Book Title

Unit Overview

Lessons

1. XML Overview and Background
2. The XML Document
3. Schemas and Validation
4. The Document Object Model
5. XSLT and XPath
6. Web Services

Lesson 1: XML Overview and Background

Author, we need an intro. to this lesson.

Objectives

At the end of this lesson you will be able to:

- Explain what XML is and how it is useful;
- Explain what markup is and distinguish between logical and procedural markup;
- Describe the differences between XML and HTML; and
- Explain how XML relates to SGML.

Key Point

XML allows you to create "smart documents" that describe their own structure and contents, greatly increasing opportunities for automatic processing and exchange across diverse systems.

What Is XML?

Ask five people "What is XML?" and you will likely hear five different answers. Extensible Markup Language (XML), after all, is a creature of the Internet era, and has grown up at a frantic pace, driven by a range of competing visions and interests. In particular, there is widespread disagreement as to whether "XML" means just XML, or XML plus a group of related technologies. We will begin with a brief, objective definition and address the complexities as they arise.

XML is a text-based syntax for encoding electronic documents[1]. Like Hypertext Markup Language (HTML), it uses tags delimited by angle brackets to identify different portions of the text. Unlike HTML, the tags do not have predefined names. The whole point of XML is to enable organizations and individual developers to create document formats that are appropriate for a particular kind of content or application, and yet easy to: 1) exchange among different applications and platforms and 2) process with generic tools. These document formats are known as XML applications or, informally, XML dialects.

Another important difference between HTML and many XML dialects is in the type of information conveyed by the tags. For the most part, HTML describes the appearance of a document. XML is more often used to describe structure and meanings. For example, an HTML document might look like this:

```
<H3>Warnings:</H3>
<P>
  Before operating this device, make sure the surrounding area is free of
  standing water. Failure to observe this precaution may result in injury
  or death.
</P>
....
```

Whereas an XML document with the same content might look like this:

```
<section>
  <warning>
    Before operating this device, make sure the surrounding area is free
    of standing water. Failure to observe this precaution may result in
    injury or death.
  </warning>
  ....
</section>
```

We should note these two important differences: first, the XML document's <warning> element identifies the type of information it contains. Second, the XML document is explicitly divided into sections by using <section> elements, whereas in HTML the division is implied by the presence of an <H3> heading.

Using semantic and structural markup as shown in this example enables us to do two very useful things. First, we can use automatic processes to format the document based on its structure and type of content. Or, taking the idea a step further, we can create different views of the same information. Of course, for information that will be published once, in a single format (a marketing brochure, for example), this may be a waste of effort. But for documents that will be reused, modified, and exchanged among different organizations or different computing environments, the ability to format the same content in different ways can be very valuable.

XML markup can also be used to intelligently categorize and search information. Part of the original thinking behind XML was that it could improve the quality of Web searches. For example, if you wanted to read a Shakespeare play, you could search for "Shakespeare as document author;" your results would consist entirely of works by Shakespeare, omitting the many documents that discuss Shakespeare (or merely mention him in passing).

The intelligent Web-search idea has gone largely unrealized. Part of the reason is that it is extremely difficult to implement. For example, because there is no single standardized way to indicate a document's author, how do you search for that information across the whole Web? Another obstacle has been that most end users are unwilling to use complex search interfaces. But the ability to give meaningful names to information items has proven useful in a way that was largely unanticipated by XML's original creators: it has enabled the creation of a whole new generation of distributed business applications that can exchange data by means of XML.

[1] As is often the case in discussions of XML, I am using the term document in a very abstract sense, which encompasses any collection of related data that may be stored, processed, or exchanged as a distinct unit.

The XML 1.0 specification defines XML. The World Wide Web Consortium (W3C) maintains this and many other XML-related standards[2]. The name "XML" is a W3C trademark, but the specification is freely available and may be redistributed with proper attribution. Further, anyone may develop XML-based languages and XML software with no obligation to the W3C.

What Is XML *Not*?

Most importantly, XML is not a programming language[3]. Thus, you cannot expect to solve any development problem simply by using XML; you must also provide appropriate processes to effectively use XML.

XML is also not a solution to every problem in the world of computing. For example, it has become popular to use XML for software configuration files. This may be a good idea for applications with very complex configurations (a Web server, for example) and for applications that are already using XML. In other applications, it is probably a waste of effort for two reasons: because it requires adding an XML parser to the application and because many configuration files are simply series of parameter names and values, which are very easy to process in most programming languages.

Even in the domain where XML has become most prominent—data exchange in distributed business applications—do not automatically assume that XML is always the solution. A standardized data format to communicate among diverse platforms, languages, and databases provides undeniable benefits. And XML can reduce development costs by moving frequently changing business rules out of compiled code and into text-based data files. But using XML also has significant costs, including staff training and increased usage of network bandwidth and storage capacity. The benefits often, but not always, outweigh these costs.

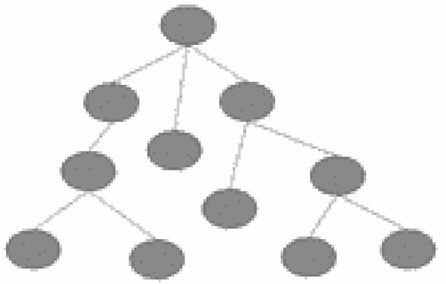
Perhaps an analogy to a physical manufacturing process will help: imagine you are a bicycle manufacturer, and one of your suppliers offers you a new alloy that is supposed to be both more flexible, lighter, and stronger than conventional materials. Buying this material will not benefit your business if buying the material is all you do. First of all, you will probably need to change your production process to use the new material. Another complicating factor is that the material may make a large difference in the quality of some parts of the bicycle (the frame, for example), but not in others (such as the pedals and handlebars).

Before making the change, you need to ask yourself which parts of your product will benefit most from the new material, and whether the improvement in quality will be enough to justify the cost of changing your production processes—and if you decide to proceed, will your customers approve? You must ask the same questions when considering developing an XML-based product or service.

The XML Information Model

It is well known that XML markup consists mainly of tags with angle brackets. Less well known, but perhaps more important is the abstract information model. XML is fundamentally designed to model hierarchical information structures—informally called trees. This basic model is illustrated in the Tree Information Structure Diagram >. At this abstract level, the document tree is said to consist of nodes. Node is simply a generic term for

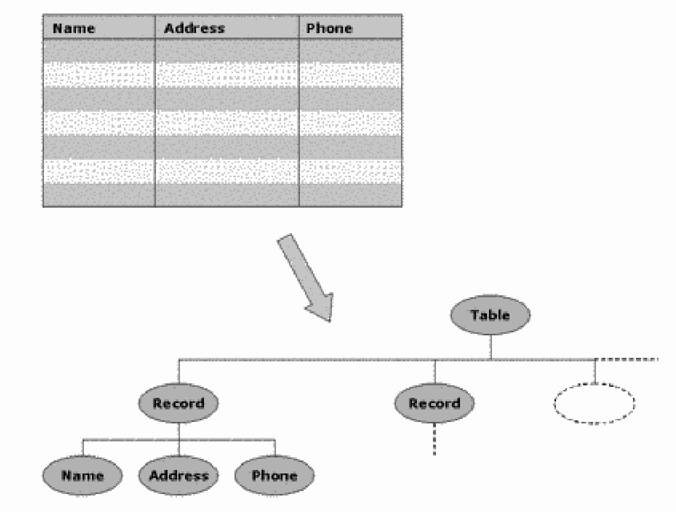
[2] I use "standards" for simplicity, but the technically correct term here is "recommendations." This is because unlike ISO or ANSI, the W3C is not legally chartered as a standards body, and its technical specifications are considered experimental. But in practice, W3C recommendations are generally seen as equivalent to standards.
[3] It is true that programming languages can be written in XML, XSLT being a case in point. But XSLT is an application of XML, and we are speaking here of XML itself.

an information item that belongs to a larger data structure, thus a node can represent a wide variety of specific items. 

Tree Information Structure

In addition to tree and node, other terms often used to describe XML models include root, branch, and leaf. Kinship terms such as parent, child, ancestor, descendant, and sibling are also common.

XML can, however, be used to represent data structures other than the tree model. But in order to represent those structures, you need to create a tree that simulates the desired structure. For example, see the XML Model of a Data Table Diagram.



XML Model of a Data Table

XML information modeling is a complex topic, and we will not discuss it in depth here. But it is important to keep the tree model in mind because many XML software architectures are based on this model.

History of XML

Author: Because this is a first level heading, please write a short paragraph introducing “History of XML”

Origins of Markup

The concept of markup originated in the typesetting world, where manuscripts were delivered to typesetters with various marks indicating how portions of the text should be typeset—they were literally "marked up." The word markup now applies to textual codes in electronic documents; these codes, too, were originally formatting instructions. This type of markup is sometimes called procedural markup, because it instructs a person or a computer to perform a specific operation that determines the appearance of the finished document. A more recent

development is logical markup, which encodes the structure and semantics of information rather than the appearance.

The following example illustrates the power of logical markup in contrast with traditional data formats. Suppose you are writing a prose document using a conventional word processor, and you come across a phrase you want to emphasize. Chances are you will italicize the phrase; in other words, you will use formatting to indicate that the phrase is special in some way. And when you save the document, the file will contain codes indicating that the text is italic.

But what do italics mean? In addition to simple emphasis, italics may be used to highlight a variety of information: book titles, foreign words, "extra" passages inserted into running text, and other items. Moreover, the usage of italics and other text styles can vary between countries and even between organizations or professional fields in a single country. And in some cases, you may not be able to use italics at all; e-mail is still primarily a plain-text medium, and some universities still require underlining for theses and dissertations.

Actually, this is seldom a problem for human readers, who can determine the significance of text styles by considering the context. But suppose your document contains valuable knowledge that may be reused and republished in several formats. Suppose you would like to find information from your document with an online search engine. In these cases, the fact that a phrase is italicized is of very little use to a computer. Although computers are more precise than humans and are very good for automating repetitive tasks, they have nowhere near the ability that humans have to interpret complex patterns of information. So we need to give them a little help.

Logical markup means using textual codes to identify the structure and semantics of information. Instead of storing the information that a phrase is italicized, we can identify it with tags such as ”emphasis,” ”book title,” ”foreign_phrase,” and so forth. , These tags enable search engines to find information based on its context, which means that logical markup greatly increases the reusability of the information.

From GML to XML

As computers became widely used in government agencies, universities, and large corporations in the 1960s, the limitations of procedural markup—and the potential for a more efficient model—became apparent. In 1969, Charles Goldfarb, Edward Mosher, and Raymond Lorie of IBM designed the first version of Generalized Markup Language, or GML, which also stands for Goldfarb, Mosher, Lorie. In the early 1980s, GML evolved into the Standard Generalized Markup Language (SGML), which became an ISO Standard in 1986.

Most early SGML users, such as aerospace manufacturers and government agencies, were organizations that maintained large document collections. SGML systems were (author: and still are?) expensive to implement and usually involved a long and laborious specification and design process. This is not to say that SGML is useless or obsolete. In its traditional market, SGML has proven its value and remains strong. But many problems that call for something that works like SGML—an open, text-based data format that represents structure and semantics—also require a tool that is simpler and less expensive to implement.

The other part of the story begins about 1990 at CERN, the Swiss physics research facility, where a young researcher named Tim Berners-Lee was creating a hypertext system to help scientists share information over the Internet. The core components of his system, HTML and Hypertext Transfer Protocol (HTTP), became the foundations of what we now know as the Web. Hypertext Markup Language (HTML) was originally a minimal language for expressing document structure: you could use it to mark up paragraphs, headings, lists, and blockquotes; include bitmap images; and that was about all. The unanticipated popularity of the Web led to the creation of many presentation-related extensions; browser vendors seeking to gain a competitive advantage in a hot marketplace added most of these in an ad hoc fashion.

As the popularity of the Web exploded in the mid-1990s, the SGML community began discussing the notion of "SGML for the Web." This discussion gave rise in 1996 to a new working group at the W3C, charged with adapting SGML to the Web. It soon became clear, however, that the complexity of full SGML was poorly suited

to the Web, whose openness allows almost anyone to obtain nearly any document, and where simple software architectures and rapid development are the norm. Thus the W3C initiative evolved from "fixing SGML" to defining a new language that would preserve the most important benefits of SGML, yet eliminate much of the complexity. The result, adopted as a W3C recommendation in February 1998, was XML 1.0. XML is often described as an "80/20" solution because like SGML, it allows you to create languages whose structure and semantics are a perfect fit for your data. Yet most optional features of the syntax have been removed, which dramatically simplifies the task of creating XML software and allows for improved performance.

Applications of XML: Present and Future

Let us take a look at some of the XML applications that have developed. (It should be noted that in this context, "application" is used in its broad sense—a usage of a technique—rather than meaning a software program.)

This survey of the field is by no means comprehensive; it is merely an attempt to suggest the range of problems to which XML is applied.

Documents

Because XML was originally conceived as a document technology, it should come as no surprise that some of the oldest and most widely used XML applications store and process documents. In fact, the two we will look at here were originally SGML applications.

DocBook

The DocBook DTD was developed for technical documentation, primarily software manuals. It provides a wide variety of elements useful to technical writers, and has a modular design for customizing local needs. Several major software and hardware vendors, including Novell, Hewlett-Packard, Sun Microsystems, and IBM, reportedly use DocBook. It is also widely used for documenting Linux and other open source software.

TEI

The Text Encoding Initiative (TEI) is a comprehensive document format for literary and linguistic texts. It is widely used among libraries, museums, and educational institutions around the world.

Content Presentation

Author, we need an introductory paragraph here. Just a sentence or two—like in “Documents” above is necessary.

XHTML

The HTML markup we all know and love is an SGML application, which has advantages and disadvantages. An advantage is that SGML allows for substantial flexibility in authoring. Depending on the DTD, for example, the end tag may be omitted on certain elements. That very flexibility, however, greatly complicates the task of creating reliable software to process the markup and is one reason browser technology has been so slow to mature.

In an effort to bring the benefits of XML standardization to the browser, the W3C created Extensible HTML (XHTML), which they intended to replace HTML. It is unclear as of this writing when XHTML will be fully implemented. It already is implemented, however, in the sense that most XHTML markup will work in most

existing browsers. However, many authoring tools require extensive modification to produce XHTML, and a strict interpretation of XHTML would require browsers to reject a great deal of markup that they currently accept.

In addition to XML simplifying (author: is simplifying) the syntax of HTML, XHTML has also introduced modularization, which is basic container elements, media objects, inline elements, and tables that are all implemented in separate modules. This is useful for the many applications that need generic formatting markup. One use, for example, is documenting other XML dialects. Rather than inventing your own XML vocabulary for documentation, you can add a single <description> element that contains XHTML paragraphs.

XForms

XForms is a W3C project intended to provide a well-structured replacement for HTML forms. HTML forms have several significant flaws. They haphazardly mix structure and data. Their structure is thus closely tied to the client interface; for example, a form designed for a desktop browser is unsuitable for a handheld device. They also provide a very limited set of controls, and have no built-in support for applying business logic.

XForms attempts to solve these problems and more by separating the structure and logic of the form from the presentation; it also includes the XForms Submit Protocol, which enables clients to submit data as XML.

SVG

Although literally hundreds of file formats are appropriate for computer graphics, all of them fit into one of two general types: vector and bitmap graphics. Until recently, Web browsers have supported only bitmap formats such as Joint Photographic Experts Group (JPEG), Graphics Interchange Format (GIF), and Portable Network Graphics (PNG). Bitmap formats are ideal for photographs and can be used for any kind of static images, but some types of graphics are better represented in vector formats. Examples include scientific and financial visualizations, animations, and interactive graphics such as image maps. Scalable Vector Graphics (SVG) is an XML-encoded vector graphics format designed for use on the Web.

SMIL

Synchronized Multimedia Interchange Language (SMIL [pronounced "smile”]) is an extreme example of presentation-oriented XML. A SMIL document describes a time-sequenced collection of multimedia objects. The media files themselves are binary objects, thus they must be stored and transmitted separately from the SMIL document. In this example, XML is acting as a kind of glue rather than directly encoding content. SMIL has not taken the world by storm, but is supported by at least one popular Web product: RealNetworks's RealPlayer.

MathML

Mathematical Markup Language (MathML) is intended to provide a complete XML vocabulary for publishing mathematics on the Web. It includes a wide variety of elements and attributes, permitting expressions to be marked-up either in a content-oriented or a presentation-oriented style. Among mainstream Web browsers, Netscape 7 and Mozilla 1.0 provide built-in support for MathML. Internet Explorer has no built-in support, but can display MathML by using one of several third-party plugins.

To learn more about MathML, visit http://www.w3.org/Math/

Communications

Author: A couple (or one) of introductory sentences needed here.

Jabber

Jabber is an XML-based protocol for real-time message exchange. The first Jabber application is an instant messaging framework comparable to ICQ or Yahoo! Instant Messaging.

To learn more about Jabber, visit http://www.jabber.org/

WAP/WML

Wireless Application Protocol (WAP) is a communication protocol for building networked applications for cellular phones and other mobile devices; WAP is the dominant standard for these applications in the North American and European markets. WAP version 1 uses the XML-based Wireless Markup Language (WML) as its message format. WML, however, has been criticized as a poor reinvention of existing XML dialectics. Therefore, the WAP Forum has adopted XHTML for messages in WAP version 2.

ECommerce

Author: We need a short intro. paragraph here.

SOAP

Simple Object Access Protocol (SOAP) is a standardized, XML-based protocol for distributed business applications. It provides a standard "envelope" that can carry any XML data as its body, or payload. SOAP was originally intended to be used as a Remote Procedure Call (RPC) mechanism, but it can also be used in a messaging style. The crucial difference between these two approaches is that SOAP-as-RPC involves exchanging objects serialized as XML, whereas the messaging approach treats the XML content as a document. Many developers have found the RPC approach too heavy for Web applications and have therefore come to prefer the messaging approach.

ebXML

Electronic Business using XML (ebXML) is intended to provide a common XML vocabulary and set of protocols for conducting business online. It is sponsored by the Organization for the Advancement of Structured Information Standards (OASIS) and United Nations (UN) Centre for Trade Facilitation and Electronic Business (UN/CEFACT).

