# XML Schemaについて

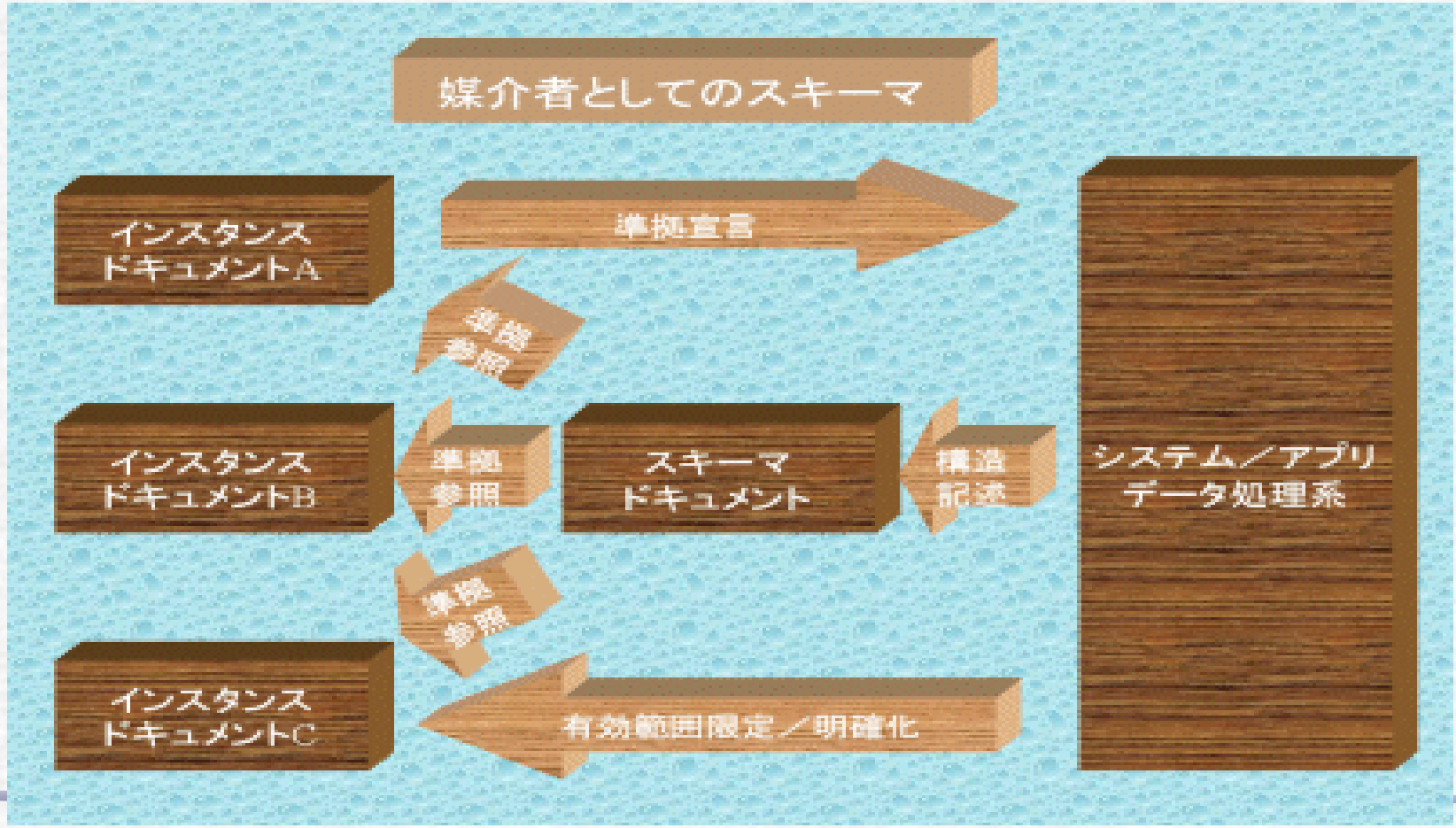
XMLｺﾝｿｰｼｱﾑ基盤技術部会
共通基盤ｸﾞﾙｰﾌﾟXMLｽｷｰﾏｻﾌﾞｸﾞﾙｰﾌﾟ

# 目　　次

# XMLスキーマとは何か

- ある特定の語彙体系に所属する一連のXMLインスタンスドキュメントの構造をXML自身によりメタ定義する。
- 多くのXMLスキーマは文法定義的であり、文書の全体構造を規定すること（主に要素と属性の出現形態を規定する）にその主眼がある（但し、後述するSchematronのようなルールベースのスキーマも存在する）。
- XMLの名前空間の考え方と合わせ、ある特定のドキュメント構造を共有する1つのシステム空間の構造定義を行う。
- システム／アプリケーションのデータ処理系とデータ（インスタンスドキュメント）との媒介的機能を持つとも考えられる。
- 1つのインスタンスドキュメントが複数の語彙体系を参照してもよい
- XMLスキーマによって定義されるデータ構造は、それを理解するある特定のシステム／アプリケーションの実装に依存しなくともよい（インターフェース的な考え方との類似性）。すなわちデータ構造とシステムの実装を明確に区分できる。

# XMLスキーマとは何か



# XMLスキーマとは何か

- **共通語彙の定義**
  明確に規定された一連のルールに基く情報処理プロセスの確立
- **フォーマルなルール定義**
  コンピュータが理解することが出来る様式でルールが記述されるべき
- **実装に依存しない語彙定義**
  特定の語彙体系を理解する特定のシステムの実装に依存しない様式で語彙が定義出来る仕組みが提供されるべき (名前空間＋スキーマ)
- **契約事項の確立**
  共通語彙の規定を可能にするが故、人間の目から見ても特定の契約事項のルールを明確化することに役立つ (たとえばSOAP,UDDI,WSDL等のXMLを利用した仕様には、XML Schemaで記述された語彙定義が付加されている)

# XMLスキーマとは何か

- DTDでは何故不足か？

・DTD自体はXMLで記述されない。
(A)固有のシンタックスを学習する必要がある。
(B)XMLアプリケーションによりXMLドキュメントとしてハンドリング出来ない。
(C)拡張性がない。
・ネームスペースのサポートが現状ではない。この為、DTDはネーミングコンフリクトを避ける手段がない為、ローカルな語彙定義或いはただ1つの語彙体系に参照することしか出来ないことになる。
・データタイプを規定する方法が殊に要素に関しては極めて限定される。
・要素の繰返し回数の指定にフレキシビリティがない。
・要素型に関して親子関係等の構造的関係の規定が出来ない（拡張性／再利用性がない）。

※最初DTDは煩雑であるという理由もあったが、この点に関してはXML Schemaの方がはるかに複雑になってしまった。

# XMLスキーマとは何か

- XMLスキーマの種類

  - DTD
  - XML Schema (W3C)
  - XML-Data (Microsoft社他)
  - XDR (Microsoft社他)
  - SOX (Commerce One社)
  - DSD (AT&T研究所他)
  - Schematron (Rick Jelliffe氏)
  - RELAX NG (村田真氏、James Clark氏)
  - その他

# XML Schemaの機能

- 2001年5月にW3CのRecommmendationとして承認

・XML Schema Part1:Structure
　　http://www.w3.org/TR/xmlschema-1/
　　**構造面（シンタックス）の定義**
・XML Schema Part2:Datatypes
　　http://www.w3.org/TR/xmlschema-2/
　　**組込みデータ型の定義、ユーザ定義型の派生方法等**
・XML Schema Part0:Primer
　　http://www.w3.org/TR/xmlschema-0/
　　**入門編**

- **その他参考サイト**

http://www.oasis-open.org/cover/schemas.html
http://www.xfront.com/BestPracticesHomepage.html
http://www.xml.com/schemas/
http://schemas.hr-xml.org/xc/canon/TSC/SchemaDesignGuidelines-1_0-20010712.pdf

