 */
 Attribute classe = new Attribute("classe","P2");
   /* Ajouter cet attrubut à l'élément etudiant */
 etudiant.setAttribute(classe);
 Element nom = new Element("nom"); /* Créer l'élément nom */
 nom.setText("Katib"); /* Définir le texte de nom */
 etudiant.addContent(nom); /* ajouter nom à etudiant */
 /* Afficher et enregistrer le fichier XML */
 XMLOutputter sortie=new XMLOutputter(Format.getPrettyFormat());
 sortie.output(document, System.out);
 sortie.output(document, new FileOutputStream("EX1.xml"));
}}
```

Parcourir un document XML avec JDOM

- Parser un fichier XML revient à transformer un fichier XML en une arborescence JDOM.
- Nous utiliserons pour cela le constructeur SAXBuilder, basé, comme son nom l'indique, sur l'API SAX.
- Créez tout d'abord le fichier suivant dans le répertoire contenant votre future classe JDOM2 :

```
<?xml version="1.0" encoding="UTF-8"?>
<personnes>
   <etudiant classe="P2">
      <nom>katib</nom>
      <prenoms>
         <prenom>mohamed</prenom>
         <prenom>amine</prenom>
      </prenoms>
   </etudiant>
   <etudiant classe="P1">
      <nom>talbi</nom>
   </etudiant>
   <etudiant classe="P1">
      <nom>santel</nom>
   </etudiant>
</personnes>
```

Afficher les noms des étudiants du fichier XML

```
import java.io.*;import java.util.List;
import org.jdom.*;import org.jdom.input.SAXBuilder;
public class App2 {
public static void main(String[] args) throws Exception {
/* On crée une instance de SAXBuilder */
 SAXBuilder sb=new SAXBuilder();
/* On crée un nouveau document JDOM avec en argument le fichier XML */
 Document document=sb.build(new File("EX2.xml"));
/*On initialise un nouvel élément racine avec l'élément racine du
   document.*/
 Element racine=document.getRootElement();
 /* On crée une List contenant tous les noeuds "etudiant" de l'Element
   racine */
 List<Element> etds=racine.getChildren("etudiant");
 /* Pour chaque Element de la liste etds*/
 for(Element e:etds){
    /*Afficher la valeur de l'élément nom */
 System.out.println(e.getChild("nom").getText());
 }
}

}
```

Modifier une arborescence avec JDOM

- Quelques méthodes de JDOM:
  - addContent : Ajoute le contenu de l'argument à la fin du contenu d'un Element. On peut spécifier un index pour l'inserer à la position voulu.
  - clone : Retourne un clone parfait de l'Element.
  - cloneContent : on ne copie que le contenu.
  - removeAttribute : Supprime un attribut d'un Element
  - removeChild : Supprime le premier enfant portant ce nom.
  - removeChildren : Supprime tous les enfants ayant ce nom.
  - removeContent : Supprime l'intégralité d'un noeud donné en argument ou par sa position. removeContent accept aussi les filtres, tout comme getContent vu précédement.
  - setAttribute : permet à la fois de créer un attribut et d'en modifier sa valeur.
  - setContent : Remplace le contenu d'un Element. On peut spécifier un index si l'on ne veut pas tout remplacer.
  - setName : Change le nom de l'Element.
  - setText : Change le text contenu par l'Element. <element>TEXT</element>
  - toString : Retourne une représentation de l'Element sous forme de chaine.

Parseur SAX DOM

- Pour traiter la totalité d'un document XML au moment de sa lecture, il faut utiliser un parseur SAX.
- JDK fournit un parseur DOM et un parseur SAX qui peuvent être utilisé sans passer par JDOM.

Interpréter un fichier XML

```
import javax.xml.parsers.*;import org.xml.sax.*;
import org.xml.sax.helpers.*;
public class SaxParser extends DefaultHandler {
public SaxParser() {
try {
 SAXParserFactory parserFactory=SAXParserFactory.newInstance();
 SAXParser saxParser=parserFactory.newSAXParser();
 saxParser.parse("EX3.xml",this);
} catch (Exception e) { e.printStackTrace(); }
}
```

Interpréter un fichier XML

```
@Override
public void startDocument() throws SAXException {
   System.out.println("Début Document");
}
@Override
public void startElement(String uri, String localName, String qName,
Attributes attributes) throws SAXException {
// Début de chaque élément
  System.out.println("Elément :"+qName);
  System.out.println("\t Attributs:");
  for(int i=0;i<attributes.getLength();i++){
  System.out.println("\t\t"+attributes.getQName(i)
  +"="+attributes.getValue(i));
}
}
```

Interpréter un fichier XML

```
@Override
public void endElement(String uri, String localName, String qName)
throws SAXException {
 // Fin de chaque élément
}
@Override
public void endDocument() throws SAXException {
 // Fin du document
}

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

}
```

Passer d'un document DOM à JDOM

- Il vous arrivera parfois de devoir travailler sur un document DOM.
- Nous allons voir comment transformer un document DOM en un document JDOM et vis versa.
- Voici une méthode qui reçoit en argument un document DOM et retourne un document JDOM.

```
org.jdom.Document DOMtoJDOM(org.w3c.dom.Document
documentDOM) throws Exception
 {
/* On utilise la classe DOMBuilder pour cette
 transformation */
     DOMBuilder builder = new DOMBuilder();
     org.jdom.Document documentJDOM =
builder.build(documentDOM);
     return documentJDOM;
 }
```

Passer d'un document JDOM à DOM

- Et maintenant, voici la fonction inverse qui reçoit en argument un document JDOM et qui retourne un document DOM.
- Vous remarquerez la similitude avec la fonction précédente.