RosettaNet

RosettaNet is a nonprofit consortium of electronics, IT, and semiconductor companies that seeks to provide standardized, XML-based message schemas and process specifications for the industries represented by its member companies.

Database

Author: We need an introductory paragraph here.

XML: DB

XML: DB (author: what does this acro. Stand for?) is an industry initiative to design and promote specifications for XML databases. The project includes XUpdate, an XML language for modifying XML documents.

XML Query

Yet another W3C initiative, Xquery, is an XPath-based language for querying performing queries on a wide variety of data sources, including documents, relational databases, and XML databases. Although XML Query syntax is different from SQL, it is functionally equivalent and could replace SQL for certain applications. XML Query 1.0 is currently in the working draft (author: ok that I made working draft lower case?) stage.

Knowledge Management

Author: We need an introductory paragraph here.

RDF

The Resource Description Framework (RDF) is a lightweight XML syntax for describing the relationships between entities. An RDF application uses triples, each consisting of a subject, an object, and a predicate. The object can be thought of as a property of the subject, and the predicate specifies a value for that property. RDF is considered a key technology for the Semantic Web (author: not sure what Semantic Web is here).

Topic Maps

XML Topic Maps (XTM) addresses some of the same problems as (author: what does RDF stand for?) RDF, but it is somewhat more complex. In XTM, every relationship is reciprocal, and relationships themselves must be defined as topics.

Dublin Core

The Dublin Core standard defines a small set of elements to be used for cataloging metadata, which is information about documents, such as an author, title, and description. The objective is to facilitate access to information by providing common semantics for information items frequently used in searches. Dublin Core is generally not used by itself, but rather in conjunction with other document types, such as RDF and Hypertext Markup Language (HTML).

How Much XML Is Enough?

If at this point you are starting to feel a bit overwhelmed, you are by no means alone. The time is long past when a single individual, even a veteran developer, could claim a comprehensive knowledge of all significant XML technologies. With all the XML initiatives under development, you are doing well if you can keep abreast of a few applications directly relevant to your work.

Tools

Before going further, we will briefly look at two tools that will be useful in learning XML. One is very familiar: Microsoft Internet Explorer. If you load an XML document into Internet Explorer, it displays the document with color coding and indentation, as shown on the XML in Internet Explorer Screen Capture:

Book Title

XML in Internet Explorer

You may also notice a number of minus (or plus) signs at the left edge of the document. By clicking on these, you can collapse and expand portions of the document, which is very helpful for analyzing complex XML structures, as shown on the Complex XML Structure Screen Capture.

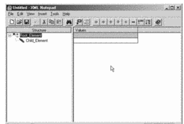
Complex XML Structure

Although you can view a raw XML document in Internet Explorer 5, for standard Extensible Stylesheet Language Transformation (XSLT) support you should use Internet Explorer 6, or obtain the (author: what does this acro stand for?) MSXML Parser. As of this writing, MSXML can be downloaded for free from:

```
http://msdn.microsoft.com/downloads/default.asp?url=/downloads/topic.asp?
URL=/MSDN-FILES/028/000/072/topic.xml
```

For easy integration with Internet Explorer, you should use MSXML version 3, not version 4.

Another free tool from Microsoft is XML Notepad. This lightweight editor makes it easy to explore the structure of existing XML documents, and is a good beginners' tool for creating simple documents. The XML Notepad Screen Capture follows.



XML Notepad

As of this writing, XML Notepad can be downloaded from:

```
http://msdn.microsoft.com/library/default.asp?
url=/library/en-us/dnxml/html/xmlpadintro.asp
```

Activities

1. Describe three important differences between HTML and XML.

2. Explain the advantages of logical markup over procedural markup. Extra: can you see any disadvantages to using logical markup?

Extended Activities

1. Find two XML applications that address similar problems. Based on information available on the Web, compare and contrast these two applications. Which one would you be more likely to use, and why? A good starting point for this activity is the XML Applications section of the CoverPages site (http://xml.coverpages.org/xmlApplications.html); it contains an enormous amount of information. Some of it will be very unclear, but with careful reading, you should be able to find two applications for this activity.

2. Create a simple document in Microsoft Word or any comparable word processor and save it in Rich Text Format (RTF). A two- or three-paragraph document with some text styling (bold, italics, or underlining) would be appropriate. Open the saved file in a text editor and examine its contents. RTF is actually a markup language, although it is very different from XML and Standard General Markup Language (SGML). Can you identify some RTF tags? Do you find the RTF language easy to understand? Do you think RTF is mainly a logical or a procedural markup language? If you were creating a new markup language, what might you do differently?

Quiz Questions

1. Which of the following are XML applications. (Choose four.)

    a. SVG

    b. Jabber

    c. Apache

    d. ebXML

    e. RDF

    a, b, d, e

2. SVG is used for:

    a. Graphics

    b. Literary texts

    c. Business-to-business data exchange

    d. Technical documentation

    e. Configuration files

    a

3. SOAP stands for:

    a. Synchronous Online Animation Process

    b. Simple Online Analysis Portal

    c. Simple Object Access Protocol

    d. Segmented Object Analysis in Parallel

    e. Synchronous Online Abstraction Protocol

    c

4. Which of the following most accurately describes the XML data model?

   a. XML documents are usually used to model table structures.

   b. XML represents sequential data.

   c. The fundamental data model of XML is tree-like, but can be used to simulate other data structures.

   d. The basic data model of XML is a multidimensional array, but it can be adapted to other structures.

   e. XML is used to model tree structures.

   c

5. Tim Berners-Lee did which of the following?

   a. Invented XML

   b. Invented the World Wide Web

   c. Invented SGML

   d. Founded Netscape

   e. Invented markup

   b

6. Which of the following organizations played a key role in creating SGML?

   a. Microsoft

   b. ISO

   c. the W3C

   d. Netscape

   e. IBM

   e

7. Which of the following is **not** true?

   a. XML is a form of logical markup.

   b. Many features were added to SGML to create XML.

   c. HTML is mainly used to specify the presentation of documents.

   d. SGML is too complex to be suitable for Web documents.

   e. Both SGML and XML enable developers to create customized document formats.

   b

8. Markup was originally used in conventional typesetting. True or False?

   T

9. XML specifies a set of tags that you must use in your applications. True or False?

   F

10. SGML is more recent than XML. True or False?

    F

11. XML is a subset of SGML. True or False?

    T

12. HTML is a subset of XML. True or False?

    F

13. HTML elements identify the type of information they contain. True or False?

    F

14. XML is more flexible than SGML. True or False?

    F

15. Anyone can develop an XML application. True or False?

    T

16. XML is a patented technology. True or False?

    F

17. The World Wide Web Consortium (W3C) maintains the XML 1.0 specification. True or False?

    T

18. The International Organization for Standardization (ISO) maintains the XML 1.0 specification. True or False?

    F

19. XML is an updated version of HTML. True or False?

    F

20. XML uses angle brackets (< and >) to delimit tags. True or False?

    T

21. XML is a programming language. True or False?

    F

Lesson 2: The XML Document

Author: Introduction needed.

Objectives

After completing this lesson you will be able to:

- Define the following terms: element, attribute, entity, processing instruction, XML declaration, and namespace;
- Explain the purpose and basic usage of XML namespaces; and
- Create a simple well-formed XML document.

Key Point

A basic XML document consists of elements, attributes, entities, and data, and must adhere to a few syntax rules called well-formed constraints.

Basic XML Syntax

An XML document consists of markup and content. Markup consists of elements, attributes, and entities. Both markup and data consist of character strings, which mean that XML cannot contain binary data. If you have ever worked with HTML, you have certainly seen elements and attributes, and perhaps entities also. But there are some significant differences between HTML and XML syntax rules.

This lesson describes the most important XML syntax rules. Documents that conform to these rules and a few other minor ones are called “well-formed documents.”

Characters

One of the major design goals of XML was to provide a fully internationalized language. Therefore, the XML character set is based on Unicode. The definition of XML characters differs slightly from Unicode, because certain control characters are excluded, as is a range of characters called “surrogates,” which are currently rather uncommon. But it is fair to say that XML uses at least all common used, printable Unicode characters.

XML documents are allowed to use local character encoding like ISO-8859-1 (for English and many European languages), Shift-JIS (for Japanese), and KOI8-R (for Russian), but XML processors are only required to support the Unicode encodings UTF-8 and UTF-16. This means that if you wish to use an uncommon encoding, your choice of XML tools may be severely limited. Fortunately for English speakers, however, ASCII is a subset of UTF-8, meaning that documents created with only ASCII characters will automatically be recognized as UTF-8.

Elements

Generally speaking, an XML element consists of a start tag and an end tag, with content between them:

```
<firstName>Bubba</firstName>
```

So far, XML is much like HTML. But XML also has a special form for empty elements:

```
<nothingHere/>
```

Elements can contain text, other elements, or a combination of the two. They can also have one or more attributes as part of the start tag.

Element names must follow a few important rules. The first character of an element name must be a letter, an underscore (_), or a colon; the remaining characters in the name must be letters, digits, or any of the characters `_`, `-`, `.`, and `:`.[4] Colons, however, are reserved for a special use, thus they should generally be avoided for ordinary element names.

Unlike HTML, all element names are case-sensitive. Element names must conform to the same upper and lower case format each time it appears. In other words, the following are all completely different element names in XML:

- `address`
- `Address`
- `ADDRESS`

Lastly, the sequence "xml" (in any combination of upper and lower case) is reserved for special element names that may be defined by the W3C, thus it should never be used to begin ordinary element names.

Attributes

An attribute associates additional information with an element. Attributes in XML are always name-value pairs, with the values in quotes:

```
version="3.48"
```

You can use either single or double quotes, provided the opening and closing quotes match on any given element. An element start tag with attributes looks like this:

```
<MusicCatalog version="3.48" updated='2002-12-05'>
```

Attribute names must be unique for any given element. Their values can contain text in almost any form, but they cannot contain elements. Attribute names follow the same rules as element names.

Entities and Character References

Many people new to XML find the concept of entities a confusing. This may be partly because of the name: in common English usage, 'entity' is essentially a fancy word for 'thing'. In the markup world, however, entity has a specific technical meaning. The term refers to a chunk of reusable data which can be referred to using a simple name. Entities come in several forms, but the type you will encounter most frequently is the **general entity**.

A general entity is a named chunk of text consisting of one or more characters. You can use it to represent characters that have special meanings in XML, like '<', or symbols that you are unable to type on the keyboard, such as the trademark symbol. You can also use it for boilerplate text such as a copyright notice or your organization's telephone number. To use an entity in an XML document, you insert an entity reference, which consists of an ampersand (&), the entity name, and a semicolon (;). Here are some examples of general entities, with entity references on the left, and values on the right:

[4] Actually, there are a few other characters that are allowed, but they are rarely used.

```
&lt;              =>    <
&trade;           =>    ™
&phone;           =>    (303) 761-8088

&disclaimer;      =>    Acme Mousetraps, Inc. disclaims any and all
                        liability for accidents, injuries, or
                        damages to you and to your friends, family,
                        pets, and property, arising in connection
                        with the use, misuse, abuse, illegal use, or
                        failure to use this product.
```

XML has five ”built-in” general entities:

- <
- >
- "
- '
- &

You must declare any other entities you wish to use in a document. The lesson entitled “Schemas and Validations” will show you how to declare entities.

Character references are used for representing foreign language characters and other special symbols. They are similar to entities and are used in the same way, but are different in two important respects: they are predefined and can always be used in any XML document, and instead of an alphabetic name, each character reference is identified by the pound symbol (#) followed by a number.

Any Unicode character can be represented using a character reference:

퐡

Character references can be written using either decimal or hexadecimal numbers, but it is common to use hexadecimal.

Comments

As in most programming languages, comments are used to include author's notes or other text that is not part of the document content. For example:

```
<!-- This is a comment. -->
```

Processing Instructions

As the name implies, processing instructions (PIs), provide directions for processing the document or parts of it. Processing instructions may appear anywhere in a document. XML processors may ignore unknown processing

instructions. An example processing instruction follows. This instruction is a common way to associate a stylesheet with an XML document, and it is recognized by most browsers:

```
<?xml-stylesheet type="text/css" href="main.css"?>
```

A PI is enclosed in the symbols '<?' and '?>'. The first word after the start delimiter is called the target of the PI; it is typically the name of a processing application or a keyword known to such an application. Following the target is additional data, which may be any string but often takes the form of attributes. You may have noticed that the XML declaration looks just like a PI. However, it is by definition not a PI. The XML declaration is considered a unique entity.

Elements vs. Attributes

Newcomers to XML often ask, "Should I use elements or attributes?" The answer is, "It depends." The traditional distinction between the two is that elements are for content, which is the primary information you want to convey to the audience, and attributes are for metadata, which is information about the information. But in reality the distinction between content and metadata is not hard-and-fast, and the following practical considerations may affect your choice:

- No element may have two attributes with the same name, but element names may be repeated.
- XML parsers must preserve the order of elements, but not of attributes.
- Element content can be structured (it can contain other elements, but attribute content cannot.)
- White space in elements can be preserved. White space in attributes may be normalized (which basically means that leading and trailing white space is removed, and any sequence of internal white space is replaced by a single space character).

Thus, for document-like applications, where human readability is a key consideration, it is recommended that you mostly use elements. For applications where readability is less important than efficient transmission (communicating between databases, for example), you may wish to use mainly attributes.

XML Document Structure

This section describes the main parts of an XML document, in order of appearance. These parts are the XML declaration, the DOCTYPE declaration, and the document body.

XML Declaration

An XML declaration identifies the document as being XML. In its simplest form, it looks like this:

```
<?xml version="1.0"?>
```

At present "1.0" is the only commonly used value for "version." Version 1.1 of the XML spec is now a Candidate Recommendation, which means that it is roughly in its final form, but it has not yet been widely implemented. There are two other optional attributes, "encoding" and "standalone." "Encoding" indicates the character encoding of the document (UTF-8 by default). "standalone" indicates whether the document depends on any external resources; its value may be "yes" or "no." An XML declaration with all three attributes looks like this:

```
<?xml version="1.0" encoding="ISO-8859-15" standalone="yes"?>
```

The XML declaration is optional, but recommended. If present, it must be at the very beginning of the document, with no preceding text or white space. Note that although the XML declaration looks exactly like a processing instruction, by definition it is not a processing instruction.

DOCTYPE Declaration

A Document Type (DOCTYPE) Declaration identifies an XML document as an instance of a certain document type as defined in a Document Type Definition (DTD). We will look more closely at DOCTYPE declarations in the lesson entitled, “Schemas and Validation,” which covers DTDs and schemas. For the moment, just be aware that an XML document may include a DOCTYPE declaration after the XML declaration and before the document body.

Document Body

The body of an XML document consists of all the elements, attributes, and text in the document. All of these must be contained in a single root element (also known as the document element).

Well-Formed Document Examples

Here is a very simple XML document:

```
<doc/>
```

This code is not very useful, of course, but it is well formed XML. So is this:

```
<?xml version="1.0" encoding="utf-8"?>
<price currency="USD">29.95</price>
```

The following code, however, is *not* well-formed. Can you see why?

```
<?xml version="1.0"?>
<name>Tweedle Dum</name>
<address>1234 Forest St.</address>
```

The example has two elements at the root level, which is illegal. To make it well formed, we have to provide a single root element, as shown below:

```
<?xml version="1.0"?>
<contact_info>
  <name>Tweedle Dum</name>
  <address>1234 Forest St.</address>
</contact_info>
```

Namespaces

One of the greatest opportunities—and at the same time one of the greatest challenges—opened up by the World Wide Web is that of using disparate sources of information to create new knowledge in ways never foreseen by their creators. In concrete terms, that can mean pulling together XML documents from different sources to create new documents. This practice, however, creates a problem: combining two documents that use the same element or attribute name with two different meanings? For example:

```
<order>
  <item>
  ....

<order>ascending</order>
```

In the first example, “order” means a request for goods or services; in the second, it means a sequence. And it is entirely possible that both could end up in the same composite document.

XML namespaces were created to prevent this sort of name collision from occurring. With namespaces, instead of the ambiguous ”order,” you can use qualified names, such as in this example:

```
<biz:order>

<sort:order>
```

But how are the prefixes determined? And is it possible that two documents could coincidentally use the same prefixes, leading to the same problem all over again? The answer is that every XML namespace is defined in terms of a URI. The following example is an example of a namespace declaration:

```
<some_element xmlns:biz="http://www.example.com/bizdocs">
```

The prefix “xmlns” denotes a namespace definition, and whatever follows the colon is the namespace prefix. Because each document using namespaces must declare them, the prefix can be anything you choose. The real value of the namespace is the URI. Although it can be any type of URI, it is most common to see HTTP URLs used to define namespaces. This is a convenient way to ensure uniqueness, because in principle only the owner of a domain name can use it in URIs.

The URI is not required to refer to any existing resource. This point has confused beginners and has been very controversial among XML developers. It means that if you were to enter a namespace URI into your browser, you *might* find an XML schema or stylesheet, or some other information related to the namespace, but only if the owner of that domain has decided to make such a resource available. Owners are not required to do so. Therefore, applications using namespaces should simply treat the URI as a unique identifier, with no expectation of using it to retrieve a resource from the network.

Activities

1. XML markup includes which of the following? (Choose three.)

    a. Elements

    b. Objects

    c. Functions

    d. Attributes

    e. Entities

A, D, E

2. Which of the following entities is *not* predefined for XML?

Book Title

a. "

b. &

c. <

d.

e. '

D

3. The script below is an example of ____________________ :

```
<dinosaur species="Diplodocus"/>
```

a. PCDATA

b. An entity reference

c. A namespace declaration

d. A comment

e. An empty element

E

4. The script below is an example of __________________ :

```
&percent;
```

a. An attribute

b. An entity reference

c. A comment

d. A namespace prefix

e. A character reference

B

5. The following is an example of ________________________:

```
xmlns:zip="http://zip.net/ns/"
```

a. An attribute.

b. A namespace prefix.

c. A namespace declaration.

d. An entity reference.

e. A URN.

