



Distributed Software Development

XML

Chris Brooks

Department of Computer Science
University of San Francisco

- About XML
- Structuring XML documents
- Using CSS to display XML
- Parsing with DOM
- Parsing with SAX

- XML is a language for describing data
 - Really more of a meta-language
- XML itself provides metadata
 - Data types, relations between data objects, etc.
- Designed to be read, created, and consumed by programs.

Advantages of XML

- Well-defined, easy-to-manipulate structure
- Human-readable
- Extensible
- Metadata can be included directly with data
- Widely used

Things to note

- An XML document has two components:
 - tags (metadata)
 - content (data)
- Metadata serves to help an application make sense of the data.

Example

```
<?xml version="1.0"?>
<book>
  <author> J.R.R. Tolkien </author>
  <title> The Lord of the Rings </title>
  <volumes>
    <volume> Fellowship of The Ring </volume>
    <volume> The Two Towers </volume>
    <volume> Return of the King </volume>
  </volumes>
  <price> 14.95 </price>
  <publisher> Ballantine </publisher>
  <isbn> 0345340426 </isbn>
</book>
```

XML documents as trees

- An XML document can also be represented as a tree.
- This makes XML very easy to parse.
- The outermost element is the root element, and elements contained within it are children of that element.
- Content is stored at the leaves
- What would the tree for our Tolkien example look like?

Outline

- About XML
- **Structuring XML documents**
- Using CSS to display XML
- Parsing with DOM
- Parsing with SAX

Elements

- XML requires that every starting tag have a corresponding closing tag.
- Everything between a starting tag and a closing tag is called an *element*
- For example, `<volume>Return of The King </volume>` is an element
- So is everything between `<volumes>` and `</volumes>`
- As is everything between `<book>` and `</book>`.
- This means that elements must be nested.

Tags and elements

- Tags form the boundaries of elements, and give processing instructions to parsers.
 - Empty elements: `<coAuthor />` All information is contained in the tag.
 - Container elements: `<price> 14.95 </price>`
 - Comments: `<!-- here's a comment -->`
 - Declaration: `<!ENTITY jrirt ``J.R.R. Tolkien>` This provides a way to define variables or constants in a single location.
 - Entity reference: `<author> &jrirt </author>`

Attributes and Values

- You can also specify that an element has *attributes*
- These attributes can take on *values*
- This is helpful when you want to specify that an object belongs to one of a few types.

```
<book genre="fantasy" size="large"> ...  
</book>
```

Attributes vs. Sub-elements

- We could rewrite the example above using subelements instead of attributes.
- When to use one over the other is largely stylistic.
 - Can always transform one into the other
- If a feature can only take on one of a few values, an attribute might make more sense.
- If we expect to extend the number of genres, a subelement is preferable.
- Also, order is preserved for subelements
 - Semantically, attribute/value pairs are treated as a dictionary.
- So, a list of authors should be done as subelements

ID attributes

- A particularly helpful attribute is ID - this lets you assign a reference to an element and refer to it later in the document.

```
<volume id="book1"> Fellowship of the Ring </volume>  
<volume id="book2"> The Two Towers. Read this book after you've  
finished <volumeref idref="book1" />.</volume>
```

- The ref tag refers to a previous volume
- This provides the XML parser with the information that this is a reference to a previous volume with id “book1”.

Document Prolog

- If you've looked at XML that's used by other applications, you've probably noticed a lot of messy-looking stuff at the top.
- This is called the *document prolog*.
- This tells a client that the document is in XML and refers it to other document that indicate which tags are valid.

```
<?xml version="1.0" encoding="US-ASCII" standalone="no">
<!DOCTYPE book
  PUBLIC "-//USF //DTD Book 1.8//EN"
  "http://www.foobar.com/DTDs/lotr.dtd"
[
  <!ENTITY jrirt "J.R.R. Tolkien">
  <!ENTITY elvish-key "elvish.xml">
]>
```

Document Prolog

```
<?xml version="1.0" encoding="US-ASCII" standalone="no">
```

- This is the XML declaration.
- It indicates that the document is XML, the encoding schema, and whether or not the client will need to fetch supporting documents.

Document Prolog

```
<!DOCTYPE book
```

- This is the document type declaration - it indicates that the root element in the XML document is a book.

Document Prolog

```
PUBLIC "-//USF //DTD Book 1.8//EN"  
"http://www.foobar.com/DTDs/lotr.dtd"
```

- These lines designate a document type definition.
- Basically, this points to a separate document (called a DTD) that describes what elements books are allowed to have.

Document Prolog

```
<!ENTITY jrirt "J.R.R. Tolkien">  
<!ENTITY elvish-key "elvish.xml">
```

- These lines declare an *internal subset*. These are sort of like C macros; they give a shorthand for elements that occur repeatedly throughout the document.
- All of the lines in the prolog except for the first are optional.

- We could then use our entity definitions later in the document by prepending a '&' to them

```
<book> ...
```

```
<description> the Author of The Lord of the Rings is &jrirt; he  
invented a grammar and semantics for Elvish, which can be found at  
&elvish-key;
```

```
</description>
```

Outline

- About XML
- Structuring XML documents
- **Using CSS to display XML**
- Parsing with DOM
- Parsing with SAX

Using CSS to display XML

- CSS can also be used to display XML documents.
- Control is limited to laying out a complete XML document.
- If we want filtering or sorting, we'll need to use XSLT.