```
org.w3c.dom.Document DOMtoJDOM(org.jdom.Document
  documentJDOM) throws Exception
 {
/* On utilise la classe DOMOutputter pour cette
  transformation */
    DOMOutputter domOutputter = new DOMOutputter();
    org.w3c.dom.Document documentDOM =
  domOutputter.output(documentJDOM);
    return documentDOM;
 }
```

JDOM et XSLT

- Grâce à l'API JAXP et TraX il est très facile de faire des transformation XSLT sur un document JDOM.
- Dans l'exemple suivant nous allons créer une méthode qui prend en entrée un document JDOM et le nom d'un fichier XSL et qui crée en sortie un fichier XML transformé.

Fichier XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<personnes>
   <etudiant classe="P2">
      <nom>katib</nom>
      <prenoms>
         <prenom>mohamed</prenom>
         <prenom>amine</prenom>
      </prenoms>
   </etudiant>
   <etudiant classe="P1">
      <nom>talbi</nom>
   </etudiant>
   <etudiant classe="P1">
      <nom>santel</nom>
   </etudiant>
</personnes>
```

Fichier XSL pour générer une sortie HTML

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
 <xsl:template match="/">
     <html>
         <head></head>
         <body>
             <table border="1" width="80%">
                  <tr>
                      <th>Nom</th><th>Prénoms</th>
                  </tr>
                 <xsl:for-each select="personnes/etudiant">
                     <tr>
                         <td><xsl:value-of select="nom"/></td>
                 <td><xsl:value-of select="prenoms/prenom[1]"/></td>
                     </tr>
                 </xsl:for-each>
             </table>

         </body>
     </html>
 </xsl:template>
</xsl:stylesheet>
```

Nom	Prénoms
katib	mohamed
talbi	
santel	

Exemple de transformation XSL avec JDOM

```
import java.io.*; import org.jdom.*;
import org.jdom.input.SAXBuilder;
import javax.xml.transform.*;
import javax.xml.transform.stream.StreamSource;
public class JDOM4{
public static void main(String[] args) {
SAXBuilder sb=new SAXBuilder();
try {
  Document jDomDoc=sb.build(new
  File("Exercice2.xml"));
  outputXSLT(jDomDoc, "classe.xsl");
} catch (Exception e) {
  e.printStackTrace();
}
}
```

Exemple de transformation XSL avec JDOM

```
public static void outputXSLT(org.jdom.Document documentJDOMEntree,String
   fichierXSL){
/*Document JDOMResult, résultat de la transformation TraX */
   JDOMResult documentJDOMSortie = new JDOMResult();
/* Document JDOM après transformation */
   org.jdom.Document resultat = null;
   try{
/* On définit un transformer avec la source XSL qui va permettre la
   transformation */
TransformerFactory factory = TransformerFactory.newInstance();
Transformer transformer = factory.newTransformer(new
   StreamSource(fichierXSL));
/* On transforme le document JDOMEntree grâce à notre transformer. La
   méthoded transform() prend en argument le document d'entree associé au
   transformer et un document JDOMResult, résultat de la transformation
   TraX */
 transformer.transform(new org.jdom.transform.JDOMSource
   (documentJDOMEntree), documentJDOMSortie);
/* Pour récupérer le document JDOM issu de cette transformation, il faut
   utiliser la méthode getDocument() */
 resultat = documentJDOMSortie.getDocument();
/* On crée un fichier xml corespondant au résultat */
 XMLOutputter outputter = new XMLOutputter(Format.getPrettyFormat());
 outputter.output(resultat, new FileOutputStream("resultat.htm"));
   }   catch(Exception e){}
}}
```

Fichier HTML généré : resultat.htm

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head />
  <body>
    <table border="1" width="80%">
      <tr>
        <th>Nom</th>
        <th>PrÃ©noms</th>
      </tr>
      <tr>
        <td>CynO</td>
        <td>Nicolas</td>
      </tr>
      <tr>
        <td>Superwoman</td>
        <td />
      </tr>
      <tr>
        <td>Don Corleone</td>
        <td />
      </tr>
    </table>
  </body>
</html>
```

Application

- 1- Créer une application java qui permet de créer le fichier XML suivant : meteo.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<meteo>
  <mesure date="2006-1-1">
    <ville nom="casa" temp="25" />
    <ville nom="rabat" temp="28" />
  </mesure>
</meteo>
```

Application

- 2- Créer une application java qui permet de lire le fichier XML créé précédemment, de lui ajouter un nouveau élément mesure marqué en rouge et d'enregistrer le nouveau fichier XML sous le nom meteo2.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<meteo>
  <mesure date="2006-1-1">
    <ville nom="casa" temp="25" />
    <ville nom="rabat" temp="28" />
  </mesure>
  <mesure date="2006-1-2">
    <ville nom="casa" temp="30" />
    <ville nom="rabat" temp="32" />
  </mesure>
</meteo>
```

Application

- 3 - Créer une application java qui permet de lire le fichier XML meteo2.xml et d'afficher toutes les mesures comme suit:

```
Date de mes:2006-1-1
  Ville:casa; Temp:25
  Ville:rabat; Temp:28
Date de mes:2006-1-2
  Ville:casa; Temp:30
  Ville:rabat;Temp:32
```

Application

- 4 – Créer une feuille de style XSL qui permet de transformer le document XML meteo2.xml en document HTML suivant :

Date de Mesure : 2006-1-1

Ville	Temp
casa	25
rabat	28
Moyenne temp:	26.5

Date de Mesure : 2006-1-2

Ville	Temp
casa	30
rabat	32
Moyenne temp:	31

- 5 – Créer une application java qui permet d'afficher la transformation du fichier XML meteo2.xml par la feuille de style XSL que vous venez de créer. Enregistrer la sortie dans un fichier m.htm

Application

- Nous disposons d'une base de données MYSQL nommée DB_CATALOGUE qui contient des produits appartenant à des catégories.
- La base de données contient deux tables:
  - CATEGORIES : contient les catégories
  - PRODUITS : contient les produits.
- La structure des deux tables est la suivante :

Field	Type	Collation	Attributes	Null	Default	Extra
ID_CAT	int(11)			No	*None*	AUTO_INCREMENT
NOM_CAT	varchar(25)	latin1_swedish_ci		No		
DESCRIPTION	text	latin1_swedish_ci		No	*None*	

Field	Type	Collation	Attributes	Null	Default	Extra
ID_PROD	int(11)			No	*None*	AUTO_INCREMENT
NOM_PROD	varchar(25)	latin1_swedish_ci		No		
PRIX	float			No	0	
ID_CAT	int(11)			No	0	

