

# SGBD, XML, amis ou ennemis ?

---

***Olivier Perrin***

*IUT Nancy-Charlemagne*

*Département Informatique*

*Université Nancy 2*

*Olivier.Perrin@loria.fr*

# SGBD

---

- Un SGBD est un ensemble de logiciels qui fournit un environnement pour :
  - décrire et structurer
    - mémoriser
    - manipuler
    - traiter des collections de données
  - tout en assurant leur
    - cohérence
    - sécurité
    - confidentialité
    - intégrité
    - indépendance (logique/physique)

# XML

---

- XML = *eXtensible Markup Language*
- XML est un langage pour structurer des contenus et définir une classe d'objets de données, par exemple:
  - un dessin vectoriel (SVG), une page Web (XHTML), un flux (RSS)...
- Le langage est basé sur le concept de balisage des données
- Un document XML:
  - contient des déclarations, éléments, commentaires, définition de caractères spéciaux et instructions (facultatives) de traitement
  - c'est un arbre: il doit avoir une racine et les éléments doivent s'imbriquer proprement
- Plus simple que SGML, plus complexe mais moins confus et plus performant que HTML
- Recommandation officielle du W3C depuis le 10 février 1998
- Idéal pour l'échange de données semi-structurées
- Utilisable entre machines

# XML (2)

---

- XML, c'est donc...
  - un méta-langage universel pour structurer les données...
  - qui permet aux utilisateurs de délivrer du contenu...
  - depuis les applications à d'autres applications (browsers par exemple)
- XML promet de standardiser la manière dont l'information est :
  - échangée (XML)
  - personnalisée/présentée (XSL/CSS)
  - recherchée (XPath/XSLT/XQuery)
  - sécurisée (Encryption, Signature)
  - liée (XLink)

# XML (2)

---

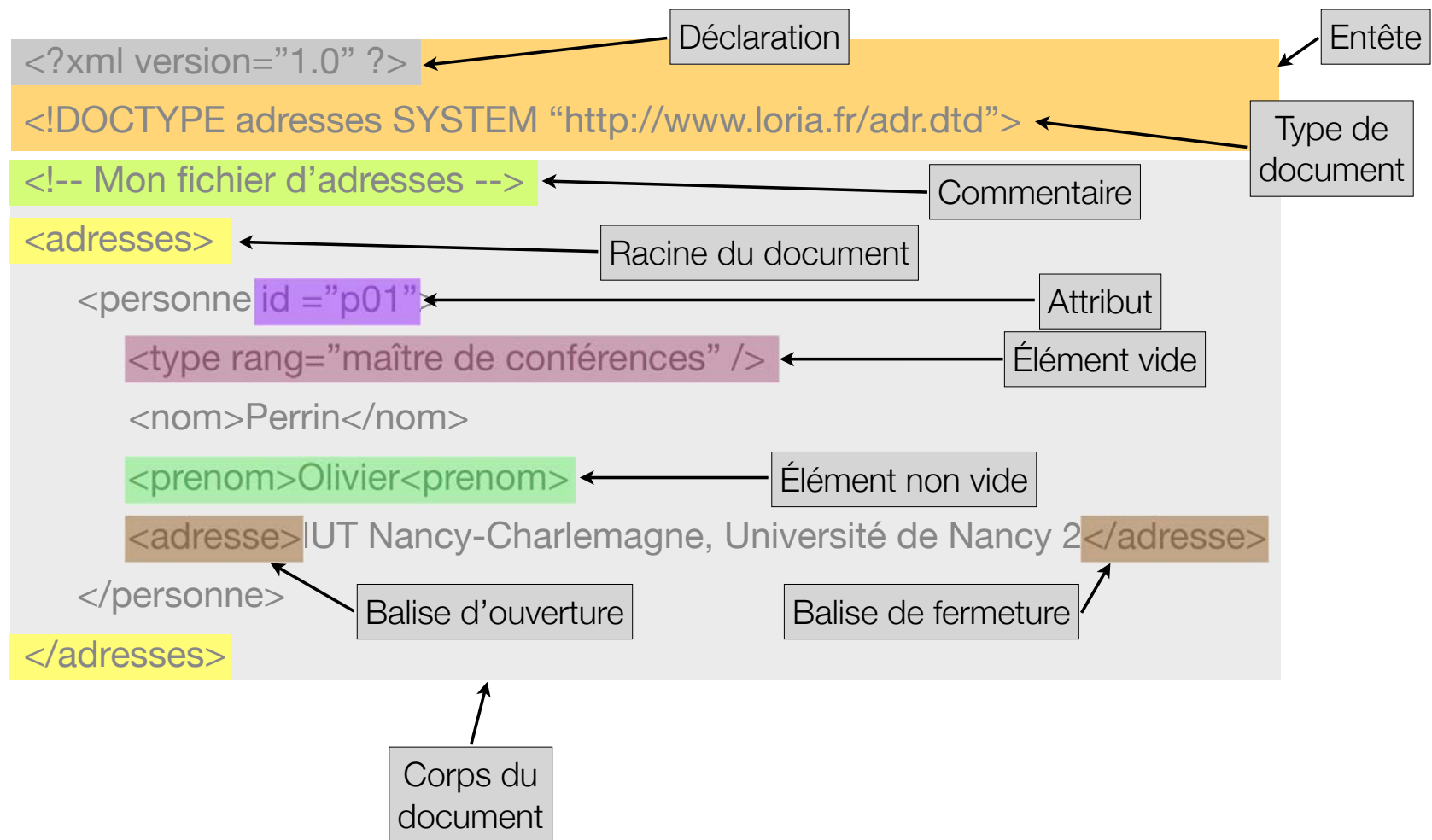
- XML, c'est donc...
  - un méta-langage universel pour structurer les données...
  - qui permet aux utilisateurs de délivrer du contenu...
  - depuis les applications à d'autres applications (browsers par exemple)
- XML promet de standardiser la manière dont l'information est :
  - échangée (XML)
  - personnalisée/présentée (XSL/CSS)
  - recherchée (XPath/XSLT/XQuery)
  - sécurisée (Encryption, Signature)
  - liée (XLink)

# XML (3)

---

- Balise (ou tag ou label)
  - marque de début et fin permettant de repérer un élément de données (textuel)
  - forme: <balise> de début, </balise> de fin
  - les balises indiquent la signification des sections marquées
- Élément de données
  - texte encadré par une balise de début et une de fin
  - les éléments de données peuvent être imbriqués
- Attribut
  - couple nom="valeur" qualifiant une balise
  - <producteur no="160017" region="Bourgogne"/>
- Les utilisateurs définissent leurs propres balises
- Il est possible d'imposer une grammaire spécifique (DTD, Schéma)

# XML: un exemple



# XML: format interne

---

- C'est un arbre



# XML: format interne

---

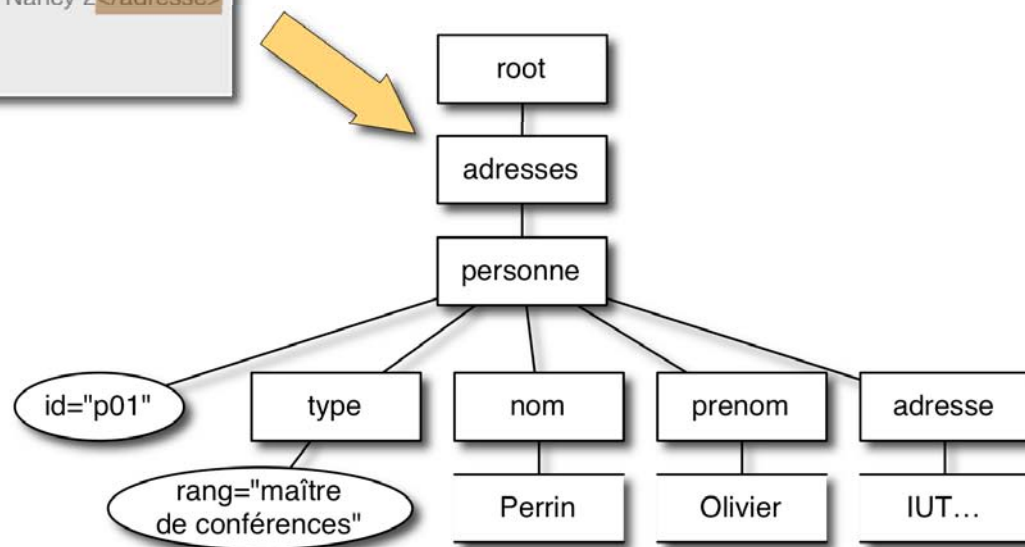
- C'est un arbre

```
<?xml version="1.0" ?>
<!DOCTYPE adresses SYSTEM "http://www.loria.fr/adr.dtd">
<!-- Mon fichier d'adresses -->
<adresses>
  <personne id="p01">
    <type rang="maître de conférences" />
    <nom>Perrin</nom>
    <prenom>Olivier</prenom>
    <adresse>IUT Nancy-Charlemagne, Université de Nancy 2</adresse>
  </personne>
</adresses>
```

# XML: format interne

- C'est un arbre

```
<?xml version="1.0" ?>  
<!DOCTYPE addresses SYSTEM "http://www.loria.fr/adr.dtd">  
<!-- Mon fichier d'adresses -->  
<addresses>  
  <personne id="p01">  
    <type rang="maître de conférences" />  
    <nom>Perrin</nom>  
    <prenom>Olivier</prenom>  
    <adresse>IUT Nancy-Charlemagne, Université de Nancy 2</adresse>  
  </personne>  
</addresses>
```



# Pourquoi XML ?

---

- Définir vos propres langages d'échange
  - commande, facture, bordereau de livraison, etc.
- Modéliser des données et des messages
  - *Document Type Definition* (DTD)
  - types et éléments agrégés (*XML Schema Definition*)
  - passerelle avec *Unified Modelling Language* (UML)
- Publier des informations
  - indépendante du format
  - mise en forme avec CSS et XSL
  - présentation possible en XHTML, PDF, WML,...
- Archiver des données
  - auto-description des archives

# Comparaison Données/Documents

---

- Approche « Donnée »
  - structuration forte et simple
  - compatibilité SGBDR existants
  - mise à jour en place
  - intégrité sémantique
  - indexation exacte
  - adapté au transactionnel et décisionnel
  - performances « moyenne » à « forte » pour une volumétrie « moyenne »
- Approche « Document »
  - structuration faible et complexe
  - systèmes documentaires spécialisés
  - gestion de versions
  - recherche textuelle
  - indexation approchée
  - accès type moteur de recherche
  - performances « moyenne » pour une volumétrie « forte »

# Avantages de XML

---

- Une technologie structurante
- Clarifie toutes les interfaces d'échange
- Transversale à l'entreprise
  - échanges de données
  - bureautique
  - GED
  - sites Web
  - EDI
  - bases de données
  - intégration e-business
  - ...
- Un choix stratégique de direction
  - ne pas rester isolé

# Faiblesses de XML

---

- Une syntaxe verbeuse
- Un méta-langage, mais de nombreux langages
- Coûteux en CPU
  - analyse
- Coûteux en mémoire
  - instanciation

# XML et BD

---

- Intégration des données et méta-données
- Standard d'échange de données universel
- Les BD ne peuvent rester indifférentes :
  - nécessité de stocker les documents XML
  - nécessité de pouvoir interroger ces documents
  - évolution ou révolution ?
- Quel modèle de données ?
- Quel langage d'interrogation ?
- Quelle intégration avec l'existant ?

# Modèles de données

---



# Document XML

---

- Un document XML peut être associé à:
  - une DTD ou un schéma pour décrire la structure du document XML
  - une feuille de style pour présenter les données
  - DTD ou/et schéma permettent de définir son propre langage basé sur XML
  - vocabulaire (balises)
  - grammaire (imbrications)
- Deux types de documents
  - *well-formed document*
  - *valid document*

# Document bien formé (*well-formed*)

---

- Commence par une déclaration XML (attribut version obligatoire) avec possibilité de choisir un encodage (le défaut est utf-8):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

- Structure hiérarchique:

- les balises d'ouverture et de fermeture doivent apparaître et correspondre
- pas de croisements de type `<i>...<b>...</i> .... </b>`
- sensible à la casse: "LI" n'est pas égal à "li" par exemple
- balises "EMPTY" utilisent la syntaxe XML "auto-fermante": `<br/>`
- les valeurs d'attributs sont quotés: `<a href="http://www.foo.fr/xml.html">`
- un seul élément racine (root):
  - l'élément root ne peut apparaître qu'une fois et ne doit pas apparaître dans un autre élément

- Caractères spéciaux (!!): `<`, `&`, `>`, `"`, `'`

- utilisez `&lt;`, `&amp;`, `&gt;`, `&quot;`, `&apos;` à la place dans un texte !
- les espaces sont préservés

# Document valide (*valid*)

---

- Un document “valide” doit être:
  - “well-formed” (formé correctement)
  - être associé à une DTD (ou une autre grammaire)
  - et être conforme à cette DTD ou une grammaire d’un autre type comme XSD (*XML Schema Definition*) ou RelaxNG

# DTD

---

- C'est une grammaire dont les phrases sont des documents XML (instances) qui :
  - permet de définir le «vocabulaire» (définit le jeu de balises utilisables ainsi que leurs attributs)
  - et la structure qui sera utilisée dans le document XML (l'imbrication des balises)
  - possibilité de décrire si les balises sont obligatoires ou optionnelles
- La DTD peut être référencé par un URI ou incluse directement dans le document XML
- Il existe d'autres types de grammaires comme XML Schema (XSD), Relax NG, etc.
  - leur puissance sémantique est plus élevée (on peut exprimer plus de contraintes)
  - Relax NG offre le meilleur rapport puissance/facilité
  - DTD est le plus répandu
  - XML Schema le plus souvent utilisé pour formaliser des langages "webservices", par ex. SOAP