C

6. DTD stands for_____________.

    a. Document Type Declaration
    b. Dynamic Type Definition
    c. Document Type Definition
    d. (a) and (c)
    e. (b) and (c)

E

7. A namespace URI is best thought of as what?

    a. A processing instruction
    b. A unique identifying string
    c. A link to a resource on the Internet
    d. A special type of entity reference

    B

8. All XML processors must support which character encodings?

    a. ISO-8859-1 and UTF-8
    b. ASCII, Shift-JIS, and ISO-8859-1
    c. All ISO-8859-* encodings
    d. GB2312, Big5, and KOI8-R

    UTF-8 and UTF-16

9. What is the XML character set?

    a. 8-bit ASCII
    b. Identical to Unicode
    c. Almost identical to Unicode
    d. A subset of ISO-Latin-1
    e. A combination of all Western-language encodings

    C

10. Which of the following is NOT a use for entities?

    a. To represent symbols that you cannot type or display on your platform
    b. To include boilerplate text in a document
    c. To include external files
    d. To validate an XML document
    e. Any of the above

D

11. Which of the following statements is true?

   a. Elements may contain child elements, but attributes may not.

   b. Whitespace is significant in elements, but not in attributes.

   c. Element and attribute names must be quoted.

   d. Element names must be unique in any given context; attribute names may be repeated.

   e. Element names are case-sensitive, but attribute names are not.

   A

12. An XML declaration is not a processing instruction. True or False?

   T

13. An XML document may have more than one root element. True or False?

   F

14. Attributes can contain elements. True or False?

   F

15. A comment may be placed anywhere in an XML document except before the XML declaration. True or False?

   T

16. A namespace URI must refer to an actual resource on the Internet. True or False?

   F

17. A DTD must be contained in a file, separate from the document. True or False?

   F

18. A DTD must be contained within the XML document that uses it. True or False?

   F

19. DTD declarations can be contained in an XML document and in an external DTD file. True or False?

   T

20. An XML document must have an XML declaration. True or False?

   F

21. If an XML document has an XML declaration, it must occur at the very beginning of the document. True or False?

   T

22. "Version" is the only required attribute of an XML declaration. True or False?

   T

23. XML elements may overlap each other. True or False?

    F

24. XML elements must be nested or in sequence. True or False?

    T

25. Attributes can contain elements. True or False?

F

Extended Activities

1. Identify structural and semantic information items in an existing document. Almost any type of document can be used for this exercise, but you should try to use something with a moderately complex and varied structure. Textbooks and manuals are appropriate, as are cookbooks, legal documents, and magazine articles with several sections. Try to identify and name at least five different information items that you think are characteristic of the type of document your sample represents.

   As much as possible, the names you give the information items should reflect their semantic and structural nature, not their presentation.

2. Create an e-mail address book in XML. If you wish to base this document on your actual address book, it is helpful to know that most e-mail programs provide a way to export addresses in plain text form. Microsoft Outlook and Outlook Express, for example, can export address books as CSV files. After you have performed the export, you can open the resulting file in a text editor and add XML markup.

When you have created your XML document, test it by loading it into Internet Explorer. If the document is not well formed, the browser will display an error message. If you have any errors, try to determine what is causing them, and edit your document until the errors disappear.

Lesson 3: Schemas and Validation

Author: We need an introductory paragraph or two here.

Objectives

After finishing this lesson you will be able to:

- Explain the purpose of a schema for XML.
- Describe important characteristics of DTDs and XSD schemas.
- Discuss the advantages and disadvantages of DTDs and XSD schemas.

Key Point

Well-formed XML allows almost any data structure imaginable, but for interoperability it is important to constrain XML structures in mutually agreeable ways. Schemas provide a way to document and enforce rules for XML structures.

Why Well-Formed XML Is Not Enough

We have seen that XML offers tremendous flexibility in developing Web applications. You can use it to represent almost any kind of data imaginable; even if your data is in binary form, you can use XML to describe it. But this flexibility is a two-edged sword: the drawback is that XML markup has no built-in semantics. Thus, without some form of coordination, the communication of two businesses exchanging XML purchase orders might fail to communicate because their purchase orders could use completely different names to describe their content. For example, if your XML-based e-commerce system is programmed to expect purchase orders in the form:

```
<purchaseorder>
  ...
</purchaseorder>
```

and your customer sends you a purchase order like this:

```
<PO>
  ...
</PO>
```

or even like this:

```
<PURCHASEORDER>
  ...
</PURCHASEORDER>
```

your system will be unable to interpret the incoming document.

Thus, it is critically important to agree upon a single markup vocabulary, or at least to be consistent in the markup you use, and to be able to communicate precisely what elements and attributes are present in your documents.

One important tool for communicating and enforcing document structure is a schema. A schema is a formal set of rules for XML documents, written in one of several schema languages. An XML document that conforms to a

schema is said to be valid. Although it is often possible to determine validity by visually inspecting a document, this is a laborious and error-prone process. Instead, it is recommended that you validate the document with a validating parser, which checks both whether the document is well-formed and whether it conforms to a particular schema[5]

In this chapter we will examine two major schema languages: the Document Type Definition (DTD) and the W3C's XML Schema language (sometimes called XSD, for XML Schema Definition). Although several other schema languages exist, these two are the most widely used.

DTDs

DTDs, part of the XML 1.0 specification, are inherited from SGML. All validating parsers are required to recognize DTDs and are supported by a wide variety of XML tools. They have very compact syntax.

A DTD consists of a series of markup declarations. There are several types of markup declarations, but the most commonly used are element, attribute, and entity declarations. We will examine each.

Element Declarations

This is how an element declaration looks:

```
<!ELEMENT Book  (BookInfo, Chapter+, Appendix*)>
```

Like all declarations in a DTD, the element declaration begins with the characters ”<!.”The keyword ELEMENT indicates an element declaration. Following that, with leading and trailing white space, is the name of the element being declared. The remainder of the element declaration is the content model, which specifies the type and order of elements or other content that is allowed within the declared element. The following special characters are used in content models:

Special Characters Used in Content Models

occurrence	?	zero or one
	*	zero or more
	+	one or more
ordering	|	choice
	,	sequence

In addition to element names, three keywords can be used in content models: #PCDATA, EMPTY, and ANY. #PCDATA stands for parsed character data, which means textual content. Here is an example of an element declaration using #PCDATA:

```
<!ELEMENT paragraph   (#PCDATA)>
```

You can also use #PCDATA in combination with element names:

```
<!ELEMENT stuff     (#PCDATA|foo|bar|spam|junk)*>
```

[5] According to the strict definition of the term, a validating parser is one that checks for validity against a DTD. But schema validators for other languages do essentially the same thing, so in an informal discussion it is reasonable to lump them together.

These declarations are examples of a mixed content model. The second example says that a “stuff” element may contain text interspersed with any number of ”foo,” “bar,” “spam,” and “junk” elements in any order. Mixed content is quite common in HTML, when a `<P>` element contains text mixed with a variety of inline elements, such as `<B>`, `<I>`, and `<SPAN>`. Note the important limitation on mixed content models: as shown in the example, the content must be a repeatable choice group (separated with ”|” and ending with ”*”); also, the #PCDATA keyword must be the first item in the group.

Some experts recommend avoiding mixed content if possible, because it makes processing considerably more complex than either element-only or text-only content.

Attribute Declarations

Attributes in a DTD are declared with reference to particular element names. For example:

```
<!ATTLIST paragraph
    id          ID                 #REQUIRED
    role        NMTOKEN            #IMPLIED
    importance  (high|normal|low)  "normal"
>
```

The declaration begins with the keyword ATTLIST, which stands for "Attribute List,” followed by the name of the element to which the declaration applies. The remainder of the declaration consists of one or more attribute definitions. Each definition consists of the attribute name, its type, and a default declaration. There are several predefined attribute types, including the ID and NMTOKEN types shown here, and CDATA, the most common. CDATA stands for arbitrary textual data; it is essentially the same as #PCDATA in elements (there is a slight difference, but it is unimportant except in very unusual cases).

An attribute can also have an enumerated type, which means that its value can be one of several choices specified in the declaration. The following example defines an enumerated type: (high|normal|low)

The third part of the attribute definition specifies whether the attribute is required and its default value, if any. #REQUIRED means the attribute must be present and has no default value. #IMPLIED means the attribute is optional and has no default value. A quoted string such as "normal" specifies a default value. A fourth option, not shown in the example, is the keyword #FIXED followed by a default value. This means that the attribute can only have the specified value. Attributes with default values (including #FIXED attributes) do not need to be explicitly included in documents; if one is omitted, an XML processor assumes it is present and has its default value.

Entity Declarations

Although the concept of entities may be difficult, defining specific entities is not. An entity declaration consists simply of the keyword ENTITY, an entity name, and a value in quotes, as shown in the following example:

```
<!ENTITY  copyright   "Copyright (c) 2002 by Risky Schemes, Inc.
                       All rights reserved.">
```

If your DTD includes the above definition, you can use the entity reference ©right; in documents. The XML parser will insert the entity's value in place of the reference.

DTDs and Documents

An XML DOCTYPE declaration is used to associate a document with a DTD. It looks like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   SYSTEM "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

The declaration is contained in the delimiters ”<!“ and ”>“ and begins with the `DOCTYPE` keyword. The next word–`html` in this case— is the name of the document type, which must be the same as the document's root element. Following this is an optional public identifier and a required system identifier. The public identifier allows an XML processor to find the DTD by means of a local catalog that associates such identifiers with DTD files. The system identifier is a reference to an actual location for the DTD file; it may point to a local file, but if you plan to exchange the document with other organizations, it is smart to use a publicly accessible URL, as in the example.

A DTD can consist of internal and external subsets. The external subset is a separate file containing markup declarations. The internal subset consists of one or more markup declarations contained in the DOCTYPE declaration itself. The above DOCTYPE declaration has only an external subset. Here is an example with only an internal subset.

```
<!DOCTYPE foo [
  <!ENTITY disclaimer
    "The manufacturer assumes no liability for damage
     caused by the use or misuse of this product">
]>
```

Your document may have either an internal or an external subset, or both. Although you can define a complete DTD in an internal subset, an external subset is preferable in most cases because it can be reused for any number of documents. The internal subset can be used to modify declarations from the external subset, or to define entities, even in a non-validated document.

Shortcomings of DTDs

DTDs have a long history and are widely supported among XML applications. But they have several limitations that have become evident as XML has encountered the real world:

- DTDs have no support for datatypes, such as numeric types and Booleans, commonly used in programming and databases. Demand for such datatypes has grown with the popularity of data-oriented uses of XML.
- DTDs are incompatible with XML Namespaces. Or, more precisely, it is possible to use names of the form `prefix:local_name` with a DTD, but all such names must be predefined in their entirety (including the prefix) in the DTD. As always, a document can only be validated as a whole against a single DTD. Thus, it is impossible to use namespaces as intended: as a means to combine document components conforming to different schemas.
- DTDs do not allow you to define different elements with the same name for different contexts.
- DTDs use a non-XML syntax, requiring completely different parsing tools from XML documents.

Because of these limitations, a widespread demand has arisen for alternative schema languages.

W3C XML Schema

The XML Schema language, sometimes abbreviated as XSD (which stands for XML Schema Description language, its original name) or WXS (which stands for W3C XML Schema), is the W3C's official answer to the demand for new schema languages. It addresses the problems mentioned earlier, and also fits well with object-

oriented programming languages. After a long development cycle and many heated debates, XSD became a W3C recommendation in May 2001, and is being promoted as a foundation for several other W3C technologies.

Simple and Complex Types

The notion of types is fundamental to working with XSD. The specification defines two categories of types: simple and complex. Simple types represent conventional data types such as integer, Boolean, date, and string. Complex types represent structures of XML elements, attributes, and data; they are roughly comparable to content models in DTDs, except that they can be defined independently of any particular element. XSD elements are defined in terms of complex or simple types, and attributes are defined in terms of simple types (as with DTDs, attributes cannot contain elements, therefore they cannot be defined as complex types). The XSD specification provides a variety of simple types, but schema designers must create complex types for each schema. They can also create new simple types using the built-in types as a base.

Structure of an XSD Schema

An XSD schema consists of a `<schema>` root element that typically contains one or more element, attribute, and simple or complex type declarations. For example,

```
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ....
</xsd:schema>
```

Declaring Elements

An easy way to get started with XSD is to look at an example. Imagine an online music catalog with a structure like this:

```
<MusicCatalog>
  <Song>
    <Title>Get Up, Stand Up</Title>
    <Artist>Bob Marley and the Wailers</Artist>
    <Genre>Reggae</Genre>
    <Downloads>
      ....
    </Downloads>
  </Song>
  ....
</MusicCatalog>
```

A DTD declaration for the `Song` element would be:

```
<!ELEMENT Song (Title, Artist, Genre+, Downloads)>
```

Note that the order of elements in the content model is fixed, and also that multiple `Genre` elements are allowed. Here is an equivalent XML Schema declaration:

```
<xsd:element name="Song">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Title">
        ....
      </xsd:element>
      <xsd:element name="Artist">
        ....
      </xsd:element>
      <xsd:element name="Genre" maxOccurs="unbounded">
        ....
      </xsd:element>
      <xsd:element name="Downloads">
        ....
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

There are several important points to note about this declaration. First of all, the type definition for this element is anonymous. Thus, it applies only to this element and cannot be used or referenced in any other context. Therefore, this usage of `complexType` is almost identical to the parentheses in a DTD content model. Second, the sub-element declarations are contained in an `xs:sequence` element, which is equivalent to a comma-separated list of element names in a DTD.

Finally, the `Genre` declaration has a `maxOccurs` attribute. Where DTDs use the symbols `?`, `*`, and `+`, XSD uses the attributes `minOccurs` (minimum occurrences) and `maxOccurs` (maximum occurrences). This provides for a great deal of flexibility; the symbols `?`, `*`, and `+` are adequate for the vast majority of cases. But suppose you needed to specify that a given element must occur at least two and at most five times in a given spot. In a DTD you would have to write:

```
<!ELEMENT foo  (bar,bar,bar?,bar?,bar?)>
```

This is awkward at best, and quite impractical for larger numbers. In an XSD Schema, however, you can specify

```
<xsd:element name="bar" minOccurs="2" maxOccurs="5"/>
```

Both `minOccurs` and `maxOccurs` have default values of 1. Thus, if both are omitted, it means that exactly one occurrence of the element is required.

One subtle, but important difference between the previous example and the DTD equivalent is that the sub-elements of `Song`—`Title`, `Artist`, `Genre`, and `Downloads`—are all declared locally, thus they can only be used within the `Song` element. In a DTD, all element declarations are global.

Elements with Simple Types

Let us look more closely at one of the subelement declarations. The first child of `Song` is `Title`. In most cases, the title of a creative work (or anything else) is going to be a simple string. Therefore, the DTD declaration for `Title` would probably be:

```
<!ELEMENT Title  (#PCDATA)>
```

The equivalent in XSD is:

Book Title

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

The name `xsd:string` refers to the built-in string data type. The namespace prefix is very important: the unprefixed name `string` would refer to a data type declared in the current schema. If no such declaration were found, the schema processor would raise an error.

Declaring Attributes

In contrast to DTDs, attributes in XSD are declared in much the same way as elements. If we want to add an `id` attribute to the `Song` element, we can add an `<xs:attribute/>` element, as shown here:

```
<xsd:element name="Song">
  <xsd:attribute name="id" type="xsd:ID"/>
  <xsd:complexType>
    ....
  </xsd:complexType>
</xsd:element>
```

Note that the data type of the attribute is `xsd:ID`. For backwards compatibility with DTDs, all the DTD attribute types are included in the XML Schema Datatypes specification.

Declaring Simple Types

XSD provides a large selection of predefined data types, but applications may need additional constraints. You can create simple types in any of several ways.Declaring Complex Types

We have seen the `xsd:complexType` element used to define anonymous types, which are roughly equivalent to DTD content models. But you can also define a named complex type, which can be referenced in the type attribute of an element reference.

Alternative Schema Languages

Although XSD solves some important problems, it has been criticized for its complexity. In particular, the data types are not necessarily appropriate for all applications, but XSD requires their use anyway. It is also unclear that the schema inheritance mechanism is an effective solution. There are also some important problems that XSD does not solve. For example, some applications benefit from co-occurrence constraints--rules that allow or prohibit certain element or attribute values based on the values of other elements or attributes. For all of these reasons, XSD will probably never be the universal solution that its supporters may have hoped for.

RELAX NG

Like XSD, RELAX NG is written in XML syntax, but it is significantly less complex. One important difference is that elements and attributes are declared directly, using `<element>` and `<attribute>` elements, rather than requiring complex type definitions. RELAX NG also has no built-in simple data types, but instead allows developers to use data type libraries of their own choice, including W3C Schema data types.

Schematron

Schematron uses a significantly different approach from the other schema languages we have discussed. Unlike the other languages, which are intended for describing complete XML data models, Schematron is a very flexible mechanism for defining validation rules. This means, for example, that it can be used to validate partial documents; also, unlike most schema languages, Schematron can validate documents using co-occurence constraints. Schematron is implemented in XSLT.

Quiz Questions

3. Which of the following statements are differences between DTDs and XSD schemas? (Choose four.)

    f. XSD schemas are written in XML.

    g. It is impossible to define attributes in an XSD schema.

    h. DTDs are incompatible with Namespaces.

    i. XSD supports numeric data types.

    j. XSD allows local element declarations.

    A, C, D, E

4. Which one of the following statements is true?

    k. A DTD must be contained within an XML document.

    l. A DTD must be stored as a self-contained file, separate from any XML document.

    m. If you use DTDs, each document must have its own DTD.

    n. A DTD file must contain a reference to each document that is to be validated against it.

    o. A DTD may be entirely included in a document, entirely stored in a separate file, or be a combination of internal and external parts.

    E

5. What is an XML parser that checks whether a document conforms to a given DTD called?

    p. A validating parser.

    q. A strict parser.

    r. A DTD parser

    s. A well-formed parser.

    t. A compliant parser.

    A

6. A content model specifies_________________ .

    u. The attributes associated with an element.

    v. The elements that may or must occur as children of an element.

w. The data type of an element.

x. All of the above.

y. None of the above.

B

7. Which of the following characters is NOT used in a DTD content model?

z. | (pipe)

aa. < (left angle bracket)

bb. * (asterisk)

cc. , (comma)

dd.) (close parenthesis)

B

8. In a DTD, how do you declare an attribute that will serve as a unique identifier for its parent element?

ee. Name the attribute `id`.

ff. Declare the attribute as having a fixed value.

gg. Declare the attribute with type `ID`.

hh. Use the `#IMPLIED` keyword.

ii. You cannot declare a unique identifier in a DTD.

C

9. Given the following element declaration, what can you say with certainty about the element it defines? (Choose two.)

