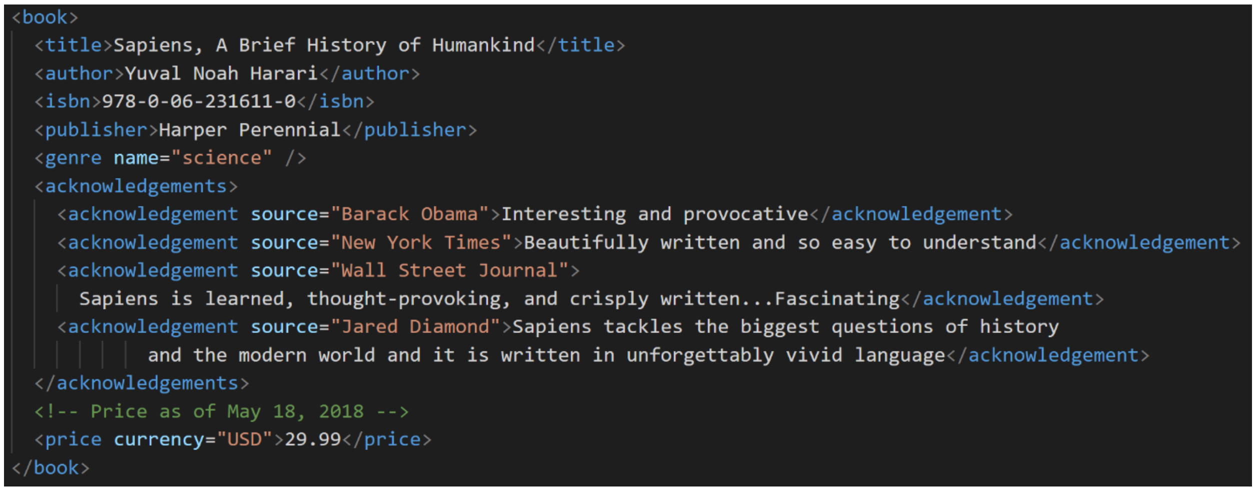
Module 1 - Creating XML

Overview

XML (*Extensible Markup Language*) is a markup language that is widely used in many computer systems, web services, and for exchanging data. JSON plays a similar role and will be discussed in a future module. In this module, we will concentrate on XML. We will not discuss every aspect of XML for 2 reasons: (a) it is way too big to fit in a single module. (b) We will be duplicating the effort of many excellent tutorials that you can find on the web. However, we will touch on the basics to enable you to create XML and query it back.

XML is human-readable and machine-readable and writeable. XML in itself doesn't do anything. You define the data you want to put in it. And you can put anything as long as you don't break XML syntax rules. To get started, let's look at the anatomy of an XML document and define some key terminology and concepts that you need to know (Figure 1).



```xml
<book>
  <title>Sapiens, A Brief History of Humankind</title>
  <author>Yuval Noah Harari</author>
  <isbn>978-0-06-231611-0</isbn>
  <publisher>Harper Perennial</publisher>
  <genre name="science" />
  <acknowledgements>
    <acknowledgement source="Barack Obama">Interesting and provocative</acknowledgement>
    <acknowledgement source="New York Times">Beautifully written and so easy to understand</acknowledgement>
    <acknowledgement source="Wall Street Journal">
      Sapiens is learned, thought-provoking, and crisply written...Fascinating</acknowledgement>
    <acknowledgement source="Jared Diamond">Sapiens tackles the biggest questions of history
            and the modern world and it is written in unforgettably vivid language</acknowledgement>
  </acknowledgements>
  <!-- Price as of May 18, 2018 -->
  <price currency="USD">29.99</price>
</book>
```

Figure 1: An example XML document containing information about a book. This document can be a string or saved as a file (e.g., book.xml).

One interesting and powerful property of XML is that, unlike HTML, there are no predefined tags such as <html>, <h2>, <br>, etc. that have special meanings to parsers (e.g., a browser that renders HTML). Instead, you define your own tags that suit your needs. For example, in the XML of Figure 1 we chose tags that are relevant to describe a book. We would have come up with different tags if our XML document was describing the characteristics of a different object or concept (e.g., a computer, an order, etc.)

