25 A XML Primer

Why XML?

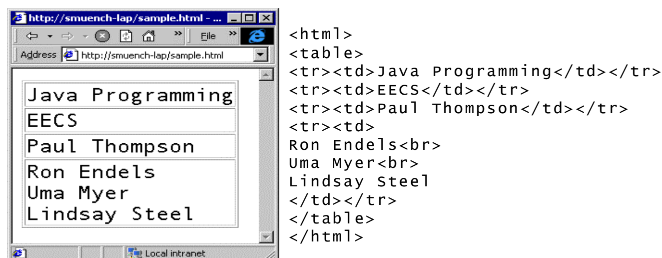
Humans are adept at figuring out the meaning of a piece of content using a few hints gathered from its structure. Going through a pile of papers on my desk, I can, after a brief scan, determine which is a phone bill, which a bank statement, and what different interpretations I need to apply to understand the contents of each. If I am in doubt, the phone company or the bank often helpfully puts headings on columns to help me along -- 'Per Minute Rate', 'Sub-Total', 'Debits' and so on.

Computers, sadly, are in need of even more explicit hints than us in order to be able to decipher what things mean and how they are related. The ubiquitous HTML used to describe pages for display in browsers employs ‘markup tags’ to provide hints to the browser on how to decipher data. For example, the string <A HREF=http://www.w3.org/></A>, is HTML code for telling a browser to treat ‘http://www.w3.org’ as a URL.

HTML is useful for marking up headlines and fonts, and for painting documents in a browser. In fact, the set of tags devised for HTML address issues like setting fonts, laying out text, creating forms. This is very useful for the browser, but not enough for other applications. Suppose your phone company sent you a bill in HTML. It would be very hard to decide, just looking at the marked up HTML document, which piece on the page is the ‘Per Minute Rate’.

So as more and more people try to put information out on the web, it is becomes clear that our interaction with the web is still quite limited, because the language which brings us web pages knows very little about the content of these pages. HTML is useful, but not so useful for taking orders, transmitting medical records or monitoring instruments. In order to make applications understand content, we must, devise a system of tags like <Order>, <Record> or <Data>, and mark up the data using these tags. Since no one can anticipate exactly what a full list of such tags is, it is important that we have a language using which we can define our own tags, for our own purposes, and for sharing with others. This is the genesis of XML.

Representation using HTML



Representation Using XML

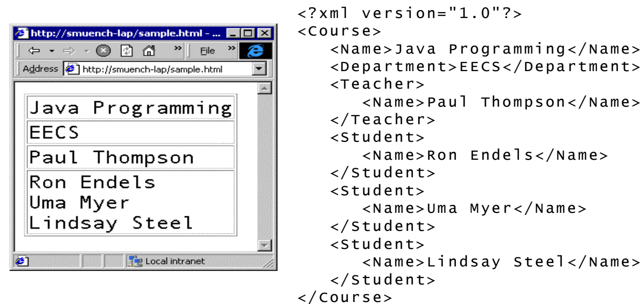


Fig 1: HTML vs XML (Need a better one)

Another motivator of XML is performance. If you change, say, the 'Quantity' field in your order on a HTML form, then in order to see the few digits in the 'Total' that change as a result, you have to ask a burdened server across a slow network to send you a brand-new, generated, graphics-rich page. This is because the meaning of ‘Quantity’ and ‘Total’ are not available to HTML. If structural and semantic information could be added to HTML, your browser (or cell-phone, or pocket planner) could do a great deal of processing on the spot. This means you would not have to hit that network or that web server as often. With the ability to add our own tags to data, a great deal of local processing could occur, and the Internet could become faster and friendlier.

W3C summarized XML’s utility as follows [971208 w3c press release. http://www.oasis-open.org/cover/xml.html#overview] "XML is primarily intended to meet the requirements of large-scale Web content providers for industry-specific markup, vendor-neutral data exchange, media-independent publishing, one-on-one marketing, workflow management in collaborative authoring environments, and the processing of Web documents by intelligent clients. It is also expected to find use in certain metadata applications. XML is fully internationalized for both European and Asian languages, with all conforming processors required to support the Unicode character set in both its UTF-8 and UTF-16 encodings. The language is designed for the quickest possible client-side processing consistent with its primary purpose as an electronic publishing and data interchange format." We will see in the succeeding sections and chapters what all this means.

So what is XML?

Formally, XML is a system for defining, validating and sharing document formats using user-specified tags to distinguish document structures, and attributes to encode document information. The tagging system of XML is quite similar to HTML, though, unlike HTML, XML’s tagging is flexible. **XML documents** are made up of storage units called **entities**, which may contain either parsed or unparsed data. Parsed data consists of characters, some of which form the character data in the document, and some of which form the tags. The set of tags in an XML document are collectively called **markup**. The markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure. A software module called an **XML processor** is used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another module, called the **application**. The XML language specification essentially describes the mandatory behavior of an XML processor in terms of how it must read XML data and the information it must provide to the application.

The concept of ‘generalized markup’ using tags had originated in the 60s, and this gradually became well established with the crystallization of a huge and comprehensive standard called SGML (Standard Generalized Markup Language.) ISO, the International Standards Organization, ratified the SGML specification in 1986. It is a measure of how comprehensive and flexible this specification was, that 10 years passed before even minor revisions had to be made. Yet, as the Internet expanded throughout the early and mid-1990s, this very power of SGML became a handicap. It was overkill. Software vendors found that it would be a huge programming task to implement even a subset of features specified in SGML. The Internet community has enthusiastically supported XML, as it seems to be the Goldilocks version of markup languages – not too simple

and not too complex, somewhere between HTML and SGML.

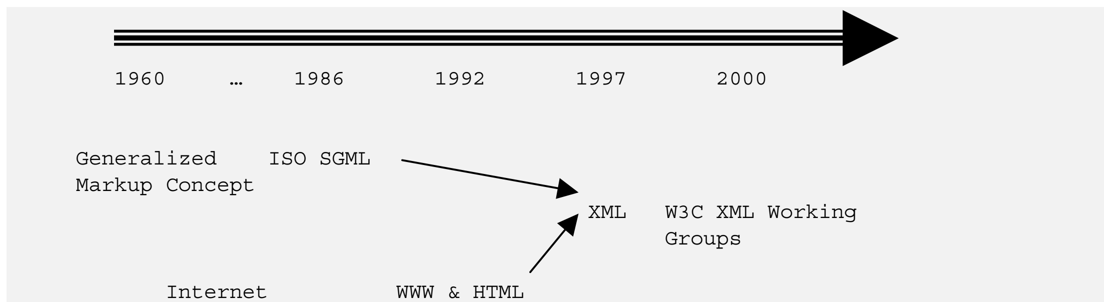


Fig. 2: The evolution of XML

XML shares similarities with SGML (Structured Graphics Markup Language), but there are also differences. In fact, valid XML documents are designed to be valid SGML documents, but XML documents have additional restrictions. Some of the differences between XML and SGML are:

XML's white space handling rules aren't as elaborate as those of SGML. XML will pass through some white space rules that an SGML processor will suppress.

XML defines the property of being **well-formed**. This feature does not correspond to any SGML concept.

XML makes a number of restrictions on the declaration of entities and characters to make life simpler. (Entities are re-usable chunks of data, much like macros and are part of XML's heritage from SGML. More on Entities later.)

XML can handle international text (non ASCII), where as few SGML processors are properly internationalized

A *well-formed* document is something that is new to XML. It is easy for a computer program to read, and ready for network delivery. For example:

All the begin/end tags match up

Empty tags use the special XML syntax (i.e. <empty/>). Empty documents can simply be spotted by the "/>" ending string.

All attribute values are nicely quoted (i.e. <A HREF=http://www.oracle.com/xml ></A>)

All the entities are declared.

Anatomy of a XML document

Here is a fragment of XML. It shows how we could mark up information for a bank statement request.

```
<?xml version="1.0" encoding="UTF8" standalone="yes" ?>
<StatementRequest>
      <BankAccount>
            <BankID>310824233</BankID>
            <AccountID>1234-56789</AccountID>
            <AccountType>SAVINGS</AccountType>
      </BankAccount>
</StatementRequest>
```

The first line is called the **XML Declaration**. It can have the version number of the XML specification that the

document assumes. Currently, everyone uses version 1.0, but this will change in the future as later specifications come out.

The XML declaration can also include information on the **encoding** scheme used in the document. Encoding refers to the process by which a file on a computer is transformed into a sequence of individual bytes for transmission. Although humans see files in terms of characters, computers really think of each character as a number. You press ‘S’ on your keyboard, but what your computer really ‘sees’ is a number corresponding to ‘S’ – in ASCII encoding this number is 83. If information were to be transferred between your computer which understands ASCII, and another computer based on some other encoding, then the receiving computer would get garbage – it would map ‘83’ not to ‘S’ but to some other character. Some of the popular encoding schemes are ASCII, EBCDIC, Unicode, UTF8, UTF16 and so on. XML documents can be recognized only when they conform to some standard encoding scheme, and we have to declare what encoding scheme we are using. If a XML Declaration does not contain encoding information, then the default is assumed to be **UTF8 encoding**.

The third parameter that the XML declaration can optionally have is a directive whether this document is **standalone**, or whether it has pieces that refer to external documents. So a fuller XML Declaration would be:

```
<?xml version=”1.0” encoding=”UTF8” standalone=”yes” ?>
```

After this declaration, which is really for the benefit of the processor rather than the reader, we get to the business of marking up the information with tags.

A **start-tag** and an **end-tag**, together with the data enclosed by them, represent an **element**. The start-tag is delimited using ‘<’ and ‘>’, and the end-tag delimited using ‘</’ and ‘>’. So in the above example

```
<BankID>310824233</BankID>
```

represents an element. Note that element name are case-sensitive, so `BankID` and `BANKID` would refer to different elements! In practice, however, it is not advised that a XML author choose elements with the same name differing in case, because of the confusion this could create. Elements can be completely enclosed by other elements, such as `BankID` by `BankAccount`, and `BankAccount` by `StatementRequest`. In fact, each element must be enclosed by some other element, except the ‘root’ or **document element**. Elements may also be repeated within enclosing elements.

```
<?xml version="1.0"?>
<StatementRequest>
     <BankAccount>
           <BankID>310824233</BankID>
           <AccountID>1234-56789</AccountID>
           <AccountType>SAVINGS</AccountType>
     </BankAccount>
     <BankAccount>
           <BankID>310824233</BankID>
           <AccountID>1234-56789</AccountID>
           <AccountType>CHECKING</AccountType>
     </BankAccount>
</StatementRequest>
```

Elements may also directly or indirectly contain instances of themselves. These structures may be recursive without theoretical limit, though of course the application programs that deal with such structures would grow complex as the recursion increased.

