Parsing XML using MATLAB®

Peter Corke

November 2018

1 Introduction

An increasing amount of data is encoded in XML format. MATLAB has built in support for parsing XML format files, but the documentation leaves something to be desired. What follows is a tutorial about XML and the MATLAB support.

2 XML 101

2.1 What's in an XML file

The most obvious feature of an XML document are the angle-bracket-delimited *tags* such as `<foo>` or `<bar>`. The file contains structured text that can be read by a person as well as being easy to parse programatically.

An XML file is frequently referred to as a *document*. The file comprises *markup* and *content*. Generally, strings that constitute markup either begin with the character `<` and end with a `>`, or they begin with the character `&` and end with a `;`. Strings of characters that are not markup are content.

2.1.1 Tags

A *tag* is a markup construct that begins with `<` and ends with `>`. Tags come in three flavours:

start-tag such as `<tag>`

end-tag such as `</tag>`

empty-element tag such as `<tag />`

where, in all cases, `tag` is the name of the tag and is given without quotes. For example `<foo>` is a tag named foo. Tag names start with a letter, underscore or colon, followed by alphanumeric, dot or underscore characters.

2.1.2 Elements

An *element* is a logical document component that either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The characters between the start-tag and end-tag, if any, are the element's *content*, and may contain markup, including other elements, which are called child elements.

An XML element has matching start and end tags like

```
<foo></foo>
```

or single empty-element tag such as

```
<bar />
```

which is equivalent to

```
<bar></bar>
```

The body of the *element* can be an arbitrary mixture of plain text or other elements, ie. elements can be nested within the body of other elements. This allows the creation of hierarchies or trees, with the first or outermost element as the root. For example:

```
<bar>some text</bar>
<foo><bar>one</bar><bar>two</bar></foo>
```

The elements must not overlap. i.e., an end tag of an element must have the same name as that of the most recent unmatched start tag.

2.1.3 Attributes

An *attribute* is a markup construct consisting of `name=value` pairs that exists within a start-tag or empty-element tag. The *attributes* are given as a name and a value. Attribute names are defined without quotation marks, whereas attribute values must always appear as a string delimited with either single or double quotes. For example

```
<img src="madonna.jpg" alt="Madonna" />
```

has two *attributes*. Their names are `"src"` and `"alt"`, and their *values* are `"madonna.jpg"` and `"Madonna"` respectively. An XML attribute can only have a single value and each attribute can appear at most once on each element. Attribute names start with a letter, underscore or colon, followed by alphanumeric, dot or underscore characters. Attribute names in XML (unlike HTML) are case sensitive.

2.1.4 Special characters &··;

Special characters are introduced using XML references. References always begin with the symbol `"&"` which is a reserved character and end with the symbol `";"`. XML has two types of references

Entity References contain a name between the start and the end delimiters. The name refers to a predefined strings such as

`&amp;`	ampersand
`&apos;`	single quote
`&gt;`	greater than
`&lt;`	less than
`&quot;`	double quote

or a larger set defined for HTML which can be found in online lists, for example, `&theta;` is the Greek letter θ.

Character References These specify a character by its Unicode code in either decimal or hexadecimal format. For example the Roman character "A" can be specified as `&#65;` in decimal or `&#x41;` in hexadecimal.

Because ampersand `"&"` indicates a reference it must always be escaped in text as `&amp;`.

2.1.5 Comments and white space

Comments are a special kind of element

```
<!-- comment stuff -->
```

and the body of the comment can also span multiple lines

```
<!--   comment
       more comment
       still more comment -->
```

Comments cannot be nested.

White space is mostly insignificant so XML files are often formatted with indenting to highlight the structure to human readers. Some editors support auto-indenting.

White space is ignored outside of quotes, so the following two lines are equivalent

```
<foo param1 = "x1" param2 = "x2"> body </tag>
<foo param1="x1" param2="x2"> body </tag>
```

2.1.6 CDATA

A CDATA block refers to character data that is considered literally. The special characters are interpreted as themselves, and *not* as markup. For example

```
<![CDATA[
   characters with markup like 3<4;
]]>
```

This section may contain markup characters (`<`, `>`, and `&`). CDATA cannot contain the string `"]]>"`. This is somewhat analogous to a LaTeX verbatim block.

2.1.7 Document

XML documents may begin with an XML declaration that describes some information about themselves. An example is

```
<?xml version="1.0" encoding="UTF-8"?>
```

which is case-sensitive and must be the first statement in the document. An XML document can have only one root element.

2.2 The DOM model

The Document Object Model (DOM) is a hierarchical or tree representation of the file called a node tree. Everything in an XML document is a node:

- The entire document is a document node
- Every XML element is an element node
- The non-markup text in the XML elements are text nodes
- Every attribute is an attribute node
- Comments are comment nodes

The document node is the root of the tree. If must have at most one (XML rule) child node. That node may have multiple children of the types mentioned above. For example the XML element `<date type="year">2005</date>` is represented by an element node `<date>` with two child nodes: an attribute node with the name `"type"` and value `"year"`, and a child text node with the value `"2005"`.