The words that appear in blue in Figure 1 are called **elements**. For example title, isbn, acknowledgement, and acknowledgements are all examples of elements. Notice that every element has a closing element. The closing element follows this syntax: </element_name> (note the forward slash). For example, consider the isbn element:

```
<isbn>978-0-06-231611-0</isbn>
```
.

<isbn>:   is the **opening element**.
</isbn> is the **closing element**.
The text between the opening and closing elements, 978-0-06-231611-0, is the **element value**.

An alternative way of closing an element is by including / before the closing > character of the opening element. This is shown in element genre. In this case the genre element doesn't have any value but has an attribute called "name" (more on attributes later).

```
<genre name="science" />
```

An element can have attributes. For example, consider the price element which has a currency attribute:

```
<price currency="USD">29.99</price>
```

currency is the **attribute name**.
USD is the **attribute value**

Note that an element can have more than one attribute. For example, let's add a second attribute to the price element to indicate if discounts are available for large orders:

```
<price currency="USD" largeOrdersDiscount="yes">29.99</price>
```

One interesting element in any XML document is the **root element** (also known as **document element**). The root element is the outer element that encloses the whole document. In Figure 1, <book> is the root element because all other elements are children of it. Although not used in Figure 1, the root element can have one or more attributes (just like any other element). Just as the root element can have children elements (e.g., title, author, isbn are all children elements of the book element), any internal element can also have children. For example, in Figure 1 the <acknowledgements> element has four <acknowledgement> children elements. In this case the children of the <acknowledgements> element are all <acknowledgement> elements, but they don't have to. Remember that you are defining the data and structure and you can organize data (the elements) in any way that suits your needs.

You can add comment in XML by using **<!--** comment goes here **-->**. XML parsers ignore anything in between <!-- and -->. In Figure 1, notice the comment:

```
<!-- Price as of May 18, 2018 -->
```

XML is **case-sensitive**. That is, <isbn>, <Isbn>, and <ISBN> are three different elements. As a result, you cannot have a closing element that doesn't match the case of its opening element. For example, the following is incorrect XML:

```
<!--Incorrect syntax -->
<language>Java</Language>.
```

The reason it is incorrect is because <language> doesn't have a closing element. The element <Language> (with capital L) is not the closing element because case-sensitivity makes it such that <language> and <Language> are two different elements.

Well-formed vs. **valid** XML:

An XML is **well-formed** if it adheres to the W3C requirements. Or in other words, it does not have syntax errors.

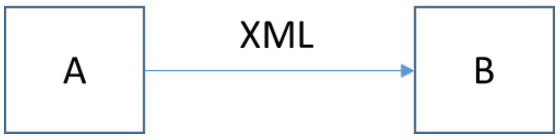
But a well-formed XML might not necessarily be **valid**, in the sense that it might have missing data that you expect. Consider for example two components A and B that exchange XML data (A creates the XML and sends it to B). Assume that B expects the ISBN of a book to complete its work. However, A did not include an <isbn> element in the XML it sent to B. In this case A sent well-formed XML (no syntax errors) but not **valid** (it didn't include an element that component B was expecting).

Document Type Definition (**DTD**):

Continuing with the above example, how can B checks that A is following the rules and adhered to what B is expecting? Or in other words, we want a consistent way for B to reject an XML that doesn't have the data and structure it is expecting. This is what a **Document Type Definition** (**DTD**) allows us to do. The DTD defines the legal structure of XML. You then validate any incoming XML against this DTD and reject all that do not pass the check. In a way, the DTD defines the grammar of the XML. For example, you can define in a DTD that the XML must contain a <price> element and the <price> element must have a currency attribute. In such a case if B receives the element <price>29.99</price>, it will reject it because such XML fails to validate against the DTD (does not have the currency attribute).

When to use a DTD?

Let's continue with the A sending XML to B example:



If A and B are written by the same person/team, you can probably do away with not using a DTD since A has intimate knowledge of what B wants and B trusts A. However, you can still use a DTD to catch innocent mistakes that you might make. However, B should always validate XML against a DTD if say A is an external system that is not under the control of B. In this case, B needs to be defensive and validate anything that A sends it. In practice, developers tend to not use a DTD for small XML since B can easily do the check without the assistance of a DTD. For example consider this small XML where B expects both an id and a quantity values to fulfill an order:

```
<order>
   <item id="22334455" quantity="1"/>
</order>
```

Now assume that A makes a mistake and sends the following to B:

```
<order>
   <item quantity="1"/>
</order>
```

In this case B can easily detect that the id attribute is missing and reject the XML. When the XML is large and complicated a DTD becomes necessary.