Another interesting concept is that of **attributes**. It is possible that an element hold some meta-information about itself. For example, `BankID` could have meta-data to specify which system of assigning IDs we were using – whether the ID was an American FDIC ID, or a Bermudan Clearinghouse ID, or a Neapolitan Chamber of Commerce ID. A single element may contain more than one attribute, though each attribute must have a name, such as `ValidUntil` or `Author`. An attribute is specified along with the element, as in:

```
<BankID>310824233  numberingSystem=”American Routing & Transit” </BankID>
```

Similarly, if we were defining an element `Address`, we might want to add an attribute specifying which addressing system it belonged to. This attribute could be used to validate, say, the format of postal codes.

```
<address addressingSystem="US">
        <name>Dukhi Desi</name>
        <street>8 Oak Avenue</street>
        <city>Fremont</city>
        <state>CA</state>
        <postcode>94555</postcode>
</address>
```

You see that XML gives us the ability to mark up information as we please. We could, for example, choose `<BankRoutingNumber>` instead of `<BankID>` as the name of the markup tag that uniquely identifies the bank for the statement request. How the tags are named depends on discussion between the application that generates the statement request and the application that services such requests. The XML language itself does not care what we call our tags – it merely lays down the rules that the tag declarations have to follow to make them recognizable to XML processors at the applications. The set of tags is **extensible** – that is, we can add new tags as we go along, perhaps to ask our ‘statement server’ that we would like our statements to be recurring:

```
<StatementRequest>
      <BankAccount>
            <BankID>310824233</BankID>
            <AccountID>1234-56789</AccountID>
            <AccountType>SAVINGS</AccountType>
      </BankAccount>
      <BankAccount>
            <BankID>310824233</BankID>
            <AccountID>1234-56789</AccountID>
            <AccountType>CHECKING</AccountType>
      </BankAccount>
      <Recurrence>
             <Number>12</Number>
             <Frequency>MONTHLY</Frequency>
             <StartDate>01/01/2000</StartDate>
      </Recurrence>
</StatementRequest>
```

Now that we have a basic understanding, we can start reading XML documents and examining their contents. Another term frequently used in this context is the **content model** of XML documents. An element that contains other elements, but no text, is said to have **element content**. In the example above, `BankAccount` has element content, and so do `StatementRequest` and `Recurrence`. An element that only contains text, such as `BankID`, is said to have **data content**. Finally, any element that contains both other elements and data possesses **mixed content**.

Here is another example:

```
<?xml version="1.0" encoding=”UTF-16” standalone=”yes” ?>
<purchaseOrder orderDate="01/01/2000">
    <shipTo addressingSystem="Canada">
        <name>Hortense Quackenbush</name>
        <street>123 Maple Street</street>
        <city>Chilliwack</city>
        <state>BC</state>
        <postcode>V2P 4P3</postcode>
    </shipTo>
```

```
        <billTo addressingSystem="Canada">
            <name>Horatio Quackenbush</name>
            <street>8 Oak Avenue</street>
            <city>Nakusp</city>
            <state>BC</state>
            <postcode>V0G 1R0</postcode>
        </billTo>
        <items>
            <item partNum="ABC-123">
                <productName>Kayak</productName>
                <quantity>1</quantity>
                <price>567.80</price>
                <comment>Prefer green with yellow trim</comment>
            </item>
            <item partNum="ABC-124">
                <productName>Paddle</productName>
                <quantity>2</quantity>
                <price>19.98</price>
                <shipDate>01/10/2000</shipDate>
            </item>
        </items>
    </purchaseOrder>
```

Such a purchase order document can be used to relay information from a browser which takes the order to a shipping subsystem that fulfils it. Of course, if the browser and the shipping subsystem do not agree on the tags, then we have a problem! Industry bodies such as OASIS (http://www.oasis-open.org) are working on standard sets of tags and XML vocabularies for specific industries. A body of bankers working as part of OASIS (or some other organization) could, for example, define a template for Bank Accounts, so that the names of the tags and the placement of elements can be validated. A significant feature for XML is its ability to specify such templates as part of document markup. We look at such templates under the section 'Logical Structure of XML document'. Before we get there, however, let's first look at the Physical Structure of XML documents.

The Physical Structure of XML Documents

The XML specification provides for a single XML document to be distributed among a number of separate data files. The main aim of this is to enable the reuse of components and fragments across documents. For example, an image of a logo that appears on each page of a corporate brochure may be stored separately, yet included in multiple documents. Each such physical unit of information is called an **entity**. Each entity must have a name by which it can be identified – except the abstract **document entity**, which is deemed to represent the entire document as it is given to the XML parser.

In the simplest situation, there may be only one entity, which is the document entity. In more complex ones, such as those involving syndicated content (say an internet portal page), a document entity may just be a framework, gluing together other entities representing images, audio, video, stock ticker feeds, weather information, horoscopes and whatever else the portal designer decides to throw in.

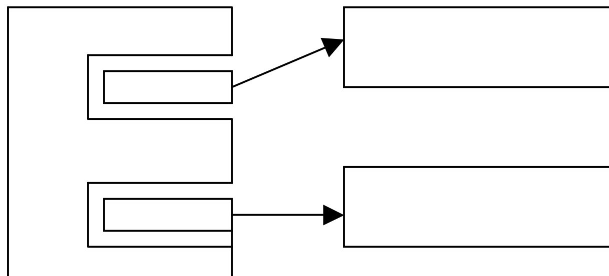


Fig. 3: Entities (Need a better diagram.)

There can be multiple types of entities. Entities that contain XML data is called a **parsed entities** – because they are ultimately parsed and validated by an XML parser. Media types such as images or audio, which do not have to be parsed, are **binary entities**. It is possible to declare **internal text entities**, which are basically short aliases for long sentences that appear in a document. There are also **general entities**, **parameter entities**, **external entities** and others, each with specific meanings that you may be able to guess.

The Logical Structure of XML Documents

One significant feature that XML inherits from SGML is the concept of a Document Type Definition or **DTD**. A DTD is quite similar to style templates found in word processors – it defines formal rules for the structure of an XML document. A DTD defines the elements that may be used, and defines where they may be placed in relation to each other. So, parallel to the document with its logical tree of elements, we have a DTD tree. Unlike the document structure tree, the DTD tree does not contain repeating entities. The DTD tree for our Bank Statement fragment would be:

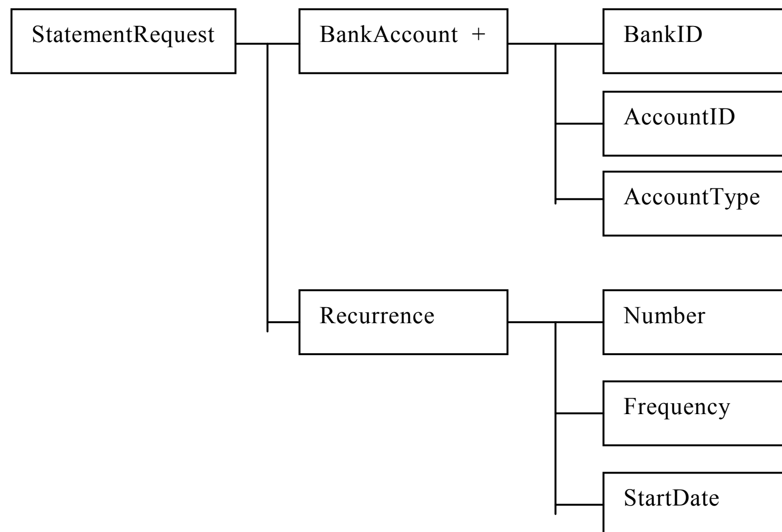


Fig. 3: DTD Logical Structure

Declaring DTDs

DTDs are declared using a special syntax. For example, the declaration for the above DTD would look something like:

```
<!-- BANK STATEMENT REQEUST DTD
               DTD for Bank Statement Requests
               Author: Dukhi Desi
               Version: 1.0 (01/01/2000)  -- >
<!DOCTYPE BANKSTATEMENTDTD SYSTEM “../DTDS/BANKSTATEMENT.DTD” >
<!ELEMENT(StatementRequest(BankAccount, Recurrence)>
<!ELEMENT BankAccount (BankID, AccountID, AccountType) +>
<!ELEMENT BankID (#PCDATA)>
<!ELEMENT AccountID (#PCDATA)>
<!ELEMENT AccountType (#PCDATA)>
```

```
<!ELEMENT Recurrence (Number, Frequency, StartDate) >
<!ELEMENT Number (#PCDATA)>
<!ELEMENT Frequency (#PCDATA)>
<!ELEMENT StartDate (#PCDATA)>
```

As you can clearly see, each element is declared separately, and after some information on the structure the DTD we add information about its child elements and attributes. (Add more explanation.)

Here’s a DTD for the PurchaseOrder example shown earlier:

```
<!—PURCHASE ORDER DTD
               DTD for Purchase Orders
               Author: Dukhi Desi
               Version: 1.0 (01/01/2000)  -- >
<!DOCTYPE BANKSTATEMENTDTD SYSTEM "../DTDS/PURCHASEORDER.DTD" >
<!ELEMENT PurchaseOrder (ShipTo,BillTo,Items)>
<!ATTLIST PurchaseOrder
     orderDate CDATA #IMPLIED
>

<!ELEMENT ShipTo (name, street, city, state, postcode)>
<!ATTLIST ShipTo
     addressingSystem CDATA #IMPLIED
>

<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postcode (#PCDATA)>

<!ELEMENT BillTo (name, street, city, state, postcode)>
<!ATTLIST BillTo
     addressingSystem CDATA #IMPLIED
>

<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postcode (#PCDATA)>


<!ELEMENT Items (Item)*>
<!ELEMENT Item (ProductName,quantity,price, shipDate)>
<!ATTLIST Item
     partNum CDATA #IMPLIED)
>

<!ELEMENT ProductName (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT shipDate (#PCDATA)>
```

This may not be the easiest to read, but you can still make out that elements and attributes are being defined, and that the order of elements and attributes can create a template using which XML documents can be validated. While DTDs certainly can do the job, and indeed have done so till now, they suffer from some deficiencies.

Shortcomings of DTDs (Enter XML Schema)

DTDs are quite useful to specify templates for logical organization of XML data, but they suffer from several disadvantages. Because of these disadvantages, they are being slowly phased out in favor of a new standard, XML Schema.

A DTD-based XML document has two structures inside it -- the XML 'instance data' (i.e. the contents marked up with tags just like HTML), and the DTD that describes rules about how the instance data is set up.

For example, an XML instance fragment about a comment might look like:

```
<Comment>Some text to serve as comment </Comment>
```

The DTD for this piece of XML might be:

```
<!ELEMENT Comment (#PCDATA)* >
```

Every new student of XML immediately notices that DTDs have a syntax that is different from the XML instance. This means that parsers, tools and programs dealing with DTDs have to implement the specific rules that govern DTDs. Clearly, it would make sense to use the same syntax for DTDs as the documents themselves. XML Schema allows models to be written in XML, which lets the same programs that read the data also read the definition of the data:

```
<element name =”Comment” type =”string”>
```

So, programmers do not have to learn the abstruse `#,* &]` conventions that DTDs carried over from SGML. XML Schema can be parsed into a tree of nodes like any other XML document. The XML DOM (see below) can then be used to navigate the schema.

Another problem with DTDs is that they really do not provide support for 'typing' of data -- to a DTD, an XML structure is a string of characters. This worked in the document-centric world of SGML, but poses major problems when you try to model dynamic content that comes from strongly typed systems such as databases. Consider the following:

```
<Speed unit = “mph”> 140.53 </Speed>
<Speed unit = “mph”> exceeds limit </Speed>
```

Clearly, the two cases would need different logic to handle them. In the absence of typing, an XML parser validating `Speed` would conclude both were correct, and the burden of adding program logic to deal with each case would fall on the developer. The XML Schema proposal provides for data-type integrity and the maintenance of bound conditions (with some minor scripting support). So, using XML Schema, we can define a generic element named `Speed`. Don’t concern yourself with the syntax yet (we will discuss XML Schema syntax in the next section), this is just to show it can be done.

```
<complexType name = “speed”>
     <element name = “value”> <base = “decimal”> </element>
     <element name = “unit” type = “string” </element>
</complexType>
```

With such a declaration, we can use the type speed wherever we like:

```
<element name=”airspeed” type = “speed”/>

<airspeed> <value> 123 </value> <unit> “knots” </unit> </airspeed>
```

Another advantage the XML Schema offers over DTDs is extensibility: XML Schema can be refined and successive schema authors can add their own elements and attributes. You can add additional constraints to the declaration of an element. This extensibility helps XML Schemas to create 'open' content models -- additional elements and/or attributes can be present within an element without one having to declare each and every

element in the XML Schema. There are, of course, limitations on what you can do to a schema as part of this open content model. You cannot add/remove elements or attributes that will break the existing content model in some way. For example, if a type is defined as a sequence of elements, valid extensions must preserve that sequence before adding any 'open' content. Undeclared elements can be added only if they belong to a different namespace. There are, of course, cases where an open content model is not desired. You can always override the default and specify the content model as 'closed', in which case any additional elements or attributes will not validate:

```
        <type name = “signature” extendable = “no”>
```

So we have seen some of the shortcomings of DTDs and how XML Schema proposes to alleviate them. Next, we take a broader look at XML Schema and dynamic content.