# XML Schemaの機能

## 用語定義

- **インスタンスドキュメント**
  XMLの形式により記述されるデータを指す。通常はファイル (*.xml) 内にストアされるが、場合によってはメモリ内に過渡的にDOM等の形式で管理されているケースも存在する。
- **スキーマドキュメント**
  XML Schemaによるドキュメント構造定義ファイル(*.xsd)を指す。バリデータ等のアプリケーションは、インスタンスドキュメントとスキーマドキュメント双方を入力としなければならない。
- **validなインスタンスドキュメント**
  あるインスタンスドキュメントが対応するスキーマドキュメントの構造と正確に一致する場合、そのインスタンスドキュメントはそのスキーマドキュメントに対してValidであると言う。尚、validでなかった場合アプリケーションがどう対応すべきかということに関しては、XML Schema仕様の関知するところではない (これはたとえばJavaの場合であるとJAXP仕様等で規定されるべきものである) 。
- **バリデーション**
  あるインスタンスドキュメントが対応するスキーマドキュメントの構造に一致するか否かを判定する処理をバリデーションと呼び、専らこのバリデーションを行うアプリケーションをバリデータと呼ぶ。
- **要素と要素定義 (当ドキュメント内での定義)**
  これは一般的な区分ではないが、当ドキュメント範囲内においてはインスタンスドキュメント内に出現する要素は単に要素と呼び、スキーマドキュメント内での<element>要素による要素の出現の定義 (後述) は要素定義と呼び区別する。

# XML Schemaの機能

## 要素定義と型定義

### (1) 要素定義

・インスタンスドキュメント中に現れる要素 (element) の出現をスキーマドキュメント中で<element>タグにより指定出来る。またグローバル定義 (後述) されている要素定義は、他の要素定義からref属性により参照することが出来る。

<element name="comment" type="string" />・・・要素定義例

<element ref="tns:comment" />・・・参照例

・要素定義のname属性に一致する要素が対応するインスタンスドキュメント中に出現した時、その要素の構造が正確に要素定義の構造に合致しないと、そのインスタンスドキュメントはそのスキーマドキュメントに対してvalidであるとは見なされない。

### (2) 型定義

・<simpleType>タグ、<complexType>タグを用いて要素を型定義することが出来る。グローバル定義 (後述) されている型は、スキーマドキュメント或いはインスタンスドキュメント (実際にはかなり特殊な場合に限られる) からtype属性により参照することが出来る。

<complexType name="PurchaseOrderType">・・・型定義例

    ・・・

</complexType>

<element name="purchaseOrder" type="tns:PurchaseOrderType" />・・・型参照例

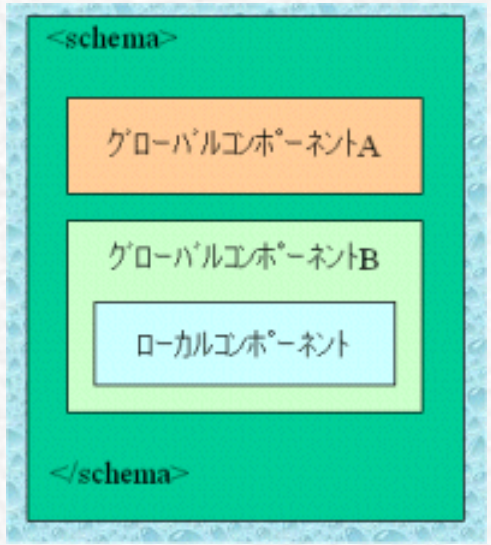
# XML Schemaの機能

## ローカルとグローバル

<element>、<simpleType>、<complexType>のいずれかにより定義される要素、型（ここではコンポーネントと呼ぶ）が、スキーマドキュメントのルート要素である<schema>要素直下に定義された場合、これらはグローバルであると呼ばれ、他の要素定義よりref又はtype属性を用いて参照することが出来る。そうではなく他のコンポーネント内に包含されるコンポーネントはローカルであると呼ばれ、それを包含するコンポーネント内にローカライズされる。

右図においては、コンポーネントA及びコンポーネントBはグローバル定義されているので互いに他のコンポーネントを参照出来る。しかしながら、コンポーネントBの内部に定義されているコンポーネントはローカルなのでコンポーネントAの内部から参照することが出来ない。

※一部のパワーユーザは、専らローカルコンポーネントによりスキーマを構成する流儀を「Russian Doll（ロシア人形）」パターン、専らグローバルコンポーネントにより構成する流儀を「Salami Slice（サラミスライス）」パターンと呼んでいる。

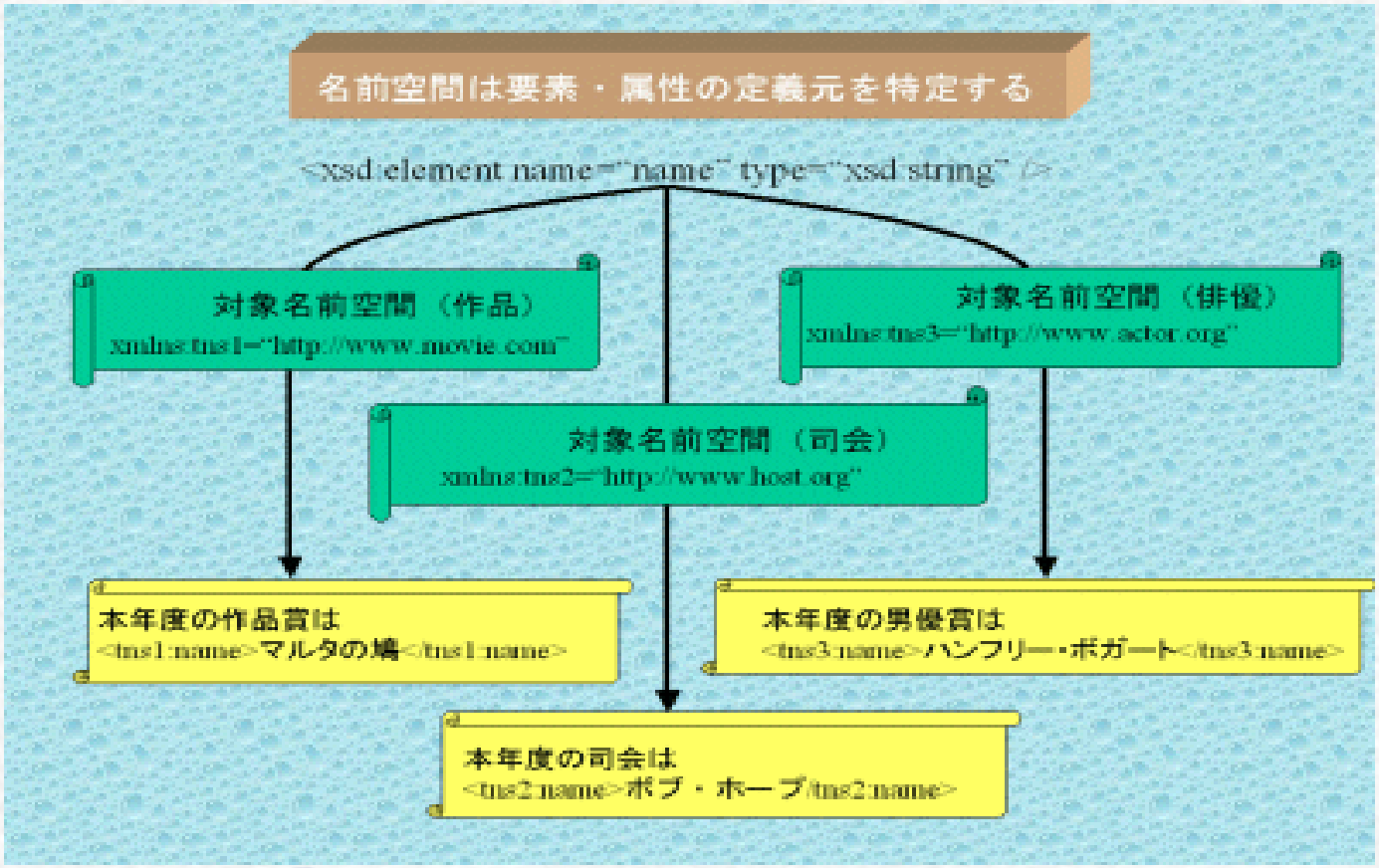


# XML Schemaの機能

## 名前空間

**・名前空間はドキュメント（インスタンス、スキーマ共）が依拠する語彙体系を明確化する。**
**・1つのドキュメントが複数の語彙体系に参照してもよい。**
**・XML SchemaはXMLの名前空間の概念をサポートする。**
（1）ある特定の名前空間（対象名前空間）の語彙構造を記述するのがXML Schemaスキーマドキュメントである。
（2）XML Schemaドキュメント内でも名前空間を指定することにより、複数の語彙体系に参照することが出来る。
**・名前空間はURI表記される。**
が、対応する物理ロケーションに当該スキーマドキュメントが存在しなければならないということではない。
**・名前空間はコロンセパレータにより分離される名前空間修飾子を要素、属性等に付加することにより識別する。**