Code SQL pour générer la base de données:

```
CREATE TABLE IF NOT EXISTS `categories` (
  `ID_CAT` int(11) NOT NULL AUTO_INCREMENT,
  `NOM_CAT` varchar(25) NOT NULL DEFAULT '',
  `DESCRIPTION` text NOT NULL,
  PRIMARY KEY (`ID_CAT`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=9 ;
INSERT INTO `categories` (`ID_CAT`, `NOM_CAT`, `DESCRIPTION`) VALUES
(1, 'ordinateurs', 'ord pc'),
(2, 'imprimantes', 'imimp');

CREATE TABLE IF NOT EXISTS `produits` (
  `ID_PROD` int(11) NOT NULL AUTO_INCREMENT,
  `NOM_PROD` varchar(25) NOT NULL DEFAULT '',
  `PRIX` float NOT NULL DEFAULT '0',
  `ID_CAT` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`ID_PROD`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=5 ;

INSERT INTO `produits` (`ID_PROD`, `NOM_PROD`, `PRIX`, `ID_CAT`) VALUES
(1, 'HP Vecrta', 7000, 1),
(2, 'PC IBM', 8000, 1),
(3, 'Imprimante HP 690', 1200, 2),
(4, 'Epson 3477', 1300, 2);
```

Travail à faire

- Ecrire une application java qui permet de :
  - Saisir le code de la catégorie :
  - Générer un fichier qui contient les produits de cette catégorie.
- la structure de ce fichier XML est la suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<catalogue codeCat="1">
  <produit idProduit="1" nomProduit="HP Vecrta" prix="7000" />
  <produit idProduit="2" nomProduit="PC IBM" prix="8000" />
</catalogue>
```

Application Java

```
import java.io.*;import java.sql.*;import java.util.*;import org.jdom.*;
import org.jdom.output.*;
public class ApplicationJDOM_JDBC {
public static void main(String[] args) {
try {
Scanner clavier=new Scanner(System.in);
System.out.print("CODE CAT:");int codeCat=clavier.nextInt();
Class.forName("com.mysql.jdbc.Driver");
Connection conn=DriverManager.getConnection
("jdbc:mysql://localhost:3306/DB_CAT2","root","");
PreparedStatement ps=conn.prepareStatement
("select * from PRODUITS where ID_CAT=?");
ps.setInt(1, codeCat);
ResultSet rs=ps.executeQuery();
```

Application Java

```
Document xmlDoc=new Document(); Element racine=new Element("catalogue");
xmlDoc.setRootElement(racine);
racine.setAttribute(new Attribute("codeCat", String.valueOf(codeCat)));
while(rs.next()){
 Element produit=new Element("produit");
 produit.setAttribute(new Attribute("idProduit", rs.getString("ID_PROD")));
 produit.setAttribute(new Attribute("nomProduit", rs.getString("NOM_PROD")));
 produit.setAttribute(new Attribute("prix", rs.getString("PRIX")));
 racine.addContent(produit);
}
XMLOutputter sortie=new XMLOutputter(Format.getPrettyFormat());
sortie.output(xmlDoc, System.out);
sortie.output(xmlDoc, new FileOutputStream("catalogue.xml"));
conn.close();
} catch (Exception e) {e.printStackTrace();}
}
}
```

Application : Web Services

- Une société de bourse dispose d'une base de données qui permet de stocker les valeurs journalières des actions des sociétés cotées en bourse.
- Cette base de données de type MySQL est nommée

« DB_BOURSE ». Elle contient la table « COTATION » dont la structure est la suivante :

  - NUM_COTATION : INT
  - CODE_SOCIETE :VARCHAR(10)
  - DATE_COTATION : DATE
  - VALEUR_ACTION : FLOAT

Application : Web Services

- On souhaite créer et déployer un web service qui permet de consulter les cotations d'une société donnée entre deux dates d1 et d2.
- Travail à faire :
  - Dessiner l'architecture du projet
  - Mettre en place la base de données.
  - Créer et tester la couche métier qui se compose de :
    - La classe Cotation
    - La classe Bourse qui contient une méthode qui permet de retourner une liste de cotations d'une société donnée entre deux dates
    - L(application TestBourse pour tester la classe Bourse
  - Créer et déployer le web service
  - Tester le web service avec l'analyseur WDSL SOAP de Oxygen
  - Créer un client Java pour le web service
  - Créer un client Dot Net pour le web service

SVG

Introduction

- SVG : Scalable Vector Graphics. Ce langage permet d'écrire des graphiques vectoriels 2D en XML.
- Il a été inventé en 1998 par un groupe de travail (comprenant Microsoft, Autodesk, Adobe, IBM, Sun, Netscape, Xerox, Apple, Corel, HP, ILOG...) pour répondre à un besoin de graphiques légers, dynamiques et interactifs.
- Une première ébauche du langage est sortie en octobre 1998 et en juin 2000 apparaît la première version du **Viewer Adobe** (*plugin* permettant de visualiser le SVG).
- Le SVG s'est très vite placé comme un concurrent de **Flash** et à ce titre, Adobe a intégré ce langage dans la plupart de ses éditeurs (dont les principaux sont **Illustrator** et **Golive**).

Pourqoui SVG

- Les raisons pouvant pousser à l'adoption d'un format comme SVG sont nombreuses ;
  - Liées aux avantages du graphisme vectoriel :
    - Adaptation de l'affichage à des media variés et à des tailles différentes ;
    - Taille de l'image après compression est très faible ;
    - ….
  - Liées aux avantages particuliers du format SVG :
    - Insertion dans le monde XML/XHTML :
    - Modèle de couleurs sophistiqué, filtres graphiques
    - Possibilité d'indexage par les moteurs de recherche;
    - Possibilité de partager du code ;
    - Meilleures capacités graphiques dans l'ensemble.

Outils

- **Edition**
  - Comme SVG est un format XML, n'importe quel éditeur de texte suffit.
  - Il est cependant possible d'utiliser un éditeur dédié comme **WebDraw** de Jasc, ou InkScape. **Adobe Illustrator**, **Corel Draw** permettent aussi d'exporter au format SVG.
- **Visualisation**
  - SVG n'est actuellement pas supporté en natif par tous les navigateurs. On distingue plusieurs degrés d'avancement :
  - Internet Explorer ne le supporte pas en natif. Il est donc nécessaire d'installer un *plugin*. le *plugin* **SVG Viewer 3.03**, téléchargeable gratuitement sur le site d'Adobe : http://www.adobe.com/svg/
  - FireFox, à partir de sa version 1.5, a entrepris de supporter en natif le format SVG,
  - Opera a commencé à prendre en charge le SVG dans sa version 8.0. Le support SVG d'Opera 8.5 inclut maintenant des animations.

Structure d'un fichier SVG

- **A- Prologue:**
- **Un fichier SVG commence par une déclaration de version XML standard, comme par exemple**
  - **<?xml version="1.0“ encoding="ISO-8859-1" ?>**
  - **Il faut alors ajouter le "DocType" correspondant à la version SVG utilisée ;**
  - **<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN" "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">**
  - **Il est cependant préférable, en cours de développement, de copier localement la DTD de SVG, et d'y accéder en local :**
  - **<!DOCTYPE svg SYSTEM "svg10.dtd">**

Structure d'un fichier SVG

- **B. Elément racine**
- **La racine d'un fichier SVG est un élément <svg>. Mais il est nécessaire de définir deux "espaces de noms", un par défaut et un second permettant d'avoir accès à d'autres fonctionnalités que nous verrons par la suite, comme suit ;**
  - ```
    <svg xmlns="http://www.w3.org/2000/svg"
    xmlns:link="http://www.w3.org/1999/xlink">
    ```
    - ```
      (...)
      ```
  - ```
    </svg>
    ```
  - **La taille de la fenêtre SVG est définie par les attributs width et height de l'élément svg**
  - ```
    <svg width="400" height="250"
    xmlns="http://www.w3.org/2000/svg">
    ```