XML Schema

As this book goes to press, XML Schema is in the final stages of deliberation within W3C. This chapter’s treatment of XML Schema is based on the public working draft XML Schema 1.0 made available by the XML Schema Working Group of the W3C. Note that the draft is still subject to change, and any final syntax that is adopted may differ from what you see in this chapter. However, the basic principles of XML Schema – such as stronger typing than DTDs – are unlikely to be dramatically modified.

In XML Schema terms, the term **instance document** is often used to describe an XML document that conforms to a particular schema. One such instance document may be our Bank Statement Request:

```
<?xml version="1.0"?>
<StatementRequest>
      <BankAccount>
            <BankID>310824233</BankID>
            <AccountID>1234-56789</AccountID>
            <AccountType>SAVINGS</AccountType>
      </BankAccount>
      <BankAccount>
            <BankID>310824233</BankID>
            <AccountID>1234-56789</AccountID>
            <AccountType>CHECKING</AccountType>
      </BankAccount>
      <Recurrence>
             <Number>12</Number>
             <Frequency>MONTHLY</Frequency>
             <StartDate>01/01/2000</StartDate>
      </Recurrence>
</StatementRequest>
```

The bank statement request consists of a main element `StatementRequest`, with sub-elements like `BankAcount` and `Recurrence`, each of which has sub elements of its own.   XML Schema for this instance document would look like:

```
<schema>

 <annotation>
  <documentation>
   Bank Statement Request Schema.
   Copyright 2000 Isle of Man Taxpayers’ Bank. All rights reserved.
  </documentation>
 </annotation>

 <element name="StatementRequest" type="StatmentRequestType"/>
```

```
 <complexType name="StatementRequestType">
  <element name="BankAccount" minOccurs=”1” maxOccurs=”unbounded”
type="BankAccountType"/>
  <element name="Recurrence" type="RecurrenceType"/>
 </complexType>

 <complexType name="BankAccountType">
  <element name="BankID"  type=”string” >
  <element name="AccountID" type="string"/>
  <element name="AccountType"  type="string"/>
 </complexType>

<complexType name="RecurrenceType">
  <element name="Number"  type="positiveInteger"/>
  <element name="Frequency" type="string"/>
  <element name="StartDate"  type="date" />
 </complexType>
</schema>
```

Simple & Complex Types

XML Schema provides for both **simple types** and **complex types**. Simple type definitions provide for atomic types, such as 'integer'. Complex types, whose expression in XML documents consists of elements with attributes and/or other elements, may also be defined. The basic difference between these is that complex types may allow elements in their content and may carry attributes, while simple types can neither have element content nor carry attributes.

Definitions for complex types typically contain a set of element declarations, element references, and attribute declarations. The declarations associate a name to constraints that govern the appearance of that name in documents. For example, `RecurrenceType` is defined as a complex type, and within the definition of `RecurrenceType` we see three element declarations. This means that any element appearing in an instance whose type is declared to be `RecurrenceType` must consist of three elements called, in order, `number`, `frequency` and `date`.

Another interesting feature of XMLSchema is that it is possible to place constraints on the schema which mandate how many times an element may occur. An element is required to appear when the value of `minOccurs` is 1. The maximum number of times an element may appear is specified by the value of a `maxOccurs` attribute in its declaration. The value of this attribute may be any positive integer, or the term `unbounded`, which indicates there is no specified maximum number of occurrences. The default value for `minOccurs` is 1, but there is none for `maxOccurs`. If there is no `maxOccurs`, its value is assumed to be the same as `minOccurs`. If both values are omitted, the element must appear exactly once.

Each aspect of a simple type that can be so constrained is called a **facet**. For example, a simple string can be constrained by the minimum length it must be, the maximum length it can have, the pattern it must conform to and so on. The pattern constraint is particularly important – we can require, for example, that a string representing a US telephone number be of the form (123)456-789 -- i.e. nine digits, with parentheses before the first and after the third digit to separate the area code, and a hyphen between the sixth and seventh digits. We could express this pattern as

```
<simpleType name="USPhone" base="string">
  <pattern value="(\d{3})\d{3}-\d{3}"/>
 </simpleType>
```

Another interesting facet is enumeration. Enumeration constrains a simple type to a set of distinct values. For example, we can have an enumeration for the days of the week:

```
<simpleType name="DaysOfTheWeek" base="string">
```

```
    <enumeration value="Sun"/>
    <enumeration value="Mon"/>
    <enumeration value="Tue"/>
    <enumeration value="Wed"/>
    <enumeration value="Thu"/>
    <enumeration value="Fri"/>
    <enumeration value="Sat"/>
  </simpleType>
```

XML Schema specifies dozens of simple types, and about fourteen facets. Not all facets apply to all types. Here is a table of some common types and the facets that apply to them.

Facets

	length	**minLength**	**maxLength**	**pattern**	**enumeration**	**maxExclusive**	**minExclusive**	**minExclusive**	**minInclusive**
string	y	y	y	y	y	y	y	y	y
integer				y	y	y	y	y	y
date				y	y	y	y	y	y
time				y	y	y	y	y	y
boolean				y					

Table 1: Some Simple Types & Some of Their Applicable Facets

Clearly, all this is much more sophisticated than DTDs. In many e-commerce applications, dynamic data is generated from a database. Such data is typically expressed in terms of a database type system. The most popular type systems are SQL:92 and the recently adopted SQL:1999. SQL92 provides for much richness in data types -- such as NULL-ness, variable precision (e.g. NUMBER(7,2)), check constraints and so on. SQL:1999 adds to the capabilities of database type systems by providing user-defined types, inheritance, references between types, collections of types and so on. XML Schema can capture a wide spectrum of schema constraints that go towards better matching generated documents to the underlying type-system of the data.

The usefulness of rich schema constraints is not limited to data-driven applications. There are more and more document-driven applications that exhibit dynamic behavior. A simple example might be a memo, which is routed differently based on markup tags. A more sophisticated example is a technical service manual for an intercontinental aircraft. Based on complex constraints provided by XML Schema, one can ensure that the author of such a manual always enters a valid part-number, and one might even ensure that part-number validity depends on dynamic considerations such as inventory levels, fluctuating demand and supply metrics, or changing regulatory mandates.

Let’s look the schema for the earlier purchase order example. The schema for this is derived from the XML Schema 1.0 draft from W3C:

```
<?xml version="1.0"?>
<purchaseOrder orderDate="01/01/2000">
    <shipTo addressingSystem="Canada">
        <name>Hortense Quackenbush</name>
        <street>123 Maple Street</street>
        <city>Chilliwack</city>
        <state>BC</state>
        <zip>V2P 4P3</zip>
    </shipTo>
    <billTo addressingSystem="Canada">
        <name>Horatio Quackenbush</name>
        <street>8 Oak Avenue</street>
        <city>Nakusp</city>
        <state>BC</state>
        <zip>V0G 1R0</zip>
    </billTo>
    <items>
        <item partNum="ABC-123">
            <productName>Kayak</productName>
            <quantity>1</quantity>
            <price>567.80</price>
            <comment>Prefer green with yellow trim</comment>
        </item>
        <item partNum="ABC-124">
            <productName>Paddle</productName>
            <quantity>2</quantity>
            <price>19.98</price>
            <shipDate>01/10/2000</shipDate>
        </item>
    </items>
</purchaseOrder>
```

XML Schema for this case could look like:

```
<schema>

 <annotation>
  <documentation>
   Purchase order schema
  </documentation>
 </annotation>

 <element name="purchaseOrder" type="PurchaseOrderType"/>

 <element name="comment" type="string"/>

 <complexType name="PurchaseOrderType">
  <element name="shipTo" type="Address"/>
  <element name="billTo" type="Address"/>
  <element name="items"  type="Items"/>
  <attribute name="orderDate" type="date"/>
 </complexType>

 <complexType name="Address">
  <element name="name"   type="string"/>
  <element name="street" type="string"/>
  <element name="city"   type="string"/>
  <element name="state"  type="string"/>
  <element name="zip"    type="decimal"/>
```

```
     <attribute name="country" type="NMTOKEN"/>
    </complexType>
```

```
    <complexType name="Items">
     <element name="item" minOccurs="0" maxOccurs="unbounded">
      <complexType>
       <element name="productName" type="string"/>
       <element name="quantity">
        <simpleType base="positiveInteger">
         <maxExclusive value="100"/>
        </simpleType>
       </element>
       <element name="price"   type="decimal"/>
       <element ref="comment"   minOccurs="0"/>
       <element name="shipDate" type="date" minOccurs='0'/>
      <attribute name="partNum" type="partSKU"/>
      </complexType>
     </element>
    </complexType>

   <simpleType name="partSKU" base="string">
     <pattern value="\[A-Z]{3}-d{3}"/>
    </simpleType>

   </schema>
```

Derivation

In XML Schema, a type definition may **derive** from another type. Derivation is the process through which more specific definitions are created from general ones. The type that we derive from is called the **base**. For example, we derive `partSKU` from the base string type, and, as part of the derivation, add a specific pattern facet to string. This makes a `partSKU` a special kind of string, one that always exhibits a particular pattern. Through derivation, it is possible to build a **hierarchy of types**, each derived type refining or specializing its base.

For simple types, as we have seen, derivation is frequently equivalent to **restriction** - the new type must constrain the legal range of values obtained from the existing type. However, it is also possible achieve the **extension** of a type through derivation, by adding more complex aspects. In the purchase-order example above, we have a simple element named price, of simple type decimal. It is possible to create a complex type globalPrice by deriving from decimal:

```
   <element name="globalPrice">
       <complexType base="decimal" derivedBy="extension">
           <attribute name="currency" type="string" />
       </complexType>
   </element>
```

We use a `complexType` element to define the new type, making `decimal` the base attribute. To the decimal value, we add a `currency` attribute using the standard attribute declaration. Because we want to <u>add</u> this attribute to the simple type, we signal our intent using `derivedBy=”extension”.`

Extension allows us to design types intelligently. Earlier, we has defined an address as:

```
   <address addressingSystem="US">
           <name>Dukhi Desi</name>
           <street>8 Oak Avenue</street>
           <city>Fremont</city>
           <state>CA</state>
           <postcode>94555</postcode>
```

```
</address>
```

The attribute addressingSystem allowed us to interpret elements like postcode according to different locales, but such a scheme does not allow, say, different addressing system to have different elements. What if we were trying to use an Indian addressing system, which additionally uses an element Post Office?

```
<complexType name="Address">
  <element name="name"     type="string"/>
  <element name="street"   type="string"/>
  <element name="city"     type="string"/>
 </complexType>

 <complexType name="US-Address" base="Address"
              derivedBy="extension">
  <element name="state" type="US-State"/>
  <element name="zip"   type="US-ZIP+4"/>
  <attribute name="country" type="NMTOKEN"
        use="fixed" value="USA"/>
 </complexType>

 <complexType name="UK-Address" base="Address"
              derivedBy="extension">
  <element name="postcode" type="UK-Postcode"/>
  <attribute name="country" type="NMTOKEN"
        use="fixed" value="UK"/>
 </complexType>

<complexType name="Indian-Address" base="Address"
              derivedBy="extension">
  <element name="postOffice" type="string"/>
  <element name="postcode" type="Indian-Postcode"/>
  <attribute name="country" type="NMTOKEN"
        use="fixed" value="India"/>
 </complexType>

 <!-- simple type definition for US-State -->
 <!-- simple type definition for US-ZIP+4 -->
 <!-- simple type definition for UK-Postcode -->
 <!-- simple type definition for Indian-Postcode -->


 <!-- other Address derivations for more countries -->
```

By deriving `Indian-Address` from `Address`, we are saying that the elements of `Indian-Address` consist of the elements of `Address` plus the declarations for a `postOffice` element and `postCode` element, as well as the fixed country attribute. This is like defining `Indian-Address` from scratch as:

```
<complexType name="Indian-Address">
      <!-- elements of Address -->
      <element name="name" type="string"/>
      <element name="street" type="string"/>
      <element name="city" type="string"/>
      <!-- appended declarations -->
      <element name="postOffice" type="string"/>
      <element name="postCode" type="Indian-Postcode"/>
      <attribute name="country" type="NMTOKEN"
        use="fixed" value="India"/>
 </complexType>
```

Why are derivation and extension important for dynamic content? DTD mechanisms use content models to

specify part-of relations, but they only specify kind-of (i.e. derivation) relations implicitly or informally. Type-hierarchies with explicit kind-of relations make both the understanding and maintenance of types easier.

