CMSC 420 - Processing XML Files for the Programming Project

The input and output for our programming project will be as XML documents. We will provide you with some useful utilities for processing XML documents. Before discussing the utility procedures, let us present a brief introduction to XML.

1 A Brief Introduction to XML

XML stands for the *eXtensible Markup Language*. It is a standard for encoding data and documents in a format that is both human-readable and machine-readable. Suppose, for example, you are creating a database for a bookstore, and you wish to encode the following information:

Book:	Category: Novel; Format: Paperback	
	Title:	Great Expectations (Language: English)
	Author:	Charles Dickens
	Year:	1860
	Price:	\$6.48 (Currency: US Dollars)

This might be encoded in XML as follows:

```
<book category="novel" format="paperback">
  <title lang="en">Great Expectations</title>
  <author>Charles Dickens</author>
  <year>1860</year>
  <price currency="usd">26.48</price>
</book>
```

The strings within angled brackets "`<...>`" are called *tags*. There are three types of tags: *opening tags* (such as `<title>`), *closing tags* (such as `</title>`). It is also possible to have *empty-element tags* (such as `<line-break/>`).

The stuff enclosed within an opening and closing tag (or just an empty-element tag itself) is called an *element*. Here are three examples of elements:

```
<year>1860</year>
<par>Hello, world!</par>
<add-contact name="Emily Dickinson" phone="240-555-1212"/>
```

Elements may be nested within one another, and we use basic tree terminology (child, parent, sibling) when referring to such elements. In the above example `title`, `author`, `year`, and `price` are siblings of each other and all are children of `book`.

It is possible to provide additional information through name–value pairss, called *attributes*, in start tags and empty-element tag. Examples include `category="novel"` and `currency="usd"` from above. In this case, "category" and "currency" are the *names* of the attributes and "novel" and "usd" are the respective *values*. Values can come in various types, including strings, numbers, file names, hyperlinks, etc.

1.1 XML Document Structure

An XML file has a standard structure. It is not required, but most XML document begin with a declaration that provides general information about the document. For example, the following indicates the version of XML used and the character encoding.

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

Because XML data is naturally tree based, the tree should have a unique root element. For example, our bookstore example above might have the following global structure (where `<bookstore>` is the root):

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="novel" format="paperback">
    <title lang="en">Great Expectations</title>
    <author>Charles Dickens</author>
    <year>1860</year>
    <price currency="usd">26.48</price>
  </book>
  <book category="novel" format="audiobook">
    <title lang="fr">Les Misérables</title>
    <author>Victor Hugo</author>
    <year>1862</year>
    <price currency="eur">19.76</price>
  </book>
  ...
</bookstore>
```

1.2 Schema and Error Checking

In order to allow error-checking in an XML file, it is possible to provide a definition of what tags and attribute names and values are allowed. Such a file is called an *XML Schema Definition* or XSD. We will be providing you with a schema definition for the input files for each part of the programming project. While you do not need to understand the contents of the schema file, when creating your own test files you should be mindful to abide by the schema's rules.

For example, our programming project will involve processing objects on a map called *cities*. Each city element will have a name (like "Chicago"), a location specified by (x, y)-coordinates, a radius to indicate its size, a color (for illustration purposes), and an optional numeric identifier. Here is a typical command for creating a new city for our data structure:

```
<createCity name="Baltimore" x="76" y="39" radius="10" color="green"/>
```

In order to check that this is a valid XML input, we will provide you with an XSD that contains (among other things) the following definition of what the `createCity` tag expects for its arguments:

```
<xs:complexType name="cityType">
<xs:attribute name="name" type="cityName" use="required"/>
<xs:attribute name="x" type="xs:integer" use="required"/>
<xs:attribute name="y" type="xs:integer" use="required"/>
<xs:attribute name="radius" type="xs:integer" use="required"/>
```

```
<xs:attribute name="color" type="colorType" use="required"/>
</xs:complexType>
```

This indicates that the city name is a string, the x, y coordinates and radius value are integers, and the color is of a type `colorType` (defined elsewhere in the schema). All of these elements are required.

2 CMSC 420 XML Utilities

As mentioned above, the input to your project will be presented as an XML document. We provide you file `420util.jar` that contains (among other things) some XML utilities for inputting and outputting XML document files. To access these functions, include the following line at the start of your source file.

```
import cmsc420.xml.XmlUtility;
```

Each XML element in the input corresponds to a command for your program to process, and each element in the output file will be a summary of the result of processing the command. For example, the following XML elements ask your program to create two city objects and then list them.

```
<createCity name="Baltimore" x="76" y="39" radius="10" color="green"/>
<createCity name="Chicago" x="87" y="41" radius="15" color="blue"/>
<listCities sortBy="coordinate"/>
```

Before explaining how your program will process this input, we should say a bit about Java's *Document Object Model* (DOM). This is a Java package that provides a number of functions for manipulating XML documents. Our XML utilities provide a few additional functions to simplify your use of the DOM functions.

The DOM represents an XML document as an object, called `Document`. The items contained with a document are stored as `Node` objects. From our perspective, the most important type of node will be an `Element` object, which stores our XML elements. For further information about the DOM, see:

https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/package-summary.html

2.1 Processing and XML Input Document

In order to process an XML document, we need to (1) parse the input document, validate its structure and map it into DOM's internal tree-based format, (2) access individual elements from the document and their attributes, and (3) understand how to inform the DOM parser of the schema we will be applying.

Parsing and validating the document: As mentioned above, a document is represented internally as a `Document` object. Our XML utilities provide a function, `validateNoNamespace`, which inputs the XML file, tests it for validity, and returns the associated document. Once the document has been input, we can use DOM functions such as `getDocumentElement` to access the root element and `getChildNodes` to obtain the children of the root. For our project, each of these elements contains a command for your program to process. This process is illustrated in the following code.