# Déclaration élément simple

---

- <!ELEMENT balise (définition) >
  - le paramètre définition représente
    - soit un type de données prédéfini,
    - soit un type de données composé constitué lui-même d'éléments
- Types prédéfinis
  - ANY: l'élément peut contenir tout type de donnée
  - EMPTY: l'élément ne contient pas de données spécifiques (élément vide)
  - #PCDATA: l'élément doit contenir une chaîne de caractères dans l'encodage courant (non interprétée par XML)
- Exemple

```
<!ELEMENT Nom (#PCDATA)>
<Nom>Victor Hugo</Nom>
```

# Éléments composés

---

- Élément composé d'une séquence ou d'un choix d'éléments
- Syntaxe spécifique avec opérateurs de composition d'éléments:

<!ELEMENT balise (composition) >

A et B	Explication	Exemples
<b>A?</b>	A (un seul) est une option, (donc: A ou rien)	<!ELEMENT personne (nom, email?)
<b>A+</b>	Il faut un ou plusieurs A	<!ELEMENT personne (nom, email+)
<b>A*</b>	A est une option, il faut 0, 1 ou plusieurs A	<!ELEMENT personne (nom, email*)
<b>A   B</b>	Il faut A ou B, mais pas les deux	<!ELEMENT personne (email fax)
<b>A , B</b>	Il faut A suivi de B (dans l'ordre)	<!ELEMENT personne (nom, email ?)
<b>(A, B)+</b>	Les parenthèses regroupent. Ici, un ou plusieurs (A suivi de B)	<!ELEMENT liste (nom, email)+

- Semi-structuré car on peut avoir un *mixed content*
  - (#PCDATA | e1 | ... | en)

# Exemple

---

- DTD

<!ELEMENT personne (nom, prenom+, tel?, adresse >

<!ELEMENT nom (#PCDATA) >

<!ELEMENT prenom (#PCDATA) >

<!ELEMENT tel (#PCDATA) >

<!ELEMENT email (#PCDATA) >

<!ELEMENT Adresse (ANY) >

- Document associé

<personne>

<nom>Hugo</nom>

<prenom>Victor</prenom>

<prenom>Charles</prenom>

<tel>0383000000</tel>

<adresse><rue/><ville>Paris</ville></adresse>

</personne>

# Attributs

---

- `<! ATTLIST balise attribut type mode >`
  - *balise* spécifie l'élément auquel est attaché l'attribut
  - *attribut* est le nom de l'attribut déclaré
  - *type* définit le type de donnée de l'attribut choisi parmi:
    - CDATA: chaînes de caractères entre guillemets ("aa") non analysées
    - Enumération: liste de valeurs séparées par |  
`<! ATTLIST balise Attribut (Valeur1 | Valeur2 | ... ) >`
    - NMTOKEN: un seul mot
    - ID et IDREF: clé et référence à clé
  - *mode* précise le caractère obligatoire ou non de l'attribut
    - #REQUIRED: obligatoire (l'attribut doit être valué)
    - #IMPLIED: optionnel (l'attribut peut être valué)
    - #FIXED valeur: attribut avec valeur fixe (valeur fixée dans la DTD)



# Exemple

---

<!ELEMENT personne (nom, prenom+, tel?, adresse >

<!ELEMENT nom (#PCDATA) >

<!ELEMENT prenom (#PCDATA) >

<!ELEMENT tel (#PCDATA) >

<!ELEMENT email (#PCDATA) >

<!ELEMENT Adresse (ANY) >

<! ATTLIST personne

num ID,

age CDATA,

genre (Masculin | Feminin ) >

<!ELEMENT auteur (#PCDATA) >

<!ATTLIST auteur

genre (Masculin | Feminin ) #REQUIRED

ville CDATA #IMPLIED>

<!ELEMENT editeur (#PCDATA) >

<!ATTLIST editeur

ville CDATA #FIXED "Paris">

# Attributs vs. éléments

---

- Il s'agit ici une FAQ classique sans réponse précise...
- Il faut plutôt utiliser un élément
  - lorsque l'ordre est important (l'ordre des attributs est au hasard)
  - lorsqu'on veut réutiliser un élément plusieurs fois (avec le même parent)
  - lorsqu'on veut (dans le futur) avoir des descendants / une structure interne
  - pour représenter un type de données (objet) plutôt que son usage, autrement dit: une "chose" est un élément et ses propriétés sont des "attributs"
- Il faut plutôt utiliser un attribut
  - lorsqu'on désire faire référence à un autre élément  
`<compagnon genre="giraffe">` fait référence à `<animal cat="giraffe">`
  - pour indiquer l'usage/type/etc. d'un élément  
`<adresse usage="prof"> ... </adresse>`
  - lorsque vous voulez imposer des valeurs par défaut dans la DTD
  - lorsque vous voulez un type de données (pas grand chose dans la DTD)

# Exemple de DTD

---

<!ELEMENT carnetAdresses (personne)+>

<!ELEMENT personne (nom,email\*)>

<!ELEMENT nom (famille,prenom)>

<!ELEMENT famille (#PCDATA)>

<!ELEMENT prenom (#PCDATA)>

<!ELEMENT email (#PCDATA)>

```
<carnetAdresses>
  <personne>
    <nom><famille>Perrin</famille><prenom>Olivier</prenom></nom>
    <email>Olivier.Perrin@loria.fr</email>
  </personne>
  <personne>
    <nom><famille>Lagrange</famille><prenom>Anne</prenom></nom>
    <email>Anne.Lagrange@mozilla.org</email>
  </personne>
</carnetAdresses>
```

# ID et IDREF

---

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE Document [
  <!ELEMENT Document (Personne*)>
  <!ELEMENT Personne (#PCDATA)>
  <!ATTLIST Personne PNum ID #REQUIRED>
  <!ATTLIST Personne Mere IDREF #IMPLIED>
  <!ATTLIST Personne Pere IDREF #IMPLIED>
]>
< Document >
  < Personne PNum = "P1">Marie</PERSONNE>
  < Personne PNum = "P2">Jean</PERSONNE>
  < Personne PNum = "P3" Mere ="P1" Pere ="P2">Pierre</PERSONNE>
  < Personne PNum = "P4" Mere ="P1" Pere ="P2">Julie</PERSONNE>
</Document >
```

# Association DTD/document XML

---

- Il existe 4 façons d'utiliser une DTD
  - 1. On déclare pas de DTD (dans ce cas le fichier est juste "bien formé")
  - 2. On déclare la DTD et on y ajoute les définitions dans le fichier (DTD interne)
  - 3. On déclare la DTD en tant que DTD "privé": la DTD se trouve quelque part dans votre système ou sur Internet (répandu pour les DTDs "faits maison")
  - 4. On déclare une DTD "public". On utilise un nom officiel pour la DTD. Cela présuppose que votre éditeur et votre client connaissent cette DTD (répandu pour les DTDs connues comme XHTML, SVG, MathML, etc.)
- Lieu de la déclaration
  - la DTD est déclarée entre la déclaration de XML et le document lui-même
  - la déclaration de XML et celle de la DTD font parti du prologue (qui peut contenir d'autres éléments comme les *processing instructions*)
  - attention: l'encodage de la DTD doit correspondre à celui des fichiers XML !

## Association DTD/document XML (2)

---

- Syntaxe de la déclaration
  - chaque déclaration de la DTD commence par `<!DOCTYPE ...` et fini par `>`
  - la racine de l'arbre XML (ici: `<hello>`) doit être indiquée après `<!DOCTYPE`
  - syntaxe pour définir une DTD interne (seulement !)
    - la DTD sera insérée entre `[ ... ]`

```
<!DOCTYPE hello [  
  <!ELEMENT hello (#PCDATA)>  
]>
```
  - syntaxe pour définir une DTD privée externe:
    - la DTD est dans l'URL indiqué après le mot clef "SYSTEM"

```
<!DOCTYPE hello SYSTEM "hello.dtd">
```
    - déclaration de la DTD dans le fichier XML et PAS dans le fichier \*.dtd.
- Définition de la racine de l'arbre
  - le mot "hello" après le mot clef DOCTYPE indique que "hello" est l'élément racine de l'arbre XML

```
<!DOCTYPE hello SYSTEM "hello.dtd">
```

## Association DTD/document XML (3)

---

- Hello XML sans DTD

```
<?xml version="1.0"standalone="yes"?>  
<hello> Hello XML et hello cher lecteur ! </hello>
```

- Hello XML avec DTD interne

```
<?xml version="1.0"standalone="yes"?><!DOCTYPE hello [  
  <!ELEMENT hello (#PCDATA)> ]>  
<hello> Hello XML et hello cher lecteur ! </hello>
```

- Hello XML avec DTD externe

```
<?xml version="1.0" encoding="ISO-8859-1" ?><!DOCTYPE hello SYSTEM "hello.dtd">  
<hello> Hello XML et hello cher lecteur ! </hello>
```

- Un fichier RSS (DTD externe public)

```
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE rss PUBLIC "-//Netscape  
Communications//DTD RSS 0.91//EN" "http://my.netscape.com/publish/formats/  
rss-0.91.dtd">  
<rss version="0.91"> <channel>...</channel></rss>
```

# Concepts additionnels (non développés ici)

---

- Validation (par ex. <http://validator.w3.org/>)
  - analyse le document en entrée (en particulier s'il est *well formed*)
  - vérifie l'élément racine
  - pour chaque élément, vérifie le contenu et les attributs
  - vérifie l'unicité et les contraintes référentielles (attributs ID/IDREF(S))
- Entité
  - une "entity" est un bout d'information stocké quelque part
  - les entités d'un document sont remplacées par le contenu référencé (macro)
  - distinction entre entités générales et entités paramétrées
- Espaces de noms
  - on peut dans un même document de mélanger plusieurs grammaires (si l'application le permet), par exemple: XHTML + Svg + MathML + XLink
  - pour éviter qu'il y ait confusion entre différentes balises, on définit un "namespace" pour chaque grammaire et on l'utilise pour les éléments (soit pour tout l'arbre, soit pour une partie seulement)



# Exercice

---

- Un carnet d'adresses plus complet
  - la personne possède un identifiant unique (obligatoire), un nom, un prénom
  - on veut connaître le sexe de la personne (attribut optionnel)
  - on veut connaître son email (optionnel)
  - on veut connaître le lien entre le chef et ses employés

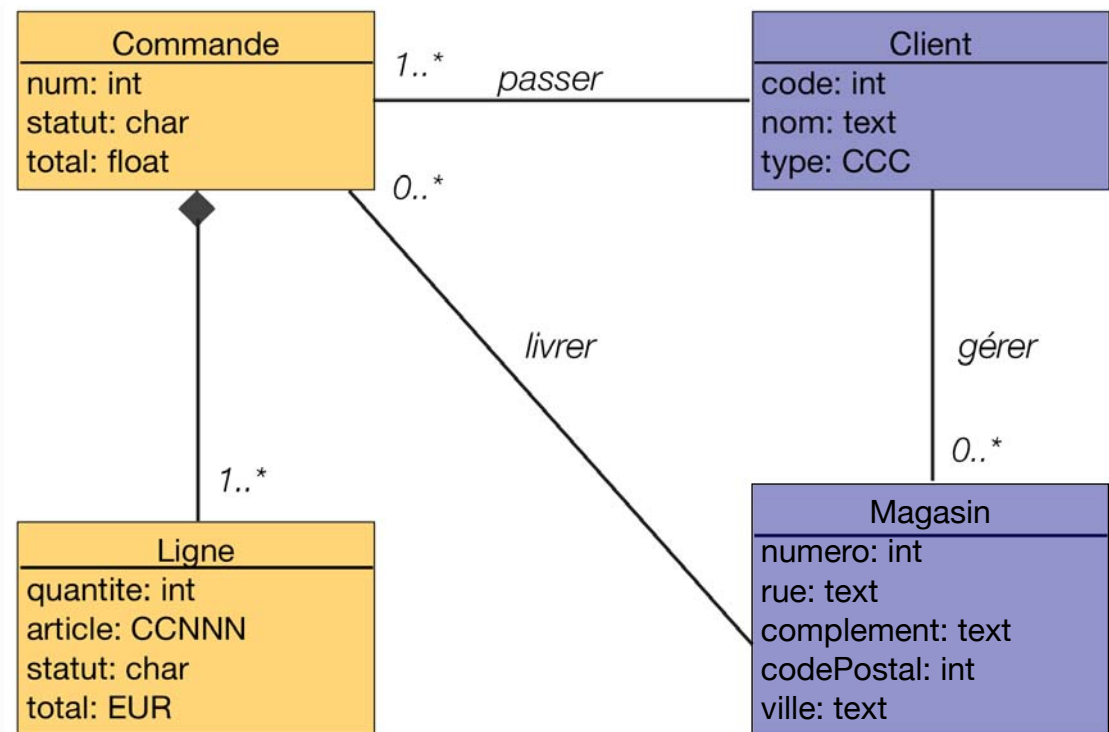
# Exercice

---

- Un carnet d'adresses plus complet
  - la personne possède un identifiant unique (obligatoire), un nom, un prénom
  - on veut connaître le sexe de la personne (attribut optionnel)
  - on veut connaître son email (optionnel)
  - on veut connaître le lien entre le chef et ses employés

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT carnetAdresses (personne)+>
<!ELEMENT personne (nom,email*)>
  <!ATTLIST personne id ID #REQUIRED>
  <!ATTLIST personne sexe (masculin|feminin) #IMPLIED>
<!ELEMENT nom (#PCDATA|famille|prenom)*>
<!ELEMENT famille (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT lien EMPTY>
  <!ATTLIST lien chef IDREF #IMPLIED employes IDREFS #IMPLIED>
```

# Passage UML vers DTD



## Passage UML vers DTD (2)

---

```
<!-- Types de base-->
<!ENTITY % int "(#PCDATA)">
<!ENTITY % float "(#PCDATA)">
<!ENTITY % char "(#PCDATA)">
<!ENTITY % string "(#PCDATA)">
<!-- Classe Commande -->
<!ELEMENT Commande (cstatut, ctotat, Ligne+)>
<!ATTLIST Commande NUM ID #REQUIRED>
<!ELEMENT cstatut %char;>
<!ELEMENT ctotat %float;>
<!-- Classe Ligne -->
<!ELEMENT Ligne (article, quantite, statut?, total?)>
<!ELEMENT article %string;>
<!ELEMENT quantite %int;>
<!ELEMENT lstatut %char;>
<!ELEMENT ltotal %float;>
```

