

# Semi-structured Data: Programming

Introduction to Databases

CompSci 316 Fall 2019



**DUKE**  
COMPUTER SCIENCE

# Announcements (Mon., Oct. 28)

- Homework 3 due next Monday
- Project milestone 2 due next Wednesday
  - But remember your weekly update on Piazza this Wed.!

# Announcements (Wed., Oct. 30)

- Homework 3 due next Monday
- Project milestone 2 due next Wednesday
  - But remember your weekly update on Piazza today!

# APIs for working with JSON/XML

If data live in a database, use a “database API,” e.g.:

- pymongo : MongoDB ≈ psycopg2 : PostgreSQL

If data live in files or messages, then parse data into a representation in your application programming language and use APIs to access/query data:

- Low-level XML APIs: SAX, DOM
- XML APIs with query functionality
  - JavaScript DOM Level 3 (supports XPath)
  - Python lxml (supports XPath/XSLT)
  - Java Saxon (supports XPath/XQuery/XSLT)

# SAX & DOM

Both are basic API's for XML processing

- **SAX (Simple API for XML)**
  - Started out as a Java API, but now exists for other languages too
- **DOM (Document Object Model)**
  - Language-neutral API with implementations in Java, C++, python, etc.

# SAX processing model

- Serial access
  - XML document is processed as a stream
  - Only one look at the data
  - Cannot go back to an early portion of the document
- Event-driven
  - A parser generates **events** as it goes through the document (e.g., start of the document, end of an element, etc.)
  - Application defines **event handlers** that get invoked when events are generated

# A simple SAX example

- Print out text contents of title elements

```
import sys
import xml.sax
from io import StringIO

class PathHandler(xml.sax.ContentHandler):
    def startDocument(self):
        .....
    def startElement(self, name, attrs):
        .....
        .....

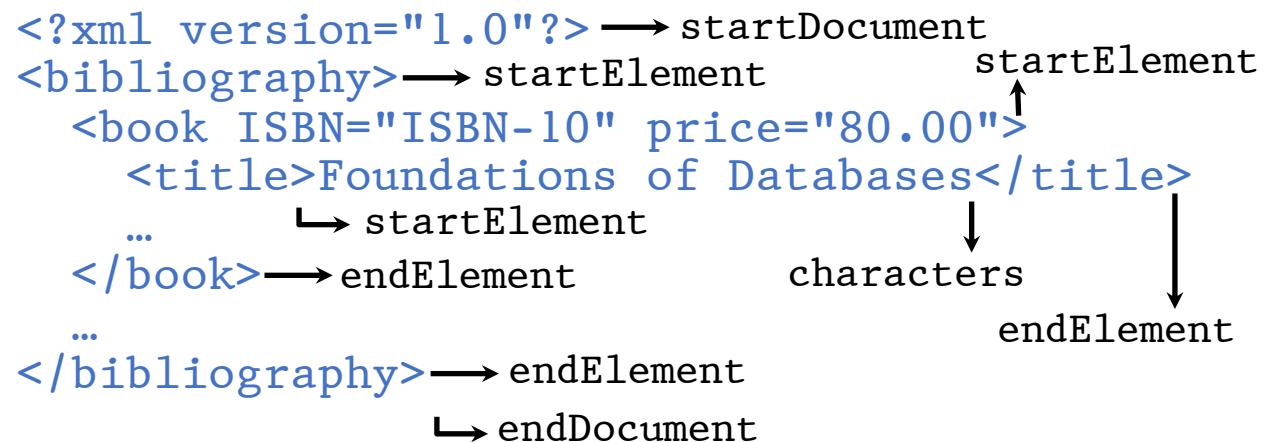
xml.sax.parse(sys.stdin, PathHandler())
```

# SAX events

Most frequently used events:

- **startDocument**
- **endDocument**
- **startElement**
- **endElement**
- **characters**

- Whenever the parser has processed a chunk of character data (without generating other kinds of events)
- Warning: The parser may generate multiple **characters** events for one piece of text



Whitespace may come up as **characters** or **ignorableWhitespace**, depending on whether a DTD is present

# A simple SAX example (cont'd)

```
def startDocument(self):
    self.outBuffer = None

def startElement(self, name, attrs):
    if name == 'title':
        self.outBuffer = StringIO()
    else:
        self.outBuffer = None
    self.outBuffer.write(name)
    self.outBuffer.write(' has attributes ')
    self.outBuffer.write(str(attrs))

def endElement(self, name):
    if name == 'title':
        print(self.outBuffer.getvalue())
    self.outBuffer = None

def characters(self, content):
    if self.outBuffer is not None:
        self.outBuffer.write(content)
```

Annotations:

- Tag name: An arrow points from the `name` parameter in the `startElement` method to the `name` parameter in the `endElement` method.
- A map from attribute names to values: An arrow points from the `attrs` parameter in the `startElement` method to the `content` parameter in the `characters` method.
- Characters read: An arrow points from the `content` parameter in the `characters` method back to the `content` parameter in the `startElement` method.