Where is the DTD defined?

The DTD can be defined inside the XML itself (known as **Internal DTD**) or as a separate file (known as **External DTD**). There are tools (desktop and online ones) that generate the DTD for you. You give them valid XML and they will output the corresponding DTD. For example, the DTD of the order XML looks like this:

```
<!ELEMENT order (item)>
<!ATTLIST order
  xmlns CDATA #FIXED ''>

<!ELEMENT item EMPTY>
<!ATTLIST item
  xmlns CDATA #FIXED ''
  id CDATA #REQUIRED
  quantity CDATA #REQUIRED>
```

How to Include in XML Data That Itself May Contain Characters That Make the XML Not well-formed?

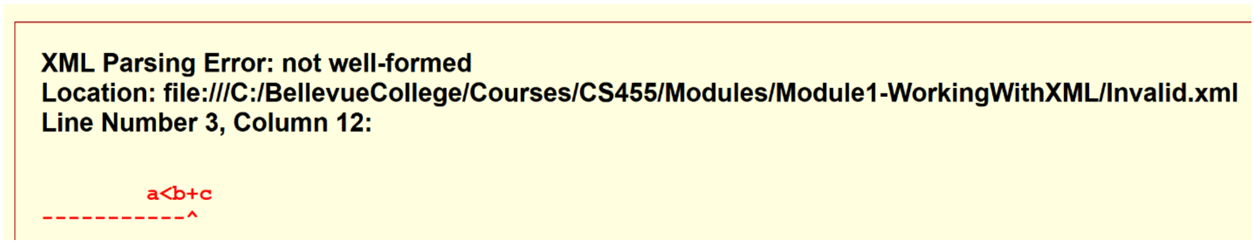
Assume for example that A wants to send the following data (a series of Math equations) to B:

a<b+c
x+y>z

How can this be done? The following won't work:

```
<data>
    <equation>
        a<b+c
        x+y>z
    </equation>
</data>
```

XML parsers (not knowing what our data mean) will get confused and think that <b is an element. If you save the above XML in a file and drag that file into Firefox, you get the below (what we expected – XML that is not well-formed):



```
XML Parsing Error: not well-formed
Location: file:///C:/BellevueCollege/Courses/CS455/Modules/Module1-WorkingWithXML/Invalid.xml
Line Number 3, Column 12:

        a<b+c
-----------^
```

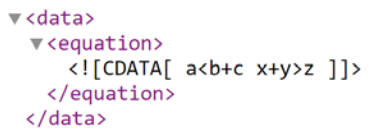
The solution to the above is to use a special element called **CDATA section** (definition found here and pasted below)

```
[Definition: CDATA sections may occur anywhere character data may occur; they
are used to escape blocks of text containing characters which would otherwise
be recognized as markup. CDATA sections begin with the string " <![CDATA[ "
and end with the string " ]]> ":]
```

Let's fix the above and include the Math equations inside a CDATA section. Our XML should now look like the below.

```
<data>
    <equation>
        <![CDATA[
        a<b+c
        x+y>z
        ]]>
    </equation>
</data>
```

And the browser now happily loads the XML:



```
▼<data>
  ▼<equation>
     <![CDATA[ a<b+c x+y>z ]]>
   </equation>
 </data>
```

There are more to XML than the above 5 pages. We just covered the basics, which is enough for us to experiment with it programmatically, what we will do next.

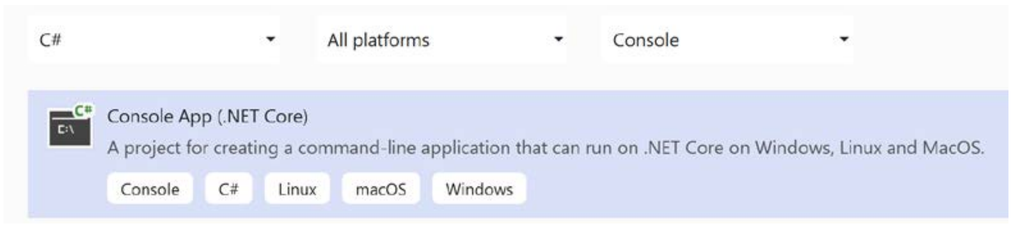
Creating XML

Before you start on the items below, make sure you have completed the following 2 module files:

A. InstallVisualStudioAndAWSToolkit.pdf
B. VisualStudioSolutionsAndProjects.pdf

Resume below after completing the above 2 PDF:

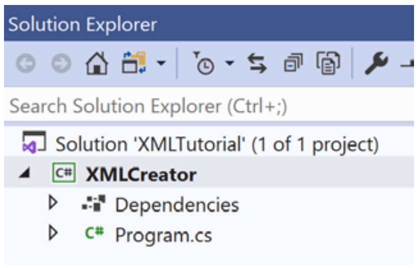
1. Open Visual Studio and create a solution named XMLTutorial.
2. To the above solution, add a project of type Console App (.NET Core). Name the project XMLCreator.



C# | All platforms | Console

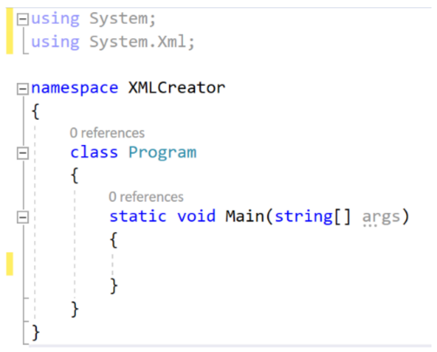
Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Console | C# | Linux | macOS | Windows

Your Solution Explorer should now look like this:



Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'XMLTutorial' (1 of 1 project)
XMLCreator
Dependencies
Program.cs

3. Double-click the program.cs file to open it. Remove the printing of Hello World and add the namespace System.Xml at the top of the file:

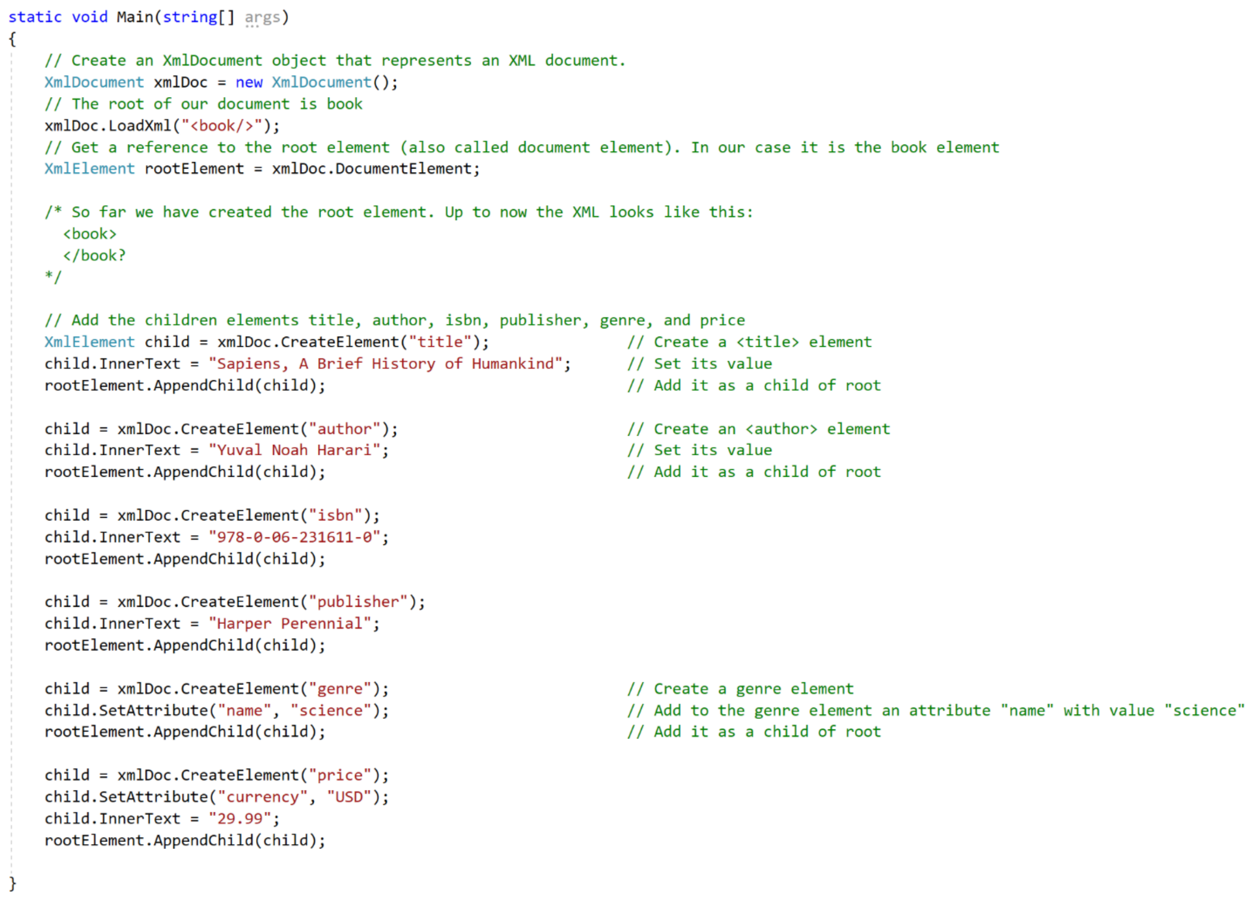


```
using System;
using System.Xml;

namespace XMLCreator
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {

        }
    }
}
```