# XML Schemaの機能



# XML Schemaの機能

## 名前空間（スキーマドキュメント内）

### ・XML Schema固有の名前空間

XML Schemaで定義されている語彙（たとえば<element>、<complexType>等）を定義。
URI＝http://www.w3.org/2001/XMLSchemaで固定。

### ・対象名前空間

定義対象となるスキーマドキュメントが所属する名前空間。targetNamespace属性により指定される。

### ・その他の名前空間

後述するインポート機能により他の名前空間に属するスキーマドキュメントを読込むことが出来る。

### ・デフォルト名前空間

1つのスキーマドキュメントは、上記のように複数の名前空間から構成されるのが普通である。しかしながら、全ての要素／属性定義に名前空間修飾子を付加するとドキュメントが見にくくなる。この為名前空間修飾子を付加する必要のないデフォルト名前空間を定義することが出来る。たとえばXML Schema固有の名前空間をデフォルト名前空間とすれば、たとえば<element>、<complexType>というようなスキーマドキュメントに最も頻繁に現れる要素定義に対し名前空間修飾子を付加する必要がなくなる。

※対象名前空間は必ずしも指定される必要はないが、この場合対応するスキーマドキュメント中の要素／型定義は全て名前空間修飾なしで参照される。この為、XML Schema固有の名前空間をデフォルト空間に指定するとどちらの名前空間に属するか曖昧になるケースが発生するので、XML Schema固有の名前空間を名前空間修飾する必要が生じる。

# XML Schemaの機能

## 名前空間（スキーマドキュメント例）

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://www.example.com/PO1"
        targetNamespace= "http://www.example.com/PO1">
  <element name="purchaseOrder" type="tns:PurchaseOrderType" />
  <complexType name="PurchaseOrderType">
    ・・・複合型の定義
  </complexType>
</schema>
```

※上記例において「xmlns="http://www.w3.org/2001/XMLSchema"」の部分がデフォルト名前空間の指定にあたる。すなわちXML Schema固有の名前空間がデフォルト名前空間に指定されているので、<schema>、<element>、<complexType>等には名前空間修飾が行われていない。
それに対し対象名前空間はデフォルト名前空間に指定されていないので、
「type="tns:PurchaseOrderType"」のように名前空間修飾が行われている。

# XML Schemaの機能

- **名前空間（インスタンスドキュメント内）**

・インスタンスは自身が語彙定義を行うわけではないのでそれ自身に対する対象名前空間指定はない。

・要素に関してXML Schema固有の名前空間に所属する要素を参照することはないが属性に関してはtype、nil等の若干の例外がある。この名前空間は以下のURIにより指定される。
http://www.w3.org/2001/XMLSchema-instance

・対応するスキーマドキュメントでローカル定義されている要素及び属性に関して、名前空間修飾が必要である場合とない場合がある。この指定は、対応するスキーマドキュメント内で、elementFormDefault（要素）、attributeFormDefault（属性）を記述することにより変更可能である。これらの属性がunqualified（デフォルト）に指定されているとローカル要素／属性に対して名前空間修飾は不要であるが、qualifiedに指定されていると必要になる（例1－1、2、3参照）。

# XML Schemaの機能（例1－1）

- **ローカル要素／属性の名前空間修飾**

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1">
  <element name="purchaseOrder" type="tns:PurchaseOrderType" />
  <complexType name="PurchaseOrderType">
    ・・・複合型の定義（内容省略）
  </complexType>
</schema>
```

※上記スキーマドキュメントには、elementFormDefaultもattributeFormDefaultも指定されていないので要素／属性双方に関してunqualifiedであると見なされローカルの要素／属性に関して名前空間修飾をする必要がない。対応するvalidなドキュメントインスタンスは以下のようになる。

```
<tns:purchaseOrder xmlns:tns="http://www.example.com/PO1">
  <shipTo country="US">
    <name>Hiroshi Takahashi</name>
    ・・・
  </shipTo>
</tns:purchaseOrder>
```

# XML Schemaの機能（例1－2）

- **ローカル要素／属性の名前空間修飾**

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1"
        elementFormDefault="qualified">
  <element name="purchaseOrder" type="tns:PurchaseOrderType" />
  <complexType name="PurchaseOrderType">
    ・・・複合型の定義（内容省略）
  </complexType>
</schema>
```

※付録例1-2とは違い上記スキーマドキュメントでは、elementFormDefaultがqualifiedに指定されているので、validなインスタンスドキュメントは以下のようになる。全ての要素が名前空間修飾されていることに注意のこと。

```
<tns:purchaseOrder xmlns:tns="http://www.example.com/PO1">
  <tns:shipTo country="US">
    <tns:name>Hiroshi Takahashi</tns:name>
    ・・・
  </tns:shipTo>
</tns:purchaseOrder>
```

# XML Schemaの機能（例1－3）

- **ローカル要素／属性の名前空間修飾**

  **しかしながらインスタンスドキュメント中で対応名前空間をデフォルト名前空間に指定すると以下のように名前空間修飾が不要になる。**

```
<purchaseOrder xmlns=“http://www.example.com/PO1”>
    <shipTo country=“US”>
        <name>Hiroshi Takahashi</name>
        ・・・
    </shipTo>
</purchaseOrder>
```

# XML Schemaの機能

- **インクルード**(include)**とインポート**(import)

  あるスキーマドキュメントから他のスキーマドキュメントを参照する方法。

  **(1) インクルード**

  参照元と参照先双方の対象名前空間が同一であるか、参照先の対象名前空間が未指定でなければならない。後者の場合、参照先の対象名前空間は参照元の対象名前空間と同一であると見なされる。一部のパワーユーザはこの後者のケースが利用される場合、カメレオンデザインと呼んでいる。

  **(2) インポート**

  参照元と参照先双方の対象名前空間が異なる場合は、インポートが用いられねばならない。

# XML Schemaの機能

- **単純型** (simpleType) **と複合型** (complexType)
  - **単純型**

    **それ自身の内に更に要素、属性、型定義を含むことが出来ない型を言う。ベースとなる組込みデータ型に後述する制限による派生を適用して新しい単純型を定義することが可能である (例2－1参照)。**

  - **複合型**

    **それ自身の内に更に要素、属性、型定義を含むことが出来る型を言う。従って典型的には、複合型定義は該当タイプに所属する要素を定義する<element>タグ、所属属性を定義する<attribute>タグ、包含階層を含む場合には包含される複合型を定義する<complexType>タグから構成される (例2－2参照)。**

# XML Schemaの機能（例2－1）

- minInclusive、maxInclusiveによる制限例

```
<simpleType name=“myInteger”>
   <restriction base=“integer”>
      <minInclusive value=“10000” />
      <maxInclusive value=“99999” />
   </restriction>
</simpleType>
```

※組込み型integerから新たなユーザ定義単純型myInteger（10000以上99999以下整数値）を派生

- Enumerationによる制限例