# Le document XML associé

---

```
<?xml version="1.0" ?>
<message>
  <Commande num="1">
    <cstatut>A</cstatut>
    <ctotal>1000</ctotal>
    <Ligne>
      <article>212</article>
      <quantite>100</quantite>
    </Ligne>
  </Commande>
  <Commande num="2">
    <cstatut>B</cstatut>
    <ctotal>1000</ctotal>
    <Ligne>
      <article>212</article>
      <quantite>300</quantite>
    </Ligne>
    <Ligne>
      <article>312</article>
      <quantite>400</quantite>
    </Ligne>
  </Commande>
</message>
```

# Faiblesse des DTDs

---

- Pas de types de données
  - difficile à interpréter par le récepteur (indépendant du contexte)
  - pas d'expressions rationnelles pour les valeurs
  - spécification des valeurs d'attributs simpliste
  - support pour la modularité et la réutilisation limité
  - difficile à traduire en schéma objets
- Pas en XML
  - langage spécifique
- Propositions de compléments
  - XML Schema Definition du W3C
  - Relax NG

# XML Schema

---

- Un schéma d'un document définit:
  - les éléments possibles dans le document
  - les attributs associés à ces éléments
  - la structure du document
  - les types de données
- Le schéma est spécifié en XML
  - pas de nouveau langage
  - balisage de déclaration
  - espace de nom spécifique xsd: ou xmlns:
- Présente de nombreux avantages
  - structures de données avec types de données (expressivité accrue)
  - extensibilité par héritage et ouverture
  - analysable à partir d'un parseur XML standard (c'est du XML)

# Objectifs

---

- Reprendre les acquis des DTD
  - plus riche et complet que les DTD
- Permettre de typer les données
  - éléments simples et complexes
  - attributs simples
- Permettre de définir des contraintes
  - existence, obligatoire, optionnel
  - domaines, cardinalités, références
  - patterns, ...
- S'intégrer à la galaxie XML
  - espace de noms
  - infoset (structure d'arbre logique)



# Modèle des schémas

---

- Déclaration des éléments et attributs
  - nom
  - typage similaire à l'objet
- Spécification de types simples
  - grande variété de types
- Génération de types complexes
  - *sequence*
  - *choice*
  - *all*

# Types simples

---

- string, normalizedString, token
- byte, unsignedByte
- base64Binary, hexBinary
- integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger, int, unsignedInt
- long, unsignedLong
- short, unsignedShort
- decimal, float, double
- boolean
- time, dateTime, duration, date, gMonth, gYear, gYearMonth, gDay, gMonthDay
- Name, QName, NCName, anyURI
- language
- ID, IDREF, IDREFS
- ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS

# Commandes de base

---

- element: association d'un type à une balise
  - attributs name, type, ref, minOccurs, maxOccurs,...
- attribute: association d'un type à un attribut
  - attributs name, type
- type simple: les multiples types de base
  - entier, réel, string, time, date, ID, IDREF,...
  - extensibles par des contraintes
- type complexe: une composition de types
  - définit une agrégation d'éléments typés

# Exemples de types simples

---

- Exemple

```
<simpleType name="entier0_a_100">  
  <restriction base="integer">  
    <minInclusive value="0"/>  
    <maxInclusive value="100"/>  
  </restriction>  
</simpleType>
```

## Exemple 2

```
<simpleType name="listeEntier">  
  <list itemType="integer"/>  
</simpleType>
```

# Types complexes

---

- Définition d'objets complexes
  - `<sequence>`: collection ordonnée d'éléments typés (concaténation)
  - `<all>`: collection non ordonnée d'éléments typés
  - `<choice>`: choix entre éléments typés (union)
  - `<any ...>`: joker

- Exemple

```
<xsd:complexType name="AdresseFR">
  <xsd:sequence>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="rue" type="xsd:string"/>
    <xsd:element name="ville" type="xsd:string"/>
    <xsd:element name="codepostal" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="pays" type="xsd:NMTOKEN" fixed="FR"/>
</xsd:complexType>
```

# Héritage

---

- Définition de sous-types par héritage
  - par extension: ajout d'informations
  - par restriction: ajout de contraintes
- Possibilité de contraindre la dérivation
- Exemple :

```
<complexType name="AdressePays">
  <complexContent>
    <extension base="Adresse">
      <sequence>
        <element name="pays" type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

# Patterns

---

- Contraintes sur type simple prédéfini
- Utilisation d'expression rationnelles (regular expressions)
  - similaires à celles de Perl
- Exemple

```
<xsd:simpleType name="pourcentage">  
  <xsd:restriction base="xsd:string">  
    <xsd:pattern value="([0-9]|[1-9][0-9]|100)%"/>  
  </xsd:restriction>  
</xsd:simpleType>
```

# Conclusion

---

- Schémas flexibles et irréguliers
  - optionnels, avec ou sans DTD
- Données auto-descriptives
  - balises et attributs
- Modèle de type hypertexte
  - support des références
- Éléments atomiques ou complexes
  - composition par agrégation
- Types de données variés et extensibles
  - textes, numériques, ..., types utilisateur



# Langages de requêtes

---

- XPath
- XSLT
- XQuery

# XPath

---

- XPath: l'adressage XML
  - permet de définir la manière pour adresser un ou plusieurs nœuds d'un document XML (expression de chemins d'accès)
    - un chemin d'accès ressemble un peu à celui des noms de fichiers (chemins), d'où le nom "XPath"
  - traiter des chaînes de caractères, des nombres et des booléens
- XML Path Language
  - recommandation W3C
  - version 2 disponible
- Expressions de chemins communes à :
  - XSLT, XLink/XPointer (liens), XQuery (requêtes), XML Schema (clés/références)
- XPath permet donc
  - de naviguer dans un arbre XML
  - de rechercher un ou plusieurs éléments dans un document
  - de référencer tout fragment d'un document

## XPath (2)

---

- XPath opère sur une représentation arborescente du document
- Chaque élément de l'arbre est appelé un nœud (*node*)
- Une expression de chemin spécifie une traversée de l'arbre du document :
  - depuis un nœud de départ (appelé également nœud contexte)
  - vers un ensemble de nœuds cibles (triés dans l'ordre du document)
  - les cibles constituent la valeur du cheminement
- Notion de contexte
  - pour comprendre ce que fait une expression XPath, il faut toujours faire attention à son contexte d'utilisation
- Un chemin est une séquence d'étapes et peut être :
  - absolu (le nœud contexte est la racine)
    - commence à la racine: /étape1/.../étapeN
  - relatif (le nœud contexte est un nœud du document)
    - commence à un nœud courant: étape1/.../étapeN

# Syntaxe et sémantique

---

- Une étape est de la forme: [axe::]filtre[prédicat]\*
  - l'axe définit la relation entre les nœuds et le sens de parcours – optionnel
  - le filtre sélectionne un type de nœud (élément ou @attribut)
  - les prédicats doivent être satisfaits par les nœuds retenus – optionnel
- Exemples:
  - child::para
  - child::figure[attribute::id="fr vesca"]
  - child::\*[position()=last()]
- Syntaxe simplifiée
  - @name équivalent à attribute::name
  - para[1] équivalent à child::para[position()=1]
  - ../para équivalent à self::node()/descendant-or-self::node()/child::para
  - . équivalent à self::node()
  - ../para équivalent à parent::node()/child::para

# Le contexte

---

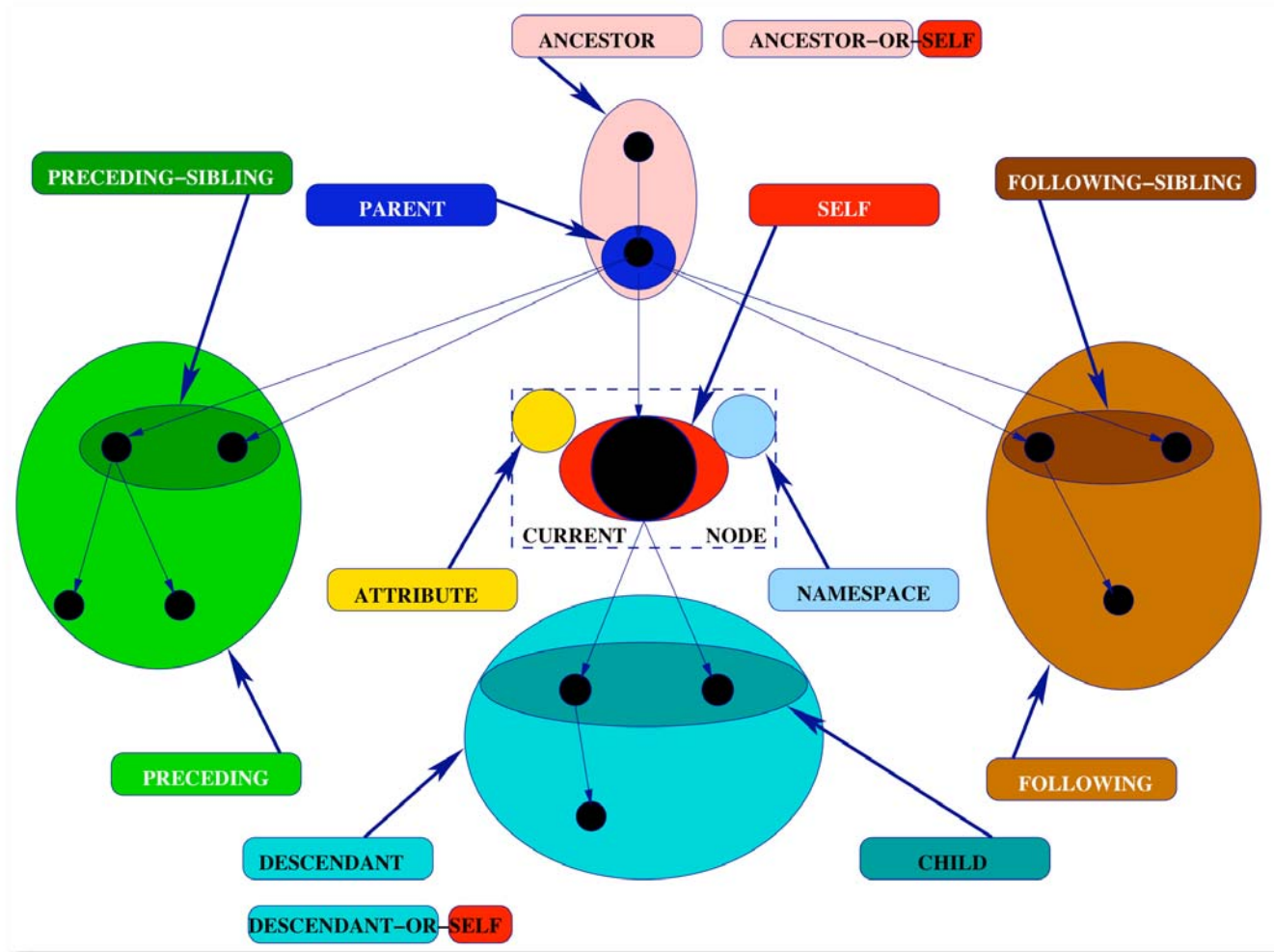
- Le contexte d'une évaluation XPath consiste
  - un nœud contexte (un nœud de l'arbre)
  - une position et une taille
  - un ensemble de variables associées
  - une librairie de fonctions
  - la déclaration d'un ensemble de namespace
- L'application détermine le contexte initial
- Si le chemin commence à la racine ("/")
  - le contexte initial est le nœud racine
  - la position est 1, et la taille est 1

# Les axes

---

- XPath supporte 12 axes
- Les axes en avant
  - child, self, descendant, descendant-or-self
- Les axes en arrière
  - parent, ancestor, ancestor-or-self
- Les axes à gauche et à droite
  - following-sibling, preceding-sibling
- Les axes avant et après
  - following, preceding
- Les axes pour les attributs
  - attribute

## Les axes (2)



# Les filtres

---

- Filtrer les nœuds
  - **nom**: les nœuds de l'axe qui portent ce nom
  - **\***: les nœuds de type Element ou Attribute de l'axe
- Filtrer les nœuds textuels
  - **text()**: tous les nœuds de type Text de l'axe
- Filtrer les commentaires
  - **comment()**: tous les nœuds de type Comment de l'axe
- Filtrer les instructions de traitement
  - **processing-instruction()**: tous les nœuds de type instruction de traitement de l'axe
- Filtrer les nœuds
  - **node()**: tous les types de nœud (sauf les attributs) de l'axe sauf la racine
  - **id(label)**: le nœud repéré par une étiquette



# Les prédicats

---

- Expression logique vraie ou fausse qui affine le résultat obtenu avec le chemin de recherche
- Condition d'existence
  - expression XPath: vraie si l'expression retourne un ensemble non vide de nœuds
  - chercher un élément qui a un attribut:
    - `nom_element_XML [ @nom_attribut ]`
  - chercher un élément qui a un attribut avec une certaine valeur:
    - `nom_element_XML [ @nom_attribut = 'valeur']`
- Condition de position
  - numéro (par ex. [1]): vrai si le nœud courant est à cette position dans le contexte courant
- Expressions booléennes: `and`, `or`, `not()`, `true()`, `false()`

# Les prédicats (2)

---

- Comparaisons
  - valeurs générales:  $\leq$ ,  $<$ ,  $\geq$ ,  $>$ ,  $=$ ,  $\neq$ 
    - vrai si une paire de valeurs satisfait la comparaison ( $8 = 4+4$ ,  $(1,2) = (2,4)$ )
    - $\text{\$livre/auteur} = \text{"Kennedy"}$  est vraie si  $\text{\$livre}$  possède un ou plusieurs auteurs et qu'au moins un est Kennedy
  - valeurs atomiques: eq, ne, lt, le, gt, ge
    - comparaison stricte des valeurs atomiques ( $8 \text{ eq } 4+4$ )
    - $\text{\$livre/auteur eq "Kennedy"}$  est vraie ssi  $\text{\$livre}$  possède exactement un auteur et qu'il s'agit de Kennedy
- nœuds: is,  $<<$ ,  $>>$ 
  - l'opérateur is compare deux nœuds (s'agit-il du même nœud ?)
  - les opérateurs  $<<$  et  $>>$  comparent l'ordre des nœuds dans le document
  - $\text{\$livre/auteur is key('auteurs', 'kennedy')}$  est vraie ssi  $\text{\$livre}$  possède exactement un auteur et que cet élément auteur est le même que celui retourné par l'expression key

# Fonctions XPath

---

- XPath définit un certain nombre de fonctions (106 !)
- Chaque fonction retourne soit une valeur booléenne (vrai/faux), un nombre, une chaîne de caractères, ou encore une liste de noeuds
- Fonctions pour les ensembles de noeuds
  - number **last**() : retourne le nombre de noeuds qui se trouvent dans le contexte (qui ont le même parent)
  - number **position**() : retourne le nombre de la position contextuelle (context position) d'un élément par rapport à son parent
  - number: **count**(node-set): retourne le nombre de noeuds de l'ensemble de noeuds passés en arguments
  - node-set **id**(object): retourne les éléments à partir de leur ID unique
  - string **name**(node-set?), string **local-name**(node-set?), string **namespace-uri**(node-set?): retourne le nom/le nom local/l'espace de nom du nœud en paramètre

## Fonctions XPath (2)

---

- Fonctions sur les chaînes
  - boolean **starts-with**(string, string): retourne TRUE si le deuxième string se trouve au début du premier
  - boolean **contains**(string, string): retourne TRUE si le deuxième string se trouve dans le premier
  - number **string-length**(string?): retourne la longueur d'un string
  - string **string**(object ?): retourne la version chaîne de l'objet
  - string **concat**(string, string, string\*): retourne la concaténation des deux chaînes
  - string **substring-before**(string, string): retourne la chaîne res tq  $ch1 = res + ch2$
  - string **substring-after**(string, string): retourne la chaîne res tq  $ch1 = ch2 + res$
  - string **substring**(string, number, number ?): retourne l'extraction de la sous-chaîne
  - string **normalize-space**(string ?): retourne une version normalisée (sans espace)
  - string **translate**(string, string, string): retourne une chaîne construite à partir de ch1 dans laquelle les caractères présents dans ch2 sont remplacés par les caractères de même position dans ch3

## Fonctions XPath (3)

---

- Fonctions booléennes
  - boolean **boolean**(object): teste si l'objet vaut True
  - boolean **not**(boolean): vraie si le paramètre est faux
  - boolean **true**()
  - boolean **false**()
- Fonctions numériques
  - number **floor**(number): arrondi à l'entier inférieur
  - number **ceiling**(number): arrondi à l'entier supérieur
  - number **number**(object?): transforme un objet en nombre
  - number **sum**(node-set): la somme de nombres trouvés dans un ensemble de noeuds. Effectue une conversion de strings si nécessaire, comme number()
  - number **round**(number): arrondit un nombre selon les conventions habituelles: 1.4 devient 1 et 1.7 devient 2
- Expressions: calculs arithmétiques
  - on utilise: + - \* div mod

# Quelques chemins simples: éléments enfants, parents, cousins

---

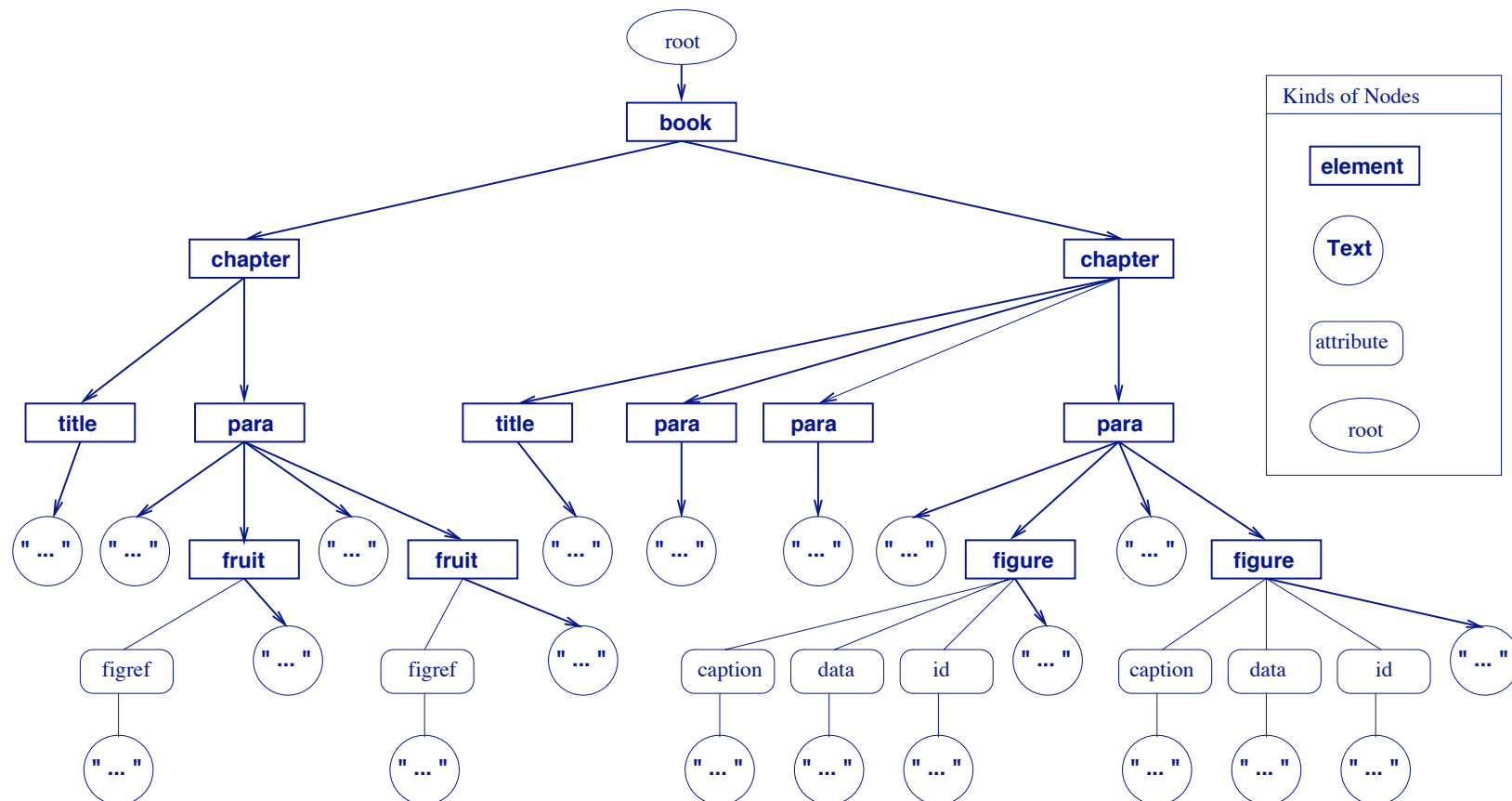
- Noeud racine:  
/ retourne le premier noeud trouvé dans un arbre
- Élément enfant direct:  
nom\_element
- Élément enfant direct du noeud racine:  
/nom\_element\_enfant
- Enfant d'un enfant:  
nom\_element\_pere/nom\_element\_enfant
- Descendant arbitraire du noeud racine:  
//nom\_element\_descendant
- Descendant arbitraire d'un noeud:  
nom\_element\_ancetre//nom\_element\_descendant
- Un parent d'un noeud:  
../
- Un cousin lointain d'un noeud:  
../../nom\_element\_XML/nom\_element\_XML/nom\_element\_XML

# Exercise

---

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<book>
  <chapter>
    <title>Various Fruits</title>
    <para>The next chapters introduce different kinds of fruits, like <fruit
      figref="fr_virg">strawberries</fruit> or <fruit figref="apple">apples</fruit>. </para>
  </chapter>
  <chapter>
    <title>Strawberries</title>
    <para>stre[a]w berige; stre[a]w straw + berie berry; perhaps from the resemblance of
      the runners of the plant to straws. </para>
    <para>A fragrant edible berry, of a delicious taste and commonly of a red colour.</
      para>
    <para>The common American strawberry is <figure caption="Fragaria virginiana"
      data="fr_virg.jpg" id="fr_virg">Fragaria virginiana</figure>, the European is <figure
      caption="Fragaria vesca" data="fr_vesca.jpg" id="fr_vesca">Fragaria vesca</
      figure>.</para>
  </chapter>
</book>
```