# A common mistake

What is wrong with the following?

```
def endElement(self, name):  
    # print the last chunk of chars before </title>:  
    if name == 'title':  
        print(self.chars)  
  
def characters(self, content):  
    self.chars = content
```

# A more complex SAX example

- Print out the text contents of top-level section titles in books, i.e., / /book/section/title
  - Old code would print out all titles, e.g., / /book/title, / /book/ /section/title
  - For simplicity, assume that if we have the pattern / /book/section/title/ /book/section/title, we print the higher-level title element
- Idea: maintain as state the path from the root

```
def startDocument(self):  
    self.path = list()  
    self.pathLenWhenOutputStarts = None  
    self.outBuffer = None
```

# A more complex SAX example (cont'd)

```

def startElement(self, name, attrs):
    self.path.append(name) # maintain the path
    if len(self.path) >= 3 and \
        self.path[-3:] == ['book', 'section', 'title']:
        # path matches //book/section/title:
        if self.outBuffer is None:
            self.pathLenWhenOutputStarts = len(self.path)
            self.outBuffer = StringIO()

def endElement(self, name):
    if self.outBuffer is not None and \
        len(self.path) == self.pathLenWhenOutputStarts:
        print(self.outBuffer.getvalue())
        self.outBuffer = None
    self.path.pop() # maintain the path

def characters(self, content):
    if self.outBuffer is not None:
        self.outBuffer.write(content)

```

Would it work if we remove  
this check?

Would it work if we change this  
check to name == "title"?