```
<!ELEMENT purchaseOrder  (customer, date, lineItem+, comments?)>
```

jj. This is the root element of a document.

kk. The element has no attributes.

ll. This element must contain at least one child named `lineItem`.

mm. A child element called `comments` is required.

nn. This element may not directly contain textual content.

C, E

10. Given the following (partial) element declaration, what can you say with certainty about the element it defines? (Choose two.)

```
<xsd:element name="memberRecord">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      ....
```

oo. This is the root element of a document.

pp. The `memberRecord` element has at least one required child element.

qq. The `name` element may not have any child elements.

rr. The `memberRecord` element has at least one attribute.

ss. None of the above.

B, C

11. An anonymous `complexType` definition occurring within an XSD element declaration is roughly equivalent to ________________.

tt. A content model in a DTD.

uu. An ATTLIST declaration in a DTD.

vv. A Doctype Declaration in an XML document.

ww. An ENTITY declaration in a DTD.

xx. None of the above.

A

12. What is the DTD equivalent of `type="xsd:string"`?

yy. CDATA for elements or NMTOKEN for attributes.

zz. #PCDATA for elements or CDATA for attributes.

aaa. <!ENTITY ....>

bbb. ANY

ccc. None of the above.

B

13. An ATTLIST declaration is independent of any particular element. True or False?

F

14. A Doctype Declaration in an XML document identifies both the root element name and a DTD to which the document should conform. True or False?

T

15. If a DTD declares an attribute named `id`, the value of that attribute must be unique within any given document. True or False?

16. In an XSD schema, an element may be defined in terms of a simple or complex data type. True or False?

T

17. It is possible to define entities in an XSD schema. True or False?

    F

18. In an XSD schema, an attribute may have a complex type. True or False?

    F

19. In an XSD schema, a named complex type defined at the global level may be used as the type for one or more elements defined in the schema. True or False?

    T

20. A user-defined data type in XSD must be based, directly or indirectly, on one of the built-in data types. True or False?

    T

21. For backwards compatibility, XSD supports the data types used for attributes in DTDs. True or False?

    T

22. DTDs support numeric data types. True or False?

    F

23. XSD schemas support numeric data types. True or False?

T

Lesson 4: The Document Object Model

Objectives

After completing this lesson you will be able to:

- Explain the purpose of the W3C DOM.
- Name the principal objects defined by the DOM specification, and how they relate to each other.
- Describe important methods for programming with the DOM.
- Discuss the limitations of the standard DOM API.

Key Point

The Document Object Model provides programmers with a versatile and easily navigable view of XML data structures.

.

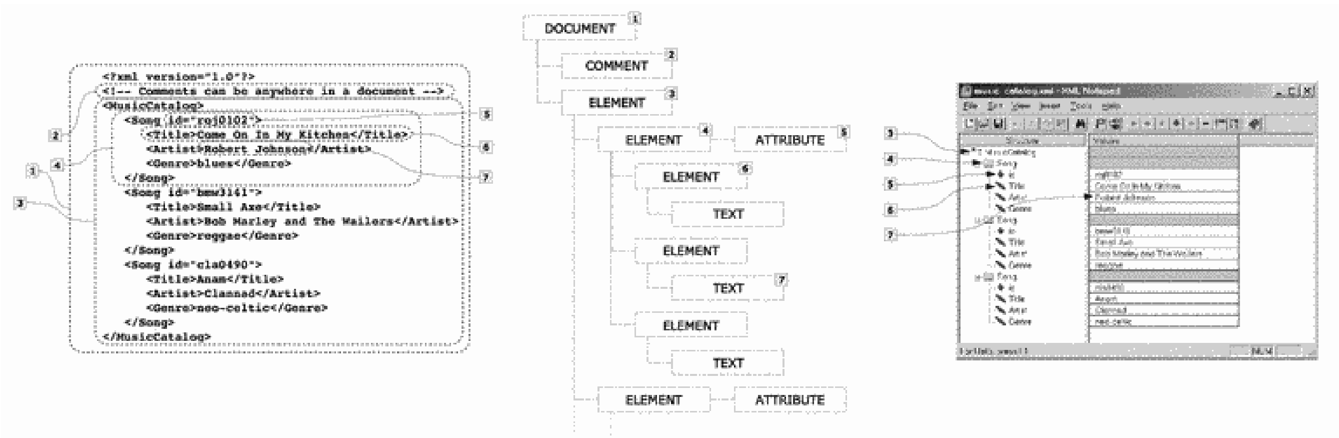
What Is a DOM?

An object model is a way of conceptualizing a set of data as a structured collection of objects; it gives developers a reliable and understandable interface for manipulating the data. A document object model (DOM), therefore, is a model for the components of a document. The W3C DOM is a standardized object model for HTML and XML. It is not the only document object model(another is Microsoft Word object model, for example), but when the name is capitalized or abbreviated as DOM, it usually refers to the W3C DOM.

The DOM as we know it grew out of the need for browsers to provide a standardized scripting interface. Because XML was originally conceived as a browser-based technology, it seemed only natural to extend DOM to cover XML as well as HTML. The DOM has since been extended to become a general-purpose application program interface (API) for XML programming, and most current XML parsers support either the W3C DOM or something similar. The need for a standardized API in custom-built server applications is questionable, but the DOM's tree-like interface has become familiar to many developers, and it is useful for a wide variety of applications.

DOM Objects

The DOM Interface Diagram shows a document object and its relationship to the XML source. Also shown is the same document in Microsoft's XML Notepad, which presents a graphical view of the DOM tree (note that XML Notepad does not show all node types, only the most common ones).



DOM Interface Diagram

As the above diagram suggests, the DOM attempts to provide a very thorough model of an XML document, with different node types representing elements, attributes, text, and other items of interest. In all there are 12 node types.   The DOM Node Types Table summarizes their names and outstanding characteristics.

DOM Node Types

Type	Name	Value	Comments
Document	#document	null	A top-level container for the entire document. Note that a document node is abstract and does not correspond to any structure in the document.
Element	element name	null	
Attribute	attribute name	attribute value	
Text	#text	textual content	Represents a contiguous extent of textual (PCDATA) content.
Entity	entity name	null	Represents an entity definition.
Entity Reference	name of entity referenced	null	
Comment	#comment	content of comment	
Processing Instruction	target	entire content except target	
CDATA Section	#cdata-section	textual content	A CDATA section is output exactly as is. If the content includes markup characters (e.g. '<','>','&'), they are treated as literal text instead of markup.
Document Type	document type name	null	Represents a Document Type Declaration.
Notation	notation name	null	Notations are a way of representing content types of unparsable files (such as images and other binary file types).
Document Fragment	#document-fragment	null	Part of a document tree, usually used in the course of major structural transformation.

Methods

DOMImplementation.createDocument

DOM applications generally rely on a `DOMImplementation` object. Although the means of creating this object is beyond the scope of the DOM specification, after it is obtained it is generally possible to write a complete program using standardized methods. At any rate, a DOM object (representing a single document) can be obtained with the `createDocument()` method. This method has three arguments: a namespace URI, a root element name, and a document type. However, they may all be passed null values, and the root element name can be assigned later.

Document.createElement, Document.createAttribute, Document.createTextNode

As their names indicate, these three methods are used for creating, respectively, new elements, attributes, and text nodes. Similar methods also exist for creating other types of nodes. It is important to note that the new nodes created are "floating"—they are not placed at any particular spot in the document.

Node.appendChild

When you obtain a node by creating it with one of the above methods or by some other means, you use the `appendChild()` method to attach it to a particular point in the document tree. As indicated by the name, the node becomes a child of the node on which the method is invoked. For example, if you have variables `a`, representing an existing element, and `b` representing a newly created element, the method call

```
a.appendChild(b)
```

makes `b` a child of `a`.

Node.getNodeName

This method returns the name of any node. Whether or not this is a useful value depends on the node type. For example, for an element node this method returns the element name, and for a text node it always returns the generic identifier `#text`.

Node.getNodeValue

This method returns either the string value of a node or null. It is important to note that the value of an element is always null. If the element has textual content, it is contained in one or more text nodes that are children of the element node, rather than in the element node itself.

Node.getChildNodes

This method returns all child nodes of a given node in the form of a node list (an ordered collection of nodes). Most DOM implementations provide some convenient way to iterate over a node list.

Element.getElementsByTagName

This method returns a node list containing all descendant elements with a particular name. Note that an equivalent `getElementsByTagName()` method is also defined for the Document object.

Element.getAttribute

Given the name of an attribute, **`getAttribute()` returns the value of the named attribute as a string.**

Node Types

Document	A top-level container for the entire document. Note that a document node is abstract and does not correspond to any structure in the document.
Element	
Attribute	
Text	Represents a contiguous extent of textual (PCDATA) content.
Entity	Represents an entity definition.
Entity Reference	
Comment	
Processing Instruction	
CDATA Section	A CDATA section is output exactly as is. If the content includes markup characters (e.g. “<”,”>”,”&”), they are treated as literal text instead of markup.
Document Type	Represents a Document Type Declaration.
Notation	Notations are a way of representing content types of unparsable files (such as images and other binary file types).
Document Fragment	Part of a document tree, usually used in the course of major structural transformation.

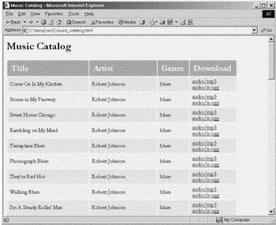
Extensions

If you have been reading carefully, you may have noticed that DOM Levels 1 and 2 are missing some features that many programmers find essential: in particular, there is no straightforward way to load or save an XML document in DOM Levels 1 and 2. In addition, Levels 1 and 2 lack convenient means for performing the common task of obtaining and manipulating a collection of nodes. For these and other reasons, almost all DOM implementations include extensions to provide convenient functionality for programmers. In the following example you will see some of the extensions provided by Microsoft's MSXML, which provides XML functionality for Internet Explorer, and by Mozilla and Netscape 7.

XML DOM Programming: A JavaScript Example

We will now look at the Example of XML DOM Programming With JavaScript Screen Capture. Important DOM features illustrated in this example include the following:

- Creating new nodes and attaching them to the DOM tree
- Testing node types
- Accessing elements by name
- Navigating the document structure



Example of XML DOM Programming With JavaScript

In theory, it should be possible to directly load an XML document into the browser and process it with JavaScript. The reality is problematic; in this case, there is no standardized way to associate a script with an XML document. The most portable way to display an XML document with JavaScript is to do it indirectlyby creating an HTML wrapper that runs the script on loading.

The following example will illustrate one technique for displaying XML content in an HTML document. The procedure involves the following six steps:

24. Create an empty XML DOM object.

25. Load an XML document into the DOM.

26. Build a new subtree of HTML elements. Note that:

    - This subtree exists within the DOM object, but by default is not attached to any particular point in the main tree.

    - Although the elements have HTML names, with the main tree they are technically still considered XML elements.

27. Replace the original root element with the root of the new subtree. This action causes the next step to produce only elements with HTML names.

28. Retrieve the contents of the DOM object as a string.

29. Insert the string into the body of the main HTML document.

Note that steps 3 and 4, which represent the bulk of processing, use standard DOM methods that are implemented in both Internet Explorer and Mozilla. Step 6 also uses standard DOM methods, but the other steps differ between the two browsers.

Creating a DOM Object

The W3C DOM specification defines an object called `DOMImplementation`, which has a method called `createDocument()`. This object and method are implemented in Mozilla, so we can use the following code to create a DOM object:

```
document.implementation.createDocument("","",null);
```
[6]

Internet Explorer, however, uses an ActiveX object provided by MSXML, the Microsoft XML Core Services library:

```
new ActiveXObject("MSXML2.DOMDocument");
```

[6] The arguments to the function are (namespaceURI, qualifiedName, doctype). The first two are the namespace and name of the root element. The name can be omitted and set later.

Thus, we need a function that determines which method to use based on the browser type. For example:

```
function createDOM() {
  var dom = null;
  if (winMSIE5) {
    dom = new ActiveXObject("MSXML2.DOMDocument");
  } else if (standardDOM) {
    dom = document.implementation.createDocument("","",null);
  } else {
    // Error message.
  }
  return dom;
}
```

Note the variables `winMSIE5` and `standardDOM`. Because several functions will behave differently depending on the browser, it is a good idea to apply the browser detection functions once and capture their results in these global variables.

Loading the XML Document

When loading an XML document, we again find differences between Internet Explorer and Mozilla. Although both browsers support the `Document.load()` method, which populates a DOM tree from the contents of an XML document, they have different ways of invoking further processing after the document is fully loaded. Internet Explorer supports the convenience of directly calling the next method in sequence after loading the document. In Mozilla, you need to use the `addEventListener()` method. As shown in the following script, this method notifies the XML processor that a further process should be triggered when the document is loaded:

```
function loadXML() {
  var xmlDOM = createDOM();
  if (standardDOM) {
    xmlDOM.addEventListener("load",digestDOMMoz,false);
  }

  xmlDOM.async = true;
  xmlDOM.load("music_catalog.xml");
  if (winMSIE5) {
    digestDOM(xmlDOM);
  }
}
```

Note that a different function is called for each browser. The `digestDOMMoz()` method receives an event object as its argument, and calls `digestDOM()`. See the next section for more detail.

Building the Tree

Now that we have created a DOM object and loaded it with XML, we can proceed to build a tree of "HTML" elements (remember, these are actually XML elements, but they have HTML names):

```
function digestDOM(xmlDOM) {
  var catalogNode = xmlDOM.documentElement;
  var tableElt = makeTable(catalogNode,xmlDOM);
  ....
}
```

The function starts by capturing the document element, which is the familiar `<MusicCatalog>`, in the variable `catalogNode`. Then we call the `makeTable` function, which uses the data contained in `catalogNode` to construct a tree of nodes representing an HTML table.

These steps are not quite sufficient, however, for a cross-browser application. We need a separate function, as follows, to act as an event listener in Mozilla:

```
function digestDOMMoz(event) {
  digestDOM(event.currentTarget);
}
```

This separate function simply receives an event object and calls `digestDOM()`. But what is `currentTarget`? The answer can be found by referring to the code in `loadXML()`:

```
xmlDOM.addEventListener("load",digestDOMMoz,false);
```

The target of the event is simply whichever object the `addEventListener()` method was called on, which in this case was the `xmlDOM` object.

Because we are processing a moderately complex document, we will need a few more functions to build a complete tree. Some of these functions are as follows:

```
function makeTable(catalogNode, docNode) {
  var songs = docNode.getElementsByTagName("Song");
  var tableElt = docNode.createElement("table");
  tableElt.appendChild(makeHeaderRow(columnNames,docNode));

  var odd = true;
  for (var i = 0; i < songs.length; i++) {
    tableElt.appendChild(makeRow(songs[i],docNode,odd));
    odd = (! odd);
  }
  return tableElt;
}
```

Replacing the Root Element

After the previous steps are completed, we will have a tree of elements with HTML names. However, it is unusable because it is not attached to the main DOM object. If we tried to access the contents of the DOM in their current state, we would retrieve a copy of the original XML document. To use the content in its transformed state, we need to replace the root element of the DOM (which contains all the original content) with the root element of the new tree. Here is how that replacement is performed:

```
function digestDOM(xmlDOM) {
  var catalogNode = xmlDOM.documentElement;
  var tableElt = makeTable(catalogNode,xmlDOM);
  xmlDOM.replaceChild(tableElt,catalogNode);
          ....
```

We are now ready to extract the HTML in text form.

Retrieving an XML String

The last line of the following example shows the usage of the `getXML ()` function, which returns the contents of a DOM as a single string:

```
function digestDOM(xmlDOM) {
  var catalogNode = xmlDOM.documentElement;
  var tableElt = makeTable(catalogNode,xmlDOM);
  xmlDOM.replaceChild(tableElt,catalogNode);
  var xmlStr = getXML(xmlDOM);
  ....
```

This is another case where we need a wrapper function for portability across browsers. The `getXML()` function is shown in the following script:

```
function getXML(node) {
  if (winMSIE5) {
    return node.xml;
  } else if (standardDOM) {
    var serializer = new XMLSerializer();
    return serializer.serializeToString(node);
  } else {
    return "<null/>";
  }
}
```

Here again, for better or worse, the Microsoft implementation is nonstandard but very convenient, while the Mozilla version complies somewhat more with standards, but is slightly more work for the developer.

Inserting Content in the Document Body

The final step in creating a visible HTML representation is to insert the markup text into the body of the current document. We can complete this step by using the `innerHTML` property of the `document` object:

```
document.body.innerHTML = "<h1>Music Catalog</h1>" + xmlString;
```

An alternative would be to use `document.write()`. The advantage of using `innerHTML` is that it enables you to output only the visible content of the document, or only a portion of it. In contrast, `document.write` replaces the entire document—including scripts and stylesheets that you may wish to keep for further interaction.

All that remains is to provide an appropriate way to activate the JavaScript. Because the XML document is the source of all the content we want to display, it makes sense to display it as the document is loaded:

```
<body onLoad="loadXML();">
</body>
```

Activities

30. Name at least 5 of the 12 DOM node types.

31. Do you think it is important to have a standard API for XML programming? Why or why not?

Extended Activities

32. Using only standard DOM methods such as those discussed in this lesson, write a program (in pseudocode or a language of your choice) that will construct the DOM equivalent to the following document:

```
<dry_cleaning_bill date="2003-03-25">
  <amount>9025.00</amount>
</dry_cleaning_bill>
```

33. Using a text editor of your choice, inspect the address book you created in Lesson 2, and diagram its structure as a tree of nodes. Try to identify the type, name, and value of each node in the tree. Then load the document into XML Notepad and see whether the tree displayed there matches your diagram. Note that XML Notepad does not display all node types (for example, processing instructions are not shown). But at least all elements, attributes, and text nodes should be the same.

Quiz Questions

34. DOM stands for:

    ddd. Document Object Model

    eee. Dynamic Object Model

    fff. Document Operation Model

    ggg. Desktop Operation Machine

    hhh. Differential Object Model

    A

35. Which of the following is *not* represented by one of the standard DOM node types?

    iii. Attribute

    jjj. Entity reference

    kkk. XML declaration

    lll. Text (PCDATA)

    mmm. DOCTYPE declaration

    C

36. The number of node types defined by the DOM specification is:

    nnn. 5

    ooo. 8

ppp. 10

qqq. 12

rrr. 15

D

37. The `nodeValue` property of an attribute node contains the attribute value.

T

38. The `nodeName` property of an element node is null.

F

39. The DOM represents textual content as a property of an Element node.

F

40. A document and its root element are represented by separate nodes.

T

41. The DOM is used only for Web scripting.

F

42. The W3C DOM specification includes Java and ECMAScript bindings.

T

43. Some essential programming tasks, such as loading and saving documents, are undefined in DOM Levels 1 and 2.

T

44. Which of the following statements is true?

sss. Both Internet Explorer and Mozilla use an ActiveX-based DOM implementation.

ttt. Both Internet Explorer and Mozilla are 100% compliant with DOM Level 2.

uuu. Internet Explorer implements DOM Level 2, but Mozilla does not.

vvv. Internet Explorer does not implement the W3C DOM.

www. The DOM implementations in Internet Explorer and Mozilla are mostly compatible, and mostly compliant with DOM Level 2, but differ from each other in a few crucial areas.

E

45. The Javascript function used to retrieve a set of elements with a particular name is:

xxx. elementsWithNameOf()

yyy. getAllElementsNamed()

zzz. getElementsByTagName()

aaaa. findElementsWithName()

bbbb. there is no such function in Javascript.

D

46. The `.xml` property, used to retrieve the content of a DOM node in serialized form:

cccc. is a standard feature of the W3C DOM

dddd. is a Mozilla extension to the DOM.

eeee. does not exist.

ffff.violates the DOM standard.

gggg. is a Microsoft extension to the DOM.

E

47. In order to a useful application with the DOM, it is usually necessary to use extension features provided by various implementations.

T

48. DOM Level 3 includes load and save functions.

T

49. The DOM specification defines a method for retrieving the contents of a DOM tree as an XML document.

F

50. The `innerHTML` property is part of the DOM specification.

F

51. The `getElementsByTagName()` method is part of the DOM specification.

T

52. The normal technique for adding a new child element is to create the element using `Document.createElement()`, then attach it to the parent element using `Node.appendChild()`.

T

53. Given a DOM element node, it is possible to add a new child element to it without reference to the document node.

FNotes

54. Internet Explorer 5.x by default uses an outdated version of MSXML that is noncompliant with standards. If you intend to do any serious XML development with IE, you should install IE 6.x or a recent version of MSXML.

Lesson 5: XSLT and XPath

Author: We need text here.

Objectives

When you have finished this lesson, you will be able to:

- Explain the purpose and the basic design of the XSLT language.
- Interpret simple XPath expressions.
- Describe a simple XSLT stylesheet.

Key Point

XSLT is a powerful language for transforming XML into HTML and other useful formats.

What Is XSLT?

XSLT stands for eXtensible Stylesheet Language Transformations. This rather awkward name reflects the history of the language; what we now call XSLT began as part of XSL (eXtensible Stylesheet Language). The original concept of XSL was that an XML document would be subjected to a two-stage process that would first produce a collection of formatting objects (an XML vocabulary specifies page formatting in a very abstract, media-independent way); the formatting objects would then be rendered in any of a variety of output formats—HTML, PDF, and RTF, to name a few.

Although the ability to simultaneously publish the same content to multiple media is very appealing in concept, it is not always workable in practice. The process of formatting printed documents is a very complex, and it is not always possible for automated formatting systems to produce good results. Furthermore, because people use online documents differently from printed ones, it is often inappropriate to use identical content for both print and online versions. In any case, the demand for HTML and XML output is much greater than that for print formats. Early implementers of XSL, including Microsoft, believed that they could satisfy most use cases by implementing only the first stage of the XSL process, known as transformation. After much debate, the XML community and the W3C came to agree with this view, and the XSL specification was split into two parts: XSLT and XSL (informally known as XSL-FO , for XSL Formatting Objects). Subsequently, XSLT was rapidly implemented by a number of vendors and open-source developers, and has been widely adopted by business.

An XSLT processor uses an XSLT stylesheet to transform a well-formed XML document into HTML or a different XML dialect[7]. You can use XSLT simply as a means to display the document content on the Web, but you can also filter a document based on either its structure or its content, change the order of elements, and even insert new, generated content in the output.

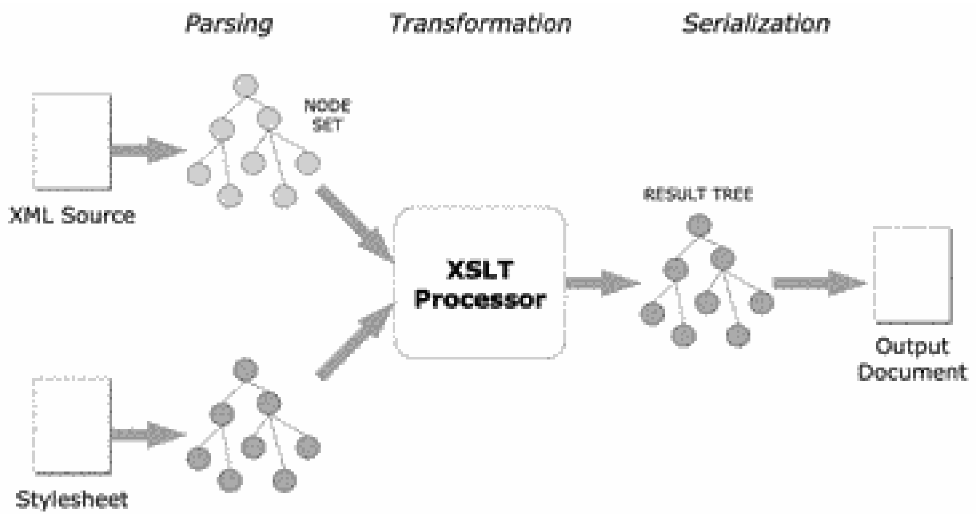
[7] Actually, XSLT can output any text-based format, but is primarily used for XML and HTML output.

In order to use XSLT effectively, it is helpful to understand what takes place when you run an XSLT processor. The following section provides a general description of the XSLT processing model.

The XSLT Processing Model

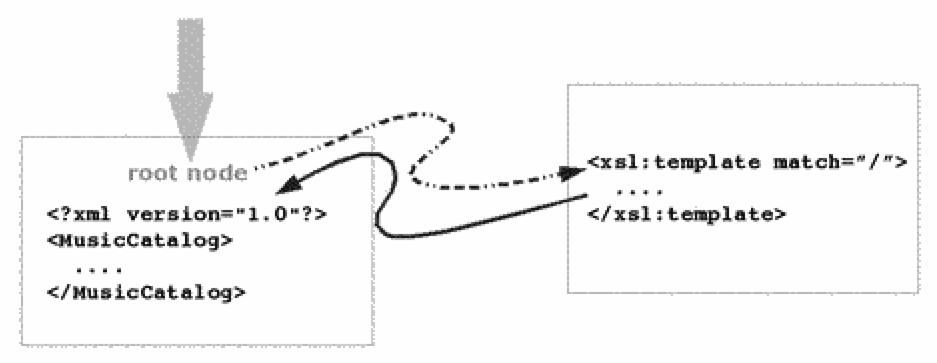
As mentioned above, an XSLT transformation requires two inputs, an XML source document and an XSLT stylesheet; and it produces a single output document, usually in XML or HTML. An XSLT processor performs the transformation per se. Before the transformation can take place, the source document and stylesheet must be parsed, and the result of the transformation must be serialized. The process is illustrated in Figure XSLT Processing Diagram. Strictly speaking, the parsing and serialization steps are separate from transformation, but most XSLT processors allow you to perform all three steps with a single command.

Note that the input to transformation is called a node set, and the output is called a result tree. These two objects are quite similar, but by definition a result tree cannot be subject to further XSLT processing (this restriction has been removed in XSLT 1.1, but most XSLT processors presently implement version 1.0).



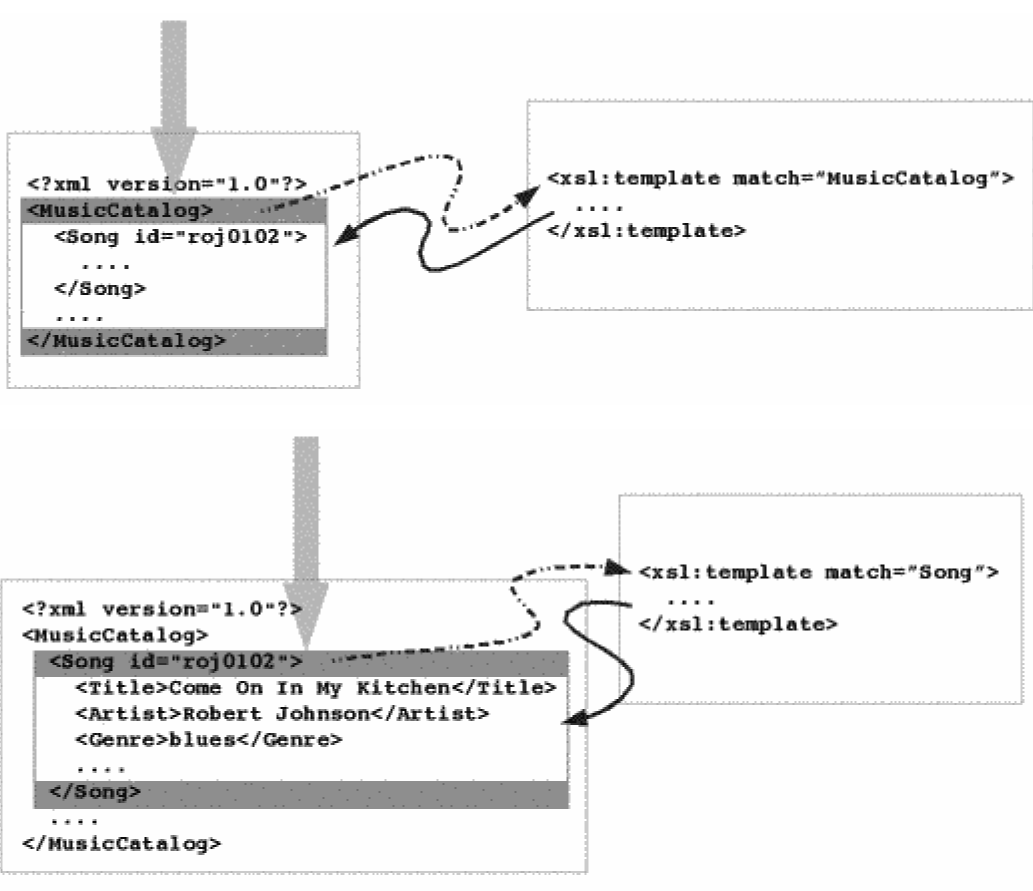
XSLT Processing

Let us look a little more closely at this process. When you feed a node set to an XSLT processor, the processor walks through the nodes and processes each according to a rule defined in a stylesheet. The transformation always begins at the document's root node (remember that the root node represents the whole document and does not correspond to any element). As shown in Figure [?], when the processor encounters the root node, it looks for a matching template rule (represented in XSLT by `match="/"`. If no rule is found, the default behavior is to continue processing all child elements of the root node.



XSLT Processing at the Root Node

The root node has a single child, which is the root element (or document element). Again, the processor looks for a matching template, as shown on the Matching Templates in XSLT Processing Diagram. Again, the contents of the template are used to process the input; if there is no template, or the template includes an `xsl:apply-templates` instruction, processing proceeds to the children of the current element.



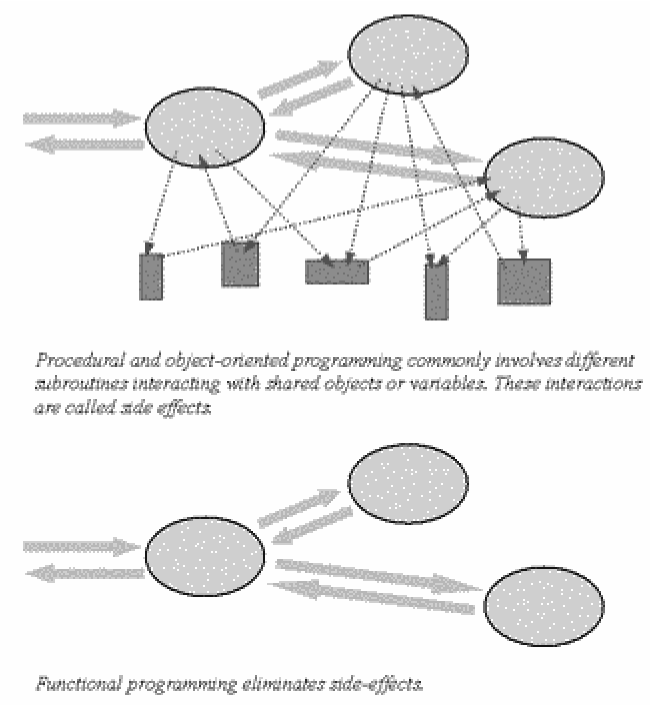
Matching Templates in XSLT Processing

A Note on Language Design

Although XSLT is a programming language and is said to be Turing-complete (which means in theory you can use it to implement any algorithm), it has two important characteristics that are important to note. One is that it is a special-purpose language—it can only process XML input, and has relatively few ways of handling output.

The other characteristic is that it is a functional programming language. Space does not permit a thorough discussion of functional programming, but what is most important to understand here is that functional languages, including XSLT, use a declarative, rather than an imperative style of programming. This means that most or all instructions consist of descriptions of the output for any given input, rather than precise sequences of operations. It also eliminates an entire class of program bugs—those that are caused by inconsistent or unexpected state. A declarative approach makes it easier to accommodate the structural variations very often found in XML documents.

Closely related to declarative programming is the concept of being side-effect free. Imperative languages (including C++, Java, Perl, and Visual Basic) commonly make use of side-effects, which are changes in the states of variables, objects, or external resources such as files and databases. If you eliminate side effects, it becomes unnecessary to control the sequence of execution because the output of a function is dependent only on the input. This concept is illustrated on the Side-Effect Free Programming Diagram.



Side-Effect Free Programming Diagram

The most obvious manifestation of side-effect-freeness is that variables in XSLT work differently from those in imperative languages. In XSLT a variable takes a value when it is defined, and it cannot be assigned a new value. Programmers new to XSLT often find this disconcerting, but—whether you agree with it or not—the lack of variable assignment is quite intentional. XSLT is not purely functional; it also includes imperative constructs such as for loops and if statements. But in many cases a functional style will produce simpler and better-performing code.

XSLT Stylesheets

An XSLT stylesheet is a well-formed XML document containing of one or more templates; each template describes rules for processing a particular portion of the input document. The root element of a stylesheet is:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  ....
</xsl:stylesheet>
```

A template consists of an `<xsl:template>` element and a body, which can contain a combination of XSLT instructions, literal result elements, and text. XSLT instructions are a set of predefined elements belonging to the XSLT namespace. Any other XML elements found within a template are literal result elements. When literal result elements or text are encountered in a template, they are sent to the output as is. This means that if you want a template to generate an HTML table, for example, you can simply write something like this script:

```
<xsl:template>
  <TABLE>
    ....
  </TABLE>
</xsl:template>
```

Like a template body as a whole, the contents of the table element can be a combination of XSLT instructions, literal elements, and text.

We should now consider how templates are combined to make up a stylesheet. XSLT is actually a programming language, although a highly specialized one, and its templates play the role of functions. Like any programming language, you need a mechanism to combine functions into a whole program. XSLT provides two means of doing this. One is to define named templates that can be explicitly called with an `xsl:call-templates` instruction. This technique certainly has its uses; however, because it is much like defining and calling functions in any programming language, we will not discuss it further here.

The other approach, which is usually more effective, uses pattern matching. As mentioned in the discussion of the processing model, an XSLT processor manipulates a tree of nodes. At any point in the tree, you can give the processor an `xsl:apply-templates` instruction, which means in essence, "take this node set and find matching templates to process it." You may well ask what "this node set" is. The answer depends on the form of the `xsl:apply-templates` element. If it has a `select` attribute, the node set to be processed can be almost anything in the document, as specified by the value of that attribute. The default behavior, however, is to select all child elements of the current element. The key to making a template available for matching is the `match` attribute on the `xsl:template` element. This attribute, like the `select` attribute of `xsl:apply-templates`, takes an XPath expression as its value. Because a good understanding of XPath is essential to working with XSLT, let us take a look at XPath before discussing XSLT any further.