2.3 Reading XML with MATLAB

Consider a short XML file that represents employees at a company

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

  <person employee_number="123" gender="female" >
    <name>Alice</name>
    <address>11 Alice St</address>
    <commenced date="2016-01-01" />
  </person>

  < person employee_number =  "237" gender = "male" status="on vacation" >
    <name>Bob</name>
    <commenced date="2015-07-07" />
    <address>13  Bob St</address>
  </person>

</people>
```

MATLAB provides a function to read and parse an XML file and return a DOM node-tree

Node type	nodeName()	nodeType()	nodeValue()
Element	tag name	ELEMENT_NODE = 1	null
Attribute	attribute name	ATTRIBUTE_NODE =2	attribute value
Text	#text	TEXT_NODE = 3	the string
CDATA	#cdata-section	CDATA_SECTION_NODE=4	the CDATA text
Entity	entity name	ENTITY_NODE=6	null
Comment	#comment	COMMENT_NODE=8	the comment
Document	#document	DOCUMENT_NODE=9	null

Table 1: Names and values for different node types. The symbolic value in column 3 is a property of any node, ie. `node.ATTRIBUTE_NODE` is the value 2.

```
>> root = xmlread('simple.xml');
```

The type of a node is indicated by an enum

```
root.getNodeType()
ans =
    9
```

which Table 1 indicates is a document node. The Table also shows the *name* and *value* associated with different types of node. These are the display value of the variable, shown when it is evaluated without a semicolon on the end of the line

```
>> root
root =
[#document: null]
```

This root node has at least one child node

```
>> root.hasChildNodes
ans =  logical
   1
```

and we can get a list of the child nodes

```
>> rootChildNodes = root.getChildNodes;
>> rootChildNodes.getLength
ans =
     1
```

The first (and only) child node from this list is

```
>> node = rootChildNodes.item(0)
ans =
[people: null]
>> node.getNodeType
ans =
     1
```

which is an element node and the display value shows the tag name: people.

The syntax looks quite unMATLAB-like and that's because the implementation is based on the standard Java DOM processor and the objects are native Java objects. Elements in the list are indexed starting at zero, and we use properties and methods of the Java objects.

This node has a name, the tag name, but no value

```
>> node.getNodeName
ans =
people

>> node.getNodeValue
ans =
     []
```

The people node has five children

```
>> persons = node.getChildNodes();
>> persons.getLength()
ans =
     5
```

which is somewhat surprising since the people node has only two children, the two person nodes: Alice and Bob. We can list the child nodes

```
>> for i=0:node.getLength-1
   persons.item(i)
 end

ans =
[#text:

    ]

ans =
[person: null]

ans =
[#text:

    ]

ans =
[person: null]

ans =
[#text:

]
```

and what we've overlooked is the non-markup text in the XML file, the three blank lines. Since we are only interested in the `person` nodes we can extract them specifically[1]

```
>> persons = node.getElementsByTagName('person');
>> persons.getLength
ans =
     2
```

The first person is

```
>> person = persons.item(0)
person =
[person: null]
>> nodes = person.getChildNodes;
>> nodes.getLength
ans =
     7
```

which has seven child nodes. Once again, we see that this is due to white space in the file

```
>> for i=0:nodes.getLength-1
    nodes.item(i)
end

ans =
[#text:
      ]

ans =
[name: null]

ans =
[#text:
      ]

ans =
[address: null]

ans =
[#text:
      ]

ans =
[commenced: null]

ans =
[#text:
    ]
```

The address node could be obtained by

```
>> address = nodes.item(3)
address =
[address: null]
```

but the index could vary depending on the formatting (white space) in the XML file. It can be obtained more robustly by

```
>> address = nodes.getElementsByTagName('address').item(0)
address =
[address: null]
```

noting that the `.item(0)` is required because the `getElementsByTagName` method always returns a list. The address node has just one child which is a text node whose value is Alice's address

```
>> txt = address.getChildNodes.item(0)
txt =
[#text: 11 Alice St]
>> txt.getNodeValue
ans =
11 Alice St
```

The person node also has attributes. We can obtain a list of those by

```
>> person.hasAttributes
ans =  logical
   1
>> attributes = person.getAttributes;
>> attributes.getLength
ans =
     2
```

The first attribute

```
>> att0 = attributes.item(0)
att0 =
employee_number="123"
```

has a display value that shows both its name and its value. We can get those strings by

```
>> att0.getNodeName
ans =
employee_number

>> att0.getNodeValue
ans =
123
```

Attributes can also be obtained directly

```
>> person.getAttribute('employee_number')
ans =
123
```

This is all somewhat tedious to do manually but it lends itself to a programmatic solution.

3 Conclusions

We have demonstrated the main parts of the MATLAB API for parsing XML files. More detail can be found online in tutorials and references for the Java DOM Parser.