```
<simpleType name=“USState”>
   <restriction base=“string”>
      <enumeration value=“AK” />
      <enumeration value=“AL” />
      ‘ ‘ ‘
   </restriction>
</simpleType>
```

※組込み型stringから新たなユーザ定義単純型USState（合衆国短縮州名リスト）を派生

# XML Schemaの機能（例2－2）

- **複合型の定義**

```
<element name=“shipTo type=“tns:Address”>
<complexType name=“Address”>
    <sequence>
        <element name=“name” type=“string” />
        <element name=“street” type=“string” />
        <element name=“city” type=“string” />
        <element name=“state” type=“string” />
    </sequence>
    <attribute name=“country” type=“NMTOKEN” />
</complexType>
```

※上記例は、4つの要素と1つの属性から構成される複合型Addressを定義。<sequence>とは、下位定義されている要素が、記述の順番にインスタンスドキュメントに出現しなければならないことを意味し、他には<all>（出現順序不定）、<choice>（複数要素からの単一要素選択）などがある。尚上記スキーマにvalidであるインスタンスドキュメントは以下のようになります。

```
<shipTo countrt=“US”>
    <name>Hiroshi Takahashi</name>
    <street>Second Avenue</street>
    <city>New York</city>
    <state>New York</state>
</shipTo>
```

# XML Schemaの機能

- **拡張（extention）及び制限（restriction）による派生**

**（1）拡張**

ベースとなる型の持つ要素や属性に新たに要素や属性を追加することにより新たな型を生成する（例3－1、2参照）。

**（2）制限**

**・単純型の場合**

ベースとなる単純型に制限記述子を適用することにより、より値の包含範囲の狭い新たな単純型を定義することが出来る。

**・複合型の場合**

ベースとなる型の要素／属性の限定或いは特定要素の出現回数等の減少方向修正により新たな複合型を定義することが出来る。尚複合型の制限を行う場合、制限の対象にならない要素／属性は全て派生される型定義に再記述される必要がある。

# XML Schemaの機能（例3－1）

- **複合型の拡張による派生例**

```
<element name="shipTo" type="tns:JPAddress" />
<complexType name="Address">
   <sequence>
      <element name="name" type="string" />
      <element name="street" type="string" />
      <element name="city" type="string" />
   </sequence>
</complexType>
<complexType name="JPAddress">
   <complexContent>
      <extention base="tns:Address">
         <sequence>
            <element name="prefecture" type="string" />
         </sequence>
      </extention>
   </complexContent>
</complexType>
```

※上記例では、拡張派生複合型であるJPAddressは要素として<name>,<street>,<city>,<prefecture>から構成されることになる。これに対応するvalidなインスタンスドキュメントは付録例3－2のようになる。

# XML Schemaの機能（例3－2）

- Validなインスタンスドキュメント

```
<shipTo>
    <name>高橋洋</name>
    <street>豊岡1－6－12</street>
    <city>入間</city>
    <prefecture>埼玉</prefecture>
</shipTo>
```

# XML Schemaの機能

- **要素出現回数指定**

DTDには要素の出現回数を指定する方法として無修飾（必ず1回）、？修飾（0又は1回）、＋修飾（1からn回）、＊修飾（0からn回）があったが、それ以上の指定を行うことが出来なかった。たとえばDTDで「要素A中に要素Bが3回以上10回以下出現する」を表現するには、

<!ELEMENT A (B,B,B,B?,B?,B?,B?,B?,B?,B?)>

のように記述するしかなかった。

これに対しXML Schemaでは、要素の出現回数を正確に記述することが出来る

minOccurs（最少出現回数）、maxOccurs（最多出現回数）という属性が用意されている。それぞれのデフォルトは1であり、maxOccurs="unbounded"と指定されると最多出現回数は無限となる。

<element name="B" minOccurs="3" maxOccurs="10" />

# XML Schemaの機能

## 抽象要素定義

要素をabstractとして定義する。スキーマドキュメント中でabstractとして定義された要素定義に対応する要素は、インスタンスドキュメント中に出現することは出来ないが、その要素定義を代替グループとして参照する要素定義に対応する要素が、abstractとして定義されている要素定義に対応する要素の代わりに出現することが出来る。これによりインスタンスドキュメントの特定箇所に型を共有する複数の要素がポリモルフィックに出現可能になる（例4－1、2、3参照）。

# XML Schemaの機能（例4－1）

- **抽象要素定義例（前半）**

```
<element name="VehicleCatalogue">
   <complexType>
      <sequence>
         <element ref="tns:Vehicle maxOccurs="unbounded" />
      </sequence>
   </complexType>
</element>
<element name="Vehicle" abstract="true" type="tns:VehicleType">
<complexType name="VehicleType">
   <sequence>
      <element name="name" type="string" />
      <element name="manufacturer" type="string" />
      <element name="price" type="positiveInteger" />
   </sequence>
</complexType>
```

※<VehicleCatalogue>要素には、<Vehicle>要素が1回以上出現する。しかしながら、<Vehicle>要素はabstract="true"と指定されているので、インスタンスドキュメント中ではそのまま出現することが出来ない。

# XML Schemaの機能（例4－2）

## 抽象要素定義例（後半）

```
<element name="Car" substitutionGroup="tns:Vehicle" type="tns:CarType">
<complexType name="CarType">
   <complexContent>
      <extention base="tns:VehicleType">
         <sequence>
            ・・・（自動車固有要素追加）
         </sequence>
      </extention>
   </complexContent>
</complexType>
<element name="AirPlane" substitutionGroup="tns:Vehicle" type="tns:AirPlaneType">
<complexType name="AirPlaneType">
   <complexContent>
      <extention base="tns:VehicelType">
         <sequence>
            ・・・（飛行機固有要素追加）
         </sequence>
      </extention>
   </complexContent>
</complexType>
```

※<Car>要素、<AirPlane>要素は<Vehicle>要素の代替グループに所属するので、インスタンスドキュメント中の<Vehicle>要素が論理的に出現可能な箇所には<Car>要素、<AirPlane>要素が出現可能である。

# XML Schemaの機能（例4－3）

- **抽象要素定義例**（validなｲﾝｽﾀﾝｽﾄﾞｷｭﾒﾝﾄ）

```
<VehicleCatalogue xmlns="Some URI">
    <Car>
        <name>乗用車A</name>
        <manufacturer>乗用車メーカA</manufacturer>
        <price>5000000</price>
        ・・・（自動車固有要素）
    </Car>
    <AirPlane>
        <name>ジェット機A</name>
        <manufacturer>ジェット機メーカA</manufacturer>
        <price>300000000</price>
        ・・・（飛行機固有要素）
    </AirPlane>
</VehicleCatalogue>
```

# XML Schemaの機能

- **抽象型定義**

  型をabstractとして定義する。Abstract定義されている型をtypeによって参照する要素定義に対応するインスタンスドキュメント中の要素については、このabstractとして定義されている型を派生して定義されるabstractではない型をtype属性を用いて指定しなければならない（例5－1、2参照）。

# XML Schemaの機能（例5－1）

- **抽象型定義例**

```
<element name="VehicleCatalogue">
   <complexType>
      <sequence>
         <element ref="tns:Vehicel" maxOccurs="unbounded" />
      </sequence>
   </complexType>
</element>
<element name="Vehicel" type="tns:VehicelType">
<complexType name="VehcleType" abstract="true">
   <sequence>
      <element name="name" type="string" />
      <element name="manufacturer" type="string" />
      <element name="price" type="positiveInteger" />
   </sequence>
</complexType>
```

※抽象要素定義例と違い、要素<Vehicle>は抽象指定されておらず、要素<Vehicle>が参照する型VehicleTypeがabstract="true"により抽象指定されている。尚、CarTypeとAirPlaneTypeの定義は抽象要素定義例と同様であるが、対応する要素定義は抽象型定義の場合には必要ない。

# XML Schemaの機能（例5－2）

- **抽象型定義例**（validなｲﾝｽﾀﾝｽﾄﾞｷｭﾒﾝﾄ）