## Exercise (2)





## Exercice (3)

---

- Sections d'un paragraphe

`/child::book/child::chapter/child::para`

`/book/chapter/para`

- Texte du paragraphe 2 du chapitre 1

`/descendant:: chapter[position() = 1]/child:: para[position() = 2]/child::text()`

`//chapter[1]/para[2]/text()`

## Exercice (4)

---

- Sélectionner les éléments figure sans attributs
- Sélectionner le chapitre qui possède 3 paragraphes
- Sélectionner le premier paragraphe de chaque chapitre
- Sélectionner le premier paragraphe de tous les chapitres
- Sélectionner les figures possédant un élément caption 'Fragaria virginiana' dans le deuxième chapitre
- Sélectionner les figures des chapitres 2 à 5

## Exercice (4)

---

- Sélectionner les éléments figure sans attributs

`//figure[not(@*)]`

- Sélectionner le chapitre qui possède 3 paragraphes

`//chapter[count(./para) = 3]`

- Sélectionner le premier paragraphe de chaque chapitre

`//chapter//para[1]`

- Sélectionner le premier paragraphe de tous les chapitres

`(//chapter//para)[1]`

- Sélectionner les figures possédant un élément caption 'Fragaria virginiana' dans le deuxième chapitre

`//chapter[2]//figure[@caption = 'Fragaria virginiana']`

- Sélectionner les figures des chapitres 2 à 5

`//chapter[position() >= 2 and position() <= 5]//figure`

## Exercice (5)

---

- Sélectionner les captions des figures qui sont référencées par l'attribut figref des éléments fruit dans le premier chapitre
- Sélectionner les chapitres dans lesquels le mot 'Strawberry' est mentionné dans au moins un paragraphe
- Sélectionner les chapitres dans lesquels le mot 'Strawberry' est mentionné dans chaque paragraphe

## Exercice (5)

---

- Sélectionner les captions des figures qui sont référencées par l'attribut figref des éléments fruit dans le premier chapitre

```
id(//chapter[1]//fruit/@figref)[self::figure]/caption
```

- Sélectionner les chapitres dans lesquels le mot 'Strawberry' est mentionné dans au moins un paragraphe

```
//chapter[.//para[contains(.,'Strawberry')]]
```

- Sélectionner les chapitres dans lesquels le mot 'Strawberry' est mentionné dans chaque paragraphe

```
//chapter[count(.//para) = count(.//para[contains(.,'Strawberry')]) and .//para]
```

```
//chapter[not(.//para[not(contains(.,'Strawberry'))]) and .//para]
```

# Comparaison avec SQL

---

```
<invoice number="2145" date="2001-11-01">
  <customer>
    <name>Tom Jones</name>
    <address>33, Plaza Av.</address>
    <zip>14345-650</zip>
  </customer>
  <items>
    <item>
      <productID>MONO</productID>
      <qty>1</qty>
      <sale-price>65.00</sale-price>
    </item>
    <item>
      <productID>BIKE-12</productID>
      <qty>2</qty>
      <sale-price>290.00</sale-price>
    </item>
  </items>
</invoice>
```

SQL: Select discount from invoice;

SQL: Select \* from invoice where date="2001-11-02";

SQL: Select invoiceNumber from invoice where discount > 0;

# Comparaison avec SQL

---

```
<invoice number="2145" date="2001-11-01">
  <customer>
    <name>Tom Jones</name>
    <address>33, Plaza Av.</address>
    <zip>14345-650</zip>
  </customer>
  <items>
    <item>
      <productID>MONO</productID>
      <qty>1</qty>
      <sale-price>65.00</sale-price>
    </item>
    <item>
      <productID>BIKE-12</productID>
      <qty>2</qty>
      <sale-price>290.00</sale-price>
    </item>
  </items>
</invoice>
```

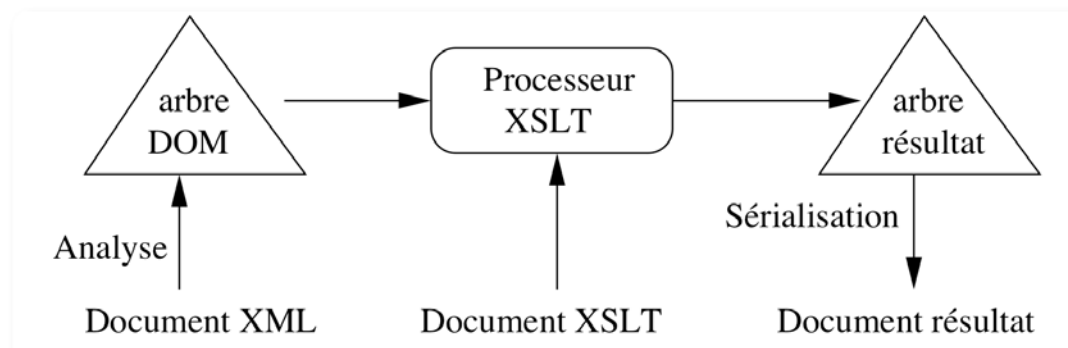
SQL: Select discount from invoice;  
SQL: Select \* from invoice where date="2001-11-02";  
SQL: Select invoiceNumber from invoice where discount > 0;

XPath: //discount  
XPath: //invoice[@date='2001-11-02']  
XPath: //invoice[//items/item/discount/number()>0]/@number

# XSLT

---

- Objectif: transformer un document XML en
  - un ou plusieurs documents XML, HTML, WML, SMIL
  - un document papier: PDF (XSL-FO), LaTeX
  - un texte simple