```
Document input = XmlUtility.validateNoNamespace(System.in); // read & validate input
Element rootNode = input.getDocumentElement();              // get root node
NodeList nl = rootNode.getChildNodes();                     // get list of commands
for (int i = 0; i < nl.getLength(); i++) {                  // run through the list
    if (nl.item(i).getNodeType() == Document.ELEMENT_NODE) {// ignore XML comments
        Element command = (Element) nl.item(i);             // access next command
        // ... process the command
```

Note that the function `validateNoNamespace` throws an exception if the file does not have the proper format. Thus, this code should all be placed within `try-catch` block.

Accessing elements and their attributes: Now that we have seen how to access the document's elements, let us see how to determine the element's tags and access its various elements. The DOM provides the following useful functions. Recall that `Node` is the parent type of `Element`, so all these functions apply to `Element` objects:

- `String Node::getNodeName()`: Get an node's name (tag)
- `String Node::getNodeValue()`: Get an node's value
- `Node Node::appendChild(Node newChild)`: Append a new child to given parent node
- `String Element::getAttribute(String name)`: Get the attribute of the given name
- `boolean Element:hasAttribute(String name)`: Test whether the attribute exists
- `void Element::setAttribute(String name, String value)`: Sets attribute value

As an example, consider `createCity` element:

```
<createCity name="Baltimore" x="76" y="39" radius="10" color="green"/>
```

The following code accesses the information associated with this element:

```
Element command = // ... see above
if (command.getNodeName().equals("createCity")) {           // true
    String name = command.getAttribute("name");             // name = "Baltimore"
    float x = Float.parseFloat(command.getAttribute("x"));  // x = 76
    float y = Float.parseFloat(command.getAttribute("y"));  // y = 39
    // ... and so on
}
```

Where is the schema defined? How does the program know which schema to use for the validation? This is indicated in the first few lines of the input file, which we omitted above. Here is the full input file for this small example:

```
<commands
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="part1in.xsd"
    spatialWidth="512"
    spatialHeight="512">
 <createCity name="Baltimore" x="76" y="39" radius="10" color="green"/>
 <createCity name="Chicago" x="87" y="41" radius="15" color="blue"/>
 <listCities sortBy="coordinate"/>
</commands>
```

The line "`xsi:noNamespaceSchemaLocation="part1in.xsd"`" tells the XML utilities to look at the file `part1in.xsd` for the schema. We will provide you with this file.

2.2 Generating and Outputting an XML Document using DOM

In addition to provided functions for inputting XML files, our XML utilities and the DOM also provide methods for outputting XML documents. The process involves the following steps:

- Generate a new `Document` object
- Create a root object for the document
- For each child element of this root:
  - Create a new element
  - Set its attributes
  - Append this element to the root

The following DOM functions are the ones that will be relevant for the project. The first creates a new `Document` object:

```
Document myDocument = XmlUtility.getDocumentBuilder().newDocument();
```

To create a new `Element` object:

```
Element elt = myDocument.createElement("elementName");
```

To set one of the element's attributes:

```
elt.setAttribute("attributeName", "attributeValue");
```

Test whether an element contains a given attribute (returns a boolean):

```
elt.hasAttribute("attributeName")
```

To append the element `elt` as a child of an existing element `parent`:

```
parent.appendChild(elt);
```

Finally, once our document is created, we provide a utility function that "pretty prints" it to `System.out`:

```
XmlUtility.print(myDocument);
```

Through an appropriate combination these, we can generate and print an XML document. For example, suppose that we wanted to create an XML document that stores a contact list with the hierarchical structure shown in Fig. 1.

To do this, we will first create the document (`myDocument`), and add a root element (`root`) and then add child to the root containing the contact list (`contactList`). Then, we create the two entries, set the attributes for each, and attach them as children to the contact list. Finally, we print the resulting document. (This is straight-line Java code, but I have added indentation to make it more readable.)

```
Document myDocument = XmlUtility.getDocumentBuilder().newDocument();
Element root = myDocument.createElement("results");
myDocument.appendChild(root);
  Element contactList = myDocument.createElement("my-contacts");
  root.appendChild(contactList);
```

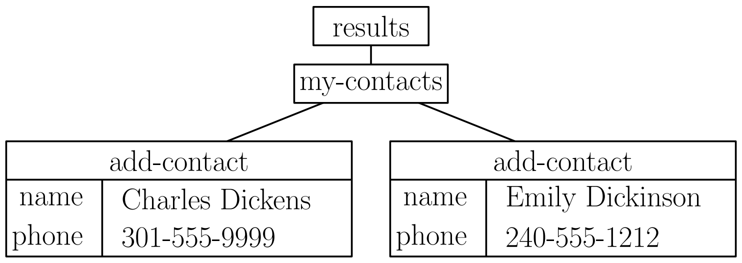

Figure 1: Tree-structure for XML contact-list document.

```
    Element contact1 = myDocument.createElement("add-contact");
    contact1.setAttribute("name", "Charles Dickens");
    contact1.setAttribute("phone", "301-555-9999");
  contactList.appendChild(contact1);
    Element contact2 = myDocument.createElement("add-contact");
    contact2.setAttribute("name", "Emily Dickinson");
    contact2.setAttribute("phone", "240-555-1212");
  contactList.appendChild(contact2);
XmlUtility.print(myDocument);
```

The output of this code is shown below. Notice that our print command automatically generates a header line for the XML document, it nicely indents nested elements.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<results>
  <my-contacts>
    <add-contact name="Charles Dickens" phone="301-555-9999"/>
    <add-contact name="Emily Dickinson" phone="240-555-1212"/>
  </my-contacts>
</results>
```