```
<VehicleCatalogue xmlns="Some URI"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Vehicle> xsi:type="CarType">
    <name>乗用車A</name>
    <manufacturer>乗用車メーカA</manufacturer>
    <price>5000000</price>
    ・・・（自動車固有要素）
  </Vehicle>
  <Vehicle xsi:type="AirPlaneType">
    <name>ジェット機A</name>
    <manufacturer>ジェット機メーカA</manufacturer>
    <price>300000000</price>
    ・・・（飛行機固有要素）
  </Vehicle>
</VehicleCatalogue>
```

# XML Schemaの機能

## 一意性指定

特定の属性／要素或いはそれらの組合わせをXPath指定される範囲内においてユニークでなければならないものとして定義することが出来る。primerにおける例を引用すると、

```
<unique>
    <selector xpath=“r:region/r:zip” />
    <field xpath=“@code”>
</unique>
```

<selector>要素のxpath属性で指定されるXPath表記が、ユニーク指定される対象範囲を定義する。<field>要素は、指定範囲内でユニークでなければならない要素又は属性を定義する。
<field>要素を複数指定して複数の要素／定義の組合わせをユニークでなければならないものとして定義することが出来る。

# XML Schemaの機能

## キー参照定義

ある特定の属性／要素或いはそれらの組合わせをユニークなキー（且つnilであってはならない）として定義することが出来る。またこのキーに対して<keyref>指定を用いて他の要素からの参照を設定することが出来る。primerにおける例を引用すると、

```
<key name="pNumKey">
    <selector xpath="r:parts/r:part" />
    <field xpath="@number" />
</key>
```

ここではpNumKeyという名称を持つキーを定義している。このキーは<parts>要素の下位要素<part>のnumberから構成される。rrimer例ではこのキーに対して以下のような参照が他の要素定義内から設定されている。

```
<element name="regions" type="r:RegionsType">
    <keyref refer="r:pNumKey">
        <selector xpath="r:zip/r:part" />
        <field xpath="@number" />
    </keyref>
</element>
```

この指定により<regions>要素の下位要素<zip>の下位属性<part>の属性numberの値に正確に一致する値をnumber属性として持つ<parts>要素の下位要素<part>が存在しなければ、このスキーマドキュメントにvalidなインスタンスドキュメントではないと見なされることになる。Validなインスタンスドキュメントは例6参照。

# XML Schemaの機能（例6）

- キー参照定義ｲﾝｽﾀﾝｽﾄﾞｷｭﾒﾝﾄ例（primerより）

```
<purchaseReport xmlns="http://www.example.com/Report">
   <regions>
      <zip code="95819">
         <part number="872-AA" quantity="1" />
         <part number="926-AA" quantity="1" />
         <part number="833-AA" quantity="1" />
         <part number="455-BX" quantity="1" />
      </zip>
      <zip code="63143">
         <part number="455-BX" quantity="4" />
      </zip>
   <reigions>

   <parts>
      <part number="872-AA">Lawnmower</part>
      <part number="926-AA">Baby Monitor</part>
      <part number="833-AA">Lapis Necklace</part>
      <part number="455-BX">Sturdy Shelves</part>
   </parts>
</purchaseReport>
```

# バリデーションと文法構造規定

- バリデーション対象には大きな分類として以下の項目が考えられる。

A.ドキュメント構造のバリデーション（マークアップシンタックスチェック）
B.リーフノード（最下位要素）コンテンツのバリデーション（データタイピング）
C.各要素／属性間のシンタックスレベルでの関連に関するバリデーション（たとえば要素Bは要素Aが出現した時のみに出現する等）
D.各要素／属性間のコンテンツレベルでの関連に関するバリデーション（キー参照定義など）
E.上記どれにもあてはまらない業務ルールに関連するバリデーション

# バリデーションと文法構造規定

- DTDとの比較（バリデーション能力に関して）

| | A | B | C | D | E |
|---|---|---|---|---|---|
| DTD | ○ | × | × | × | × |
| XML Schema | ○ | ○ | × | Partial | × |

※Dに関してXML Schemaがpartialと記されているのは、XML Schemaにおけるこの点の仕様がかなり恣意的な印象がある故である。たとえば要素<MAX>の値は要素<MIN>の値以上でなければならないというような比較的単純なDに属する二項関係のバリデーションは出来ないにも関わらず、それよりも遥かに複雑な多項関係のキー参照定義のような機能は用意されている。

# バリデーションと文法構造規定

- XML Schemaはあくまでも文法規定中心の考え方に基く

・文法の全体構造を規定し、インスタンスドキュメント中にその構造規定からはずれる要素が1つでも存在した場合、それをinvalidであると見なすような考え方である。
※確かにanyを用いて構造表記を曖昧にしておくことも可能であるが、このanyの存在そのものが基本的にはXML Schemaがドキュメントの全体構造を一貫して規定するのがその目的であることを示している（これに対して、後述するSchematronのようなルールベースのスキーマにおいてはanyという概念がそもそも必要ない）。

・インスタンスドキュメントのインターフェース的な共通構造を記述するにはDTDよりも遥かに文法表現能力の高いXML Schemaの方がより適切であると言える。最近のXML関連仕様たとえばSOAP、UDDI、WSDL等にXML Schemaによる構造規定が付加されているのはこのXML Schemaの持つ文法表現能力の故である。

・これ故文法表現能力に極めて優れるXML Schemaではあるが、これは必ずしもXML Schemaがバリデーション能力に優れているということを意味するわけではない。何故ならば、業務ルールに関連するようなバリデーションは、ドキュメントの包括的な文法構造には還元出来ないような要素をチェックする場合の方が遥かに多いからである。
※これに関しては「XML Schema Part 1:Structures」の「目的」の項に「この仕様によって定義される言語は、あらゆるアプリケーションによって必要とされるあらゆる機能を提供することが意図されているわけではありません。当言語では表現することの出来ない制限機能がアプリケーションによっては必要になることもあるでしょう。その場合にはアプリケーション自身で追加的なバリデーションを実行する必要があるかもしれません。」と明確に記述されています。

# バリデーションと文法構造規定

## ルールベースのスキーマ (Schematron)

DTDやXML Schemaを始めとする多くのスキーマは、文法構造の規定に主眼を置くものがほとんどであるが、それとは対照的にルールの規定に主眼を置くSchematronのようなスキーマがある。Schematronの場合、ドキュメントの全体的な文法構造を規定する意図はなく、個別のバリデーションルールを定義することにその主眼がある。以下はSchematronルール定義例である。