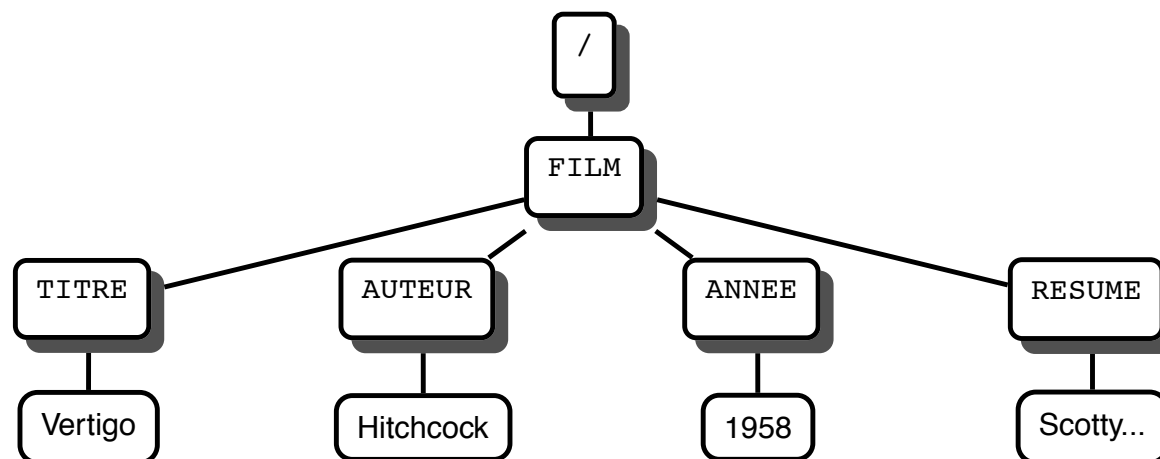


# Fonctions d'un programme XSLT

---

- Transformation d'arbres XML:
  - extraction de données
  - génération de texte
  - suppression de contenu (nœuds)
  - déplacer le contenu (nœuds)
  - dupliquer le contenu (nœuds)
  - trier

# Exemple



- Règles de transformation

```
<xsl:template match="FILM">
```

Sélecteur d'éléments à transformer

```
<p>
```

```
<h1> <i> <xsl:value-of select="TITRE"/> </i> </h1>
```

```
<i> <xsl:value-of select="ANNEE"/> </i>
```

```
<p> <xsl:value-of select="AUTEUR"/> </p>
```

```
<h3>Résumé: <xsl:value-of select="RESUME"/> </h3>
```

```
</p>
```

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

Instructions de transformation

# Fonctionnalités de XSLT

---

- Extraction de données (xsl:value-of)

```
<xsl:template match="FILM">  
  <xsl:value-of select="TITRE"/>  
</xsl:template>
```

- Génération de texte (texte brut)

```
<xsl:template match="FILM">  
  Ceci est le texte produit par application de cette règle.  
</xsl:template>
```

- Génération d'un arbre XML (fragment XML bien formé)

```
<xsl:template match="FILM">  
  <body>  
    <p>Un paragraphe</p>  
  </body>  
</xsl:template>
```

## Fonctionnalités XSLT (2)

---

- Génération d'arbre avec extraction de valeur

```
<xsl:template match="FILM">
  <body>
    <p>Titre:
      <xsl:value-of select="TITRE"/>
    </p>
  </body>
</xsl:template>
```

# Les règles

---

- C'est la structure de base
- Règle = *template*: élément de base pour produire le résultat
  - une règle s'applique dans le contexte d'un nœud de l'arbre
  - l'application de la règle produit un fragment du résultat
- Programme XSLT = ensemble de règles pour construire un résultat
- Dans une règle, on peut:
  - accéder aux fils, aux descendants, au parent, aux frères, aux neveux, aux attributs,... du noeud à transformer (grâce à XPath)
  - effectuer des tests et des boucles,...
  - "appeler" d'autres règles (récursion)

# Exemple

---

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="Salle.xsl" type="text/xsl"?>
<SALLE NO="1" PLACES="320" >
  <FILM>
    <TITRE>Alien</TITRE>
    <AUTEUR>RidleyScott</AUTEUR>
    <ANNEE>1979</ANNEE>
    <GENRE>Science-fiction</GENRE>
    <PAYS>EtatsUnis</PAYS>
    <RESUME>Près d'un vaisseau spatial échoué sur une lointaine planète, des Terriens
    en mission découvrent de bien étranges "oeufs". Ils en ramènent un à bord, ignorant
    qu'ils viennent d'introduire parmi eux un huitième passager particulièrement féroce et
    meurtrier. </RESUME>
  </FILM>
  <REMARQUE>Réservation conseillée</REMARQUE>
  <SEANCES>
    <SEANCE>15:00</SEANCE>
    <SEANCE>18:00</SEANCE>
    <SEANCE>21:00</SEANCE>
  </SEANCES>
</SALLE>
```

## Exemple (2)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="Salle.xsl" type="text/xsl"?>
<SALLE NO="1" PLACES="320" >
  <FILM>
    <TITRE>Alien</TITRE>
    <AUTEUR>RidleyScott</AUTEUR>
```

```
<SEANCES>
  <SEANCE>15:00</SEANCE>
  <SEANCE>18:00</SEANCE>
  <SEANCE>21:00</SEANCE>
</SEANCES>
</SALLE>
```

- Boucle: traduction de l'élément XML <SALLES> en élément HTML <ol><li></li></ol>

```
<xsl:template match="SALLE">
  <h2>Salle No <xsl:value-of select="@NO"/></h2>
  Film:<xsl:value-of select="FILM/TITRE"/>
  de <xsl:value-of select="FILM/AUTEUR"/>
  <ol>
    <xsl:for-each select="SEANCES/SEANCE">
      <li><xsl:value-of select="."/></li>
    </xsl:for-each>
  </ol>
</xsl:template>
```

- Remarque: c'est un fragment HTML, à intégrer dans un document complet

## Exemple (2)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="Salle.xsl" type="text/xsl"?>
<SALLE NO="1" PLACES="320" >
  <FILM>
    <TITRE>Alien</TITRE>
    <AUTEUR>RidleyScott</AUTEUR>
```

```
<SEANCES>
  <SEANCE>15:00</SEANCE>
  <SEANCE>18:00</SEANCE>
  <SEANCE>21:00</SEANCE>
</SEANCES>
</SALLE>
```

- Boucle: traduction de l'élément XML <SALLES> en élément HTML <ol><li/></ol>

```
<xsl:template match="SALLE">
```

```
  <h2>Salle No <xsl:value-of select="@NO"/></h2>
```

```
  Film:<xsl:value-of select="FILM/TITRE"/>
```

```
  de <xsl:value-of select="FILM/AUTEUR"/>
```

```
  <ol>
```

```
    <xsl:for-each select="SEANCES/SEANCE">
```

```
      <li><xsl:value-of select="."/></li>
```

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

```
  </ol>
```

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

```
<h2>Salle No 1</h2>
```

```
Film: Alien
```

```
de Ridley Scott
```

```
<ol>
```

```
<li>15:00</li>
```

```
<li>18:00</li>
```

```
<li>21:00</li>
```

```
</ol>
```

- Remarque: c'est un fragment HTML, à intégrer dans un document complet



# Appel des règles

---

- En général, on produit un résultat en combinant plusieurs règles:
  - la règle initiale s'applique à la racine du document traité ('/')
  - on produit alors le cadre du document HTML
  - on appelle d'autres règles pour compléter la création du résultat

- Exemple

```
<xsl:template match="/">
<html>
  <head><title>Programme de <xsl:value-of select="CINEMA/NOM"/></title></head>
  <body bgcolor="white">
    <xsl:apply-templates select="CINEMA"/>
  </body>
</html>
</xsl:template>
```

# Appel des règles

---

- En général, on produit un résultat en combinant plusieurs règles:
  - la règle initiale s'applique à la racine du document traité ('/')
  - on produit alors le cadre du document HTML
  - on appelle d'autres règles pour compléter la création du résultat

- Exemple

```
<xsl:template match="/">
<html>
  <head><title>Programme de <xsl:value-of select="CINEMA/NOM"/></title></head>
  <body bgcolor="white">
    <xsl:apply-templates select="CINEMA"/>
  </body>
</html>
</xsl:template>
```



Nouvelle règle

# Règle CINEMA

---

- Exploitation de l'élément CINEMA, puis appel à la règle SALLE

```
<xsl:template match="CINEMA">
  <h1><i><xsl:value-of select="NOM"/></i></h1><hr/>
  <xsl:value-of select="ADRESSE"/>,
  <i>Métro:</i> <xsl:value-of select="METRO"/>
  <hr/>
  <xsl:apply-templates select="SALLE"/>
</xsl:template>
```

# Règle CINEMA

---

- Exploitation de l'élément CINEMA, puis appel à la règle SALLE

```
<xsl:template match="CINEMA">  
  <h1><i><xsl:value-of select="NOM"/></i></h1><hr/>  
  <xsl:value-of select="ADRESSE"/>,  
  <i>Métro:</i> <xsl:value-of select="METRO"/>  
  <hr/>  
  <xsl:apply-templates select="SALLE"/>  
</xsl:template>
```



Appel de la règle SALLE

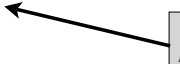
# Règle CINEMA

---

- Exploitation de l'élément CINEMA, puis appel à la règle SALLE

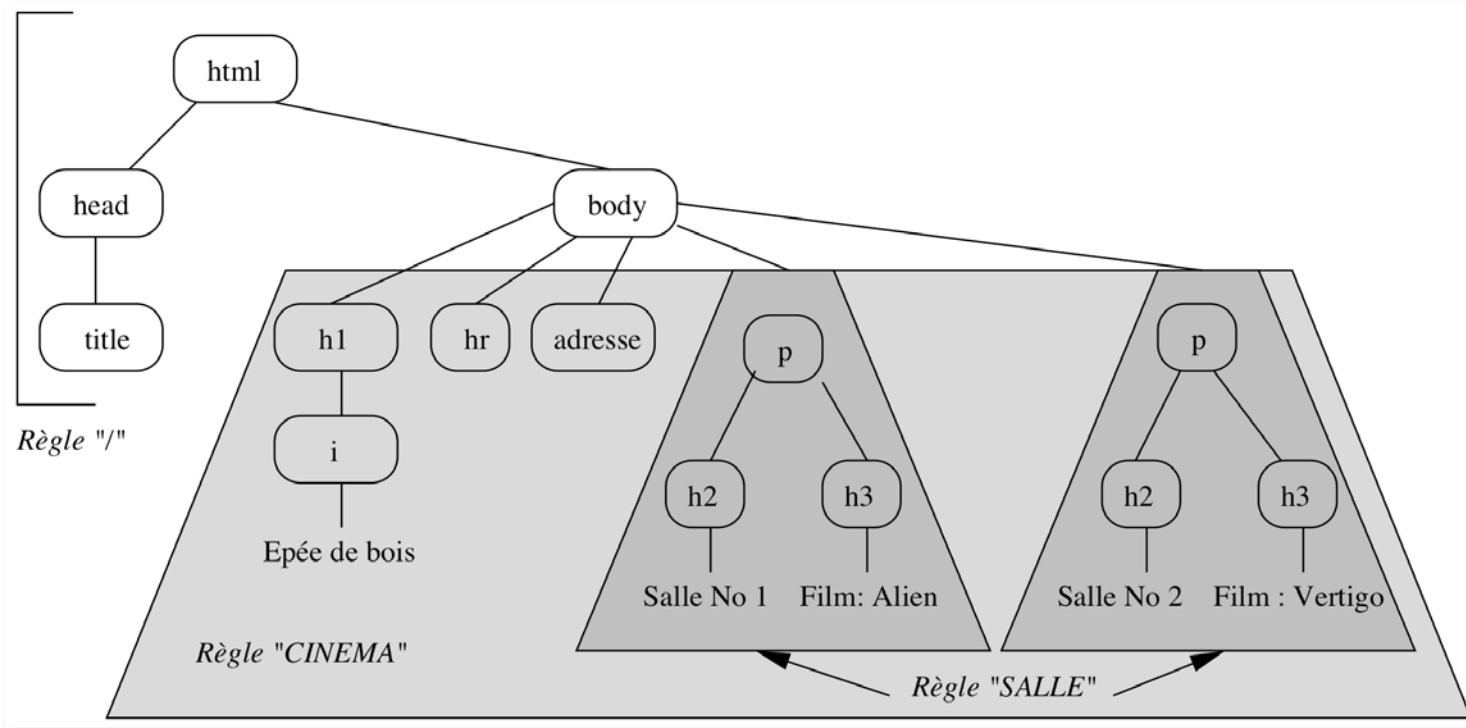
```
<xsl:template match="CINEMA">
  <h1><i><xsl:value-of select="NOM"/></i></h1><hr/>
  <xsl:value-of select="ADRESSE"/>,
  <i>Métro:</i> <xsl:value-of select="METRO"/>
  <hr/>
  <xsl:apply-templates select="SALLE"/>
</xsl:template>
```

Appel de la règle SALLE



```
<xsl:template match="SALLE">
  <h2>Salle No <xsl:value-of select="@NO"/></h2>
  Film:<xsl:value-of select="FILM/TITRE"/>
  de <xsl:value-of select="FILM/AUTEUR"/>
  <ol>
    <xsl:for-each select="SEANCES/SEANCE">
      <li><xsl:value-of select="."/></li>
    </xsl:for-each>
  </ol>
</xsl:template>
```

# Vue d'ensemble



# Programme XSLT

---

- Un programme XSLT consiste à produire un document résultat à partir d'un document source
- Un programme XSLT est un document XML
- Les éléments XSLT sont différenciés grâce à un espace de noms xsl:

```
<?xml version="1.0"encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="COURS">
  <html>
    <head><title>Fiche du cours</title></head>
    <body bgcolor="white">
      <p>
        <h1><i><xsl:value-of select="SUJET"/></i></h1>
        <hr/>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