Structure d'un fichier SVG

- **Imbrication d'un fichier SVG dans HTML**
  - **La version canonique (déconseillée avec le visualisateur d'Adobe) demande d'utiliser la balise <embed>, sous la forme**
    - `<embed src="..." width="..." height="..." type="image/svg+xml">`
    - **mais elle ne marche pas parfaitement.**
  - **La solution la plus souple d'emploi reste d'utiliser un environnement <iframe>. Par exemple :**
  - ```
    <iframe src="fichier.svg" height="540" width="95%" frameborder="0">
      (Contenu alternatif: image+texte, texte seulement...)
    </iframe>
    ```
    ```

Eléments graphiques de base

- **SVG définit un certain nombre d'éléments graphiques de base.Voici la liste des éléments les plus importants :**
  - **Texte avec <text> ;**
  - **Rectangles <rect> ;**
  - **Le cercle <circle> et l'ellipse <ellipse> ;**
  - **Lignes <line> et poli-lignes <polyline> ;**
  - **Polygones <polygon>;**
  - **Formes arbitraires avec <path>.**

Mécanismes principaux

- **a. Attributs**
  - Il faut se référer à la spécification pour connaître tous les détails. Ici, nous ne montrons en règle générale qu'un petit extrait, car leur nombre est énorme !
  - La plupart des éléments se partagent un nombre commun d'attributs comme par exemple l'attribut id (identificateur) ou encore style (les propriétés de style suivent les règles du CSS2).
  - La plupart des valeurs d'attributs sont assez intuitives
- **b. Positionnement et unités**
  - Les objets SVG se positionnent dans un système de coordonnées qui commence en haut et à gauche (pratique standard en graphisme informatique). Il est possible de travailler avec des coordonnées locales.
- **c. Transformations**
  - Chaque objet peut être translaté, orienté et changé de taille. Il hérite des transformations de l'objet parent.

Le rendu

- **SVG définit quelques dizaines d'attributs-propriétés applicables à certains éléments. En ce qui concerne les éléments graphiques, voici les deux plus importants :**
  - **stroke, définit la forme du bord d'un objet ;**
  - **fill, définit comment le contenu d'un objet est rempli.**
- **SVG possède deux syntaxes différentes pour définir la mise en forme d'un élément :**
  - **L'attribut style reprend la syntaxe et les styles de CSS2 ;**
  - **Pour chaque style, il existe aussi un attribut de présentation SVG, cela simplifie la génération de contenus SVG avec XSLT.**
- **Exemple :**
  - ```
    <rect x="200" y="100" width="60" height="30" fill="red"
    stroke="blue" stroke-width="3" />
    ```
- **Le code précédent a le même effet que ;**
  - ```
    <rect x="200" y="100" width="60" height="30"
    style="fill:red;stroke:blue;stroke-width:3" />
    ```

Le rendu : Propriété fill

- **fill permet de gérer le remplissage de l'élément. Ses propriétés sont :**
  - la couleur, avec les mêmes conventions de représentation qu'en CSS (exemple précédent : fill="red").
  - l'URI d'une couleur, d'un gradient de couleur (pour obtenir un effet de dégradé) ou d'un motif de remplissage.
  - une valeur d'opacité (opacity), comprise entre 0 et 1.

Le rendu : Propriété stroke

- **stroke permet de gérer l'entourage d'un élément de dessin. Ses propriétés sont :**
  - la couleur, avec les mêmes conventions de représentation qu'en CSS
  - l'uri d'une couleur, d'un gradient de couleur (pour obtenir un effet de dégradé) ou d'un motif de remplissage.
  - une valeur d'opacité (opacity), comprise entre 0 et 1.
  - l'épaisseur (width) du trait ;
  - la jonction de ligne (linejoin)
  - la forme des angles (linecap) qui peut être butt(les lignes s'arrêtent brutalement à leur fin), round ou square (des carrés sont tracés en bout de chaque ligne).

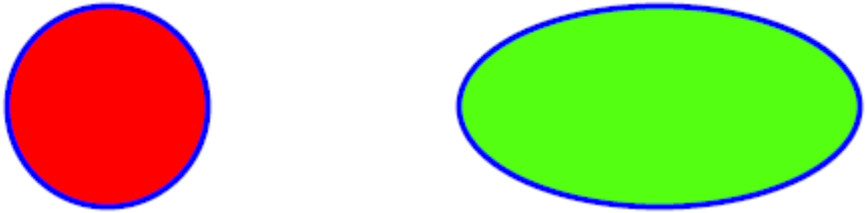
Figures géométriques (Rectangle)

- **L'élément <rect> permet de définir des rectangles, y compris avec des coins arrondis sans passer par une modification de l'attribut stroke-linejoin.**
- **Les attributs de base sont :**
  - **x et y, qui donnent la position du coin supérieur gauche.**
  - **width et height, qui permettent de définir respectivement largeur et hauteur du rectangle.**
  - **rx et ry, qui sont les axes x et y de l'ellipse utilisée pour arrondir ; les nombres négatifs sont interdits, et on ne peut dépasser la moitié de la largeur (longueur) du rectangle.**

Exemple:

```
<rect width="100" height="50" stroke="blue" stroke-width="2"
        fill="yellow" x="100" y="20" rx="10" ry="10"/>
```

Figures géométriques (Cercle)



- **Un cercle est créé par l'élément <circle> et une ellipse par l'élément... <ellipse>.**
- **Leurs attributs sont :**
  - **cx et cy qui définissent les coordonnées du centre.**
  - **r qui définit le rayon du cercle.**
  - **rx et ry qui définissent les demi-axes x et y de l'ellipse.**

Exemples:

```
<circle r="50" cx="300" cy="300" fill="red" stroke="blue" stroke-width="3"/>
<ellipse rx="100" ry="50" cx="350" cy="60" fill="#55FF11"/>
```

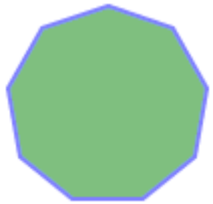
Figures géométriques (Lignes)

- **Une ligne est définie avec l'élément <line>,**
- **une poly-ligne par l'élément <polyline>.**
- **Les attributs de <line> sont :**
  - **x1 et y1, qui définissent les coordonnées du point de départ.**
  - **x2 et y2, qui définissent les coordonnées du point d'arrivée.**
- **L'attribut de base de <polyline> est :**
  - **points, qui prend comme valeur une suite de coordonnées.**

Exemples:

```
<line x1="150" y1="350" x2="250" y2="450" stroke="black" stroke-width="4"/>
<polyline points="150,160,250,200,300,130,350,200" fill="none"/>
```

Figures géométriques (Polygones)



- **Un polygone est une courbe fermée,**
- **Une poly-ligne une courbe ouverte.**
- **L'élément permettant de définir un polygone est <polygon>.**
- **L'attribut de base de cet élément est:**
  - **points, qui s'utilise de la même manière qu'avec une polyligne.**

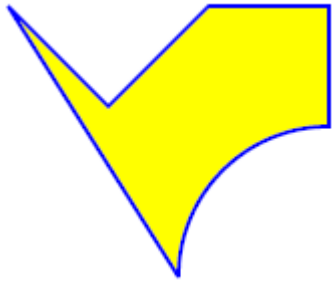
Exemples:

```
<polygon
points="350,175,382,186,399,216,393,250,367,271,332,271,
          306,250,300,216,317,186,350,175"
      opacity="0.5" fill="green" stroke="blue"/>
```

Formes arbitraires (Path)

- **a. Introduction**
  - **L'élément <path> permet de définir des formes arbitraires. Ces formes peuvent avoir un contour et servir de support pour d'autres éléments**
- **b. Attributs de base**
  - **Ces attributs sont au nombre de 2 :**
    - **d, au nom peu explicite, sert à définir les *path data*, autrement dit la liste de commande permettant de tracer le chemin.**
    - **nominalLength, facultatif, permet de définir éventuellement la longueur totale du chemin.**

Formes arbitraires (Path)



c. Path data

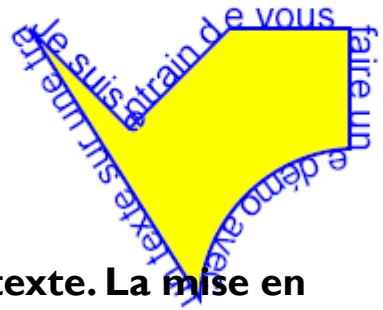
- Les *path data* suivent les règles suivantes :
  - Toutes les instructions peuvent être abrégées en un seul caractère
  - Les espaces peuvent être éliminés
  - On peut omettre de répéter une commande
  - Il existe toujours deux versions des commandes :
    - en majuscules : coordonnées absolues
    - en minuscules : coordonnées relatives
- Ces règles visent à diminuer au maximum la taille requise par la description des chemins. Les commandes sont :
  - **M** ou **m** : (***moveto***) : x,y démarre un nouveau sous-chemin
  - **Z** ou **z** : (***closepath***) ferme un sous-chemin en traçant une ligne droite entre le point courant et le dernier moveto
  - **L** ou **l** : (***lineto***) : x , y trace une ligne droite entre le point courant et le point (x,y).
  - **H** ou **h** : (*horizontal lineto*) : x trace une ligne horizontale entre le point courant et le point (x,y0).
  - **V** ou **v** : (*vertical lineto*) : y trace une ligne verticale entre le point courant et le point (x0,y).
- Il existe également des commandes permettant de tracer des courbes (courbes de Bézier, arcs...).

Exemple:

```
<path d="M 400 400 l 50 50 l 50 -50 h 60 v60 a 80,80 0 0,0 -75,75 z"
id="chemin1"></path>
```

Texte

Regardez SVG



- **A- Balise <text>**
  - **La balise <text> est considérée comme**
    - **un objet graphique, et est donc gérée comme telle.**
  - **Elle possède deux attributs :**
    - **x et y, qui donnent les coordonnées du point de départ du texte. La mise en forme est réalisée par des propriétés de style CSS.**
- **b. Eléments d'ajustement**
  - **A l'intérieur d'un élément <text>, on peut ajuster la position du texte, sa valeur ou la police grâce à l'élément <tspan>. Cet élément possède, outre les attributs x et y, des attributs dx et dy permettant de spécifier les *décalages* à apporter dans l'écriture du texte par rapport à la position par défaut.**
- **c. Lien avec les chemins**
  - **Il est possible d'écrire un texte le long d'un chemin (*path*) défini par ailleurs, par un élément <path> en appelant un élément <textPath>.**

Exemples:

```
<text x="300" y="350" font-size="50" fill="yellow" stroke="blue" stroke-
width="1">Regardez SVG</text>
<text stroke="none" fill="blue" font-size="19">
        <textPath xlink:href="#chemin1">Je suis entrain de vous faire une démo avec un texte sur une trajectoire
        </textPath>
 </text>
```

Structuration: Eléments de groupages et références

- **Chaque langage informatique de haut niveau doit permettre de regrouper des objets dans des blocs, de les nommer et de les réutiliser.**
- **SVG possède plusieurs constructions intéressantes.**
- **Il est aussi intéressant de noter que les objets SVG (comme les objets HTML) héritent le style de leurs parents.**
- **Autrement dit, les styles "cascadent" le long de l'arborescence.**
- **Voici la liste des éléments les plus importants:**
  - **Le fragment d'un document SVG : <svg> ;**
  - **Groupage d'éléments avec <g> ;**
  - **Objets abstraits <symbol> ;**
  - **Section de définition <def> ;**
  - **Utilisation d'éléments <use> ;**
  - **Titre <title> et description <desc>.**

Structuration: Eléments de groupages et références

- 1- Le fragment d'un document SVG: <svg>
  - <svg> est la racine d'un graphisme SVG. Mais on peut aussi imbriquer des éléments svg parmi d'autres et les positionner.
  - Chaque <svg> crée un nouveau système de coordonnées. Ainsi on peut facilement réutiliser des fragments graphiques sans devoir modifier des coordonnées.
- 2- Groupage d'éléments avec <g>
  - Cet élément sert à regrouper les éléments graphiques, Notez l'héritage des propriétés, mais aussi leur redéfinition locale est possible.

Structuration: Eléments de groupages et références

- **3. Objets abstraits avec <symbol>, <defs> et <use>**
  - **a. Symboles (élément <symbol>)**
    - Cet élément permet de définir des objets graphiques réutilisables en plusieurs endroits, avec l'élément **<use>**.
    - **<symbol>** ressemble à **<g>**, sauf que l'objet lui-même n'est pas dessiné.
    - Cet élément possède des attributs supplémentaires
  - **b. Définitions (élément <defs>)**
    - Cet élément ressemble un peu à **<symbol>**, mais est plus simple. De même, l'objet défini n'est pas dessiné.
  - **c. Utilisation d'objets: la balise <use>**
    - **<use>** permet de réutiliser les objets suivants : **<svg>**, **<symbol>**, **<g>**, éléments graphiques et **<use>**.
    - Cet élément se comporte légèrement différemment selon le type d'objet défini. Il s'agit donc d'un instrument de base pour éviter de répéter du code.
    - Les objets réutilisés doivent avoir un identificateur XML. Par exemple, **<rect id="rectanglerouge" fill="red" width="20" height="10"/>**.
    - Les attributs **x**, **y**, **width** et **height** permettent de redéfinir la taille et la position de l'élément appelé.
    - **xlink:href** permet de renvoyer à l'élément défini.

Structuration: Eléments de groupages et références

- **4. Titre <title> et descriptions <desc>**
  - **Ces éléments permettent de documenter le code.**
  - **Ils ne sont pas affichés tels quels. En revanche, un "client" peut par exemple décider de les afficher comme des infobulles.**
  - **Ces documentations peuvent être utiles pour principalement deux raisons :**
    - **éventuellement mieux comprendre le code (même s'il est bien sûr toujours possible de le faire *via* des commentaires <!-- (...) --!>),**
    - **mais aussi, et surtout, permettre un meilleur référencement du SVG par un moteur de recherche.**

Insertion d'images : <image>

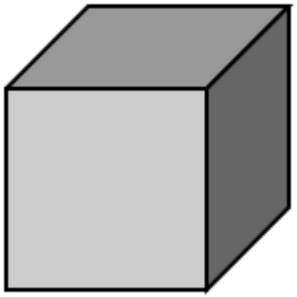
- **Les formats supportés sont jpeg et png.**
- **La balise <image> permet également d'insérer un autre fichier SVG.**
- **Les attributs sont :**
  - **x et y, qui définissent la position de l'image ;**
  - **width et height, qui définissent la taille de l'image ;**
  - **xlink:href indique l'URI de l'image (équivalent de l'attribut src de la balise <img> en HTML.**
  - **Cet élément possède également un attribut supplémentaire, preserveAspectRatio qui permet de définir la manière dont l'affichage de l'image doit s'adapter à son cadre**

Animation dans SVG

- Nous avons vu précédemment ce qu'il était possible de réaliser en SVG, uniquement de manière statique.
- On peut cependant ajouter de l'interactivité à un fichier SVG

Animation de base :<animate>

- Création d'un graphique de base:
-
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE svg SYSTEM "svg10.dtd">
<svg width="500" height="500" xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
    <symbol id="cube" stroke="black" stroke-line-join="bevel" stroke-width="2">
      <rect width="100" height="100" fill="#ccc" x="1" y="42" />
      <polygon points="1,42 42,1 142,1 101,42 1,42" stroke-width="2" fill="#999" />
      <polygon points="101,42 142,1 142,101 101,142 101,42" fill="#666" />
    </symbol>
  </defs>
  <use xlink:href="#cube" x="150" y="150" />
</svg>
```



Animation de base :<animate>

- Animation d'un attribut:
- **<use xlink:href="#cube" x="150" y="150">**

```
<animate
 attributeName="y"
 dur="2s"
 values="150; 140; 130; 120; 110; 100; 110; 120; 130; 140; 150"
  repeatCount="5"/>
</use>
```

- **Il est possible de commencer ou finir à des instants déterminés (attributs begin ou end), mais aussi de synchroniser différentes animations.**

Animation de base :<animate>

- Utilisation de la souris:
  - Il est facile d'insérer un comportement lié à la souris. Par exemple, begin="mouseover" déclenche l'animation dès que l'on survole l'élément par la souris.
  - Nous allons plutôt créer un petit bouton, afin de tester ce genre de comportement.
  - Il suffit de créer un rectangle, que l'on identifie par un id="bouton", par exemple, et d'écrire dans notre élément <animate> un begin="bouton.click", pour obtenir l'effet voulu.