In most dynamic content, structures are more complex than those in static content. Experience with other interchange formats such as EDI has shown that even familiar notions like Purchase Order can become quite complex by the time they have been captured in a standardized form. Further, while the basic layout of a Purchase Order is easy to agree upon, most organizations find that their specific case always has some idiosyncratic attribute that requires special processing. Type hierarchies provide a solution is these cases; complexity can be mitigated by creating appropriate base-type abstractions with different levels of extension, adding layers of additional meaning. Second, a standard Schema can always to extended or restricted to suit a particular idiosyncrasy.

Taken together, complex type structures and substitutability facilitate exchange between loosely coupled applications. Such exchange is currently hampered by the difficulty of fully describing the exchange data model in terms of DTDs; data model versioning issues further complicate interactions in Enterprise Application Integration (EAI) frameworks. When the data model is represented by the more expressive XML Schema definitions, the task of mapping the exchange data model to and from application internal data models is simplified.

Namespaces

In a distributed web environment, we must assume that the same type or element name may mean different things to different people. One XML document may use `Address` elements to specify where people live, and another may use `Address` elements to describe locations of computer memory. An XML application has no way of knowing how to process an `Address` element unless it has some additional information about whose definition we are dealing with.

In XML 1.0, element type names and attribute names are considered to be **local names**. The XML Namespaces Recommendation tries to improve this situation by extending the data model to allow element type names and attribute names to be qualified with a **URI**[1]. Thus a document that describes addresses of people can use `Address` qualified by one URI; and a document that describes addresses of memory locations can use `Address` qualified by another URI. The combination of a local name and a qualifying URI creates **universal names**. The role of the URI in a universal name is purely to allow applications to recognize the name. There are no guarantees about the resource identified by the URI. The XML Namespaces Recommendation does not require element type names and attribute names to be universal names; they are also allowed to be local names.

Here is an example of Namespaces:

```
<document  xmlns:acme:"http://www.acmecorp.com/addresses"
<acme:address acme:addressingSystem="US">
        <acme:name>Dukhi Desi</acme:name>
        <acme:street>8 Oak Avenue</acme:street>
        <acme:city>Fremont</acme:city>
        <acme:state>CA</acme:state>
        <acme:postcode>94555</acme:postcode>
</acme:address>
</document>
```

[1] A Uniform Resource Identifier (URI) is subtly different from a Uniform Resource Locator (URL). A URI is a string of characters for identifying, as opposed to locating, an abstract or physical resource. Some URI's are not URL's. URI's can be locators, names, or both. URLs refers to the subset of URIs that identify resources through representation of their primary access mechanism (e.g., their location on a network), rather than identifying the resource by name or by some other attribute(s) of that resource. A Uniform Resource Name (URN) differs from a URL in that it's primary purpose is the persistent labeling of a resource with an identifier. So far, there are not many URN applications – http://www.handle.net provides one of the few.

Here, we declare a **namespace** `acme` to be associated with the URI http://www.acmecorp.com/addresses. This immediately distinguishes these addresses from those developed by others. Having defined the namespace, we qualify all elements and attributes with `acme:`, thus distinguishing them from identically named elements and attributes in other namespaces.

Namespaces can also be used to keep XML and HTML tags legibly separate in documents where both occur:

```
<h:html  xmlns:acme="http://www.acmecorp.com/addresses"
         xmlns:h="http://www.w3.org/HTML/1998/html4">
 <h:head><h:title>Address Display</h:title></h:head>
 <h:body>
     <acme:address acme:addressingSystem="US">
     <h:table>
       <h:tr align="center">
         <h:td>Name</h:td><h:td>Street</h:td>
         <h:td>City</h:td><h:td>State</h:td><h:td>Postcode</h:td></h:tr>
       <h:tr align="left">
        <h:td><acme:name>Dukhi Desi</acme:name></h:td>
        <h:td><acme:street>8 Oak Avenue</acme:street></h:td>
        <h:td><acme:city>Fremont</acme:city></h:td>
        <h:td><acme:state>CA</acme:state></h:td>
        <h:td><acme:postcode>94555</acme:postcode></h:td>
     </h:tr>
  </h:table>
  </acme:address>
</h:body>
</h:html>
```

Interestingly, since XML Schemas are also well-formed XML documents, we can use namespaces inside XML Schemas as well:

```
<xmlschema:schema xmlns:xmlschema="http://www.w3.org/1999/XMLSchema">

 <xmlschema:annotation>
  <xmlschema:documentation>
   Purchase order schema
  </xmlschema:documentation>
 </xmlschema:annotation>

 <xmlschema:element name="purchaseOrder" type="PurchaseOrderType"/>

 <xmlschema:element name="comment" type="xmlschema:string"/>

 <xmlschema:complexType name="PurchaseOrderType">
  <xmlschema:element name="shipTo" type="Address"/>
  <xmlschema:element name="billTo" type="Address"/>
  <xmlschema:element name="items"  type="Items"/>
  <xmlschema:attribute name="orderDate" type="xmlschema:date"/>
 </xmlschema:complexType>

 <xmlschema:complexType name="Address">
  <xmlschema:element name="name"   type="xmlschema:string"/>
  <xmlschema:element name="street" type="xmlschema:string"/>
  <xmlschema:element name="city"   type="xmlschema:string"/>
  <xmlschema:element name="state"  type="xmlschema:string"/>
  <xmlschema:element name="zip"    type="xmlschema:decimal"/>
  <xmlschema:attribute name="country" type="xmlschema:NMTOKEN"/>
 </xmlschema:complexType>

 <xmlschema:complexType name="Items">
  <xmlschema:element name="item" minOccurs="0" maxOccurs="unbounded">
   <xmlschema:complexType>
```

```
      <xmlschema:element name="productName" type="xmlschema:string"/>
      <xmlschema:element name="quantity">
       <xmlschema:simpleType base="xmlschema:positiveInteger">
        <xmlschema:maxExclusive value="100"/>
       </xmlschema:simpleType>
      </xmlschema:element>
      <xmlschema:element name="price"    type="xmlschema:decimal"/>
      <xmlschema:element ref="comment"   minOccurs="0"/>
      <xmlschema:element name="shipDate" type="xmlschema:date" minOccurs='0'/>
     <xmlschema:attribute name="partNum" type="partSKU"/>
     </xmlschema:complexType>
    </xmlschema:element>
   </xmlschema:complexType>

  <xmlschema:simpleType name="partSKU" base="xmlschema:string">
    <xmlschema:pattern value="\[A-Z]{3}-d{3}"/>
   </xmlschema:simpleType>

  </xmlschema:schema>
```

Each of the elements in the schema has a prefix `xmlschema:` which is associated with the XML Schema namespace through the declaration, `xmlns:xmlschema="http://www.w3.org/1999/XMLSchema"`. So the prefix `xmlschema` is used to denote the XML Schema namespace, although any other prefix could be used. The same prefix appears on the names of built-in simple types, e.g. `xmlschema:string` – to guard against any possibility of confusing the string of XMLSchema with some other definition of string in the vocabulary of another author.

One of the interesting things about namespaces is that since they are URIs, people assume that they must be the address of something. They're not. Though namespaces are URIs, the namespace draft doesn't care what (if anything) they point at. The reason that the W3C decided to use URIs as namespace names is that they contain domain names (e.g. www.oracle.com), which work globally across the Internet. The XML namespaces recommendation does not require XML namespaces to exist as physical entities. They are merely qualifiers that come with the convenience of global uniqueness.

Applying XML Schema

A number of areas in which XML Schema is especially suitable have been touched upon above. Some of the major use cases are:

eCommerce: In both Business-to-Business and Business-to-Consumer eCommerce, data is largely dynamic. XML Schema maps robustly to server-driven data models such as SQL:1999, which provides user-defined types, derivation, references between types, collections etc. Libraries of schemas define business transactions within markets and between parties. XML Schema validates a business document, and also provides access to its information set.

Web Publication and Syndication: The key to syndication on the web is in highly customizable distribution between publishing and syndication services. Collections of XML documents with complex relations (cross-references, kind-of) among them will be the norm. Protocols such as **ICE**, which are built on XML, will be able to take advantage of the complex structural aspects of XML Schema.

Enterprise Application Integration (EAI): The essence of an EAI architecture is in dynamic data exchange between loosely coupled applications. DTDs cannot fully describe today's exchange data models. XML Schema is a big step forward -- its capability for metadata interchange is not only an EAI simplifier, but also an important optimization technology, as discussed below.

Process Control and Data Acquisition: In multi-vendor, distributed systems such as those in plant automation, security, devices, traffic routing etc., XML Schema can aid outgoing and incoming message validity. Controllers can determine which parts of messages they understand: when to

ignore information and when to raise errors.

Loose coupling between systems leads to all sorts of optimization issues. XML opens up the possibility of dynamic optimizations made on the basis of self-describing servers. A given database can emit a schema of itself to inform other systems what counts as legitimate and useful queries. Any query interface can inspect XML schemas to guide a user in the formulation of queries. There are a number of initiatives in the interchange of metadata (especially for database systems) and in the use of metadata registries to facilitate interoperability of database design, DBMS, query, user interface, data warehousing, and report generation tools. Examples include the **ISO 11179** and **ANSI X3.285** data registry standards, as well as OMG's proposed XMI standard. DTDs were inadequate for fully expressing database metadata; the new datatypes proposed by XML Schema, as well as the set of schema constraints the XML Schema will provide will enable dynamic description of database and run-time tuning and optimization of queries.

Another important kind of optimization will help cut down network traffic. Consider a case where someone is trying to book movie tickets using a cellphone. If you go to an on-line ticketing service and ask for all the popular movies showing in your area, you will likely get a long list that does not fit into the form factor of your cellphone display -- that is, you would get a long list that you would have to scroll up and down to inspect. To shorten this list, you need to fine-tune your showtime or your preferred movie title. In order to do this, you would have to send a request across the Web to your ticketing service and wait for a response. If, however, the list of movies in the area had been sent in XML, using XML Schema to express all the constraints about location, number of tickets available, earliest and latest showtimes, then the ticketing service you send a simple Java program along with the data that could refine and sort your choices in milliseconds, without incurring multiple network hits. Since the richness of XML Schema is comparable to programming languages, you can exploit additional processing of richly described structures at various clients. This helps dynamic content to be used with greater efficiency across scarce network resources.