```
<rule context="dog">
   <assert test="count(ear) = 2">A dog element should contain two ear elements.</assert>
   <report test="bone">This dog has a bone.</report>
</rule>
```

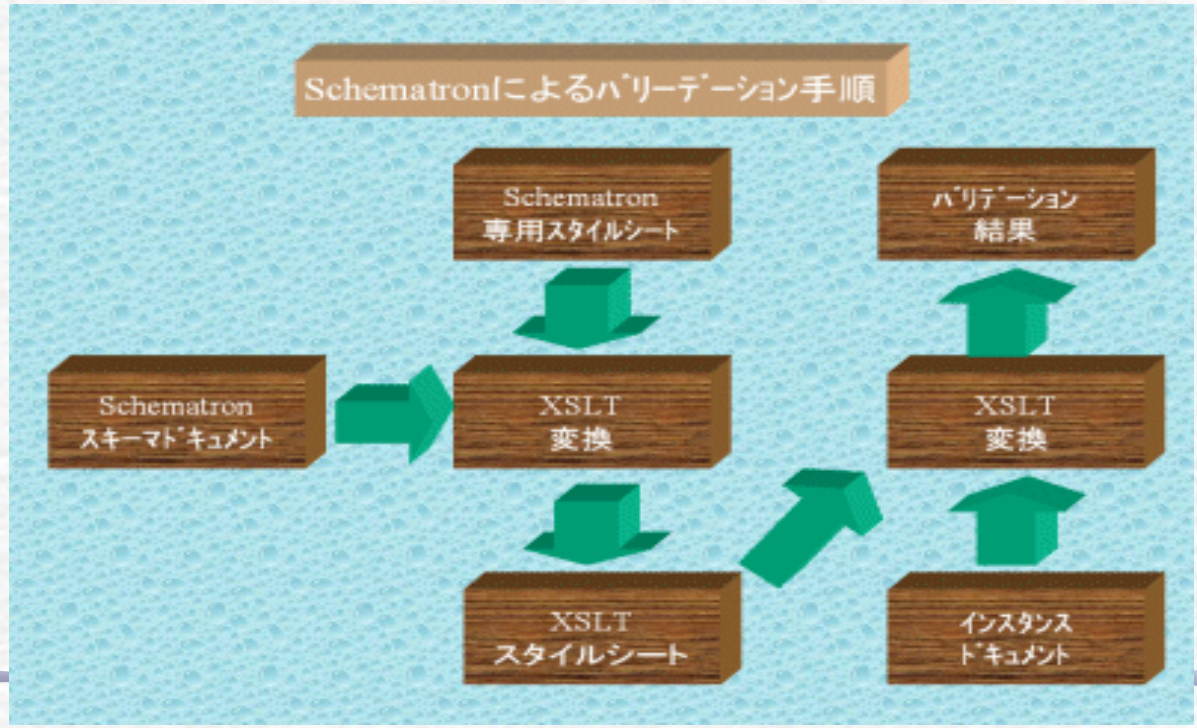
上記例を見ても分かる通り、Schematronは個別のルールを追加的な方法で定義することから始まり、定義されていない要素に関しては初めからバリデーションの対象にはならないことを意味する。これはすなわち全体の文法構造を定義することはSchematronの意図にはないことをも意味する。

尚、XML Schemaのキー参照定義などはむしろこちらのルールベースの定義の考え方に近いものと考えられる。何故ならばキー参照定義は、全体的文法構造の定義に関連するというよりは、アプリケーション的なデータ配置に関連すると言え、この規定がなくても文法構造の規定は可能であるという意味合いにおいて余剰的な機能だからである。

# バリデーションと文法構造規定

## ルールベースのスキーマ (Schematron)

Schematronは、XSLTの2度変換によって実装することが可能である。すなわち作成したSchematronスキーマをSchematronホームページで用意されている専用XSLTスタイルシートで変換し、その出力XSLTスタイルシートを使用してインスタンスドキュメントに対し更にもう一度XSLT変換を適用するという手順になる。



# バリデーションと文法構造規定

## XML Schemaの補完

前項までで述べたきたことを纏めるとスキーマの能力に関して、文法構造定義能力とバリデーション能力は別のものであるものとして考えるべきである。DTD、XML Schema、Schematron3者間で文法構造定義能力、バリデーション能力についての順位付けをすれば以下のようになるであろう。

| | 文法構造規定能力 | バリデーション能力 |
|---|---|---|
| DTD | 2 | 3 |
| XML Schema | 1 | 2 |
| Schematron | 3 | 1 |

# バリデーションと文法構造規定

- XML Schemaの補完

  1. アプリケーションロジックにより補完する。

  2. バリデーション能力の高い他のスキーマと併用する。

  3. XSLT変換を利用する (Schematronの利用は2と3双方の応用であると言える) 。

# 各スキーマの比較 (DTD)

- 特徴：昔からある。単純である
- 長所：昔から使われた実績がある
  文法が簡単なので記述が楽
- 短所：
  独自の記述ｼﾝﾀｯｸｽを持つ (XMLで記述されない)
  データ型が文字列のみ
  名前空間に対応できない
- コメント：
  ・既に広く一般的に使用されていること自体1つの長所である
  ・閉じた世界で使用する分には、要素の特徴、値の制限などは自分で把握すればいいので、十分使用可能

# 各スキーマの比較 (XML Schema)

- 特徴：構造、データ型の記述が一番厳密
- 長所：細かい文法構造定義、厳密な制限が可能
- 短所：記述が冗長、可読性が低い
- コメント：
  - 表現が冗長かつ複雑なので、気軽に扱うことは出来ない
  - 厳密な定義が可能なので、規定の表記などに適しているであろう
  - 文法構造的なバリデーションを行うには、他のスキーマよりも有効である

# 各スキーマの比較 (Schematron)

- 特徴：構造を定義するのではなく、条件文を羅列する
- 長所：構造だけでなく、値による制限が可能
- 短所：複雑な構造定義は困難
- コメント：
  - 値の比較による制限ができることはバリデーション機能においては極めて有用である
  - 構造の定義、制限を他のスキーマ言語で行い、さらにSchematronによる値のチェックを組み合わせれば強力なValidationが可能であろう

# 各スキーマの比較 (RELAX NG)

- 特徴：構造が単純で直感的に理解しやすい
- 長所：可読性が高い、構造記述が簡単、最初のとりかかりが楽
- 短所：簡略化のため、制限の厳密さにおいてXML Schemaに劣る部分がある
- コメント：
  - DTDによりは多機能で、XML Schemaよりは単純、という位置づけになる
  - 可読性が高く、初見でも構造の概要が把握できる
  - 何もないところから書き始めるのが容易
  - 社内の別グループで共同開発をするぐらいのレベルが適当なのではないか

# JavaとXML Schema

- JAXP

  現行のXMLハンドリングインターフェース仕様JAXPには、XML Schemaに関するインターフェース規定は含まれていない。この為XML Schemaに関するハンドリングは実装任せになっている。その故たとえばJAXPを実装するXercesパーサなどでもXML Schemaを用いてのバリデーション機能は既に用意されているとはいえども、たとえばメッセージのみによってしかエラーの判別が出来ない、複数エラーを一度に検出出来ないなどの不都合が存在する。XML Schemaに関するハンドリングインターフェースがJAXP等の標準仕様できちんと定義されることが待たれるところである。

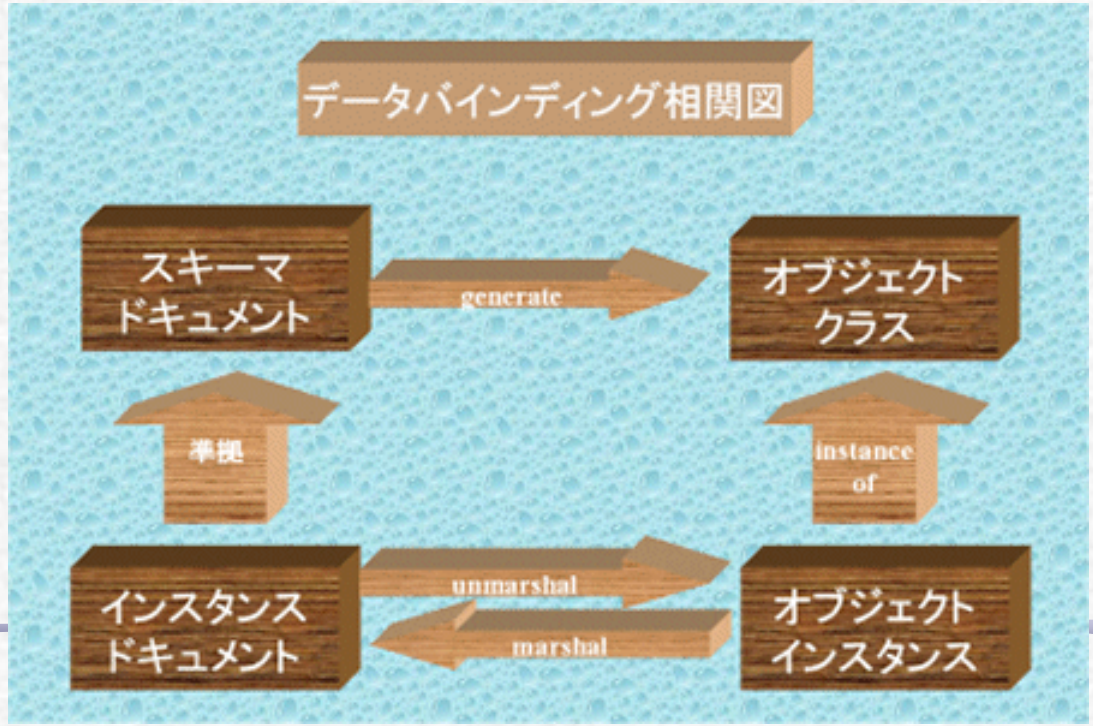
- JAXB

  JAXBは、データバインディングに関する標準仕様であり、簡潔な言い方をすればJavaオブジェクトとXMLインスタンスドキュメント間のデータ構造マッピング変換をシームレスに行う為の仕様である。本来インスタンスドキュメントの文法構造を高い精度で記述出来るXML Schemaはマッピング情報をストアする構造媒体としてはふさわしいはずであり、実際たとえばオープンソースCastor等のデータバインディングフレームワークではマッピング情報としてXML Schemaが利用されている。しかしながら現状のJAXBは独自のXMLシンタックスがマッピング情報記述子として規定されておりXML Schemaは対象になっていない。尚、JAXB仕様には将来のバージョンではXML Schemaの一部の機能をサポートするとある。

# ﾃﾞｰﾀﾊﾞｲﾝﾃﾞｨﾝｸﾞ

## データバインディングとは何か

データバインディングとはXMLインスタンスドキュメントをシームレスにメモリオブジェクト（或いはDB等）にマッピング展開する方法（unmarshal）及びその逆展開（marshal）を行う方法を言う。このデータバインディングにおいてXML Schemaが有効である理由は、個々のインスタンスドキュメントの共通構造を記述するのがXML Schemaによるスキーマドキュメントであり、XMLインスタンスドキュメントとXMLスキーマドキュメントの関係をオブジェクト指向におけるインスタンスとクラスの関係に射影することが出来るという点にある。



# ﾃﾞｰﾀﾊﾞｲﾝﾃﾞｨﾝｸﾞ(Castorの例)

- 以下の2つのﾊﾞｲﾝﾃﾞｨﾝｸﾞ方法が用意されている。

・XMLで書かれたマッピングファイルの利用（スキーマレスバインディング）
この方法では独自のマッピングファイルをXMLで記述することによりXMLとJAVAオブジェクトとのマッピングを行う。尚、デフォルトのマッピング方法が規定されているので、このマッピングファイルを記述する必要すらない場合もあるが、試してみたところあまりデフォルトマッピングにフレキシビリティはないようである。この方法はJAXBにおけるバインディングスキーマにほぼ相当するように思われる。但し、こちらの方法が使用される場合、クラス生成用ジェネレータは使用出来ない（APでマッピングファイルを読込む必要がある）。

・XML Schemaを利用したバインディング
XML Schemaをジェネレータによりスキーマコンパイルし JAVAクラスを生成し、そのクラスに自動生成されるmarshal及びunmarshalメソッドを呼出すことにより自動的にXML<->JAVAオブジェクトの変換を行うことが出来る。

# ﾃﾞｰﾀﾊﾞｲﾝﾃﾞｨﾝｸﾞ(Castorの例)

- CastorとはJavaをターゲットとしたオープンソースデータバインディングフレームワークである。
- URL＝http://www.castor.orgから自由にダウンロード可能。
- 機能としては以下のものが挙げられる。
  - ・Castor XML:JavaオブジェクトとXMLのデータバインディング
  - ・Castor JDO:JavaオブジェクトのRDBMSへのパシステンス
  - ・Castor DAX: JavaオブジェクトのLDAPへのパシステンス
  - ・Castor DSML:XMLによるLDAPディレクトリ交換
  - ・マッピングファイルによるスキーマレスバインディング
  - ・2フェーズコミット、オブジェクトロールバック、デッドロックディテクション
  - ・SQLクエリーへのOQLマッピング
  - ・EJBコンテナ管理パシステンスプロバイダ

# ﾃﾞｰﾀﾊﾞｲﾝﾃﾞｨﾝｸﾞ(Castorの例)

- XML Schemaバインディング（unmarshal）

<XML Schema例>

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="person">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="birthday" type="xsd:string"/>
        <xsd:element ref="address"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="address">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="prefecture" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="street" type="xsd:string"/>
        <xsd:element name="zip" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

# ﾃﾞｰﾀﾊﾞｲﾝﾃﾞｨﾝｸﾞ(Castorの例)

- XML Schemaバインディング（unmarshal）

＜ｼﾞｪﾈﾚｰﾀで生成されたPersonｸﾗｽ＞