Animation de base :<animate>

- Utilisation de la souris:
  - <?xml version="1.0" encoding="UTF-8"?>
  - <!DOCTYPE svg SYSTEM "../svg10.dtd">
  - <svg width="500" height="500" xmlns:xlink="http://www.w3.org/1999/xlink">
  - <defs>
  - <symbol id="cube" stroke="black" stroke-linejoin="bevel" stroke-width="2">
  - <rect width="100" height="100" fill="#ccc" x="1" y="42"/>
  - <polygon points="1,42 42,1 142,1 101,42 1,42" fill="#999"/>
  - <polygon points="101,42 142,1 142,101 101,142 101,42" fill="#666"/>
  - </symbol>
  - </defs>
  - **<rect width="80" height="15" stroke="black" fill="#0ee" id="bouton"/>**
  - **<text x="10" y="12" stroke="black" stroke-width="1" id="textbouton">Lancer!</text>**
  - <use xlink:href="#cube" x="150" y="150">
  - <animate attributeName="y" dur="2s" values="150; 140; 130; 120; 110; 100; 110; 120; 130; 140; 150" **begin="bouton.click"**/>
  - </use>
  - </svg>

Animation de base :<animate>

- Plus de contrôle:
  - **a. Attributs from et to**
    - Ces deux attributs permettent de ne pas avoir à spécifier une liste de valeurs
    - On peut par exemple ajouter **<animate attributeName="x" dur="3s" from="150" to="100"/>** à l'animation précédente pour modifier également la position horizontale.

Animation de base :<animate>

- **b. Figer une animation**
  - Pour figer l'animation dans son état final, on utilise l'attribut **fill**, avec la valeur **freeze** (geler, en anglais). On obtient ainsi le résultat visible sur l'exemple suivant...
  - <animate
    attributeName="x"
    dur="3s"
    from="150"
    to="100"
    fill="freeze"/>

Animation de base :<animate>

- **C. Répéter une animation**
  - **On peut également demander à ce qu'une animation se répète un nombre déterminé de fois, à l'aide de l'attribut repeatCount.**
  - **Cet attribut peut prendre comme valeur un nombre entier, ou bien indefinite, qui permet de boucler à l'infini.**
  - ```
    <animate
    attributeName="y"
    dur="3s"
    values="150; 140; 130; 120; 110; 100; 110; 120;
      130; 140; 150"
    repeatCount="indefinite"/>
    ```
    ```

Animations plus complexes

- **1. Changements de couleurs : l'élément <animateColor>**
  - **La couleur nécessite un traitement séparé.**
  - **Il est réalisé à l'aide de l'élément <animateColor>, mais les attributs de base restent les mêmes.**
  - **La couleur à modifier doit avoir été fixée lors de l'appel à l'élément, pas lors de la définition**
  - **<animateColor**
    attributeName="fill"
    attributeType="CSS"
    values="#ccc;#000"
    dur="3s" />

Animations plus complexes

- **2. Rotations, mises à l'échelle et translations**
  - **La syntaxe de l'attribut transform ne se prête pas à l'utilisation de l'élément <animate>.**
  - **Il a été nécessaire de développer un autre élément, <animateTransform>.**
  - **Il faut renseigner l'attribut attributeName par la valeur transform.**
  - **Cet élément utilise un attribut, type, qui lui permet de déterminer de quel type de transformation il va s'agir.**

Animations plus complexes

- **Le code suivant permet de décaler dans la direction des x et des y positifs l'élément à animer:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg SYSTEM "../svg10.dtd">
<svg width="500" height="500" xmlns:xlink="http://www.w3.org/1999/xlink">
   <defs>
      <rect width="100" height="100" id="motif"/>
   </defs>
   <use xlink:href="#motif" x="100" y="100" fill="lime">
      <animateTransform attributeName="transform" type="translate"
   from="0,0" to="100,100" dur="3s"/>
   </use>
</svg>
```

Animations plus complexes

- **Pour la rotation:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg SYSTEM "../svg10.dtd">
<svg width="500" height="500"
   xmlns:xlink="http://www.w3.org/1999/xlink">
   <defs>
      <rect width="100" height="100" id="motif"/>
   </defs>
   <use xlink:href="#motif" x="100" y="100" fill="lime">
      <animateTransform
                  attributeName="transform"
                  type="rotate"
                  from="0,100,100"
                  to="360,100,100"
                  dur="3s"/>
   </use>
</svg>
```

Animations plus complexes

- **3. Superpositions d'effets**
  - **Par défaut, à chaque fois que l'on ajoute une animation, celle-ci écrase celles qui ont été définies avant.**
  - **Afin d'éviter ce comportement, il faut renseigner l'attribut additive avec la valeur sum (addition).**
  - Exemple:

```
<use xlink:href="#rectangle" x="200" y="200">
  <animate attributeName="x" begin="bouton.click" dur="3s" to="50"
   repeatCount="indefinite" />
   <animateTransform attributeName="transform" type="rotate" from="0,220,200"
   to="720,220,200" dur="5s" additive="sum" repeatCount="indefinite"/>
   <animateTransform attributeName="transform" type="translate"
   values="0,0;0,100;0,0" additive="sum" dur="5s" repeatCount="indefinite"/>
</use>
```

Exemple de document SVG

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg SYSTEM "svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.0">
  <symbol id="logo">
    <rect width="200" height="100" x="10" y="10" stroke="blue" stroke-width="3" fill="yellow"></rect>
   <circle r="50" cx="100" cy="100" stroke="green" fill="red" stroke-width="3" fill-opacity="0.5"></circle>
  </symbol>
   <g id="logo2">
    <use xlink:href="#logo" x="100" y="100"></use>
     <use xlink:href="#logo" x="200" y="200"></use>
     <use xlink:href="#logo" x="100" y="300"></use>
   </g>
   <use xlink:href="#logo2" x="10" y="10">
      <animateTransform attributeName="transform" type="rotate" dur="3s" begin="1s" from="0,100,100"
          to="360,300,300" repeatCount="indefinite"></animateTransform>
   </use>
</svg>
```