# Programme XSLT

---

- Un programme XSLT consiste à produire un document résultat à partir d'un document source
- Un programme XSLT est un document XML
- Les éléments XSLT sont différenciés grâce à un espace de noms xsl:

Élément racine  
du programme

```
<?xml version="1.0"encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="COURS">
    <html>
      <head><title>Fiche du cours</title></head>
      <body bgcolor="white">
        <p>
          <h1><i><xsl:value-of select="SUJET"/></i></h1>
          <hr/>
          <xsl:apply-templates/>
        </body>
      </html>
    </xsl:template>
  </xsl:stylesheet>
```



# Programme XSLT

- Un programme XSLT consiste à produire un document résultat à partir d'un document source
- Un programme XSLT est un document XML
- Les éléments XSLT sont différenciés grâce à un espace de noms xsl:

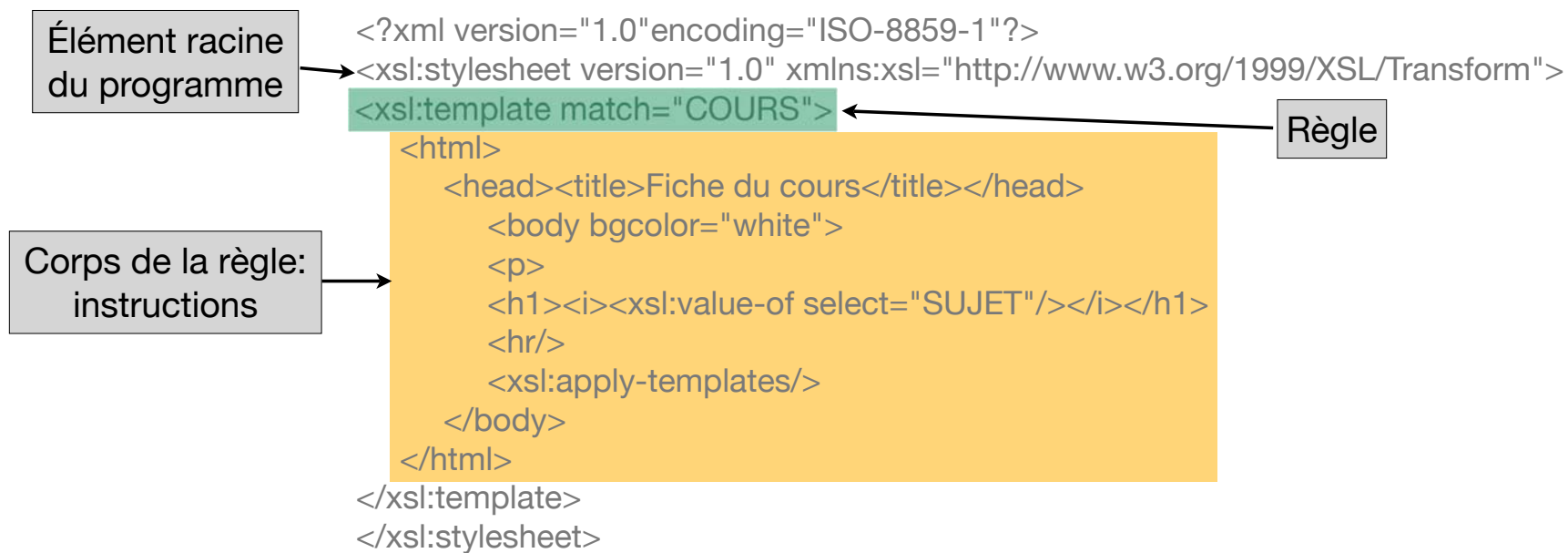
Élément racine  
du programme

```
<?xml version="1.0"encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="COURS">
    <html>
      <head><title>Fiche du cours</title></head>
      <body bgcolor="white">
        <p>
          <h1><i><xsl:value-of select="SUJET"/></i></h1>
          <hr/>
          <xsl:apply-templates/>
        </body>
      </html>
    </xsl:template>
  </xsl:stylesheet>
```

Règle

# Programme XSLT

- Un programme XSLT consiste à produire un document résultat à partir d'un document source
- Un programme XSLT est un document XML
- Les éléments XSLT sont différenciés grâce à un espace de noms xsl:



# Éléments de premier niveau

---

- **xsl:import**
  - pour importer un programme XSLT (doit être avant include, les règles importées sont prioritaires)
- **xsl:include**
  - pour inclure un programme XSLT (les règles sont au même niveau)
- **xsl:output**
  - pour définir le format de sortie
- **xsl:param**
  - pour définir un paramètre
- **xsl:variable**
  - pour définir une variable
- **xsl:template**
  - pour définir une règle

# Règles

---

- Définition

- une règle est définie par l'élément `xsl:template`
- deux possibilités
  - l'attribut `match` est une expression XPath définissant les éléments sources de la règle

`<xsl:template match="FILM">`

- l'attribut `name` donne un nom à la règle

`<xsl:template name="TDM">`

- Déclenchement

- pour le premier type de règle

`<xsl:apply-templates select="...">`

- pour le deuxième type de règle

`<xsl:call-template name="...">`

# Sélection des règles

---

- Problème
  - étant donné un nœud, comment trouver la règle qui s'applique ?
- Algorithme
  - soit N le nœud
  - soit P le motif (pattern) de la règle R
  - s'il existe quelque part un nœud C tel que l'évaluation de P à partir de C contient N, alors la règle s'applique
- Exemple: la règle pour la racine
  - le nœud contexte N est la racine du document
  - il existe une règle R dont le motif est "/"
  - en prenant n'importe quel nœud, l'évaluation de "/" est N, donc la règle s'applique
- Il est donc préférable (mais pas obligatoire) d'avoir une règle "/"

## Sélection des règles (2)

---

- Un motif de sélection est une expression XPath restreinte
  - les fils d'un élément (axe child)
  - les attributs d'un élément (axe attribute)
  - la simplification // (axe /descendant-or-self::node()/)
- Cette restriction garantit que l'on peut savoir si une règle doit être déclenchée pour un nœud N uniquement en analysant les ancêtres de N
- Cela diminue considérablement la complexité de l'algorithme de sélection
- Exemples

/COURS/ENSEIGNANTS: la règle s'applique à tous les nœuds ENSEIGNANTS fils d'un élément racine COURS

//SEANCE[@ID=2]: ... à tous les nœuds de type SEANCE ayant un attribut ID valant 2

/descendant::FILM[1]: ... au premier élément de type FILM dans le document

FILM[1]: ... aux premiers fils de type FILM (il peut y en avoir plusieurs!)

/COURS[@CODE="TC234"]: ... aux cours avec le code TC234

# Règles par défaut

---

- Lorsque aucune règle n'est sélectionnée, le moteur XSLT applique des règles par défaut
- La première règle pour les éléments et la racine du document

```
<xsl:template match="*/">
```

```
  <xsl:apply-templates/>
```

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

- on demande l'application de règles pour les fils du nœud courant
- La deuxième règle insère dans le document résultat la valeur du nœud ou de l'attribut

```
<xsl:template match="text()|@">
```

```
  <xsl:value-of select="."/>
```

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

- cela suppose (en particulier pour les attributs) d'avoir utilisé un `xsl:apply-templates` qui ait sélectionné ces nœuds
- La troisième règle concerne les processing-instructions et les commentaires

# Conséquence

---

- Si on se contente des règles par défaut, on obtient la concaténation de nœuds de type Text
- Programme minimal:

```
<?xmlversion="1.0"encoding="ISO-8859-1"?>  
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  
</xsl:stylesheet>
```



# xsl:apply-templates

---

- C'est une instruction qui possède 3 attributs
  - select
  - mode
  - priority
- **select** doit sélectionner un ensemble de nœuds
  - ces nœuds constituent le contexte d'utilisation
  - pour chaque nœud, on va rechercher la règle à instancier
- **mode** permet de choisir explicitement une des règles à instancier parmi celles qui sont candidates
- **priority** permet de définir une priorité pour le processeur puisse choisir

# Sélection d'une règle

---

- Comment gérer le fait que plusieurs règles sont éligibles pour un même nœud ?
  - il existe des priorités implicites qui permettent au processeur de choisir
  - on peut donner explicitement une priorité
  - si malgré cela, le choix est impossible, le processeur s'arrête
- Exemple: on souhaite effacer certains nœuds

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<FILM>
  <TITRE>Vertigo</TITRE>
  <ANNEE>1958</ANNEE><GENRE>Drame</GENRE>
  <MES>Alfred Hitchcock</MES>
  <RESUME>Scottie Ferguson, ancien inspecteur de police, est sujet au vertige...</RESUME>
</FILM>
<FILM>
  <TITRE>Alien</TITRE>
  <ANNEE>1979</ANNEE><GENRE>Science-fiction</GENRE>
  <MES>Ridley Scott</MES>
  <RESUME>Près d'un vaisseau spatial échoué sur une lointaine...</RESUME>
</FILM>
```

# Programme XSLT

---

- Ce programme permet d'effacer les nœuds de type RESUME

```
<!-- on ne recopie pas les nœuds RESUME dans le document résultat -->
<xsl:template match="RESUME"/>
<!-- on recopie les autres nœuds -->
<xsl:template match="@*|node()" priority="-1">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>
```

# Priorités implicites

---

- Idée: plus c'est «spécifique», plus c'est prioritaire
- Priorité 0: les motifs constitués d'une seule étape XPath, avec un nom d'élément ou d'attribut et sans prédicat
- Priorité -0.5: les filtres autres qu'un nom d'élément ou d'attribut ont une priorité égale à -0,5 (node(),\*)
- Tous les autres ont une priorité de 0.5 (prédicats, plusieurs étapes)

# Les modes

---

- Objectif
  - un même nœud peut être traité plusieurs fois
- Exemple
  - on parcourt tous les chapitres et paragraphes pour produire une table des matières
  - on les parcourt à nouveau pour publier le contenu
- Il faut donc des règles différentes qui s'appliquent aux mêmes nœuds: c'est le mode qui va permettre la distinction
- Exemple: création de liens HTML
  - on peut créer des ancres «internes» à un document

```
<a name='Alien' />
```
  - on peut ensuite créer un lien vers cette ancre

```
<a href='#Alien'>Lien vers le film Alien</A>
```
  - objectif: une règle pour créer les liens, une autre pour créer les ancres

# Règles avec mode

---

```
<xsl:template match="FILM" mode="Ancres">
  <a href="#{TITRE}"> <xsl:value-of select="TITRE"/> </a>
</xsl:template>
```

```
<xsl:template match="FILM">
  <a name="{TITRE}"/>
  <h1><xsl:value-of select="TITRE"/></h1>
  <b><xsl:value-of select="TITRE"/>,</b>
  <xsl:value-of select="GENRE"/> <br/>
  <b>Réalisateur</b>: <xsl:value-of select="MES"/>
</xsl:template>
```

# L'appel des règles

---

```
<xsl:template match="FILMS">
  <html>
    <head><title>Liste des films</title></head>
    <body bgcolor="white">
      <xsl:apply-templates select="FILM" mode="Ancres"/>
      <xsl:apply-templates select="FILM"/>
    </body>
  </html>
</xsl:template>
```

## Synthèse: sélection d'une règle

---

- Soit un **xsl:apply-templates**, et N un des nœuds sélectionné
- On ne prend que les règles avec le même mode que xsl:apply-templates
- On teste le motif XPath pour savoir si le nœud satisfait la règle
- On prend celle qui a la plus grande priorité



# Éléments de programmation

---

- Traitement conditionnel: **xsl:if**
- Syntaxe

```
<xsl:if test = "boolean-expression">  
  <!-- contenu -->  
</xsl:if>
```

- Permet de changer l'output en fonction d'un test
- Attention: il n'existe pas de "else" (utilisez "choose" à la place)
- Cas d'utilisation: traitement d'un élément en fonction de sa position, des ses attributs,...

## Éléments de programmation (2)

---

- Traitement conditionnel: **xsl:choose**

- Syntaxe:

```
<xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>
<xsl:when test = boolean-expression>
  <!-- contenu -->
</xsl:when>
<xsl:otherwise>
  <!-- contenu -->
</xsl:otherwise>
```

- Cette définition dit:
  - on peut avoir plusieurs clauses avec un test (xsl:when).
  - la première vraie est utilisée (donc la série des xsl:when correspond à "if ( ) {...}"  
elseif ( ) { ...}" elseif ( ) { ...}")
  - si aucune clause n'est vraie et s'il existe une clause xsl:otherwise, c'est cette dernière qui est exécutée (il s'agit donc du "else {...}")

# Éléments de programmation (3)

---

- Itération: **xsl:for-each**

- Syntaxe

```
<xsl:for-each select="motif-XPath">  
  <!-- contenu -->  
</xsl:for-each>
```

- Permet de parcourir un ensemble de nœuds et d'appliquer un traitement

## Éléments de programmation (4)

---

- Tri: **xsl:sort**

- Syntaxe

```
<xsl:sort
  select="motif-XPath"           <!-- par défaut, . -->
  data-type="text|number"       <!-- par défaut, text -->
  order="ascending|descending"  <!-- par défaut, asc -->
  case-order="upper-first|lower-first" <!-- par défaut, upper -->
  lang="nom_de_langue"         <!-- par défaut, langue système -->
/>
```

- Associé à un parcours (**xsl:for-each** ou **xsl:apply-templates**)
- Permet de modifier l'ordre des nœuds

# Paramètres

---

- XSLT ne connaît pas de variables au sens des langages procéduraux
- Les paramètres
  - syntaxe

```
<xsl:param  
  name = qname  
  select = expression>  
  <!-- contenu -->  
</xsl:param>
```

- Un paramètre lie un nom à une valeur. La valeur est définie soit dans l'attribut select, soit par son contenu (mais pas les deux !)
- Lorsqu'on définit un paramètre à la racine, il s'applique à tous les templates qui font appel à ce paramètre et qui ne reçoivent pas sa valeur. Cela ressemble à une constante globale

# Variables

---

- Le nom "variable" n'est pas clair. Il s'agit en fait de constantes.
- En règle générale, c'est utile pour faire des calculs "intermédiaires"
- Syntaxe