XPath

Once a well-formed and deeply nested XML document has been created, we would of course like to exploit this structure to 'traverse' to or pinpoint a certain element of interest. For example, we might want to start 'tunneling' from `Statement Request`, to `BankAccount`, to `AccountID` – in order to get at the data associated with an `AccountID`. Xpath tries to achieve such **hierarchical traversal** or **navigation** to target specific elements, attributes, text fragments etc.

XPath is a system of expressing traversal paths as strings. Its syntax is quite similar to the hierarchical directory traversal syntax familiar to command-line users of UNIX or DOS. For example, to identify the `AccountID` in our Bank Statement example, we would use the following XPath:

```
StatementRequest/BankAccount/AccountID
```

In XML terminology, a **path** is a series of steps to a target location. Paths may be **absolute** or **relative**. The starting point of an absolute path is the same as the root element of a document. An absolute path begins with the symbol /, indicating the root:

```
/StatementRequest/BankAccount/AccountID
```

A relative path, on the other hand, starts from an existing location in a document, or from some external context that identifies a starting point in a document. Such a path may appear within an expression:

```
http://www.imtb.com/getBankAccount?BankID=310824233/AccountID
```

In the above case, `http://www.imtb.com/getBankAccount?BankID=310824233` fires off a query to a database to return a `BankAccount`, and then we traverse to the `AccountID` relative to the returned element. The BankAccount element returned from the query is the **context** for starting the traversal.

Given a context, we can use the `../` construct to go to its parent. Given a `BankAccount`, the following XPath identifies the starting date of the recurring statement requested for this account:

```
../Recurrence/StartDate
```

When the names of all elements between the context element and the target are not known, the `*` symbol may be used to stand in for any element. Again, this notion (called a **wildcard**) is borrowed from traversal of directories in operating systems. So, if we know that there is an `AccountID` somewhere under `StatementRequest`, we can specify

```
StatementRequest/*/AccountID
```

Whenever we say `AccountID` without qualification i.e.

```
AccountID
```

a relative path is assumed – i.e. it always refers to an `AccountID` element that is the child of the current element in context. We can also make this more explicit, using the `child::` notation :

```
child::AccountID
```

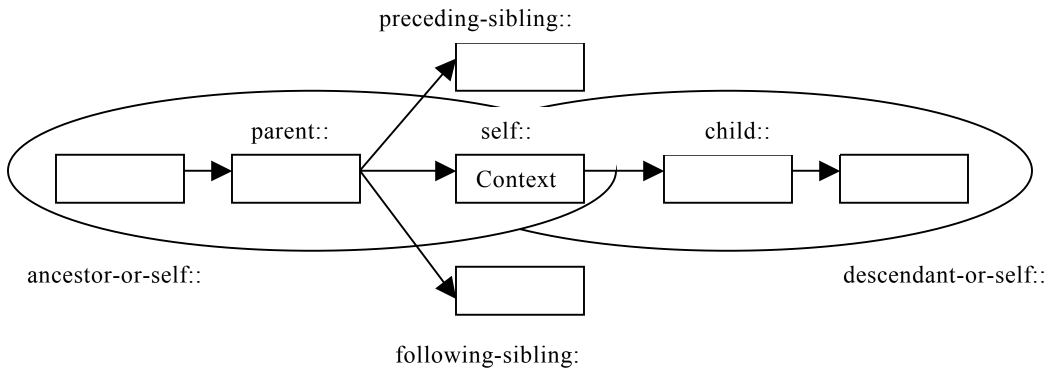
Similarly, the verbose `descendant-or-self::` notation can be used in place of the `*` for wildcards:

```
child::StatementRequest/descendant-or-self::node()/child::AccountID
```

Similarly parent:: can be used to traverse up the hierarchy. Given the context of an `AccountID`, we can specify

```
parent::node()/parent::node()/child::Recurrence/child::StartDate
```

Other navigational constructs such as preceding-sibling::, following-sibling:: etc. are also available, as is ancestor-or-self::



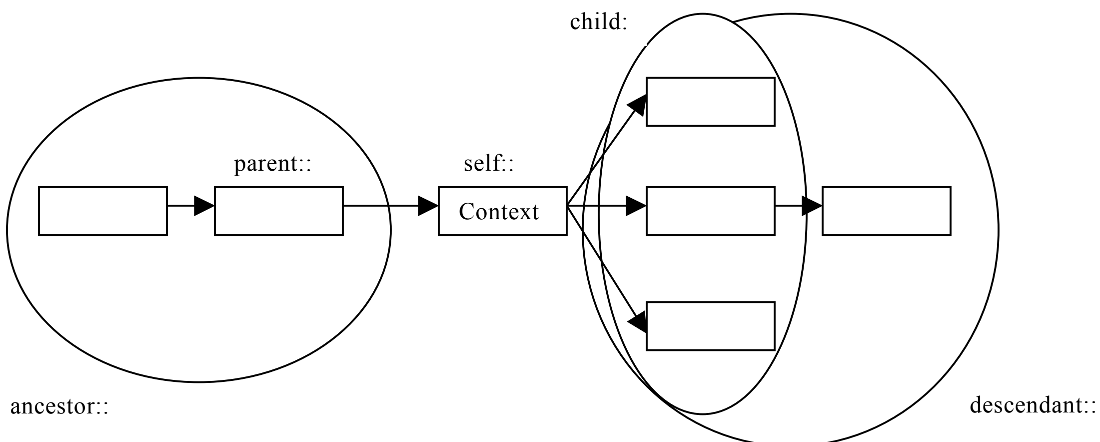


Fig. 4: XPath Hierarchical Relationships

Predicates in XPath

Just using hierarchies to pinpoint elements can quite brute-force. Suppose we have dozens of BankAccount elements in a StatementRequest. How do we find just the one we're looking for?

XPath uses predicate filteres to remove unwanted items from lists. For a given StatementRequest context, the first BankAccount can be chosen by:

```
BankAccount[1]
```

The number in this case does not refer to the position amongst all the children of StatementRequest, but to the position amongst all the BankAccount children only. To select a BankAccount only when its position is the first amongst all the children of a given Statement Request, a more complex predicate must be used:

```
*[position() = 1 AND self::BankAccount]
```

The * selects all the sibling elements one level below relative to the context, and then the filters in the predicate are applied.

Similarly, the following XPath fragment can be applied to the last BankAccount (but not the last sibling, if it happens to be BankAccount):

```
BankAccount[last()]
```

Another useful function is count(), which can be used to determine how many occurrences of a particular element are in a document. From the context of a document that has multiple StatementRequest elements, the following XPath fragment can be used to select only those StatementRequest elements that contain exactly one BankAccount:

```
child::StatementRequest[count(child::BankAccount) = 1]
```

We can also test contained elements. From the context of a StatementRequest, the following will select only those BankAccounts that directly contain a BankID:

```
child::BankAccount[BankID]
```

In addition to existence checks, the value of a contained element can be tested:

```
child::BankAccount[BankID="310824233"]
```

Attributes can be tested using the attribute keyword, which can be abbreviated using the @ symbol. The following selects the numberingSystem attribute of the BankID element:

```
BankID/attribute::numberingSystem
BankID/@numberingSystem
```

Similarly, the following will select every BankID that has a numberingSystem attribute with value

‘American Routing and Transit’:

```
BankID[@numberingSystem=”American Routing and Transit”]
```

Strings be examined for more than exact matches. To check if any of the child nodes of `BankID` contains a sub-string “American”, we can ask:

```
BankID[contains(., “American”)]
```

A number of other string operators, such as `starts-with()`, `concat()`, `substring()` etc. are provided. You can also convert strings to numbers where possible, using the `number()` function. Real numbers can be converted to integers using the `round()` function, and the customary `floor()` , `ceiling()` functions are also provided.

Hypertext Linking

On a typical document displayed in a browser, you often find text like “Click here for more Information”, and the URL underlying this text takes you to some resource which contains the additional information. HTML hardcodes tags like `A`, `IMG` etc. to link to other locations. In XML, we can't predict ahead of time what the document creator will call a linking element. Therefore, we need a standard way to put a marker (perhaps in the form of a special attribute) on an element in order for applications to recognize it as a link.

XLink is a new proposal before W3C for hyperlinking functionality. It comes in two parts: the XLink component that deals with allowing links in your XML documents to be recognized as links, and the XPointer component that allows your links to address precise sub-parts or sections of an XML document.

In other words

The **XLink** proposal governs how you insert links into a XML document, where the link might point to (e.g., a JPG file on some server), as well as what semantics link traversal might have; and

The **XPointer** proposal governs the fragment identifier that can go on a URL when you're linking to an XML document, from any other document, such as a HTML file or another XML file.

XLink

XLink uses ideas from the much more complex and sophisticated SGML HyTime standard for hypertext linking. You don’t need to know anything about HyTime, though, to understand XLink.

In HTML, `A` is a unidirectional link – you click to go from here to there. XLink considers such links to be **simple**. There can be other kinds of links, such as **extended**, **locator**, **group** or **document**. Together, these provide the capacity for:

Links with multiple destinations – When you click on such a link, you may be prompted for choosing one of multiple destinations; for example, when trying to download software, you could be prompted to choose the server closest to you by just clicking on one link.

Multi-directional links – If you want to have some way other than the Back button on your browser to return to the location of the original link, this one is for you. If multi-directional links have multiple destinations, you can start from any one of the ends and go to any of the others.

Out-of-line links – The link doesn't have to be stuck into any one the pieces of content representing the ends. This means that you can store the linking information *outside* all the information that you're hooking together; you can update your link if one of the dozens of chunks changes in any way.

Inline content replacement – XLink gives you a capability for replacing content inline with updated content from another document. This creates very flexible syndication capabilities.

Annotation capabilities – The ability to create links on documents that you don't own, as a system of 'sticky notes' of links

Databases of links – Capable of querying, sorting, and processing link collections.

We will not look at all types of XLinks, but let's consider two – beginning with simple links. Like today's HTML HREFs, and they simply point to one direction, without expectation that the pointed-to piece will somehow refer back to them. Their intelligence is limited to the page they happen to be on, and processing them requires only a simple page processor.

Here is an example of a simple link. It is about an XML element that helps you order from the website of a music label "Indie Voices":

```
<order xlink:type="simple" xlink:href="http://www.indievoices.com/order.htm"
xlink:title="Indie Voices Ordering Page" xlink:role="guest"
xlink:show="new" xlink:actuate="user" >Click here to order from the Indie Voices
website</order>
```

This would create a simple link, containing the text "Click here to order from the Indie Voices website", pointing to http://www.indievoices.com/order.htm. When you moved the mouse over this link, the title "Indie Voices Ordering Page" would pop up; in case a stylesheet (see below) asked this link for a role, the answer would be "guest". When the link was clicked on, a new window would pop up, displaying the contents of http://www.indievoices.com/order.htm.