```
public class Person implements java.io.Serializable {
    private String _name;
    private String _birthday;
    private Address _address;

    public Person() {super();}
    public Address getAddress() {return this._address;}
    public String getBirthday() {return this._birthday;}
    public String getName() {return this._name;}
    public void marshall(java.io.Writer out)
        throws MarshalException,ValidationException {
        Marshaller.marshal(this, out);
    }
    public void setAddress(Address address) {this._address = address;}
    public void setBirthday(String birthday) {this._birthday = birthday;}
    public void setName(String name) {this._name = name;}
    public static Person unmarshal(Reader reader)
        throws MarshalException,ValidationException {
        Unmarshaller.unmarshal(Person.class, reader);
    }
}
```

# ﾃﾞｰﾀﾊﾞｲﾝﾃﾞｨﾝｸﾞ(Castorの例)

- XML Schemaバインディング（unmarshal）

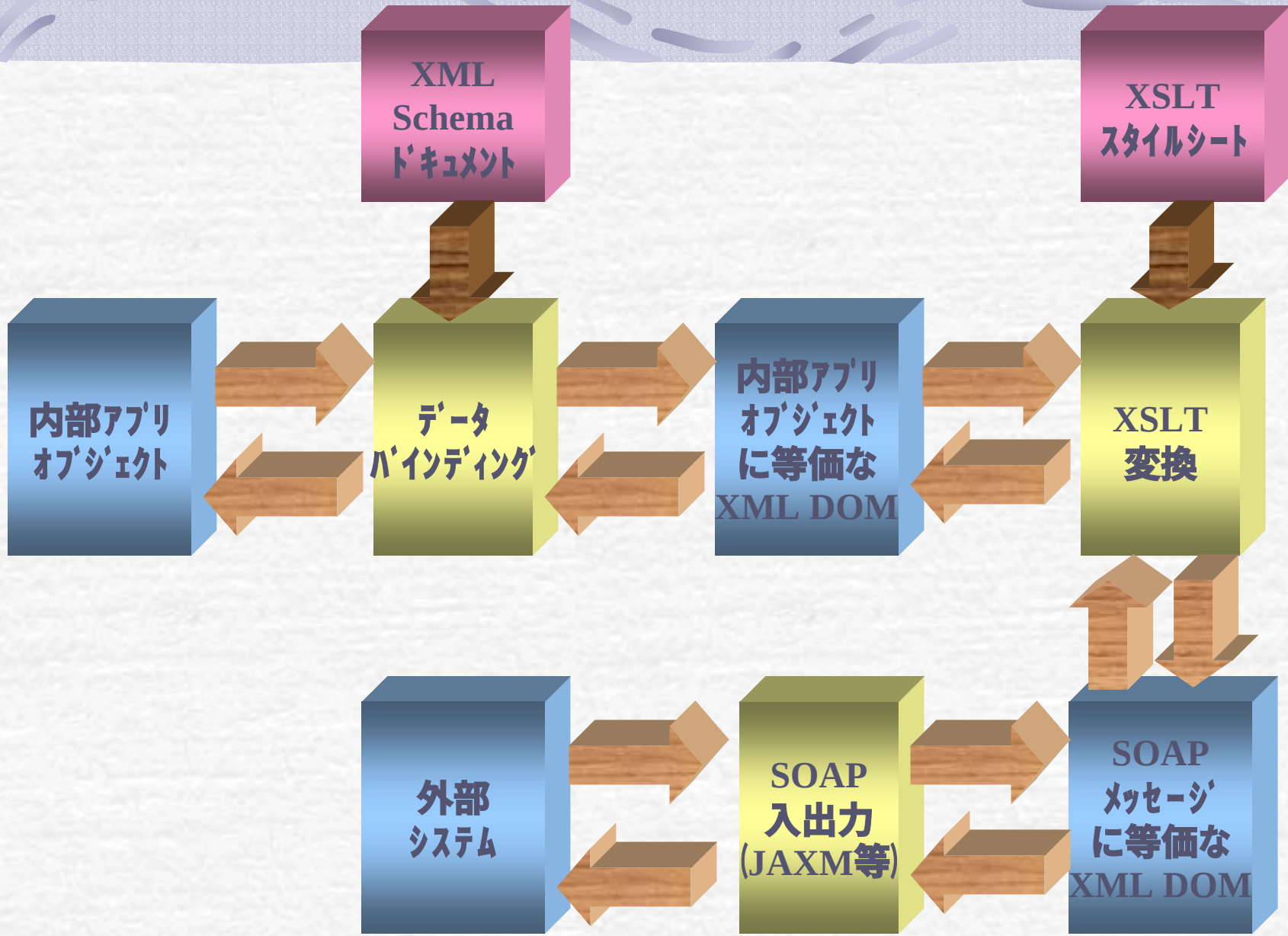
<AP例>