```
<xsl:variable  
  name = qname  
  select = expression>  
  <!-- contenu -->  
</xsl:variable>
```

# Paramètres, variables: exemple

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="text"/>
<!-- affichage du resultat -->
<xsl:template match="/factorielle">
  <xsl:variable name="x" select="valeur"/>
  <xsl:text>factorielle(</xsl:text><xsl:value-of select="$x"/><xsl:text>) = </xsl:text>
  <xsl:call-template name="factorielle">
    <xsl:with-param name="n" select="$x"/>
  </xsl:call-template>
</xsl:template>
<!-- fatorialle(n) - calcul de la factorielle -->
<xsl:template name="factorielle">
  <!-- on recupere le parametre, 1 valeur par default -->
  <xsl:param name="n" select="1"/>
  <!-- calcul -->
  <xsl:variable name="somme">
    <xsl:if test="$n = 1">1</xsl:if>
    <xsl:if test="$n != 1">
      <xsl:call-template name="factorielle">
        <xsl:with-param name="n" select="$n - 1"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:variable>
  <xsl:value-of select="$somme * $n"/>
</xsl:template>
</xsl:stylesheet>
```

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="fact.xsl"?>

<factorielle>
  <valeur>5</valeur>
</factorielle>
```

# Instructions diverses (non détaillées)

---

- `xsl:copy`
- `xsl:copy-of`
- `xsl:number`
- `xsl:element`
- `xsl:attribute`
- `xsl:attribute-set`
- `xsl:comment`
- `xsl:processing-instruction`
- `xsl:key`
- `xsl:message`
- `xsl:for-each-group`
- `xsl:function`



# XSLT étend XPath

---

- ajout de la fonction `generate-id(noeud)` qui renvoie un identifiant unique :

```
<xsl:if test="generate-id($n1) = generate-id($n2)">
  ...
</xsl:if>
```

- ajout de la fonction `current()` qui renvoie le noeud courant

```
<xsl:template ...>
  <xsl:variable name="p" select="//article[@id = current()/@id]"/>
  ...
</xsl:template ...>
```

- ajout de la fonction `key(index,clef)`,
- ajout du **ou** dans les expressions
- ajout de la fonction `format-number(nombre,format)` qui renvoie une chaîne issue du formatage d'un nombre :

```
<xsl:value-of select='format-number(500100, "###,###.00")' />
```

- ajout de la fonction `document(URI)` qui renvoie le document XML identifié par l'URI

```
<xsl:value-of select="document('stock.xml')//produit[prix > 10]" />
```

# Comment interroger des documents XML ?

---

- SQL: il faut stocker le document XML dans une base relationnelle
- XPath: extraction de fragments d'arbres
  - on navigue dans l'arbre grâce à des axes de navigation
  - un chemin de navigation est une séquence d'étapes
  - dans chaque étape, on choisit un axe, un filtre et éventuellement des prédicats
  - le résultat d'une étape (d'une séquence d'étapes) est une séquence de noeuds
- XSLT: extraction et transformation
  - on définit des règles de transformation qui transforme un fragment d'arbre en un autre fragment (copie)
  - les fragments à transformer sont choisis par des expressions XPath
  - les règles de transformations à appliquer à un fragment est choisie par une expression XPath
  - le résultat d'une transformation est un document XML
- XQuery ?

# XQuery

---

- Principe

- XQuery est un langage de "programmation" puissant pour extraire des données XML
- type de données: un seul "document" ou encore des collections sous forme de:
  - fichiers
  - bases de données XML
  - XML "en mémoire" (arbres DOM)
- permet de faire des requêtes selon la structure ou encore les contenus en se basant sur des expressions Xpath (version 2.0)
- peut générer des nouveaux documents (autrement dit: on peut manipuler un résultat obtenu et y ajouter)
- ne définit pas les mises à jour (équivalent de update/insert de SQL), à venir...

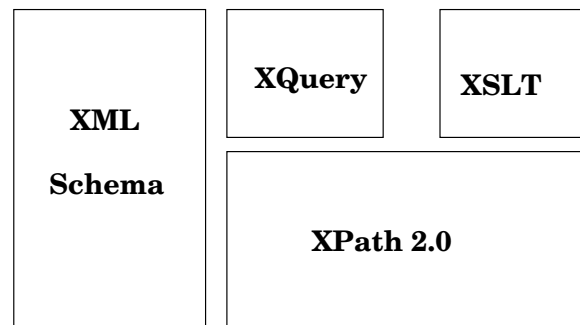
- Résumé:

- XQuery permet d'extraire des fragments XML, d'y effectuer des recherches et de générer des fragments XML

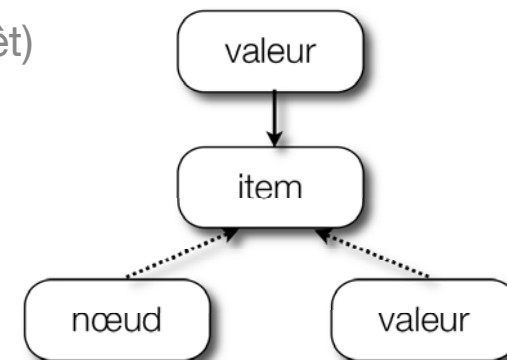
## XQuery (2)

---

- Sa place dans l'écosystème XML



- Le modèle de données XQuery
  - une valeur est une séquence ordonnée d'items (forêt)
  - un item est un nœud ou une valeur atomique
  - chaque nœud et chaque valeur a un type



# Séquences

---

- Pas de distinction entre un item et une séquence de longueur 1:  $47=(47)$
- Une séquence peut contenir des valeurs hétérogènes:  $(1, \text{"toto"}, <\text{toto}/>)$
- Pas de séquences imbriquées:  $(1, (2, 6), \text{"toto"}, <\text{toto}/>) = (1, 2, 6, \text{"toto"}, <\text{toto}/>)$
- Une séquence peut être vide:  $()$
- Les séquences sont ordonnées:  $(1, 2) \neq (2, 1)$

# Expressions XQuery

---

- Une requête XQuery est une composition d'expressions
- Chaque expression a une valeur ou retourne une erreur
- Les expressions n'ont pas d'effets de bord (par exemple, pas de mise-à-jour)
- Expressions (requêtes) simples:
  - valeurs atomiques: 46, "Salut"
  - valeurs construites: true(), date("2007-03-19")
- Expressions complexes
  - expressions de chemins (XPath 2.0): FILM//ACTEUR
  - expressions FLWOR (for-let-where-order-return)
  - tests (if-then-return-else-return)
  - fonctions: racines, XPath, utilisateurs

# Exemple

---

```
<!ELEMENT bib (book*)>
<!ELEMENT book ((author)+,publisher,price)>
<!ATTLIST book year CDATA #IMPLIED title CDATA #REQUIRED>
<!ELEMENT author (la,fi)>
<!ELEMENT la (#PCDATA)>
<!ELEMENT fi (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

```
<bib>
  <book year="2000" title="Data on the Web">
    <author><la>Abiteboul</la><fi>S.</fi></author>
    <author><la>Buneman</la><fi>P.</fi></author>
    <author><la>Suciu</la><fi>D.</fi></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
</bib>
```

# Expressions de chemin

---

- Expression: `document("bib.xml")//author`
- Résultat:  

```
<author><la>Abiteboul</la><fi>S.</fi></author>,  
<author><la>Buneman</la><fi>P.</fi></author>,  
<author><la>Suciu</la><fi>D.</fi></author>
```
- Expression: `doc("bib.xml")/bib/book/author`
- Résultat:
- Expression: `doc("bib.xml")/bib//book[1]/publisher`
- Résultat:



# Constructions de nœuds XML

---

- Le nom de l'élément est connu, le contenu est calculé par une expression

- Requête:

```
<auteurs>
  {document("bib.xml")//book[1]/author/la}
</auteurs>
```

- Résultat:

```
<?xml version="1.0"?>
<auteurs>
  <la>Abiteboul</la>
  <la>Buneman</la>
  <la>Suciu</la>
</auteurs>
```

## Constructions de nœuds XML (2)

---

- Le nom et le contenu sont calculés:

- `element {expr-nom} {expr-contenu}`
- `attribute {expr-nom} {expr-contenu}`

- Requête:

```
element{document("bib.xml")//book[1]/name(@*[2])}{
  attribute{document("bib.xml")//book[1]/name(*[4])}{
    document("bib.xml")//book[1]/*[4]
  }
}
```

- Résultat:

```
<title publisher="Morgan Kaufmann Publishers"/>
```

# Différence de séquences de nœuds

---

- Requête:

<livre>

Tous les sous-éléments sauf les auteurs: {document("bib.xml")//book[1]/(\* except author)}

</livre>

- Résultat:

<livre>

Tous les sous-éléments sauf les auteurs:

<publisher>Morgan Kaufmann Publishers</publisher>

<price>39.95</price>

</livre>

# Concaténation de séquences

---

- Requête:

<livre>

Le prix suivi des auteurs: {document("bib.xml")//book[1]/(price,author)}

</livre>

- Résultat:

<livre>

Le prix suivi des auteurs:

<price>39.95</price>

<author><la>Abiteboul</la><fi>S.</fi></author>

<author><la>Buneman</la><fi>P.</fi></author>

<author><la>Suciu</la><fi>D.</fi></author>

</livre>

- Remarque: on a changé l'ordre des nœuds (union)

# FLWR (prononcer *flower*)

---

- FLWOR = "For-Let-Where-Order-Return"
  - rappelle l'idée du select-from-where de SQL
- Format d'une requête
  - **for** \$<var> in <forest> [, \$<var> in <forest>]+
    - itération sur une liste de collections xml
  - **let** \$<var> := <subtree>
    - assignation du résultat d'une expression à une variable
  - **where** <condition>
    - élagage avec une sélection
  - **return** <result>
    - construction de l'expression à retourner
- Les forêts sont soit des collections, soit sélectionnées par des XPath
- Le résultat est une forêt: un ou plusieurs arbres

# For, Let

---

- La clause **for \$variable in expression** affecte la variable **\$variable** successivement avec chaque item dans la séquence retournée par expression
- La clause **let \$variable := expression** affecte la variable **\$variable** avec la séquence “entière” retournée par expression

- Exemple

```
for $b in document("bib.xml")//book[1]
let $al := $b/author
return <livre nb_auteurs="{count($al)}"> {$al} </livre>
```

- Résultat

```
<livre nb_auteurs="3">
  <author><la>Abiteboul</la><fi>S.</fi></author>
  <author><la>Buneman</la><fi>P.</fi></author>
  <author><la>Suciu</la><fi>D.</fi></author>
</livre>
```

# Where

---

- La clause **where expression** permet de filtrer le résultat par rapport au résultat booléen de l'expression **expression** (= prédicat dans l'expression de chemin)
- Requête:

```
<livre>  
  {for $a in document("bib.xml")//book  
    where $a/author[1]/la eq "Abiteboul"  
    return $a/@title }  
</livre>
```

- Résultat:

```
<?xml version="1.0"?>  
  <livre title="Data on the Web"/>
```

# Order

---

- La clause **order by** permet de trier les résultats

- Requête

```
for $t in //book/author/la order by $t return $t
```

- Un exemple plus complet

```
for $t in document("bib.xml")//book
```

```
let $n := count($t/author)
```

```
where ($n > 1)
```

```
order by $n
```

```
return <result> {$t/@title} possède {$n} auteurs </result>
```



# Return

---

- La clause **return** construit l'expression à retourner à chaque iteration
  - attention: chaque iteration doit retourner un seul fragment XML (pas une collection)

- Correct

```
for $t in document("bib.xml")//book
let $n := count($t/author)
return <result>
    {$t/@title} possède {$n} auteurs
</result>
```

- Incorrect

```
for $t in document("bib.xml")//book
let $n := count($t/auteurs)
return $t/@title possède $n auteurs
```

# Exemple

---

- Sélection: lister les noms et téléphones des restaurants de Cabourg
- Résultat

```
<Guide Version= "2.0">
<Restaurant type="français" categorie="***">
  <Nom>Le Moulin</Nom>
  <Adresse>
    <Rue>des Vignes</Rue>
    <Ville>Mougins</Ville>
  </Adresse>
  <Manager>Dupuis</Manager>
</Restaurant>
<Restaurant type="français" categorie="***">
  <Nom>La Licorne</Nom>
  <Adresse>
    <Rue>Des Moines</Rue>
    <Ville>Paris</Ville>
  </Adresse>
  <Téléphone>0148253278</Téléphone>
  <Manager>Dupuis</Manager>
</Restaurant>
<Bar type="anglais">
  <Nom>Rose and Crown</Nom>
</Bar>
</Guide>
```

# Exemple

---

- Sélection: lister les noms et téléphones des restaurants de Cabourg

- Résultat

```
<result>
<titre>Restaurants de Cabourg</titre>
{for $R in document("Guide")/Restaurant
where $R//Ville = "Cabourg"
return <Restaurant>
  <Nom>{$R/Nom}</Nom>
  <Tel>{$R/Téléphone}</Tel>
</Restaurant>}
</result>
```

```
<Guide Version= "2.0">
<Restaurant type="français" categorie="***">
  <Nom>Le Moulin</Nom>
  <Adresse>
    <Rue>des Vignes</Rue>
    <Ville>Mougins</Ville>
  </Adresse>
  <Manager>Dupuis</Manager>
</Restaurant>
<Restaurant type="français" categorie="***">
  <Nom>La Licorne</Nom>
  <Adresse>
    <Rue>Des Moines</Rue>
    <Ville>Paris</Ville>
  </Adresse>
  <Téléphone>0148253278</Téléphone>
  <Manager>Dupuis</Manager>
</Restaurant>
<Bar type="anglais">
  <Nom>Rose and Crown</Nom>
</Bar>
</Guide>
```