XPath

XPath, the XML Path Language, is a compact but powerful expression language used to identify nodes in an XML document. In XSLT stylesheets, XPath expressions can appear in a number of places; they are very commonly used to select nodes (as in the `select` attribute of the `xsl:apply-templates` element) and to specify the pattern that a template matches. A very simple XPath expression might look like this:

```
"title"
```

A simple name with no other qualifying characters refers to an element name. Thus, the previous expression means any elements named `title.` When used as a select expression, it will select from the child elements of the current node.

Of course, the earlier expression could potentially match any element called `title`, regardless of its context. But what if, as often happens, you have a prose document consisting of a number of sections and subsections, each with their own title—in addition to the document title? For example:

```
<article>
   <title>Identifying Shiitake Mushrooms</title>
   <author>Chanterelle Morel</author>
   <section>
      <title>Habitat</title>
      <para>
          ....
      <section>
         <title>Reading the Signs</title>
         <para>
             ....

   <section>
      <title>Commonly Confused Species</title>
      <para>
          ....
```

Chances are you will want to process these different titles in different ways—with different fonts, for example. Fortunately, XPath allows you to limit the match to elements in a particular context. The following example matches only those elements that are children of article elements:

```
"article/title"
```

This following pattern will match section titles:

```
"section/title"
```

The problem with this pattern, however, is that it will apply not only to the titles of top-level sections, but also to the titles of subsections. It turns out that XPath has a solution to this, too, as shown in the following example:

```
"section/section/title"
```

This solution will select only the titles that are children of sections that are themselves children of sections.

You may wonder what happens when two or more templates could apply to the same node. (For example, the XPath expression `section/section/title` clearly applies to subsection titles—but so does `section/title`.) The answer is that the XSLT processor will choose the most specific match it can find [1]; `section/section/title` is considered more specific than `section/title` because it specifies more of the context.

Selecting elements by name and context is only the beginning of what XPath can do. It is often important to be able to distinguish between elements based on attributes and content, and XPath can also handle these tasks. The following example will select any `lease` elements having an attribute called `specialCondition`(regardless of the value of `specialCondition`):

```
"lease[@specialCondition]"
```

Book Title

The contents of the square brackets are known as a predicate expression. Of course, you can also specify a value for the attribute, as follows:

```
"lease[@expirationDate = '20021017']"
```

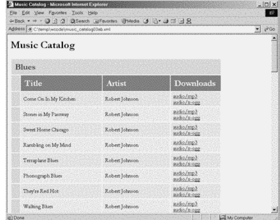
And if you need to match a range of dates rather than a single date, you can use relational and Boolean expressions as shown in the following example:

```
"lease[@expirationDate >= '20021001' and
       @expirationDate &lt;= '20021031']"
```

The less-than symbol must be represented with the `<` entity, otherwise the XML parser will read it as the start of a tag. It is also important to note that “>” and “<” in XPath are limited; they can only compare numbers. If you attempt to compare non-numeric strings using “<” or “>,” the result will always be false.

XSLT Example

As with the DOM, it is probably easier to understand XSLT by looking at a realistic example. Let us revisit the music catalog example of the previous lesson. This time, however, we will add a new feature. The Music Catalog Interface Diagram shows the desired output for this example.



Music Catalog Interface Diagram

Let us begin by creating the outer shell of our stylesheet. As with any XML document, it is a good idea to (author: more text here?)

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

</xsl:stylesheet>
```

This is actually a perfectly valid stylesheet, although not a very useful one. If you applied it as is, the output would consist of all the text in the source document with no markup. In order to generate some useful output (such as HTML), we need to start creating templates.

The following is the template that performs the top-level transformation. In other words, it matches the root node of the source document and outputs the root element of the result tree.

```
<xsl:template match="/">
  <html>
    <head>
      <title>Music Catalog</title>
    </head>
    <body>
      <h1>Music Catalog</h1>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
```

There are two important things to note about this template. One is that it will generate content that is not present in the source document (`<h1>Music Catalog</h1>`). The other is its use of `<xsl:apply-templates/>`. This element will be replaced by the result of applying templates to the descendant nodes of "/" (the root node).

Our stylesheet happens to contain a template that begins with this line:

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

The sole child of the root node is the root element, `<MusicCatalog>`. This is obviously matched by the match expression, so this will be the next template processed. As in the JavaScript example, most of the document body consists of a table that displays the song list. So, naturally, the outer elements of the template will look like this:

```
<xsl:template match="MusicCatalog">
  <table>
  ....
  </table>
</xsl:template>
```

If our stylesheet were following the document structure, we could simply continue in the same vein, working down through a series of progressively more detailed templates to build rows, cells, and their content. But it might be more convenient for site users if we were to organize the table by musical genre, as suggested in this diagram (author: what is the title of this diagram?):

Genre name				←	*Per genre*
	Title	Artist	Downloads	←	*Per song*
	Title	Artist	Downloads	←	....

The difficulty, of course, is that the source document is not organized by genres—and in fact some of the songs belong to more than one genre. Thus we will need to perform a significant transformation. This is where the power of XPath and XSLT start to really become useful.

In order to generate the right table structure, we need to:

1. Find the first occurrence of each genre;
2. Generate the header row; and
3. Generate rows for all songs that belong to that genre.

Before tackling the most difficult task in the stylesheet, let us jump ahead and look at a few more templates. The individual rows of the table are fairly straightforward; here, in simplified form, is the template that generates a header row for each genre:

```
<xsl:template match="Song">
  <xsl:param name="genre"/>
  <tr>
    <th colspan="4">
      <xsl:value-of select="$genre"/>
    </th>
  </tr>
  <tr>
    <th/>
    <th>Title</th>
    <th>Artist</th>
    <th>Downloads</th>
  </tr>
  <!-- Further rows go here -->
  ....
  <tr>
    <td colspan="4"/>
  </tr>
</xsl:template>
```

Here we see an example of parameters in action. If we used a template matching `<Genre>`, we could output its name with a simple `<xsl:value-of select="."/>`. The trouble with that is that we need to go on and process a number of `<Song>` elements. And since in the source document the `<Genre>` elements are children of the `<Song>` elements, by descending into the `<Genre>` scope we lose access to the following `<Song>` elements.

The following generates the song information rows:

```
<xsl:template match="Song">
  <tr>
    <td/>
    <xsl:apply-templates select="Title|Artist|Downloads"/>
  </tr>
</xsl:template>
```

The `select` expression here bears some explanation: as in many languages, the pipe character stands for a Boolean OR, so this is a union expression. It will select all child elements named `Title`, `Artist`, and `Downloads`. The processor will then encounter the following template to process the `<Title>` and `<Artist>` elements:

```
<xsl:template match="Title|Artist">
  <td>
    <xsl:value-of select="."/>
  </td>
</xsl:template>
```

This creates a table cell and inserts the string value of the element in it. The template matching `<Downloads>` is a bit more complex, since it must generate links, but it too produces a single cell. So if a <Song> element contains one each of `<Title>`, `<Artist>`, and `<Downloads>`, these templates will produce exactly the cells needed to complete the table row.

Now that we know how to generate various parts of the table, let us look at how we can make them work together. Again, the difficult part of the template is finding the first occurrence of each `<Genre>` element and grouping all the `<Song>` elements that have that genre. The following XPath expression will perform the first part of that task:

```
Genre[. != preceding::Genre]
```

An English translation of this expression might be "Select all Genre elements whose content is not equal to that of any preceding Genre element in this document."

How can we use the previous expression? Recalling the previous discussion, the starting point for processing the `<Song>` elements is the following template:

```
<xsl:template match="MusicCatalog">
  <table>
  ....
  </table>
</xsl:template>
```

At first glance it appears we can use the following instruction to process the first `<Song>` element in each genre:

```
<xsl:apply-templates select="Song[Genre[. != preceding::Genre]]"/>
```

But this does not work because song may belong to more than one genre. This XPath expression selects the right song, but the information about the genre is discarded. So we have to start by selecting the genre itself, as in this expression:

```
<xsl:apply-templates select="Song/Genre[. != preceding::Genre]"/>
```

The processor will then find and apply the following template:

```
<xsl:template match="Genre">
  ....
</xsl:template>
```

This template will generate the HTML for the header rows. But before leaving the template, we also need to invoke processing for all the songs belonging to this genre. Here again, XPath proves invaluable. We actually need to combine two expressions. The first is (author: What is `..` below?)

```
..
```

Just like navigating a file system, this means "go up one level," i.e. select the parent of the current node. That takes care of the first `<Song>` element. But we also need to select all the other songs in the current genre. The expression to do that looks something like this:

```
(X) following::Song[Genre = .]
```

In other words, "select all songs whose genre equals the current node." Actually, though, this is not quite correct. That is because within the braces [] of the predicate expression, the current node is the Song node that the predicate is being applied to. In order to use the value of the current Genre node within the predicate, we need a variable:

```
<xsl:variable name="this-genre" select="."/>
<xsl:apply-templates
  select="..|following::Song[Genre = $this-genre]"/>
```

Book Title

Finally, we will end up with a template looking like this:

```
<xsl:template match="Genre">
  <!-- Create header row for this genre -->
  ....
  <xsl:variable name="this-genre" select="."/>
  <xsl:apply-templates
    select="..|following::Song[Genre = $this-genre]"/>
  ....
</xsl:template>
```

Activities

1. Explain the usage and effects of the following three XSLT instructions:

- xsl:template
- xsl:apply-templates
- xsl:value-of

2. What type(s) of Web site would benefit most from storing content in XML and using XSLT for presentation? What type(s) of Web site would benefit the least?

Extended Activities

1. This activity involves experimenting with XPath using your address book document. Write XPath expressions to do the following:

- Extract all e-mail addresses from the document.
- Extract the first address record.
- Extract all records where the person's first name matches a certain string.

You can use the simple stylesheet below to test these expressions. Simply save the stylesheet as address.xsl in the same directory as your address book document. Add this processing instruction to the XML document:

```
<?xml-stylesheet type="text/xsl" href="address.xsl"?>
```

Then, insert each XPath expression in turn in place of the text `INSERT YOUR XPATH HERE`. Then, when you load your document into an XSLT-enabled browser, the results should be displayed. (author: Should this previous sentence end with “...the following results should be displayed:”?)

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/">
    <xsl:value-of select="INSERT YOUR XPATH HERE"/>
  </xsl:template>
```

</xsl:stylesheet>The following hints may be helpful:

Because there is only a single template, which matches the root node, all your XPath expressions should be absolute. Remember that absolute XPath expressions begin with a slash (/).

The third XPath expression will require a predicate, something along the lines of [name='Alicia'].

2. What do you think your browser will display if you apply the stylesheet below to your address book? Test your hypothesis by loading the document in your browser, after adding a processing instruction like the one in the previous exercise to the address book.

```
<?xml-stylesheet type="text/xsl" href="experiment.xsl"?>

STYLESHEET:

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>Experiment</title>
      </head>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="*">
    <p>
      <b><xsl:value-of select="name()"/></b>
    </p>
    <xsl:apply-templates/>
  </xsl:template>
</xsl:stylesheet>
```

Quiz Questions

1. The XSLT instruction `<xsl:apply-templates/>`, with no `select` attribute __________________ .

hhhh. Is invalid.

iiii. Is valid, but does nothing.

jjjj. Selects the current node.

kkkk. Selects the textual content of the current node.

llll. Selects all child elements of the current node.

E

2. The XSLT instruction `<xsl:value-of select="."/>`:

a. Is invalid.

b. Outputs an empty string.

c. Selects all child elements of the current node.

d. Outputs the string value of the current node.

e. Outputs the character '.'.

3. DXSLT stands for______________________ .

mmmm. Experimental Style Language Toolkit

nnnn. Extensible Syntax Lookup Table

oooo. Experimental Style Language Transformer

pppp. Extensible Style Language Transformations

qqqq. Essential Syntax Language Tree

D

4. Why is XPath is used in XSLT?

rrrr. To query databases.

ssss. To include remote files.

tttt. To select nodes for processing.

uuuu. To choose an output method.

vvvv. To identify components of the stylesheet.

C

5. The input to an XSLT transformation________________ .

wwww. Must be XML or HTML.

xxxx. Must be a valid XML document.

yyyy. Cannot have attributes.

zzzz. Must be XML.

aaaaa. Can be any type of textual data.

D

6. The output of an XSLT transformation:

bbbbb. Can only be HTML.

ccccc. Can only be XML.

ddddd. Can be any type of textual data, as long as any markup characters are used in accordance with the XML or HTML specifications.

eeeee. Can be any type of textual data, without restriction.

fffff. Cannot be XML.

C

7. The XPath expression "docinfo/date" does which of the following?

ggggg. Matches any docinfo element that has a child element named date.

hhhhh. Is invalid.

iiiii. Matches any date element that is a child of a docinfo element.

jjjjj. Retrieves the date when a document was created.

kkkkk. Matches any docinfo element that has an attribute named date.

C

8. The XPath expression "order[@status]":

lllll. Is invalid.

mmmmm. Matches any element named 'order' that has a child element of 'status'.

nnnnn. Matches any element named 'status' that is a child of an 'order' element.

ooooo. Matches any element named 'order' that has an attribute named 'status'.

ppppp. Queries the status of an 'order' element.

D

9. An XSLT variable cannot be modified after it is defined. True or False?

T

10. What is an XSLT instruction?

qqqqq. Any of a predefined set of elements in the XSLT namespace.

rrrrr. Any element whose name begins with 'xsl:'.

sssss. Any element in an XSLT stylesheet.

ttttt. An XPath expression in an XSLT stylesheet.

uuuuu. An XSLT template.

A

11. XSLT is a replacement for CSS. True or False?

F

12. XSLT can be used to generate JavaScript and CSS in addition to HTML. True or False?

T

13. XSLT eliminates the need for JavaScript. True or False?

F

14. XSLT is a special-purpose programming language. True or False?

    T

15. XSLT cannot change the order of elements. True or False?

    F

16. Only DTD-validated documents can be processed with XSLT. True or False?

F

17. If no XSLT template matches a given element, the XSLT processor will continue to process that element's child elements. True or False?

    T

18. An XSLT stylesheet *must* have one or more templates. True or False?

    F

19. The root node in XSLT represents the root element of a document. True or False?

20. An XSLT stylesheet is a well-formed XML document. True or False?

T

Notes

Author: ?Sorry, but does this note go back to a specific place? 1. Actually, the XSLT rules for resolving template conflicts are rather complex. But in general, the most specific template applies.

Lesson 6: Web Services

Author: We need and intro. here.

Objectives

After finishing this lesson you will be able to:

- Provide a definition for Web services.
- Explain what SOAP, WSDL, and UDDI are and how they relate to each other.
- Discuss the resource-based approach to Web services and how it differs from SOAP.

Key Point

Web services use XML to enable a wide variety of applications to access remote software components over the Web.

New Visions for the Web

The Web has undergone a remarkable change in just a few years. Originally conceived as an online library, it has gained popularity as a virtual shopping center and amusement park. You can still read the latest particle physics abstracts online, but you can also read newspapers from around the world, keep a journal, listen to music, buy airline tickets, and do a thousand other things.

But wait; the Web offers more. People today primarily use the Web to distribute information. Visionary developers have realized, however, that because the Web is so widely deployed and is based on simple, standardized technology, it can also function as a platform for distributed applications that tie together diverse organizations and multiple platforms. And thus the idea of Web Services was born, as defined in the following excerpts from recent books on XML.

> A software application that provides a (specialized) service and can be invoked over the Internet.
>
> —Berthold Daum & Udo Merten,
> *System Architecture with XML*

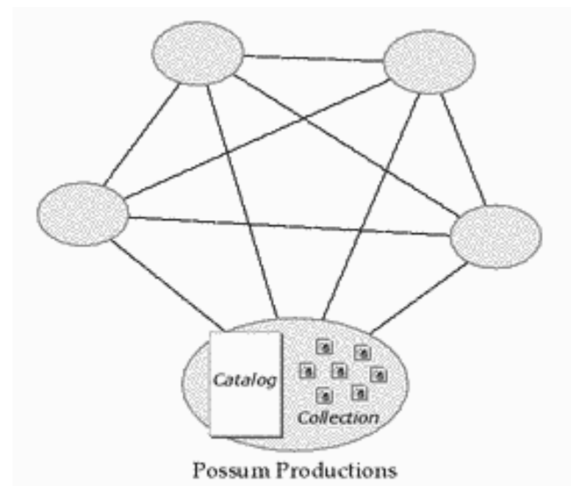
> Web Services is at once a technology, a process, and a phenomenon. As a technology it is a set of protocols that builds on the global connectivity made possible by SOAP and the synergies of XML and HTTP. As a process, it is an approach to software discovery and connection over the Web. As a phenomenon, it's an industry-wide realization that the decentralized, loosely coupled, synergistic Web can't be ignored.
>
> —Frank P. Coyle,
> *XML, Web Services, and the Data Revolution*

Although definitions of the term Web services vary widely, the common thread is that Web Services involve using Web technologies—especially Extensible Markup Language (XML) and Hypertext Transfer Protocol (HTTP)—to build applications using widely distributed, lightweight software components.

A Web Services Scenario

The following portrayal is one way in which Web Services may be used. It is by no means intended to encompass all possible Web Services applications; in particular, it ignores the idea of using Web-based components to extend conventional desktop applications. The point of this story is to show how Web Services technologies can be applied today to enhance the functionality of the existing Web.