XSL- SVG

- On considère un document XML qui présente les mesures des températures des différentes villes à une date donnée.
- Chaque mesure est qualifiée par une date et formée par une ensemble de villes
- Chaque ville est qualifiée par un nom et une température.
- Travail à faire:

1-Faire une représentation graphique de l'arbre XML

2-Créer une DTD

3- Créer un schéma XML

4- Créer le document XML

5- Créer une feuille de style qui permet d'afficher une page HTML

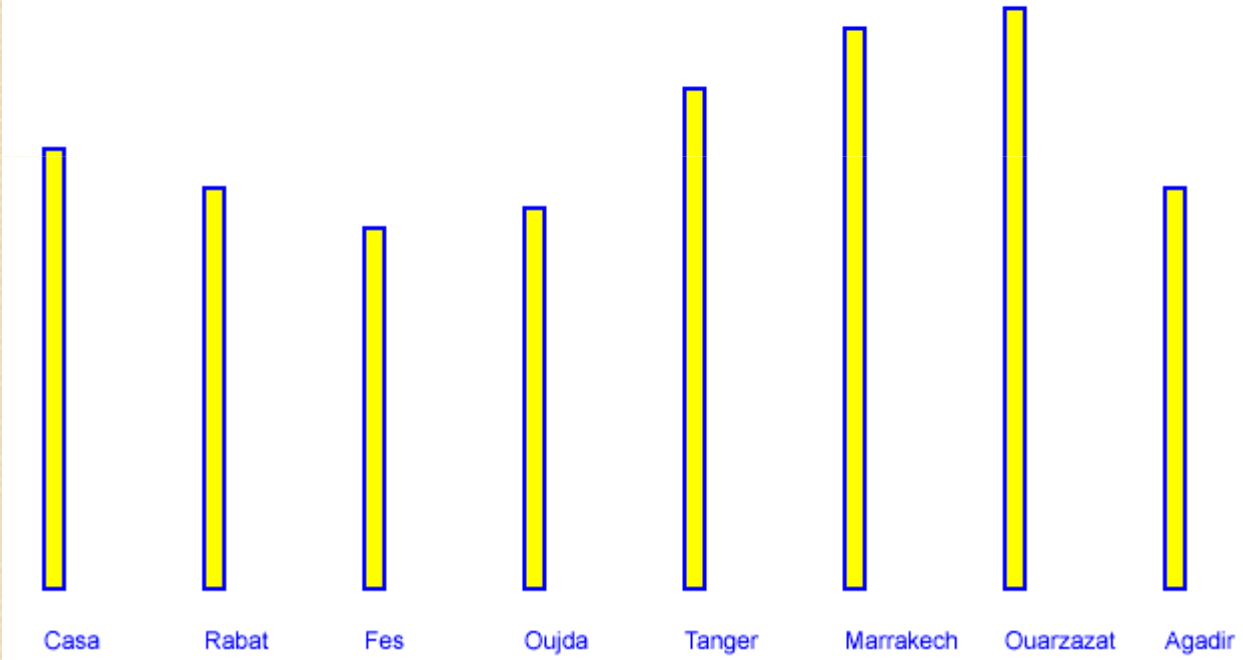
6- Créer une feuille de style qui permet de transformer ce document en un document SVG (voir fig.1)

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

<meteo>
   <mesure date="2006-1-1">
      <ville nom="Casa" temperature="22"/>
      <ville nom="Rabat" temperature="20"/>
      <ville nom="Fes" temperature="18"/>
      <ville nom="Oujda" temperature="19"/>
      <ville nom="Tanger"  temperature="25"/>
      <ville nom="Marrakech"  temperature="28"/>
      <ville nom="Ouarzazat"  temperature="29"/>
      <ville nom="Agadir"  temperature="20"/>
   </mesure>
   <mesure date="2006-1-2">
      <ville nom="Casa" temperature="21"/>
      <ville nom="Rabat" temperature="23"/>
      <ville nom="Fes" temperature="19"/>
      <ville nom="Oujda" temperature="20"/>
      <ville nom="Tanger"  temperature="23"/>
      <ville nom="Marrakech"  temperature="27"/>
      <ville nom="Ouarzazat"  temperature="25"/>
      <ville nom="Agadir"  temperature="23"/>
   </mesure>
</meteo>
```

Fig2:Résultat de transformation de la deuxième feuille de style

Températures à la date : 2006-1-1



Feuille de style XSL

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="xml" media-type="image/svg+xml" indent="yes" />
  <xsl:template match="/">
     <svg xmlns:xlink="http://www.w3.org/1999/xlink" xmlns="http://www.w3.org/2000/svg">
        <line x1="20" y1="20" x2="20" y2="400" stroke="blue" stroke-width="2"></line>
        <line x1="20" y1="400" x2="800" y2="400" stroke="blue" stroke-width="2"></line>
        <xsl:for-each select="meteo/mesure[@date='2006-1-1']">
           <xsl:for-each select="ville">
              <xsl:variable name="temp" select="@temperature"></xsl:variable>
              <xsl:variable name="H" select="$temp*10"></xsl:variable>
              <xsl:variable name="XR" select="position()*80"></xsl:variable>
              <xsl:variable name="YR" select="400-$H"></xsl:variable>
              <text x="{$XR}" y="420" stroke="blue">
                 <xsl:value-of select="@nom"/>
              </text>
              <rect width="10" height="{$H}" x="{$XR}" y="{$YR}" stroke="blue" stroke-width="2"
                 fill="yellow">
                 <animate attributeName="height" dur="3s" from="0" to="{$H}"></animate>
                 <animate attributeName="y" dur="3s" from="400" to="{$YR}"></animate>
              </rect>
           </xsl:for-each>
        </xsl:for-each>
     </svg>
  </xsl:template>
</xsl:stylesheet>
```