# Jointure

---

- Jointure: Lister le nom des Restaurants avec téléphone dans la rue de l'Hôtel Lutecia
- Résultat

```
<Guide Version= "2.0">
<Restaurant type="français" categorie="***">
  <Nom>Le Moulin</Nom>
  <Adresse>
    <Rue>des Vignes</Rue>
    <Ville>Mougins</Ville>
  </Adresse>
  <Manager>Dupuis</Manager>
</Restaurant>
<Restaurant type="français" categorie="***">
  <Nom>La Licorne</Nom>
  <Adresse>
    <Rue>Des Moines</Rue>
    <Ville>Paris</Ville>
  </Adresse>
  <Téléphone>0148253278</Téléphone>
  <Manager>Dupuis</Manager>
</Restaurant>
<Bar type="anglais">
  <Nom>Rose and Crown</Nom>
</Bar>
</Guide>
```

# Jointure

- Jointure: Lister le nom des Restaurants avec téléphone dans la rue de l'Hôtel Lutecia
- Résultat

```
for $R in collection("Guide")/Restaurant,
    $H in document(Répertoire)/Hotel
where $H//Rue = $R//Rue
    AND $H//Nom= "Le Lutecia"
return
    <Result>
        <Nom>{$R/Nom}</Nom>
        <Tel>{$R/Téléphone}</Tel>
    </Result>
```

```
<Guide Version= "2.0">
<Restaurant type="français" categorie="****">
    <Nom>Le Moulin</Nom>
    <Adresse>
        <Rue>des Vignes</Rue>
        <Ville>Mougins</Ville>
    </Adresse>
    <Manager>Dupuis</Manager>
</Restaurant>
<Restaurant type="français" categorie="***">
    <Nom>La Licorne</Nom>
    <Adresse>
        <Rue>Des Moines</Rue>
        <Ville>Paris</Ville>
    </Adresse>
    <Téléphone>0148253278</Téléphone>
    <Manager>Dupuis</Manager>
</Restaurant>
<Bar type="anglais">
    <Nom>Rose and Crown</Nom>
</Bar>
</Guide>
```

# Agrégat

---

- Combien de restaurants y-a-t-il dans les guides?
- Résultat

# Agrégat

---

- Combien de restaurants y-a-t-il dans les guides?
- Résultat

```
<result>
  <entete>Nombre total de restaurants</entete>
  {let $R := collection("Guide")/Restaurant
   return <NombreRestaurant>
    {count ($R)}
   </NombreRestaurant>}
</result>
```

# XQuery et XSLT

---

- XSLT est un peu plus verbeux que XQuery
  - c'est du XML
  - c'est un peu moins vrai en XSLT 2.0
- XSLT est un système basé sur deux langages: XPath pour les expressions et XSLT pour les instructions
- Le fait que XSLT soit en XML présente plusieurs avantages
  - réutilisation de tous les outils XML
  - la syntaxe est extensible
  - liens avec les autres langages XML (XSD,...)
- Le résultat d'une requête XSLT est un (ou plusieurs) document(s) XML, alors que cela peut être un nombre pour une requête XQuery
- En XQuery, il n'y a pas de apply-templates, donc la récursivité doit être programmée à la main
- XQuery est plus modulaire que XSLT (l'import dans XSLT est assez peu puissant)



## XQuery et XSLT (2)

---

- On peut facilement transformer presque toutes les expressions FLWR en équivalent XSLT

Construction XQuery	Équivalent XSLT
for \$var in SEQ	<xsl:for-each select="SEQ"> <xsl:variable name="var" select="."/>
let \$var := SEQ	<xsl:variable name="var" select="SEQ"/>
where CONDITION	<xsl:if test="CONDITION">
order by \$x/VALUE	<xsl:sort select="VALUE">

## XQuery et XSLT (3)

---

- Soit la requête suivante (extraite du benchmark XMark)

```
for $b in doc("auction.xml")/site/regions//item
let $k := $b/name
order by $k
return <item name="{ $k }">{ $b/location } </item>
```

- Son équivalent XSLT

```
<xsl:for-each select="doc('auction.xml')/site/regions//item">
  <xsl:sort select="name"/>
  <item name="{name}"
    <xsl:value-of select="location"/>
  </item>
</xsl:for-each>
```

## XQuery et XSLT (4)

---

- Soit la requête suivante: liste des éditeurs qui ont publié plus de 100 livres

```
<big_publishers>
  FOR $p IN distinct(document("bib.xml")//publisher)
  LET $b := document("bib.xml")/book[publisher = $p]
  WHERE count($b) > 100
  RETURN $p
</big_publishers>
```

- Son équivalent XSLT

```
<big_publishers xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:for-each select="document('bib.xml')//publisher[not(.=preceding::publisher)]">
    <xsl:variable name="b" select="document('bib.xml')/book[publisher=current()]" />
    <xsl:if test="count($b) > 100">
      <xsl:copy-of select="." />
    </xsl:if>
  </xsl:for-each>
</big_publishers>
```

## XQuery et XSLT (4)

---

- Soit la requête suivante: liste des éditeurs qui ont publié plus de 100 livres

```
<big_publishers>
```

```
  FOR $p IN distinct(document("bib.xml")//publisher)
```

```
    LET $b := document("bib.xml")/book[publisher = $p]
```

```
    WHERE count($b) > 100
```

```
    RETURN $p
```

```
</big_publishers>
```

- Son équivalent XSLT

```
<big_publishers xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
  <xsl:for-each select="document('bib.xml')//publisher[not(.=preceding::publisher)]">
```

```
    <xsl:variable name="b" select="document('bib.xml')/book[publisher=current()]" />
```

```
    <xsl:if test="count($b) > 100">
```

```
      <xsl:copy-of select="." />
```

```
    </xsl:if>
```

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

```
</big_publishers>
```

## XQuery et XSLT (4)

---

- Soit la requête suivante: liste des éditeurs qui ont publié plus de 100 livres

```
<big_publishers>  
  FOR $p IN distinct(document("bib.xml")//publisher)  
  LET $b := document("bib.xml")/book[publisher = $p]  
  WHERE count($b) > 100  
  RETURN $p  
</big_publishers>
```

- Son équivalent XSLT

```
<big_publishers xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  
  <xsl:for-each select="document('bib.xml')//publisher[not(.=preceding::publisher)]">  
    <xsl:variable name="b" select="document('bib.xml')/book[publisher=current()]" />  
    <xsl:if test="count($b) > 100">  
      <xsl:copy-of select="." />  
    </xsl:if>  
  </xsl:for-each>  
</big_publishers>
```

## XQuery et XSLT (4)

---

- Soit la requête suivante: liste des éditeurs qui ont publié plus de 100 livres

```
<big_publishers>
```

```
FOR $p IN distinct(document("bib.xml")//publisher)
```

```
LET $b := document("bib.xml")/book[publisher = $p]
```

```
WHERE count($b) > 100
```

```
RETURN $p
```

```
</big_publishers>
```

- Son équivalent XSLT

```
<big_publishers xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:for-each select="document('bib.xml')//publisher[not(.=preceding::publisher)]">
```

```
<xsl:variable name="b" select="document('bib.xml')/book[publisher=current()]" />
```

```
<xsl:if test="count($b) > 100">
```

```
<xsl:copy-of select="."/>
```

```
</xsl:if>
```

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

```
</big_publishers>
```

## XQuery et XSLT (4)

---

- Soit la requête suivante: liste des éditeurs qui ont publié plus de 100 livres

```
<big_publishers>
```

```
FOR $p IN distinct(document("bib.xml")//publisher)
```

```
LET $b := document("bib.xml")/book[publisher = $p]
```

```
WHERE count($b) > 100
```

```
RETURN $p
```

```
</big_publishers>
```

- Son équivalent XSLT

```
<big_publishers xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:for-each select="document('bib.xml')//publisher[not(.=preceding::publisher)]">
```

```
<xsl:variable name="b" select="document('bib.xml')/book[publisher=current()]" />
```

```
<xsl:if test="count($b) > 100">
```

```
<xsl:copy-of select="." />
```

```
</xsl:if>
```

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

```
</big_publishers>
```

## XQuery et XSLT (4)

---

- Soit la requête suivante: liste des éditeurs qui ont publié plus de 100 livres

```
<big_publishers>  
  FOR $p IN distinct(document("bib.xml")//publisher)  
  LET $b := document("bib.xml")/book[publisher = $p]  
  WHERE count($b) > 100  
  RETURN $p  
</big_publishers>
```

- Son équivalent XSLT

```
<big_publishers xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  
  <xsl:for-each select="document('bib.xml')//publisher[not(.=preceding::publisher)]">  
    <xsl:variable name="b" select="document('bib.xml')/book[publisher=current()]" />  
    <xsl:if test="count($b) > 100">  
      <xsl:copy-of select="." />  
    </xsl:if>  
  </xsl:for-each>  
</big_publishers>
```



Intégration avec l'existant

---

# SGBD et XML

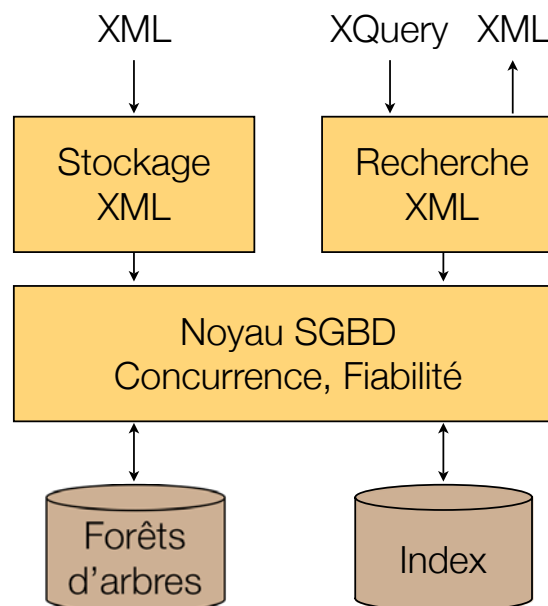
---

- Deux possibilités
  - les systèmes natifs
  - les extensions objet-relationnel
- Systèmes natifs
  - technique spécialisée de stockage et recherche
  - indexation des arbres XML
  - gèrent seulement du XML
  - langage de requêtes XQuery
- Extensions objet-relationnel
  - de plus en plus intégré aux grands SGBD
  - deux techniques:
    - colonne «objet XML »
    - mapping: 1 document → N tables
  - langage de requêtes hybride: SQL étendu avec fonctionnalités XQuery

# SGBD natif

---

- SGBD natif
  - conçu pour XML,
  - stockant les documents entiers sans les décomposer en éléments,
  - utilisant de techniques d'indexation d'arbres spécifiques



## SGBD natif (2)

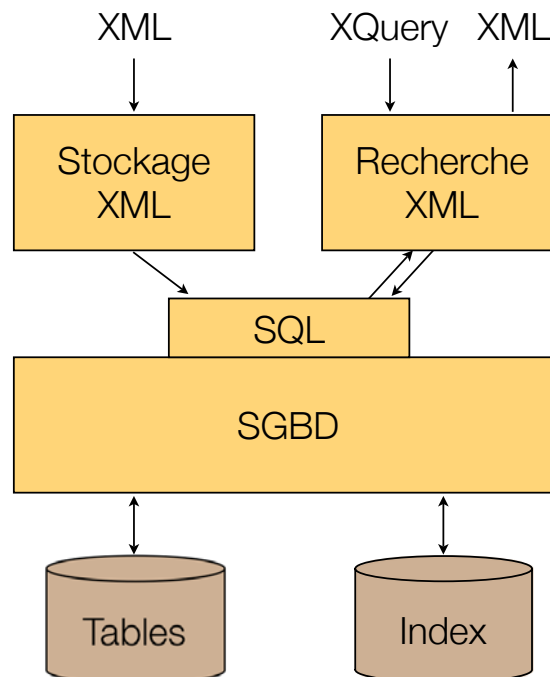
---

- Techniques
  - stockage d'arbres
  - index de structure (arbres)
  - index de contenu (mots clés)
  - algèbre XML
  - optimisation XQuery
  - passage à l'échelle
  - gestion de versions
- Produits commerciaux
  - Software A.G. Tamino, X-Hive/Db, XML Global, Coherity,...
- Produits open source
  - Exist (<http://exist.sourceforge.net/index.html>)
  - XIndice (<http://xml.apache.org/xindice/>)
  - Berkeley DB XML (<http://www.sleepycat.com/>) (racheté par Oracle en février 2006)

# Extension du relationnel

---

- Composant logiciel intégré à un SGBD (objet-relationnel) assurant:
  - le stockage et l'interrogation de documents XML
  - en transformant le XML en tables
  - et les tables en XML



## Extension du relationnel (2)

---

- Passage SQL/XML
- Intégration de fonctionnalités XQuery à SQL
- Support à la SQL3
  - type de donnée natif XML Type (colonnes XML)
  - fonctions d'extraction XPath
  - fonctions de construction de XML (pont relationnel)
  - insertion et mise à jour de XML en colonne(s)
- Exemple de requête

```
SELECT XMLElement("Emp", XMLForest( e.hire, e.dept AS "department")) AS "result"
FROM EMPLOYEE e
WHERE ExtractValue(e.XMLemp, /emp/@id) > 200;
```
- Intégré à Oracle et DB2

## Extension du relationnel (3)

---

- Stockage et publication
  - mapping de XML plat sur une table
  - mapping de XML imbriqué en tables imbriquées
  - stockage de XML en colonne (XML Type)
  - commandes PutXml et GetXml
- Interrogation
  - support de SQL/XML
  - ServletXSQL
    - document XML avec requêtes SQL/XML
    - transformation du résultat des requêtes en XML

# Comparaison

---

- Points forts xml-objet-relationnel
  - pas de nouveau SGBD
  - possibilité de normaliser les données
  - possibilité de stocker une colonne comme attribut ou élément
  - une certaine portabilité multi-SGBD
  - performance pour accès grain fin
- Points forts Natif
  - un nouveau SGBD fait pour XML
  - jamais de mapping à définir et maintenir
  - intégrité du document
  - recherche plein texte
  - performance pour accès gros grain