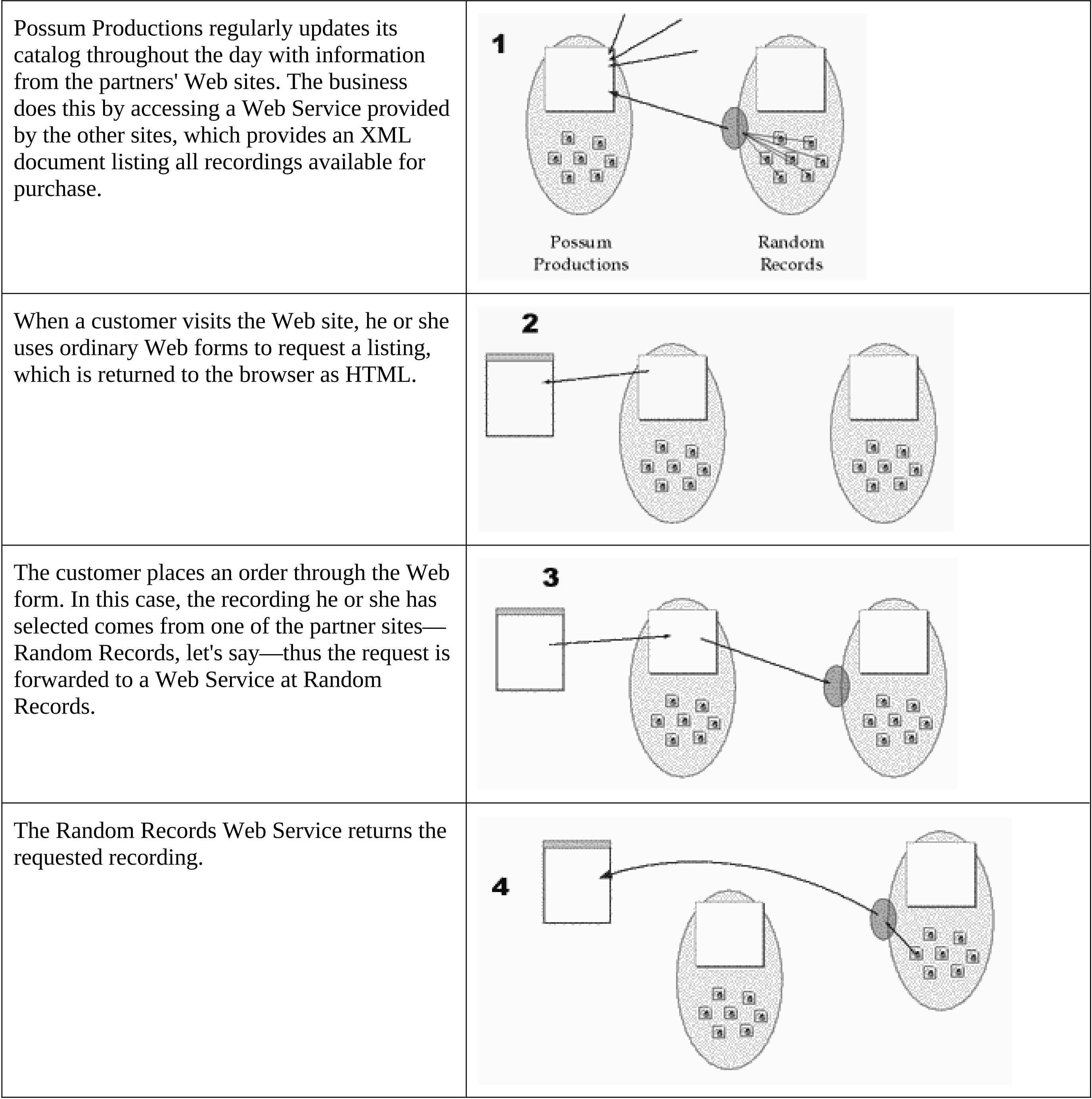
Imagine that a group of small, independent record labels have decided to form a marketing alliance. Each has a modest collection of recordings for sale, but taken together their holdings are substantial. Instead of each company trying to market its own collection in isolation, they create a shared online catalog that advertises the combined collections of all the companies.



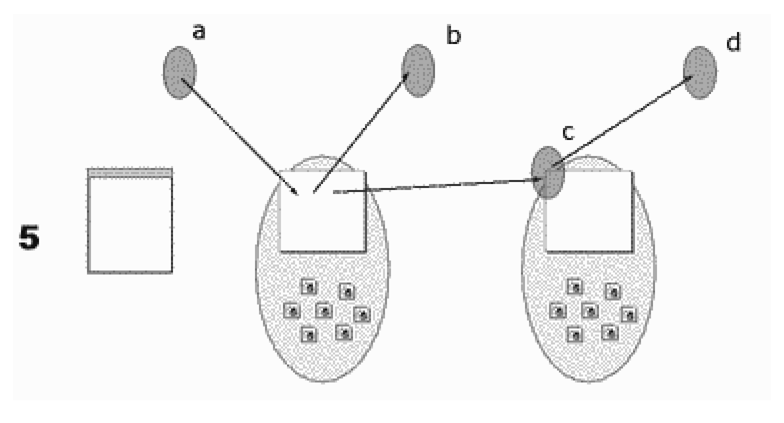
As the following diagram suggests, when customers visit the Possum Productions Web site, they find an easy-to-navigate interface to the shared music catalog. They can place an order here for any of the songs in the catalog. The actual files may be hosted at a completely different Web site, but customers do not need to know that. A user’s experience appears to take place entirely at the Possum Productions Web site.

In this scenario, the user interface is not a Web Service; in fact, from a user's point of view it probably looks just like any of the thousands of online retail sites that exist today. But Web Services are the key to combining the content of several sites into a single interface. What happens behind the scenes is explained on the Shared Catalog Web Service Table.

Shared Catalog Web Service Table

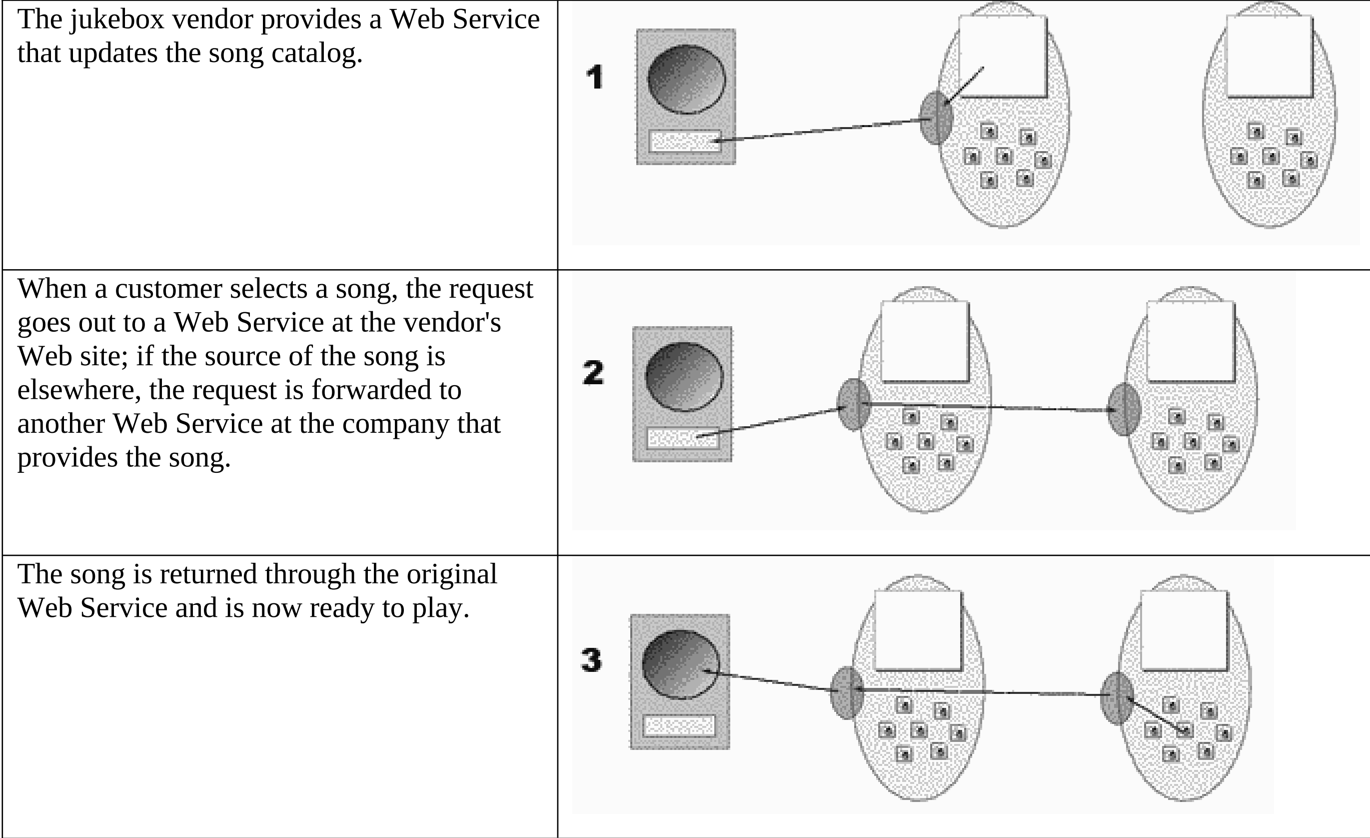


Step	Diagram
Possum Productions regularly updates its catalog throughout the day with information from the partners' Web sites. The business does this by accessing a Web Service provided by the other sites, which provides an XML document listing all recordings available for purchase.	1
When a customer visits the Web site, he or she uses ordinary Web forms to request a listing, which is returned to the browser as HTML.	2
The customer places an order through the Web form. In this case, the recording he or she has selected comes from one of the partner sites—Random Records, let's say—thus the request is forwarded to a Web Service at Random Records.	3
The Random Records Web Service returns the requested recording.	4

Finally, there is the small matter of payment. Assuming the user has provided his or her credit card number, Possum Productions now obtains payment from a Web Service provided by the credit card company (a). It then deposits its percentage of the transaction in its own account and forwards the remainder to Random Records; these operations are also accomplished through Web Services (b & c). Finally, Random Records deposits the payment in its own account, again using a Web Service (d).	

This scenario describes a business-to-consumer transaction. But Web Services can be used to even greater effect for business-to-business transactions. Imagine a company that provides Internet-enabled jukeboxes that download MP3 recordings from the Web. Obviously the browser-based interface makes little sense in this situation because it requires far too much work from a human user. Thus, as shown on the Business-to-Business Web Service Table, from the user's point of view, the jukebox is just another jukebox with a screen showing a listing of songs and a set of pushbuttons for making a selection.

Business-to-Business Web Service Table



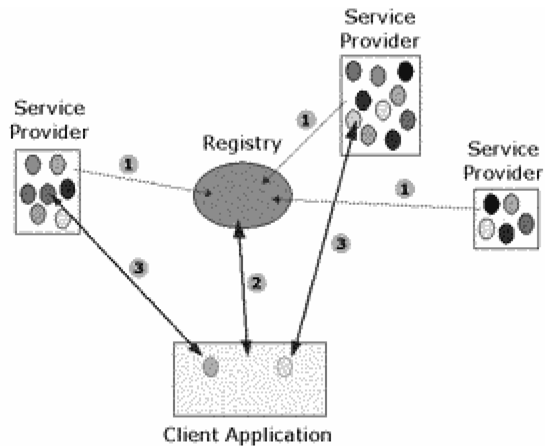
The jukebox vendor provides a Web Service that updates the song catalog.	1
When a customer selects a song, the request goes out to a Web Service at the vendor's Web site; if the source of the song is elsewhere, the request is forwarded to another Web Service at the company that provides the song.	2
The song is returned through the original Web Service and is now ready to play.	3

These examples should give you a sense of how XML can be used to integrate diverse systems among two or more organizations. But the Web services vision goes farther: although an XML-based protocol can ease the task

of developing distributed applications within existing business partnerships, perhaps we can also make it easier for businesses to discover potential partners and opportunities through Web technology..

Service Description and Discovery

After we understand how to link one system with another, we meet the next challenges on the way to fully realizing the Web Services vision. First, for clients to effectively use Web Services, there must be a commonly understood way to describe the services. Second, a discovery mechanism is needed for clients to be able to find relevant services without painstaking research. With these two goals accomplished, the scenario begins to function like the Web Services Diagram. This diagram shows that various service providers deploy Web services, which they then register with a centralized service known as the registry. A client application looking for a Web service component can query the registry to locate a service provider. Finally, the client accesses the Web service. In theory, the user need not know where the Web service is coming from, or even that the application is using Web services at all: the Web service simply behaves as a software component providing part of the application's functionality.



Web Services

Whether this vision will ever be fully realized is an open question. Service description and discovery mechanisms are gradually coming into use, and it is clear that they help to reduce the effort of building distributed business applications. But these technological advances may have only a small effect on business-to-business transactions because businesses prefer to establish trust through human contacts before developing technological links. On the other hand, there is considerable anecdotal evidence that the Web services model is proving valuable *within* organizations.

Implementing Web Services

In a technical sense, there are at least two major approaches to implementing Web services. The first, which has the support of most major software vendors, is built around three major technologies: SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery, and Integration). Because both SOAP and WSDL are based on W3C specifications, I will refer to this as "the W3C approach." The technical architecture of this approach is rooted mainly in Object Oriented Programming and associated technologies for distributed systems, such as Distributed Component Object Model (DCOM) and Common Object Request Broker Architecture (CORBA).

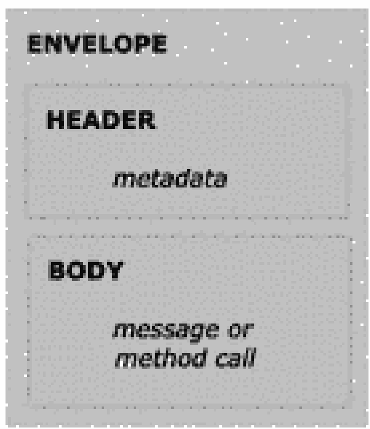
The alternative is—intentionally—much more loosely defined, but revolves around utilizing existing Internet protocols to manipulate resources, which are abstract information units in network space. Although the W3C approach is by far the dominant one at present, the resource-centric approach has strong technical foundations, and has substantial support among XML and Internet experts. Even if the W3C approach remains dominant, its development has already been influenced by the resource-centric approach.

SOAP and XML-RPC

Although the phrase "paradigm shift" has been overused in recent years, the development of SOAP (Simple Object Access Protocol) may indeed represent a paradigm shift. From a strictly technical standpoint, there is nothing particularly innovative or even interesting about SOAP. It is just a simple XML format that can carry a variety of messages (or RPC calls) over the Internet using standard protocols, such as HTTP. But it represents a significant--some would say revolutionary--change in thinking about distributed computing and how the Internet can facilitate it.

As the name suggests, the idea behind SOAP is that applications can use XML to communicate with remote objects on the Web. A service provider often creates a specialized application and uses a SOAP toolkit to install the application on a server. On the client side, a SOAP toolkit provides a proxy interface, so that when the client invokes a method on the proxy object, the method call is translated into an XML-based SOAP message, forwarded to the appropriate server, and transformed back into a method call. In theory, at least, it makes no difference what platform either side is running, or what languages they use for their applications; everything is tied together through the common language of XML.

The SOAP specification defines a standardized message format that can contain any type of XML document. The structure of a SOAP message is shown on the SOAP Message Format Diagram.



Soap Message Format

An actual SOAP message might look like the following script. If Possum Productions offered a SOAP-based service allowing users to rate songs, the message used to submit a rating might look like the following:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <app:RateSong xmlns:app="http://soap.possum.com/musicapps/">
      <songID>wjx2349</songID>
      <rating>3.5</rating>
    </app:RateSong>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

To access the service, this message is submitted using an HTTP POST request[8] to a known URL—for example:

```
http://soap.possum.com/
```

This URL is known as a SOAP endpoint. Messages sent to it are received by a SOAP server, which chooses the appropriate SOAP service—usually an application local to the SOAP server—to process the request.

In addition to the XML message format, SOAP requires a special HTTP header, SOAPAction as shown here:

```
SOAPAction: http:// soap.possum.com/music/
```

The value of the SOAPAction must be a URI, which typically corresponds to the XML namespace associated with the service being requested.

Several important points to note about this message are:

- The SOAP body contains a message in the form of an application-specific XML document. This is known as the payload, and it can follow any format at all, as long as it is well-formed XML. Naturally, both client and server must understand the message format.
- An application-specific namespace is used to identify the message. Although this is not required, it is standard practice.
- The message normally includes the name of an action (`RateSong`), the resource to be operated on (`songID=wjx2349`), and additional arguments representing whatever data is being submitted (`rating=3.5`). The significance of these facts will become clearer as we discuss Resource-centric Web Services in the next major section.

SOAP Toolkits

You may correctly feel that it would take a great deal of work to create applications that generate and process SOAP messages. However, most SOAP applications are built using one or another Web Services toolkit, thus the application you build needs at most to process the application-specific XML message in the SOAP payload. In fact, SOAP toolkits also generally provide an RPC (Remote Procedure Call) interface. Such interfaces provide proxy objects that represent remote services; developers can—at least in theory—invoke remote Web Services through these proxies as if they were local method calls, thus eliminating the need to write any XML.

WSDL

Web Services Description Language (WSDL) is an XML dialect for describing the technical characteristics of Web services. It is intended as a means for client applications to automatically determine how to access a Web

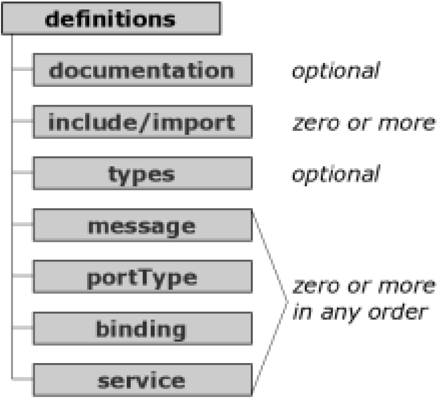
[8] Actually, SOAP allows for other Internet protocols, but HTTP is by far the most commonly used.

service. In fact, some Web services toolkits will automatically generate WSDL service descriptions, thus you may never need to create a WSDL file manually. Still, it is a good idea to have at least passing familiarity with the structure of a WSDL file.

The root element of a WSDL document is `<definitions>`; the following script is an example[9] of such an element. As suggested by the large number of namespaces declared, WSDL relies on a variety of supporting technologies:

```
<wsdl:definitions
  targetNamespace="http://soap.possum.com/music/MusicCatalog3.jws"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:apachesoap="http://xml.apache.org/xml-soap"
  xmlns:intf="http://soap.possum.com/music/MusicCatalog3.jws"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ....
</wsdl:definitions>
```

This element contains a rather flexible assortment of elements describing various aspects of the service, such as the messages it accepts and produces, and the protocol used to access it. The high-level structure of a WSDL document is shown on the WSDL Document Structure Diagram.