4. We will now use classes in the System.Xml namespace to create the XML document shown in Figure 1 on page 1. NOTE: you should never create XML by concatenating strings (what most beginners working in XML do). Type the following while reading the comments and trying to understand how we are using various objects to construct the XML of Figure 1.



```
static void Main(string[] args)
{
    // Create an XmlDocument object that represents an XML document.
    XmlDocument xmlDoc = new XmlDocument();
    // The root of our document is book
    xmlDoc.LoadXml("<book/>");
    // Get a reference to the root element (also called document element). In our case it is the book element
    XmlElement rootElement = xmlDoc.DocumentElement;

    /* So far we have created the root element. Up to now the XML looks like this:
      <book>
      </book?
    */

    // Add the children elements title, author, isbn, publisher, genre, and price
    XmlElement child = xmlDoc.CreateElement("title");             // Create a <title> element
    child.InnerText = "Sapiens, A Brief History of Humankind";    // Set its value
    rootElement.AppendChild(child);                               // Add it as a child of root

    child = xmlDoc.CreateElement("author");                       // Create an <author> element
    child.InnerText = "Yuval Noah Harari";                        // Set its value
    rootElement.AppendChild(child);                               // Add it as a child of root

    child = xmlDoc.CreateElement("isbn");
    child.InnerText = "978-0-06-231611-0";
    rootElement.AppendChild(child);

    child = xmlDoc.CreateElement("publisher");
    child.InnerText = "Harper Perennial";
    rootElement.AppendChild(child);

    child = xmlDoc.CreateElement("genre");                        // Create a genre element
    child.SetAttribute("name", "science");                        // Add to the genre element an attribute "name" with value "science"
    rootElement.AppendChild(child);                               // Add it as a child of root

    child = xmlDoc.CreateElement("price");
    child.SetAttribute("currency", "USD");
    child.InnerText = "29.99";
    rootElement.AppendChild(child);

}
```

Note that the order in which elements are added and appear is not important to XML. For example, Figure 1 shows that the <acknowledgements> element and its children are before the <price> element. But in the code above we added the <price> element before <acknowledgements>. In general, the following 2 XML are identical:

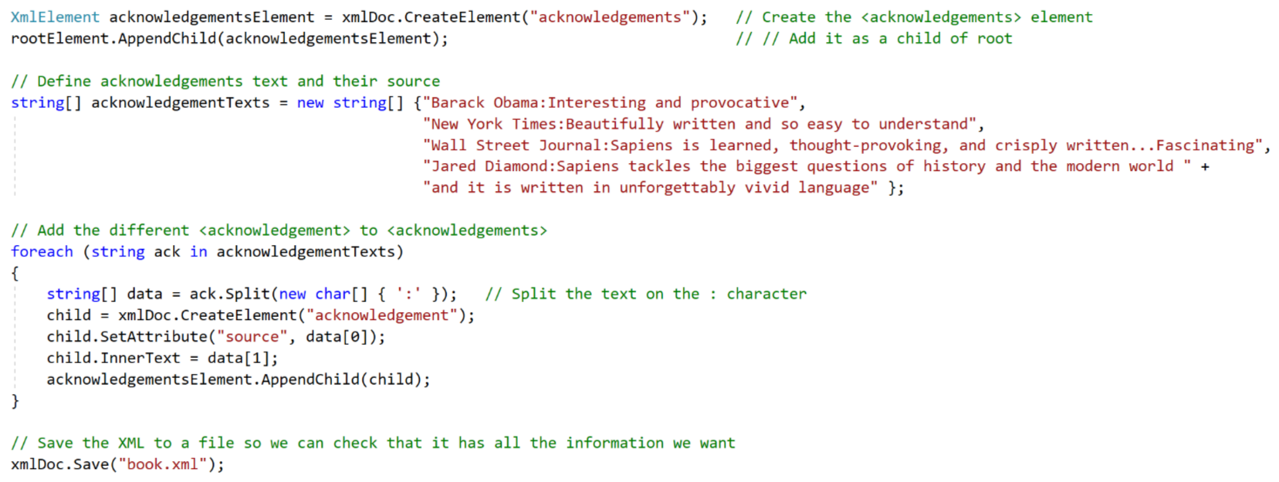
```
<order>
    <item>xyz</item>
    <quantity>2</quantity>
</order>

<order>
    <quantity>2</quantity>
    <item>xyz</item>
</order>
```

Both encode the same information. It does not matter that in the top one <item> comes before <quantity> while in the bottom one it is the reverse. **You should never design XML where the order in which elements appear in a document is meaningful to you**. If order is important, then you should encode the ordering information in a different way. For example, in the XML below the order attribute is used to indicate some ordering that is meaningful to my application. In this case I can query the order attribute to know that the blue house (order=1) comes before the brown house (order=2), which comes before the grey house (order=3).

```
<street>
   <house order=”1”>blue</house>
   <house order=”3”>grey</house>
   <house order=”2”>brown</house>
</street>
```

5. Let’s now continue and add the remaining elements (the <acknowledgements> element and its children). Add the following code to what you already have:



```
XmlElement acknowledgementsElement = xmlDoc.CreateElement("acknowledgements");   // Create the <acknowledgements> element
rootElement.AppendChild(acknowledgementsElement);                                // // Add it as a child of root

// Define acknowledgements text and their source
string[] acknowledgementTexts = new string[] {"Barack Obama:Interesting and provocative",
                                               "New York Times:Beautifully written and so easy to understand",
                                               "Wall Street Journal:Sapiens is learned, thought-provoking, and crisply written...Fascinating",
                                               "Jared Diamond:Sapiens tackles the biggest questions of history and the modern world " +
                                               "and it is written in unforgettably vivid language" };

// Add the different <acknowledgement> to <acknowledgements>
foreach (string ack in acknowledgementTexts)
{
    string[] data = ack.Split(new char[] { ':' });   // Split the text on the : character
    child = xmlDoc.CreateElement("acknowledgement");
    child.SetAttribute("source", data[0]);
    child.InnerText = data[1];
    acknowledgementsElement.AppendChild(child);
}

// Save the XML to a file so we can check that it has all the information we want
xmlDoc.Save("book.xml");
```

The last line (xmlDoc.Save()) saves the content of the document you have been building (represented by xmlDoc) into a file. Look in some subdirectory of the bin folder of your project to find book.xml.

6. Drag book.xml into a browser and see if it can open it without errors. Browsers are great to quickly test if XML is well-formed. If the XML is not well-formed, the browser fails to display it, and often tells you where the problem is. This is better than opening it in a text editor because a text editor doesn't check its well-formedness.

   In the next module we will practice how to search XML for specific data.

Exercise to Complete

1. Enclosed in the module folder is a file called "catalog.xml"
2. Add to the XMLTutorial solution a second project (also of type Console App (.NET Core)) and name it XMLCreator2.
3. Write code that creates the content of "catalog.xml".
4. Running your program should output file "catalog.xml". That is, if I run your program from the command like as shown below, it should output "catalog.xml" in the same directory where XMLCreator2 is running from.

   ```
   command prompt >  XMLCreator2.exe
   ```

   your program should output file catalog.xml

5. OPTIONAL work: look on the web and find some small XML file that is a bit more complicated than catalog.xml (maybe it has more children nodes, and children of children, etc.) and practice creating it.

What to Submit

Nothing to submit. Complete quiz Module1-Quiz.