Next, let's take a brief look at extended links. These links use containment to identify sets of locators (which could be local or global). Each set has a parent element, which contains the properties for the set, while the locators or local resources are elements within this parent.

```
<MusicCD xlink:type="extended: xlink:title="Acme Salsa Band" xlink:role="CD"
xlink:showdefault="new" xlink:actuatedefault="user">
<frontcover xlink:type="locator" xlink:href="acmesalsafront.jpg" xlink:title="Acme
Salsa Band: The Best Years"/>
<backcover xlink:type="locator" xlink:href="acmesalsaback.jpg"
xlink:title="Tracks"/>
<linernotes xlink:type="locator" xlink:href="acmesalsaliner.htm" xlink:title="The
Acme Salsa Story"/>
<order xlink:type="simple" xlink:href="order.htm?Acme Salsa Band"
xlink:title="Indie Voices Ordering Page" xlink:role="guest" xlink:show="new"
xlink:actuate="user"> Click here to order from the Indie Voices website</>
</MusicCD>
```

Extended links may also contain arcs. Arcs identify connections between locators, providing information on traversal paths:

```
<MusicCD xlink:type="extended: xlink:title="Acme Salsa Band" xlink:role="CD"
xlink:showdefault="new" xlink:actuatedefault="user">
<frontcover xlink:type="locator" xlink:href="acmesalsafront.jpg" xlink:title="Acme
Salsa Band: The Best Years" xlink:role="front"/>
<backcover xlink:type="locator" xlink:href="acmesalsaback.jpg" xlink:title="List
of Tracks" xlink:role="back"/>
<linernotes xlink:type="locator" xlink:href="acmesalsaliner.htm" xlink:title="The
Acme Salsa Story" xlink:role="liner"/>
<order xlink:type="simple" xlink:href="order.htm?AcmeSalsaBand" xlink:title="Indie
Voices Ordering Page" xlink:show="new" xlink:actuate="user"
xlink:role="order">Click here to order from the Indie Voices website</>
<link xlink:type="arc" from="front" to="order" />
<link xlink:type="arc" from="back" to="order" />
<link xlink:type="arc" from="liner" to="order" />
</MusicCD>
```

Note the references to role values. Connections here are between the front cover, back cover and liner notes to

the ordering page.

The method of actuation of a link, as well as the behavior of a link once it has been actuated, are controlled by the `actuate` and `show` attributes of the link. A value of "`user`" for the `actuate` attribute indicates that the link is to be traversed only when the user explicitly chooses it. A value of "`auto`" means that the link is actuated automatically on presentation. The `show` attribute determines how the material that the link points to will be presented. A value of "`replace`" implies the familiar browser behavior – the material currently on display will be discarded, and the new material displayed; a value of "`new`" creates a new display of the new material without replacing the old; and a value of "`embed`" indicates that the new material is to be brought in a embedded in the current material at the position of the XLink.

So XLink is, in sum, really a language that allows you to invent your own link elements, specify your own traversal behavior, and provides a set of primitives to do so efficiently.

XPointer

Once you have found the document you want, perhaps by following a link, you need to home in to the part of the document that interests you. XPointer builds on XPath to provide better location specification for XML documents. XLink can use it as part of specifying where within a document its link-ends reside.

HTML frequently uses **fragment identifiers** on the end of URLs to take you to a specific part of a HTML document. Any characters following a # character in a URL constitute a fragment identifier. For example, you could specify the following HTML to refer to an **anchor** named `xptr` in a document:

```
<p> See: <a href="xmlprimer.html#xptr">XPointer</a> </p>
```

XPointer adds to the fragment identifier capability in several ways:

- XPointers can point to specific places inside documents.
- XPointers provide finer-grained addressing into elements, string selections, and ranges inside documents.
- XPointers can navigate hierarchies (i.e. the structure of XML documents), so that locations are human-readable and writable.

XPointer expressions are simply appended to a URL. XPointer makes heave use of XPath – XPath expressions are used inside Xpointers for navigation. For example, in order to navigate within a XML document containing StatementRequests, we can specify:

```
http://acmecorp.com/statements/StatementRequest.xml#xpointer(/BankAccount/AccountI
D)
```

The range functionality of XPointer is very useful. For example, we could specify the following to select a range of account elements:

```
xpointer(AccountID("000000000")/range-to(AccountID("999999999")))
```

There are a number of quite useful XPointer functions that go beyond XPath. The `unique()` function of XPointer can be used to determine whether an expression actually locates a single object in the document.

```
http://acmecorp.com/statements/StatementRequest.xml#xpointer(/BankAccount[unique()
]
```

Similarly, XPointer adds functions such as `here()` and `origin()`, which provide for addressing relative to

the location of a XPointer expression itself; functions `start-point()` and `end-point()`, to identify beginning and ending locations, and so on. Together, these make XPointer ideal for pointing into the sub-parts of a document.

DOM

Earlier, we had discussed XML processors, used to read XML documents and provide access to their content and structure to XML applications. As an application developer, once you have some XML documents, you will need to access the information contained in them from your application code.

You could treat the XML document is just a text file, and write a text file reader that interprets the information in the XML document in a way that your application code can use. This would, however, be quite tedious, and require you to understand all the constructs of XML. Also, such code would have to be written over and over again for by different application developers all over the world who want to access information in XML documents. W3C realized that this would be a problem, and they created a standard way to create these XML document processors or XML parsers. Typically, XML processors parse an XML document, build a tree model of the elements in the document, and then allow the application to access this tree by means of a standard API called the Document Object Model or **DOM**.

A DOM XML parser is, quite simply, a Java program that converts your XML documents into a Java object model. You point a DOM XML parser to an XML document; it parses the document, and gives you a bunch of objects in the memory of your Java Virtual Machine. When you need to manipulate any information stored in the XML document, you can do so through these objects in memory. So a DOM XML parser creates a Java document object *representation* of your XML document file. There are lots of free DOM XML parsers out there, including one from Oracle as part of the XDK.

The parser also performs some simple text processing as it creates the object model. It expands all entities, compares the structure of the information in the XML document to a DTD (if one is used), and if this processing is successful, the XML document is converted into a tree of nodes in memory. The tree of nodes contains all the data and structure of the information contained in the XML document. This tree of nodes can be accessed and modified using the DOM API.

The DOM API consists of a set of **interfaces.** The XML parser interprets these interfaces. If you want to access XML documents from within Java, you need to import the org.w3c.dom package in Java, and then simply use these classes to get at the tree of nodes.

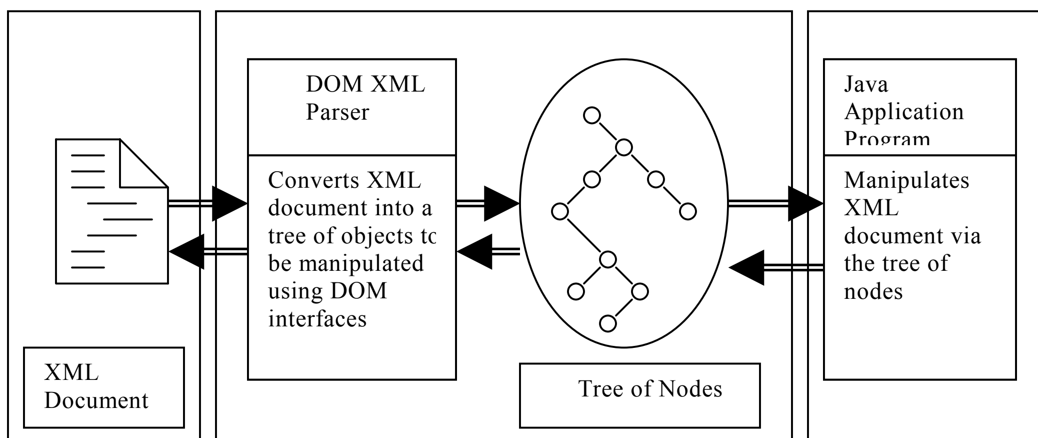


Fig. 5 : DOM Processing

Interestingly, when DOM was being developed, many Web browser developers wanted a similar capability for accessing HTML elements. So W3C decided DOM would apply to common constructs like elements, comments, processing instructions, text content etc. that are present in both HTML and XML, and would in addition have some HTML-specific extensions.

In the document object tree, everything is a **node**. A node may have other nodes nested inside it. The node can hold information, like its tag-name, its value, and its child nodes (if any). This hierarchical organization reminds one of a file-system view of data, where items are organized hierarchically, a folder may have files in it or other folders, and everything is descended from one root folder. Of course, we can encounter such ‘tree-structured’ data in other areas as well, such as bills of material in manufacturing, where assemblies are made of parts that have sub-assemblies and sub-parts.

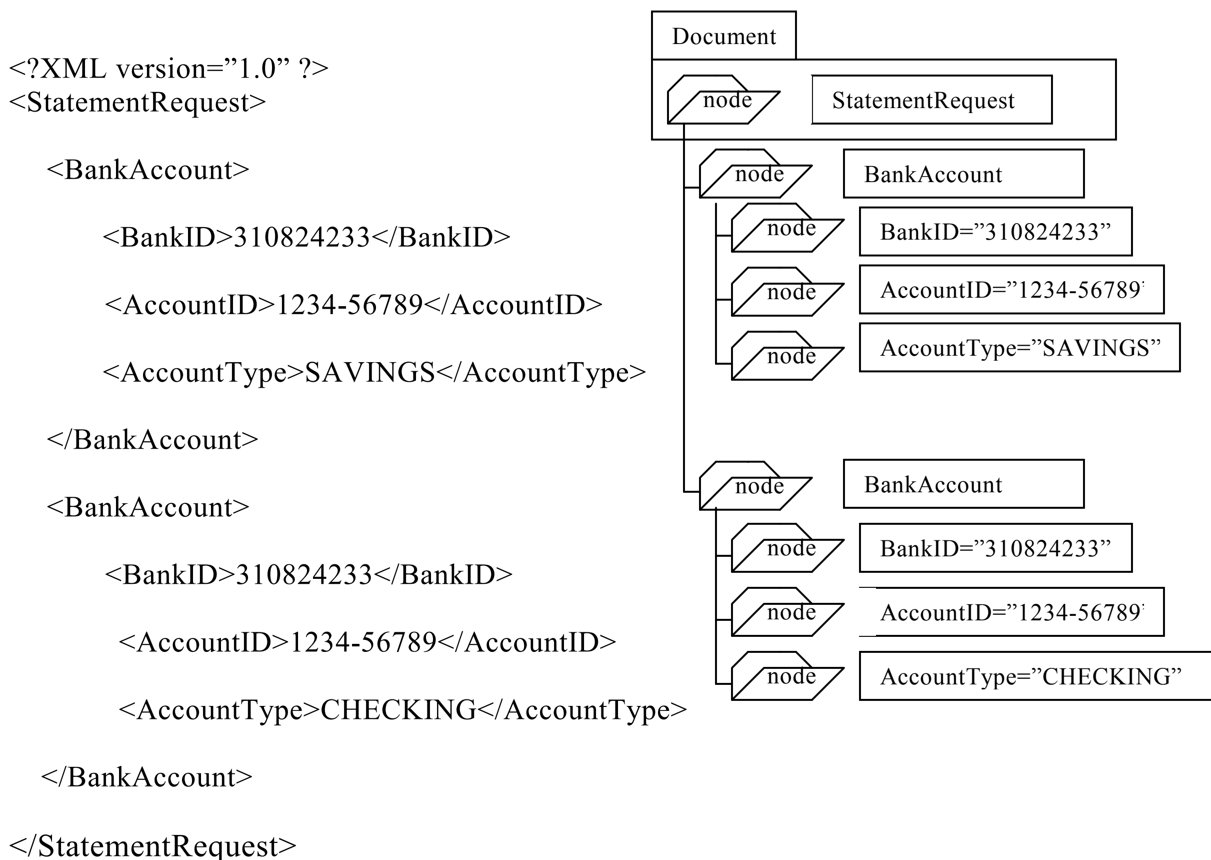


Fig 6: Tree Structure of DOM Nodes

A document object itself is a node, it is descended from one node, and it may have other nodes inside it. The Node interface is central to DOM – most of the time, you can get by using just this interface. There are also there are other interfaces like DocumentType. Most of the interfaces are, however, subclasses of Node, and extend it to provide specific functionality.