Document Structure Diagram

The functions of the `documentation` and `include/import` components are fairly self-explanatory. The remaining components are described in the following sections.

types

The types component, represented with a `wsdl:types` element, contains definitions of data types and elements in XML Schema syntax. It is, in fact, an internal schema for the relevant services. This component may not be needed if all messages use only simple data types. But some common uses cases will require type definitions. One example is a method that returns an array of values. Because XML Schema does not provide any built-in type for lists, sets, or other collections, you would need to define a complex type for this case.

[9] This example is taken from a real WSDL document generated by Apache Axis from Java source code.

message

Each message component, represented by a `wsdl:message` element, defines an incoming or outgoing message used in communicating with the Web service.

```
<wsdl:message name="rateSongRequest">
  <wsdl:part name="song_id" type="xsd:string"/>
  <wsdl:part name="rating" type="xsd:float"/>
</wsdl:message>
<wsdl:message name="rateSongResponse">
</wsdl:message>
```

This WSDL defines two messages that will be transmitted as SOAP payloads. The `rateSongRequest` message will have two child elements: `song_id`, whose value must be a string, and `rating`, whose value must be a floating point number. The `rateSongResponse` message will be empty. It is important not to read too much into these declarations. Judging from the message names, it would appear they are intended to be used together, as the input and output for some method of the Web service. And in fact these message comprise the interface to a Java method whose signature is:

```
public void rateSongRequest(String song_id, float rating)
```

But nothing in the message declarations associates the messages with this code, or even with each other. At this stage all we can be sure of is that this SOAP server will recognize these messages.

portType

The `portType` component defines a set of public operations, and associates them with input and output messages. Although these operations are abstract—that is, they do not necessarily correspond directly to any real methods—this at least gives us a method name, and defines which messages are used for the method's input and output. The `portType` component is used in the following script:

```
<wsdl:portType name="MusicCatalog3">
  ....
  <wsdl:operation name="rateSong" parameterOrder="song_id rating">
    <wsdl:input message="intf:rateSongRequest" name="rateSongRequest"/>
    <wsdl:output message="intf:rateSongResponse"
      name="rateSongResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

binding

As shown in the following script, the `binding` component provides the link between the logical operations specified in the `portType` section and the implementation in SOAP (or, theoretically, another messaging protocol):

```
<wsdl:binding name="MusicCatalog3SoapBinding"
  type="intf:MusicCatalog3">
  <wsdlsoap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  ....
  <wsdl:operation name="rateSong">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="rateSongRequest">
      <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://soap.possum.com/music/MusicCatalog3.jws"
        use="encoded"/>
    </wsdl:input>
    <wsdl:output name="rateSongResponse">
      <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://soap.possum.com/music/MusicCatalog3.jws"
        use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

service

Finally, the `service` component associates a binding component with the physical location of the service provider—a SOAP endpoint, assuming the service is SOAP-based. This is how the server announces to the world that a particular Web service is available. The `service` component is used in the following script:

```
<wsdl:service name="MusicCatalog3Service">
  <wsdl:port
    binding="intf:MusicCatalog3SoapBinding" name="MusicCatalog3">
    <wsdlsoap:address
      location="http://soap.possum.com/music/MusicCatalog3.jws"/>
  </wsdl:port>
</wsdl:service>
```

If you have been reading carefully, you may have noticed that although the operations defined in this WSDL file happen to have the same names as the Java methods used to implement them, there is no explicit association between the two. That is because WSDL is only intended to specify a public Web services interface. Of course, in most cases it makes sense to use the same names for the public interface that are used in the implementation code. But actually specifying that relationship is outside the scope of WSDL.

UDDI

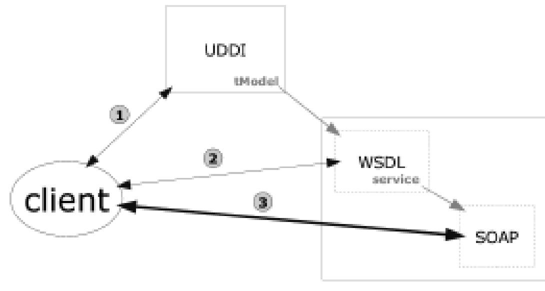
Universal Description, Discovery, and Integration (UDDI) is designed to be a comprehensive framework for locating Web services providers. The information provided by UDDI is often described using a telephone directory metaphor: there are "white pages," providing basic contact information for businesses; "yellow pages," where businesses are classified according to standard industry taxonomies; and "green pages," which expose the technical information needed for business to interact using Web services. This information is provided through the UDDI Business Registry; the Registry is logically centralized but is physically made up of multiple nodes. As

of this writing, four companies operate public UDDI registry nodes: IBM and Microsoft in the US, SAP in Germany, and NTT Communications in Japan.

It should be noted that the telephone directory metaphor only describes the types of information UDDI provides; it does not directly correspond to the actual structure of an UDDI registry. Also, UDDI is not designed to provide complete information about registered businesses. Its most important value is in discovering the technical requirements for online collaboration.

The key information structures in the UDDI registry are called `businessEntity, businessService, bindingTemplate,` and `tModel`. The `businessEntity` is a container for basics such as a company's name and contact information; it also may contain one or more `businessService` objects, which provide an overview of the business-provided Web services. Each service is described in some detail by a `bindingTemplate`, which in turn references technical details described in the `tModel` (technical model).

Those details, however, are not directly stored in the `tModel`. Rather, the `tModel` will reference an information resource at an address specified by a service provider. In many cases this resource will be a WSDL document, but UDDI is designed to accommodate other types of service specification.



(Author: Please explain what this diagram illustrates.)

UDDI, when fully implemented, will complete the Web services architecture as conceived by Microsoft, IBM, the W3C, and others. As illustrated above, a client application accesses the UDDI registry to determine what services are available and how to access them. WSDL then describes precisely how the client can communicate with the service. Finally, using the interface described in WSDL, the client is able to access the Web service to perform useful work.

Resource-Centric Web Services

Although the W3C approach to Web services has captured the imagination of the trade press and is supported by most major software vendors, not everyone agrees that it is an effective solution for distributed applications on the Web. In addition to the interoperability problems mentioned above, SOAP raises security concerns that have not been fully addressed. Because it deliberately bypasses firewalls, SOAP requires separate security measures based on an understanding of what the SOAP methods are doing. Finally, it is doubtful that an RPC-based Web Services architecture is a viable basis for realizing the promised ease of use. Because each SOAP service has a unique API, you need prior knowledge of a service's methods to develop a client for it.

In response to these issues, some developers advocate an alternative approach. Because this approach has several variations and is generally independent of any specific technology, this course refers to it using the general term “resource-centric Web services.” The common thread of the resource-based approach is that rather than using an object-oriented model where clients obtain services through application-specific methods, XML documents (also called data collections) are exposed to the client through commonly understood protocols, such as HTTP. It is

then up to the client how to process the data. This approach may seem to be a step backwards, but it has a significant advantage: given the address of a resource, a client can access its information with no prior knowledge of its content—and thus with no custom programming.

REST

One variant of the resource-based approach that has been gaining momentum lately is known as Representational State Transfer (REST). REST is mainly a descriptive model of how the Web actually works. In simple terms, the REST model consists of resources, components, and connectors. Resources are abstract units of information that can be addressed with URIs; components are software programs such as Web servers and clients; and connectors are protocols that allow the components to exchange information. Note, however, that what is exchanged is not resources. Because resources are by definition abstract entities, a client does not know —and does not need to know—whether the resource represents information stored in a file, or the records in a database, or the output of a program.

The actual information that is exchanged by the components is in the form of representations. Any given resource can have a number of different representations: for example, data stored in the form of SVG might be represented as a JPG image or an HTML file describing the contents, or even a "raw" SVG representation. When accessing the resource's URI, a client should request a particular representation, and if possible, the server should provide that representation. Again, the client does not need to know how that representation is produced.

But what really makes REST powerful is that all actions are performed through a minimal set of HTTP methods: GET, POST, PUT, and DELETE. Thus, there is no need to learn application-specific APIs. In place of SOAP methods, such as `getSongList`, `getPrice`, and `getSong`, various methods for obtaining various types of information—applications send HTTP GET requests to different URIs representing the different types of information. In place of SOAP methods like `sendEmail`, `updatePrice`, and `convertDollarsToYen`—all of which add new information to a collection or manipulate existing information, a REST application can simply send the information to an appropriate URI using HTTP POST (or perhaps PUT).

Thus, to revisit the song-rating example we looked at earlier: a minimal SOAP message for rating a song takes about 10 lines of XML and a special HTTP header. Very likely the XML would be hidden behind an API method call such as `rateSong("wjx2349",3.5)`, but a developer still has to know the API, which is likely different for each service. To accomplish this using REST might be as simple as POSTing this message:

```
<rating>3.5</rating>
```

to the following URI:

```
http://www.possum.com/consumer-ratings/wjx2349
```

Comparison of RPC and Resource-Centric Approaches

The following list shows how several actions related to the music store scenario might be implemented in SOAP-RPC and REST. We should remember, of course, that the SOAP API hides a substantial amount of XML, which is omitted here for brevity.

- Obtain a catalog listing

  -RPC:

  ```
  getSongListByGenre("hip-hop")
  ```

-REST action:

```
GET http://www.possum.com/catalog/hip-hop
```

- Log in:

-RPC:

```
login("joebob","xj233i8Dibfa3Q2o7coe4rGnuf34xZ34289")
```

-REST:

```
POST
 <member>
   <user>peggysue</user>
   <pass>bIerf0342fET09eb349BurZ47O43oq457</pass>
 </member>
to
 "http://www.possum.com/member-entrance"
```

- Download a song

-RPC:

```
getSong("wjx2349")
```

-REST:

```
GET "http://www.possum.com/songs/wjx2349"
```

- Rate a song

-RPC:

```
rateSong("wjx2349",3.5)
```

- REST

```
POST:

  <rating>3.5</rating>
to
  "http://www.possum.com/consumer-ratings/wjx2349"
```

It is impossible to predict how Web services technologies will develop over the next few years. It seems likely that major software vendors will continue to support the W3C approach, especially because a great deal of money has already been invested in developing products associated with that approach. On the other hand, the arguments of REST advocates and other skeptics are already helping to improve Web services specifications and practices. For example, SOAP versions up to and including 1.1 allow only the HTTP POST method, even for tbe many actions that consist only of retrieving information. SOAP 1.2, currently under development, will allow for both POST and GET.

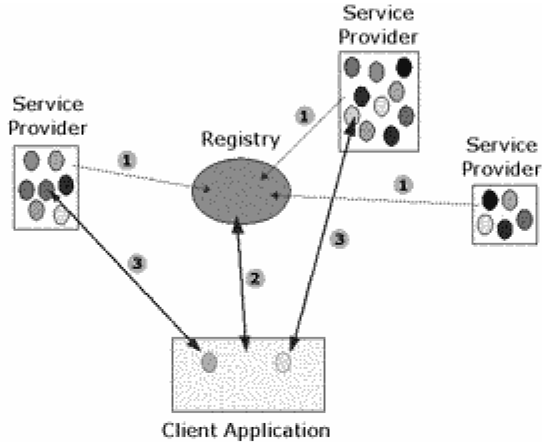
It has also become clear in practice that the RPC model, which relies on the availability of remote objects and methods, is often unsuitable for Internet-based applications, where remote services are sometimes unavailable and

often involve considerable delay. Thus, many SOAP advocates now recommend a document-oriented approach, where request and response methods are treated as data to be exchanged rather than function calls and their results.

Web services may never live up to their early hype, but XML is gradually gaining ground as a cross-platform integration technology and is likely to play a role in distributed development for some time to come.

Service Description and Discovery

Whether you choose SOAP or a resource-based alternative, you will encounter two additional challenges on the way to fully realizing the Web Services vision. First, in order for clients to effectively use Web Services, there must be a commonly understood way to describe the services. Second, a discovery mechanism is needed in order for clients to be able to find relevant services without painstaking research. Author: What is the title of the diagram below? And is this section done?



Other Technologies Related to Web Services

Given that the very idea of Web services is still less than fully formed, it should come as no surprise that there is no consensus as to what constitutes a Web services technology. There are a variety of technologies that are relevant, and they tend to represent a variety of interests and world views. In any case, there are at least two other initiatives that should be mentioned in connection with Web services.

ebXML

Although ebXML (Electronic Business using XML) is not being marketed under the Web services label, conceptually it has much in common with the W3C approach. The overall vision for ebXML

The most notable difference between ebXML and the W3C approach is that ebXML provides a standardized semantic framework for e-commerce, with a variety of XML document formats for expressing business processes and agreements.

ebXML is a joint project of OASIS (the Organization for the Advancement of Structured Information Systems) and UN/CEFACT (the United Nations Centre for Trade Facilitation and Electronic Business).

.NET

Readers will undoubtedly have heard of .NET, Microsoft's next-generation development platform. But what is .NET, and how does it relate to Web services? The first question is rather hard to answer, because the name .NET encompasses a confusing variety of components and technologies. Perhaps the most sensible statement one can make is that .NET, rather than representing any specific technology, is an integration strategy that aims to ease the tasks of integrating systems across platforms, languages, and networks. Although .NET is not synonymous with Web services, it shares with Web services a basic organizing concept: software constructed from loosely coupled, reusable components.

Two specific .NET components that are worthy of mention in connection with Web services are ASP.NET and Visual Studio .NET. ASP.NET is of course an updated version of Microsoft's popular ASP, but it also incorporates some significant advances: one is the inclusion of form libraries to ease dynamic Web page development. The other, more relevant to this discussion, is that it serves as a host for managed code, one of the key technical features of .NET. This means that it is possible to create software components in any .NET-enabled language (Visual Basic .NET and C#, for example) and deploy them on the Web through ASP .NET. Visual Studio .NET allows developers to create Web services projects, in which the details of deployment, including generating WSDL service descriptions, are handled automatically by Visual Studio.NET and ASP .NET.

It should also be noted that .NET implementations for the Linux and FreeBSD operating systems are under development, thus we can expect that the framework will be at least partly a cross-platform technology.

Activities

1. Define the term "Web services."

2. Describe the three major components of the W3C approach to Web services, and their respective roles.

Extended Activities

1. Write a one-page essay describing a potential use case for Web services in the travel industry. Extra: Disregarding the availability and cost of tools and expertise, which of the two major approaches to Web services do you think is more appropriate for this application? Why?

2. A real-world example of Web services can be found at Google, which provides a SOAP API for its search services. We will use some examples from Google for this activity.

   Download the Google Web APIs Developer's Kit from

   ```
   http://www.google.com/apis/download.html
   ```

   The download package is a ZIP file containing a soap-samples directory; this contains three pairs of SOAP request and response messages. Choose one of these pairs and determine the following:

   vvvvv. The name of the method being accessed.

   wwwww. The arguments to the method and their data types.

   xxxxx. The data type of the response.

   Based on the above information, write a method signature for the service represented by the request and response. For example,

   ```
   string foo(integer x)
   ```

represents a method named foo, which has a single required argument called x (an integer), and returns a string.

Quiz Questions

1. SOAP stands for ________________ .

   a. Symbolic Object Access Protocol

   b. Simple Object Access Protocol

   c. Synchronous Or Asynchronous Parsing

   d. Symbolic Online Analysis Protocol

   e. Simple Online Analytic Programming

   B

2. The names of top-level components in a WSDL document include which of the following? (Choose one.)

   a. portType, schema, binding

   b. stylesheet, message, portType

   c. service, portType, binding

   d. types, model, message

   e. service, message, protocol

   C

3. Which HTTP method does SOAP generally use?

   a. GET

   b. HEAD

   c. PUT

   d. DELETE

   e. POST

   E

4. Which of the following statements is NOT true?

   a. WSDL was designed for use with SOAP, but can be used with other forms of Web services.

   b. An UDDI registry is accessed using SOAP.

   c. WSDL service descriptions are accessed using SOAP.

   d. An UDDI `tModel` may, but need not, reference a WSDL document.

   e. WSDL helps a client understand how to use a Web service, but not how to locate the service.

   C

5. ____ is divided into white, yellow, and green pages.

   a. UDDI

   b. SOAP

   c. REST

   d. WSDL

   e. XSLT

   A

6. __ is used to describe the interface of a particular Web service or group of related services to clients.

   yyyyy. SOAP

   zzzzz. XML Schema

   aaaaaa. UDDI

   bbbbbb. WSDL

   cccccc. ebXML

   D

7. ___ provides a standardized format for exchanging messages in Web services.

   dddddd. SOAP

   eeeeee. HTML

   ffffff. XML Schema

   gggggg. XSLT

   hhhhhh. WSDL

   A

8. REST relies primarily on ____ methods.

   iiiiii. RPC

   jjjjjj. SOAP

   kkkkkk. .NET

   llllll. HTTP

   mmmmmm. DOM

   D

9. By default, SOAP messages use XML Schema data types.

   T

10. SOAP messages can only use XML Schema data types.

   F

11. SOAP messages can have binary data attached.

    T

12. ebXML provides schemas for various types of business documents.

    T

13. The name of a message defined by a WSDL `message` component must match the name of a method in the implementation code.

    F

14. The `portType` component in WSDL defines the relationship between messages and operations.

    T

15. REST is a programming API.

    F

16. Using UDDI, intelligent Web service agents can discover and use Web services anywhere on the Internet. True or False?

F

17. Web services require the use of SOAP. True or False?

    F

18. Web services require the use of WSDL. True or False?

    F

19. Web services must use RPC. True or False?

    F

20. SOAP messages can be transmitted over various Internet protocols. True or False?

T

Unit Summary

Author: Please write a summary. To do this, you can simply write a couple of sentences or a brief paragraph about each lesson.