An example

- Let's say we have an XML-based CD database:
- We can use CSS to display it in a web browser.
- (see separate examples)

- About XML
- Structuring XML documents
- Validating XML with schema
- Using CSS to display XML
- **Parsing with DOM**
- Parsing with SAX

Parsing XML

- XML also has the advantage of being easy for programs to parse and construct.
- There are two different approaches to parsing and manipulating XML.
- SAX: Simple API for XML
 - Event-driven parser
 - User defines actions to take when an element is found during parsing.

- DOM: Document Object Model
 - Tree parser: Entire document is instantiated in memory as a tree.
 - Nice for random-access applications
 - Large documents may consume a large amount of memory
- Most languages provide support for both. We'll start with DOM.

- The DOM model is specified in a language independent way.
- Implementations then follow this specification.
 - This means that they all work very similarly.
- Java
 - javax.xml.parsers built into Java 1.5
 - Apache's Xerces parser provides support for both SAX and DOM.
 - Xerces also has C++ and Perl implementations
 - JDOM is also a popular tool for parsing and creating XML in Java.
- Python
 - Built-in support for SAX, DOM, and minidom
 - ElementTree is a DOM-like parser.
 - 4suite provides third-party implementations

Libraries

- Perl
 - LibXML provides SAX and DOM functionality.
- C#
 - .NET has built-in support for SAX and DOM
- Ruby
 - The REXML library provides tree parsing, but not with the DOM interface.

Parsing a document in Python

- Example:

```
from xml.dom import minidom
doc= minidom.parse('library.xml')
```

- Reads in and parses a document
- creates a Document object.
- toxml() show the XML version.

Traversing the tree

- childNodes, firstChild, lastChild, parentNode
- childNodes can have childNodes.
- Leaves are text nodes,
 - Respond to 'data', which gives up the data they store.
- This is useful if you need to process an entire document, but annoying if you're searching.

Finding specific elements

- `getElementsByTagName` finds all elements according to name:

```
eltlist = doc.getElementsByTagName('key')
```

- Can search at any node

Finding attribute/value pairs

- Nodes have a dictionary-like structure that holds attribute/value pairs:

```
eltlist = doc.getElementsByTagName('key')
node1 = eltlist[0]
attrs = eltlist[0].attributes
keys = eltlist[0].attributes.keys()
```

An example

- Let's build a simple program for reading and displaying XML

```
#!/usr/bin/python

from xml.dom import minidom
import sys

doc = minidom.parse('./cdcat.xml')
def showCD(cd) :
    for item in cd.childNodes :
        if not item.nodeType == item.TEXT_NODE :
            print '<p>', item.tagName, item.firstChild.data, '<p>'

print '<html><body>'
print 'CDs in my catalog: '
cds = doc.getElementsByTagName('cd')
for item in cds:
    showCD(item)
```

- Often, you will want a more flexible way to find nodes in a tree.
 - All titles underneath a 'cd' node.
 - All song titles for songs with a rating of '5'.
- We want to be able to specify a *pattern* that names nodes of interest based on their position in the DOM tree.
- XPath is a language for doing this.

- In XPath, everything is dealt with as a path from the root of the tree.
- To find a node, we'll use a *location path*, which consists of a series of *location steps*.
- A location step consists of:
 - An axis that tells us which direction to travel
 - A node test that specifies which types of nodes apply
 - predicates that use boolean tests to help filter nodes.

Examples

- `/cd/title` - matches title nodes underneath cd nodes.
- `/cd[rating=5]/title` - matches the title of CDs with a rating of 5.
- You can use XPath expressions inside a `getElementsByTagName()` method.
- We'll return to XPath in two weeks when we study XSLT.

Outline

- About XML
- Structuring XML documents
- Validating XML with schema
- Using CSS to display XML
- Parsing with DOM
- Parsing with SAX

Parsing with SAX

- DOM is very convenient to use in many cases, but not all
 - Document is too large to hold in memory
 - Document is malformed
 - Document is being produced (and should be consumed) incrementally
- In these cases, a SAX parser may be more appropriate.

SAX: Simple API for XML

- SAX is an interface that was developed to provide an uniform way to integrate different XML parsers.
 - Interesting contrast in origin to DOM.
 - SAX developed 'bottom-up' by XML developers
 - DOM developed 'top-down' by the W3C.
- SAX is an *event-driven parser*
 - You define an event handler that is passed to the parser.
 - Describes how to handle particular types of elements.
 - Document is processed sequentially. State must be maintained by hand.

Using SAX within Python

- (Note: Java looks very similar)
- Most of the work involves creating *handlers*
- For example, to deal with processing content, override the *content handler*

```
import xml.sax
from xml.sax.handler import *
class CDHandler(ContentHandler) :
    def __init__(self) :
        self.books = []
        self.buffer = ''
        self.inTitle = False

    def startElement(self, name, attrs) :
        if name == 'title' :
            self.inTitle = True

    def endElement(self, name) :
        if name == 'title' :
            self.inTitle = False
            print self.buffer
```

Using SAX within Python

- To use this, we then register the handler with a SAX parser.

```
parser = xml.sax.make_parser()
handler = CDHandler()
parser.setContentHandler(handler)
parser.parse('cdcat.xml')
```

SAX comments

- You must keep track of 'where you are' yourself.
 - No access to the enclosing context
 - It's hard with SAX to, for example, print the corresponding artist for each title node.
- SAX has more modest memory requirements than DOM
 - Nodes are discarded after parsing
- More flexible recovery from parsing errors.
- Use the parser that best fits your needs.