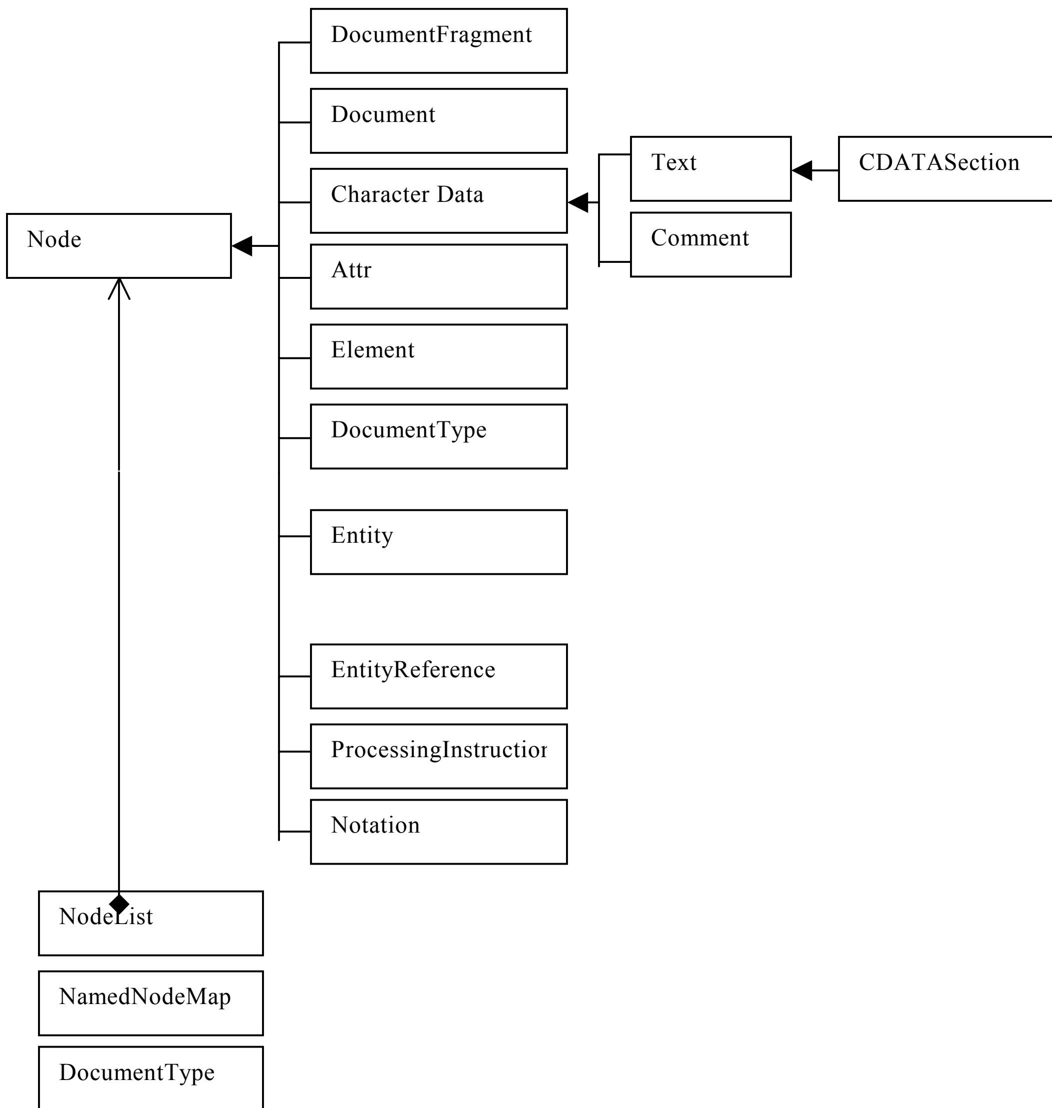


Fig. 7: DOM Interfaces

What kinds of behavior or methods do these interfaces have? For the most part, programmers want to get a Node's value, its child Nodes and so on, and of course such methods are provided. In other situations, programmers want to know what the type of a node is (e.g. whether it is an Element or a piece of Notation), so these methods are provided as well.

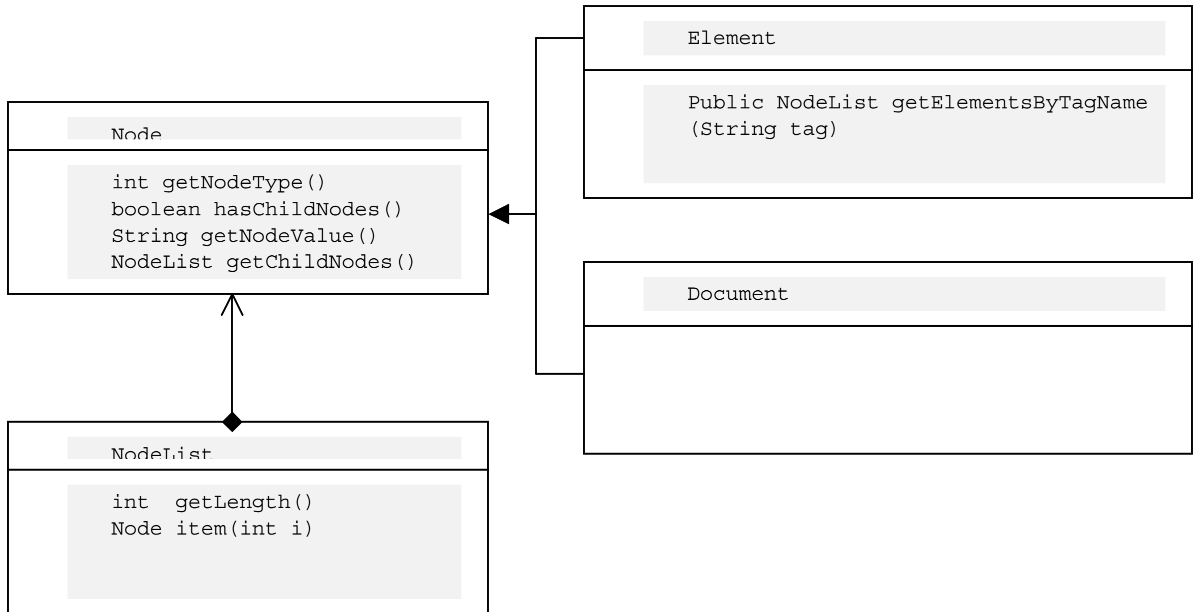


Fig. 8: Some popular DOM Interfaces and their methods

What answers would we get if we called these methods on the Nodes in our XML fragment?

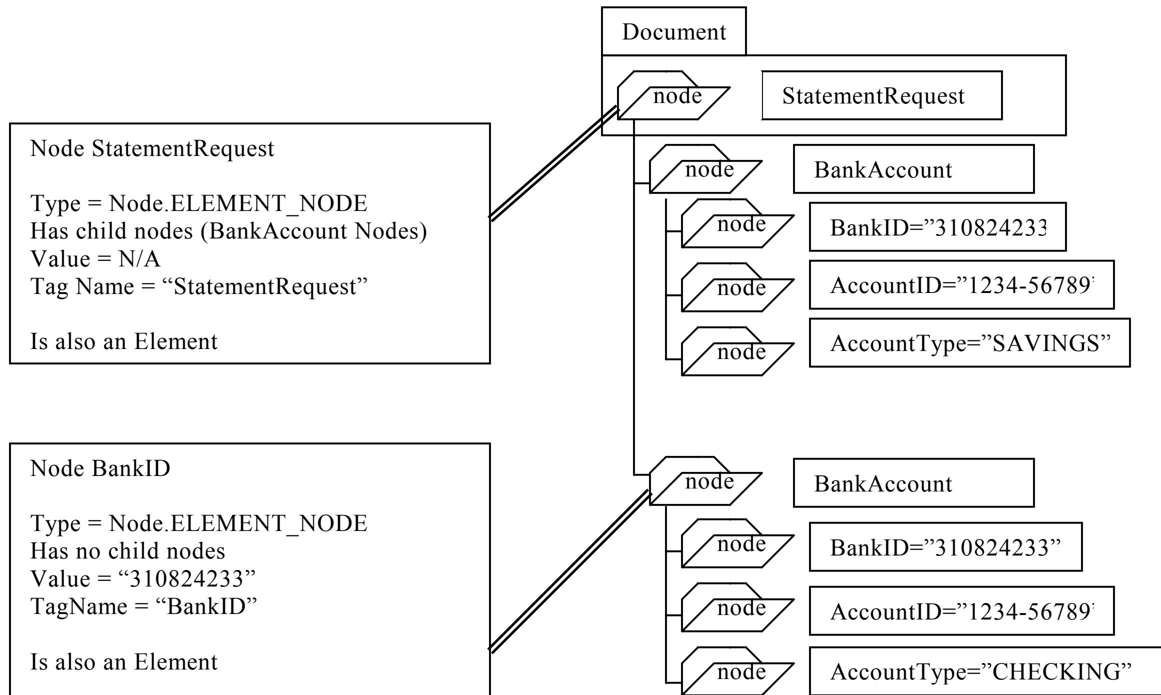


Fig. 9: DOM Example

Simple API for XML (SAX)

There's another API we will mention in passing. The SAX API is an **event driven** API, rather than **document-model driven** like DOM. The event model of SAX works like this:

> the SAX XML parser processes elements serially
>
> the XML application registers for those events that it is interested in and provides callback functions to handle the element
>
> when the events of interest are encountered, the callback functions are invoked.

The callback functions are defined by the interfaces `DocumentHandler`, `ErrorHandler`, `DTDHandler` and `EntityResolver`. For example, an application might be interested in the `DocumentHandler` interface and hence provide callbacks for the methods in this interface – viz. `StartDocument`, `EndDocument` , `StartElement`, and `EndElement`. The application might also want to implement the methods defined by the `ErrorHandler` interface e.g. `Warning` or `FatalError`. The SAX parser then informs the application when it encounters these events, and the application in turn invokes methods to process the events.

The SAX API is a fast API. It has a lighter computational footprint than DOM – i.e. it consumes less memory and CPU resources. The DOM API must construct a tree out of the whole document before any processing can be done. DOM is ideal for interactive applications that require an object representation in memory. SAX is more suited for server side applications that do not need to create a tree structure. SAX may be a good choice for network-oriented programs that send and receive XML documents. SAX, in general requires a lot more coding than the DOM interface.

StyleSheets – XSL and XSLT

While XML documents are marked up to be easily read by both people and software, people are typically more interested in seeing the content rather than the markup. Further, they are interested in seeing the content on different devices or media – PCs, TVs, PDAs, paper etc. Clearly, the display of XML on each of these would need different kinds of formatting. To display the content of XML documents on these devices, it necessary to replace the tags with appropriate text styles, on occasion to transform between styles. The Extensible Stylesheet Language (XSL), along with the XSL Transformation (XSLT) standard, is designed to address these requirements.

The content of an XML element such as Address has no explicit style. A single style would not be acceptable to all forms of publications – the weight of font, number of lines of display, purpose of presentation – all have to be taken into consideration when determining a style. A **style sheet** lists the rules according to which the elements associated with a DTD or schema are to be formatted for display. Such style sheets may be shared by a number of documents, reducing the effort needed to deal with large number of documents that need to be formatted in the same way.