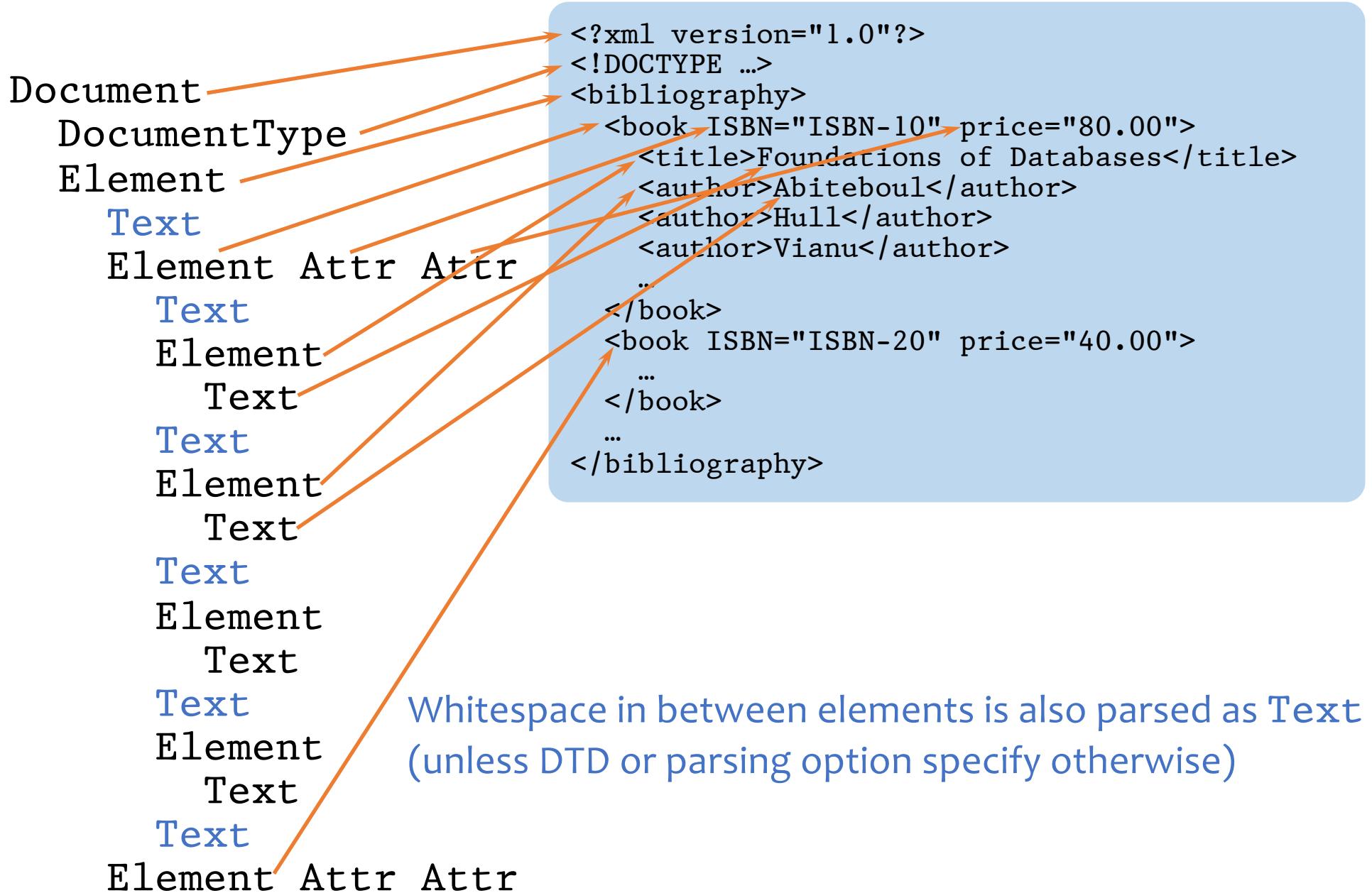
# DOM processing model

- XML is parsed by a parser and converted into an in-memory DOM tree
- DOM API allows an application to
  - Construct a DOM tree from an XML document
  - Traverse and read a DOM tree
  - Construct a new, empty DOM tree from scratch
  - Modify an existing DOM tree
  - Copy subtrees from one DOM tree to another
  - etc.

# DOM Node's

- A DOM tree is made up of Node's
- Most frequently used types of Node's:
  - **Document**: root of the DOM tree
    - Not the same as the root element of XML
  - **DocumentType**: corresponds to the DOCTYPE declaration in an XML document
  - **Element**: corresponds to an XML element
  - **Attr**: corresponds to an attribute of an XML element
  - **Text**: corresponds to chunk of text

# DOM example



# Node interface

`n.nodeType` returns the type of Node n

`n.childNodes` returns a list containing n's children

- E.g., subelements are children of an Element;  
DocumentType is a child of the Document
- `n.appendChild(c)` adds Node c as the last child of n

`d.documentElement` returns the root Element of Document d

`e.nodeName` returns the tag name of Element e

`e.attributes` returns a NamedNodeMap containing e's attributes

- Attributes are not considered children!
- Loop through attributes using  
`for i in range(e.attributes.length):`  
`a = e.attributes.item(i)`
  - `a.nodeName` returns the attribute name
  - `a.nodeValue` returns the attribute value
- Given e, `e.hasAttribute(name)`, `e.getAttribute(name)`,  
`e.setAttribute(name,value)` are also available

`t.nodeValue` returns the content of Text t

For convenience: `n.parentNode`, `n.previousSibling`,  
`n.nextSibling`, `n.ownerDocument`, etc.

# Constructing DOM from XML

```
import sys
from xml.dom.minidom import parse

dom = parse(sys.stdin)
# now print it back out:
print(dom.toprettyxml(indent=' '*4, encoding='utf-8'))
```

# Traversing DOM

- Compute the string value of an XML node

```
def nodeToString(n):  
    # string value of a Text node is just its content:  
    if n.nodeType == n.TEXT_NODE:  
        return n.nodeValue;  
    # string value of a Node of another type is the  
    # concatenation of its children's string values:  
    return ''.join(\n        nodeToString(child) |\n            for child in n.childNodes |\n    )\n}
```

# Traversing DOM

- Print out text contents of title elements

```
def outputTitle(n):
    if n.nodeType == n.ELEMENT_NODE and \
        n.nodeName == 'title':
        print(nodeToString(n))
    else:
        for child in n.childNodes:
            outputTitle(child)
```

- How do you print out just /book/section/title?
  - Use parentNode to check for section parent and book grandparent

# Constructing DOM from scratch

- Construct a DOM Document showing all titles as follows:

```
<result>
  <title text="title1"/>
  <title text="title2"/>...
</result>
```

```
from xml.dom.minidom import getDOMImplementation

def addTitles(n, newdoc):
    if n.nodeType == input.ELEMENT_NODE and \
        n.nodeName == 'title':
        e = newdoc.createElement('title')
        e.setAttribute('text', nodeToString(n))
        newdoc.documentElement.appendChild(e)
    else:
        for child in n.childNodes:
            addTitles(child, newdoc)

newdom = getDOMImplementation().\
    createDocument(None, 'result', None)
addTitles(dom, newdom)
```

# Copying subtrees in DOM

- Construct a DOM Document showing all title elements from the input XML

```
from xml.dom.minidom import getDOMImplementation

def addTitles2(n, newdoc):
    if n.nodeType == input.ELEMENT_NODE and \
        n.nodeName == 'title':
        e = newdoc.importNode(n, True) —————→ A Document can import (copy) a
        newdoc.documentElement.appendChild(e)      Node from another element; the
    else:                                         second argument specifies
        for child in n.childNodes:             whether to copy recursively or not
            addTitles2(child, newdoc)

newdom = getDOMImplementation().\
    createDocument(None, 'result', None)
addTitles2(dom, newdom)
```

# Summary: SAX versus DOM

- SAX
  - Because of one-pass processing, a SAX parser is fast, consumes very little memory
  - Applications are responsible for keeping necessary state in memory, and are therefore more difficult to code
- DOM
  - Because the input XML needs to be converted to an in-memory DOM-tree representation, a DOM parser consumes more memory
    - Lazy materialization of DOM tree helps alleviate this problem
  - Applications are easier to develop because of the powerful DOM interface
- Which one scales better for huge XML input?