```
import java.io.FileReader;
public class TestAP2 {
   public static void main(String[] args) {
      try {
         Person person = Person.unmarshal(new FileReader("person.xml"));
         System.out.println("NAME:" + person.getName());
         System.out.println("BIRTHDAY:" + person.getBirthday());
         Address address = person.getAddress();
         System.out.println("PREFECTURE:" + address.getPrefecture());
         System.out.println("CITY:" + address.getCity());
         System.out.println("STREET:" + address.getStreet());
         System.out.println("ZIP:" + address.getZip());
      } catch (Exception ex) {
         System.out.println("Error Occurred:" + ex.getMessage());
      }
   }
}
```

# ﾃﾞｰﾀﾊﾞｲﾝﾃﾞｨﾝｸﾞ

## 応用例

記載例は、単純にJavaアプリを利用したデータバインディングの例であるが、アプリケーションサーバコンテクスト内或いはアプリケーションサーバを利用する何らかのフレームワークコンテクスト内にデータバインディング機能を用意しておいて、たとえば外部システムとSOAPによりデータ交換を行うケースなどにおいて、SOAPでやり取りする入出力メッセージの構造をXML Schemaで記述しておいて（但しXSLT等で補完する必要があるかもしれません）アプリケーションの仲介なしにSOAPメッセージ＜－＞メモリオブジェクト間のシームレスな変換を実現する等の利用形態も考えられるであろう（尚、このCastorはアプリケーションサーバではないのでそのような機能は有していない）。次ページ図を参照のこと。


XML
Schema
ﾄﾞｷｭﾒﾝﾄ
XSLT
ｽﾀｲﾙｼｰﾄ
内部ｱﾌﾟﾘ
ｵﾌﾞｼﾞｪｸﾄ
ﾃﾞｰﾀ
ﾊﾞｲﾝﾃﾞｨﾝｸﾞ
内部ｱﾌﾟﾘ
ｵﾌﾞｼﾞｪｸﾄ
に等価な
XML DOM
XSLT
変換
外部
ｼｽﾃﾑ
SOAP
入出力
(JAXM等)
SOAP
ﾒｯｾｰｼﾞ
に等価な
XML DOM

# バリデーション (製品)

- XSV
  W3Cとエジンバラ大学が共同開発したオープンソースの検証ツール
- MSXML
  XML Schemaのvalidationとタイプ判別をDOMとSAXの両方でサポートしている。
- Schematron Validator
  フリーのWindowsベースのValidationツールだが、DTD、XML Schema、Schematronを扱うことができる。
- Xerces
  XML Schema勧告をサポートしているXML parser
- XML Validate
  TIBCO社の開発したSAXベースのランタイムvvalidationツール
- XML Spy
  XML SchemaベースのValidation、graphicalなスキーマデザイン、スキーマ変換などをサポートする。

# バリデーション (JAXP)

- DTDは対応可能
- XML Schemaはサポート外(1.1.3現在)
- JAXP 1.2の要求項目にXML Schema対応が含まれている
- 現在でもparserによっては可能
  - →環境変数でparserを固定
    - ＋parser依存のAttribute設定が必要
  - →結果的にparser依存のコーディングになる？

# バリデーション(xerces)

- Xerces-J 1.4.4を使用
- Parseの時点でvalidationを実行
- エラーは全てSAXParseExceptionとしてスローされる

# バリデーション (XML Schema)

```
<?xml version="1.0" encoding="Shift_JIS" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="顧客リスト">
  <xs:complexType>
     <xs:sequence>
       <xs:element name="顧客" type="customerData"
              minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="customerData">
  <xs:sequence>
     <xs:element name="氏名" type="xs:string"/>
     <xs:element name="生年月日" type="xs:date"/>
     <xs:element name="性別" type="xs:string"/>
     <xs:element name="電話番号" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
     <xs:element name="電子メール" type="xs:string" maxOccurs="unbounded"/>
     <xs:element name="サービスポイント" type="xs:nonNegativeInteger"/>
  </xs:sequence>
  <xs:attribute name="会員番号" type="xs:string"/>
</xs:complexType>
</xs:schema>
```

# バリデーション（XML インスタンス）

```
<?xml version="1.0" encoding="Shift_JIS"?>
<!DOCTYPE 顧客リスト SYSTEM "customers.dtd">

<顧客リスト>
  <顧客 会員番号="S016823">
      <氏名>山田太郎</氏名>
      <生年月日>1980-01-28</生年月日>
      <性別>男性</性別>
      <電子メール>taro@xxx.xx.xx</電子メール>
      <電子メール>yamada@xxxx.xx.xx</電子メール>
      <サービスポイント>580</サービスポイント>
  </顧客>
  <顧客 会員番号="S200312">
      <氏名>鈴木花子</氏名>
      <生年月日>1970-05-14</生年月日>
      <性別>女性</性別>
      <電話番号>xxx-xxxx-xxxx</電話番号>
      <電子メール>flower@xxx.xx.xx</電子メール>
      <サービスポイント>8210</サービスポイント>
  </顧客>
</顧客リスト>
```

# バリデーション (ﾌﾟﾛｸﾞﾗﾑｿｰｽの例)

```
DOMParser parser= new DOMParser();
ErrorHandler errorh = new TestErrorHandler();
Try{
    parser.setErrorHandler( errorh );
    parser.setFeature("http://xml.org/sax/features/namespaces", true );
    parser.setFeature("http://xml.org/sax/features/validation", true );
    parser.setProperty("http://apache.org/xml/properties/schema/external-
    schemaLocation", xsd );
    parser.parse(xml);
}catch(Exception e){
    System.out.println( e.getMessage() );
}
Document doc = parser.getDocument();
```

Validationエラーは、全てSAXParseExceptionとしてスローされる

# バリデーション (エラーメッセージの例)

**予定外の**element**が現れた場合**

Element type "**氏名**s" must be declared.

- **要素の出現順序、回数が間違っている場合**

The content of element type "**顧客**" must match "(**氏名**,**生年月日**,**性別**,**電話番号***,**電子メール**+,**サービスポイント**)".

- **文字列の正規表現にマッチしない場合**

Datatype error: Value 'S01682' does not match regular expression facet '[A-Z]¥d{6}'..

- **数値の制限範囲を超える場合**

Datatype error: In element '**サービスポイント**' : 82100 is out of bounds:[ 0 <= X < 10000 ].

# バリデーション (苦労した点など)

- エラーメッセージがわかりにくい
  エラーの内容は示されるが、エラー発生箇所を特定できる情報がない。
  (参考：.NET環境ではエラー発生箇所の情報も表示される)

- エラーが1回につき1件しか表示されない
  1件修正するたびに実行しなくてはいけない。

- エラーが全て同じ例外にマッピングされる。
  エラーの種類に応じた処理分岐のためには、メッセージの文字列を解析する必要がある。