XSL is basically a set of formatting objects. The most popular formatting objects are table cells, blocks and so on. Each formatting object has a number of properties that can be used to layout the document according to the tastes of the designer – for example, the block object has a property that determines the spacing between this block and the next. XSL makes the XML elements 'flow' into the formatting objects. We could assign the

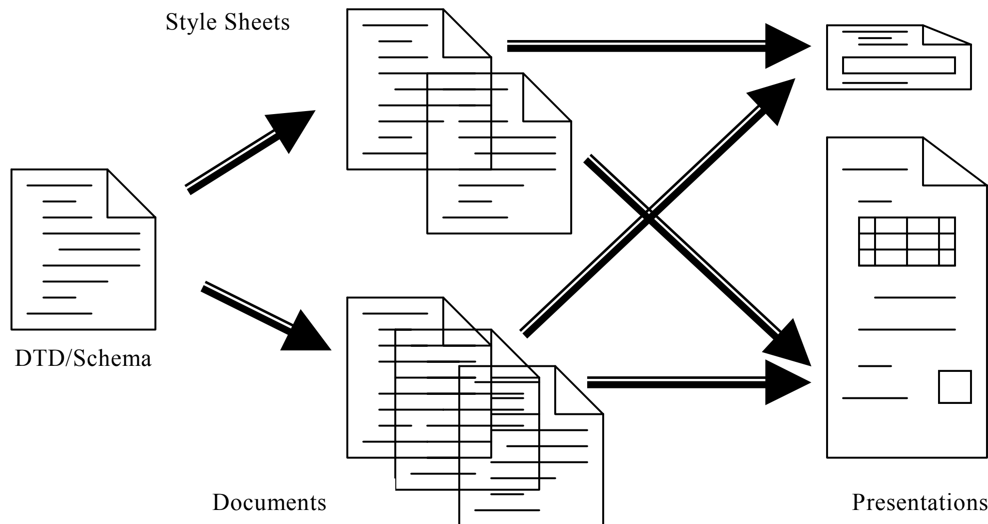


Fig. 10: Style Sheets

individual elements of Address to different table cells. We could reduce the font if the display was on a device with a small screen, such as a cellphone. Some of the other tasks performed by stylesheets could be:

Leave a gap of 2 lines between paragraphs.

Increase the volume on a cellphone if emphasized text is to be spoken.

Display hypertext links in black.

Make a stock quote blink on a screen.

XSL has many more features, such as being able to process the same element twice, suppress elements in one location and display them in another, add generated text before displaying, and even changing the order of elements. Together, these capabilities are very useful for creating different presentations for the same data.

XSLT is a standard for transformations. In addition to XSL formatting, it can

Transform data from one XML format to another XML format.

Specify which XSL formatting objects need to be applied to an element.

Reorder and sort source elements.

Add prefixes or suffixes to elements in the source content.

XSLT transformation instructions are distinguished from XSL formatting instructions by the use of the namespace http://www.w3.org/1999/XSL/Transform. Let us see how XSL and XSLT work in practice.

Suppose we have a simple XML document containing a directory entry – a person and his phone number:

```
<?xml version="1.0"?>
<DirectoryEntry>
       <name>Dukhi Desi</name>
       <number>(123)456-789</number>
```

```
</DirectoryEntry>
```

A simple stylesheet for this would display the name, followed by number. In order to do so in a HTML browser, we have to create HTML out of the XML:

```
<xsl:stylesheet
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:template match="/">
<H1><xsl:value-of select="//name"/></H1>
<H2><xsl:value-of select="//number"/></H2>
</xsl:template>
</xsl:stylesheet>
```

The above XSL directives transform the XML document into the following code with HTML tags:

```
<H1>Dukhi Desi</H1>
<H2>(123)456-789</H2>
```

As you can deduce, the XSL above uses the value-of directive to put the value of element <name> between HTML tags <H1> and </H1>, and then the same directive to put the value of element <number> between <H2> and </H2> tags. This results in a display where the name is displayed bolder by a browser than the number.

If you wanted to create a different display, one in which the number precedes the name and both had the same font size, you could create another stylesheet:

```
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:template match="/">
<H1><xsl:value-of select="//number"/></H1>
<H1><xsl:value-of select="//name"/></H1>
</xsl:template>
</xsl:stylesheet>
```

This would generate a different arrangement of HTML tags:

```
<H1>(123)456-789</H1>
<H1>Dukhi Desi</H1>
```

Let's discuss the XSL processing model. An XSL processor begins with a style sheet and a **source tree**, which is akin to a DOM Node representation of the XML document. In the simplest case, the XSL processor starts at the root node of the source tree and processes it by finding the template in the style sheet that describes how that element should be displayed. Each node is then processed in turn until there are no more nodes left. Processing may invoke various XSLT transformation instructions.

In more complex cases, each template can specify which nodes to process, so some nodes may be processed more than once and some may not be processed at all.

All this processing creates a **result tree**. XSL allows the result tree to be composed of any kind of elements, such as XSL formatting objects, HTML element names etc. When HTML is used in the result tree, XSL will transform a XML source document into an XML document that looks very much like HTML but is still XML! In fact, it's impossible for XSL to produce a document that isn't well-formed XML.

Suppose we had the item element from the Purchase Order example in its own separate document:

```
<?xml verison = "1.0" ?>
<item partNum="ABC-123">
            <productName>Kayak</productName>
            <quantity>1</quantity>
            <price>567.80</price>
```

```
            <comment>Prefer green with yellow trim</comment>
</item>
```

We might decide that when displaying the item fragment in a HTML browser, we want to choose different fonts, indentation etc. for the different elements. Specifically, we want to display the `productName` and `partNum` in bold font, the `quantity` and `price` in italics, and the `comment` in, say, blue to make it stand out. In XSL, we can use **template rules** to define generic directives for accomplishing such formatting. The XSL processor parses a XML source and tries to find a matching template rule. If one is found, the instructions inside the matching template are evaluated.

The XSL for generating the HTML fragment for displaying item in a browser could look like:

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

<xsl:template match="productName">
<P><B><xsl:value-of select="."/></B></P>
</xsl:template>

<xsl:template match="@partNum">
<P><B><xsl:value-of select="."/></B></P>
</xsl:template>

<xsl:template match="quantity">
<P><i><xsl:value-of select="."/></i></P>
</xsl:template>

<xsl:template match="price">
<P><i><xsl:value-of select="."/></i></P>
</xsl:template>

<xsl:template match="comment">
<P style="color:blue"><xsl:value-of select="."/></P>
</xsl:template>

</xsl:stylesheet>
```

Note that by using a preceding @ symbol, attributes can be selected in the same way as elements. The result would be:

```
<P><B>Kayak</B></P>
<P><B>ABC-123</B> </P>
<P><i>1</i></P>
<P><i>567.80</i></P>
<P style="color:blue">Prefer green with yellow trim. </P>
```

We can create very complex result trees from very complex (or simple!) sources, but hopefully the above will give you a flavor of how this is all done.

Databases & XML

XML has several specific advantages in the context of database applications, which are worth recapitulating:

Fine grained content model

Because each element of an XML document can be “extracted” and stored in the database, XML documents can essentially be “broken” in to many pieces, with each piece being used in other XML documents. This separates data elements from the original document it appeared in and greatly enhances content reuse.

Self describing, open format for data & metadata exchange

XML has essentially been able to provide an inexpensive substitute for EDI for medium and small companies that cannot afford their own private networks or participate in commerce networks for large corporations. This has important implications for content syndication. Businesses can syndicate their information to customers, as well as utilize information from their dealers and suppliers, as long as the data is self-describing, XML-based. Since XML Schemas are XML documents themselves, it is also possible to exchange metadata in the same way. Namespaces, support for various encodings, other globalization features – all make XML attractive as a trans-national format.

Flexible presentation

XSL and XSLT allow you to reformat the same XML data generated from a query for any desired presentation - a printed invoice, an HTML screen displayed in a web browser, a PDA screen, or a cell phone. With HTML, authors need to create new pages for each new rendering option – so any change to the underlying content generates a cascading editing effort. XML's separation of content from presentation drastically reduces this effort.

Some Disadvantages of Using XML!

For a number of reasons outlined above, XML is becoming popular rapidly. While we keep all of these in mind, it is also instructive to reflect on some of the disadvantages.

XML requires significant development investment. While parts of it such as DOM harmonize with existing programming languages, a large part of the XML stack is bewilderingly new. Programmers have to learn many different aspects of this new technology before they can be productive.

Industry-group support is still emerging, to standardize on vocabularies and schemas for various segments. Without such support, it is not clear if the 'exchange' capabilities of XML will be fulfilled.

Not all the infrastructure for XML has been defined, let alone developed. Many of the technologies discussed in this chapter, -- such as XML Schema, Namespaces, XSL and XSLT, XLink and XPointer – are recommendations put forward to W3C or working drafts of proposals. The more innovative users of XML face heroic pioneering efforts.

XML encodes the (metadata) self-describing details with the physical data content (actual data being sent) for each message. This has the impact of increasing the physical byte count of messages as they traverse the network backbone, sometimes by a factor of 3. The database community is still figuring out the optimizations that need to be made to XML storage and retrieval to work around this 'bloat'.

Single purpose applications do not derive a significant return on investment with XML; multi-purpose applications, where provision for future growth and expansion is needed, might.

Oracle and XML

Database Server

Anything that you could want to do with XML and Oracle8*i* is provided by Oracle. This section describes the major possibilities in brief – they are taken up in detail in Chapters 26 and 27.

Oracle tables produced as XML documents

If your data is currently in Oracle8*i* and you want to use XML, there is no need to try a new XML data store or to begin creating XML documents by hand. Conventional scalar data, tables with rows and columns, can be produced from Oracle8*i* as XML documents.

Oracle tables produced as styled XML documents

If all of your data is currently in Oracle8*i* and you want to display your Oracle data as XML in web browsers, there is no need to use HTML or other web technology. Conventional scalar data, tables with rows and columns, can be produced from Oracle8*i* as styled XML documents.

XML documents stored in Oracle tables

If you are creating XML documents or receiving XML documents from third parties, these XML documents can be stored in an Oracle table. Such **structured storage** means than you can invoke SQL queries on you XML data. Oracle lets you choose relational tables or object-relational ones when it comes to deciding how you want to decompose the XML documents. Object-relational storage results in the most faithful retention of the structure of the XML documents -- tag names are mapped to columns, text-only data is mapped to scalar columns; sub-elements to object types and lists to collections. XML documents can also easily be reconstructed.

XML documents stored in Oracle as LOBs

If you do not care about structured storage or SQL search, the XML document can be stored as a BLOB or CLOB in Oracle. You can still search on CLOBs using the text search capabilities of *inter*Media. *inter*Media Text can be also be used for XML **section-searching**, i.e. searching for text within specific tags.

XML documents stored in Oracle, accessed as files

If you would like to act on the XML document as if it were a file but have the security, scalability, multi-user access and integrity of Oracle8*i*, the Oracle Internet File System (*i*FS) can be used. *i*FS gives you a file metaphor for manipulating XML. It is most useful when you have document-driven (rather than data-driven) applications, and prefer a higher-level solution rather than making your own XML storage decisions.

Tools

Oracle Tools support XML comprehensively. The most popular tools for Oracle developers are:

- Oracle XSQL Page Processor Servlet
- Oracle XML SQL Utility for Java and PL/SQL
- Oracle XML Parser for Java and PL/SQL
- Oracle XML Parser for C and C++
- XML Class Generator for Java

Process XML pages – querying and transforming

The Oracle XSQL Page Processor makes it very easy to do the following (with the help of the database capabilities described above):

Produce dynamic XML documents based on one or more SQL queries

Optionally transform the resulting XML document in the server or client using XSLT.

Insert XML documents into Oracle tables.

The Oracle XSQL Page Processor is available as an XSQL Servlet or as a Command Line Utility. It can be used with any Servlet capable web server.

Parse XML

The Oracle XML Parsers for Java, PL/SQL, C and C++ makes it very easy to do the following:

Parse XML into DOM or SAX

Transform XML documents using XSLT style sheets.

Transform XML documents into other XML documents.

Generate Java Classes

The Oracle XML class generator written in Java, will generate a set of Java source files based on an input DTD. The generated Java source files can then be used to construct, validate and print a XML document that is compliant to the DTD specified.