Introduction to XML

Introduction to XML

Welcome to OST's Introduction to XML course! In this course, you will learn the fundamentals of XML for use with XML-enabled applications or general web use.

Course Objectives

When you complete this course, you will be able to:

- demonstrate knowledge of XML, SGML, and HTML.
- find and fix syntax errors in XML elements and attributes.
- create and declare a Document Type Definition (DTD) in an XML file.
- build schemas, namespaces, restrictions, and data types in XML.
- perform tranformations using advanced XSL.
- navigate an XML document with XPath.
- develop a personal information manager using XML, XSL, DTD and HTML.

From beginning to end, you will learn by doing your own XML-based projects. These projects, as well as the final project, will add to your portfolio and will contribute to certificate completion. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

How to Learn Using O'Reilly School of Technology Courses

Welcome to the O'Reilly School of Technology (OST) XML Course. Since this may be your first course with us, we'd like to tell you a little about our teaching philosophy. We believe in a hands-on, practical approach to learning.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from

misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.

- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Note Understanding HTML is a prerequisite of this XML course. If it has been a while since you've used HTML, or you just need to brush up, check out our Introduction to HTML and CSS course.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```

The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.

```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

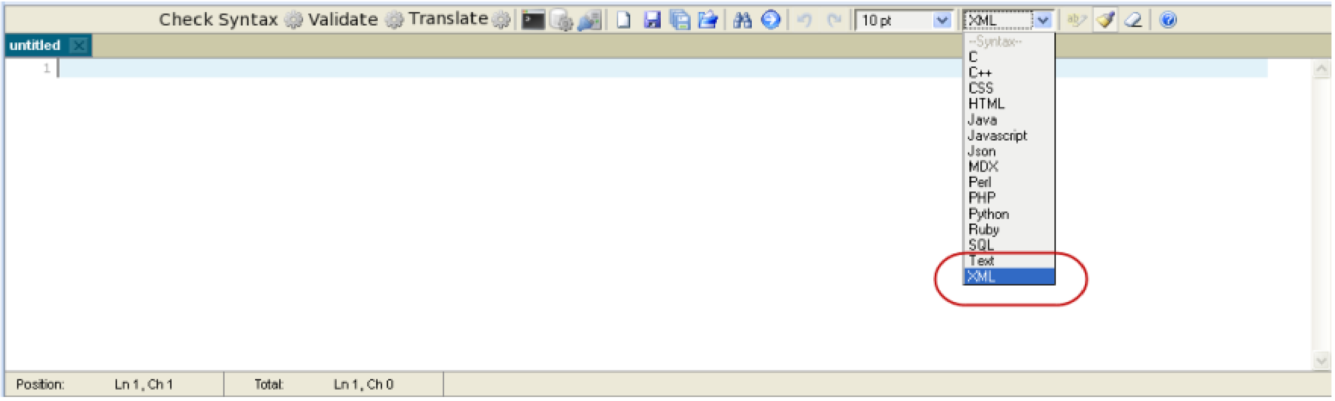
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

Setting XML Mode

Since CodeRunner is a multi-purpose editor, you'll need to use the correct *Syntax*. In this course, you will use **XML** for xml, xsl, and xsd files. To change to XML, select it from the drop-down menu as shown:



Position: Ln 1, Ch 1 Total: Ln 1, Ch 0

Throughout the course, we will give you more tips for using the learning environment effectively.

What is XML?

XML is essentially an agreement among people to store and share textual data using standard methods.

- XML stands for e**X**tensible **M**arkup **L**anguage.
- XSL stands for e**X**tensible **S**tylesheet **L**anguage.

Data is stored in XML documents. XSL documents describe how to change XML documents into other kinds of documents (such as HTML, TXT or even XML.) The process of transformation is called XSLT, or sometimes XSL transformations.

For example, suppose I gave you the following information:

Alex Chilton
52
Arkansas

Given just this information, you wouldn't know what it was for or what it meant. XML provides a way to *mark up* this data so that it can be interpreted by other people, as well as other computer programs. So, the data above might be marked up like this:

```
<PLAYERINFO>
<PLAYERNAME>Alex Chilton</PLAYERNAME>
<PLAYERNUMBER>52</PLAYERNUMBER>
<PLAYERUNIVERSITY>Arkansas</PLAYERUNIVERSITY>
</PLAYERINFO>
```

Now you know a lot more about the meaning of this information. After the XML tags are added, you can tell that Alex Chilton is some kind of sports player, that his number is 52, and that he plays for the University of Arkansas. It might look a lot like HTML to you, but the markup above doesn't work exactly like HTML. XML doesn't tell us how content should look; instead, it gives it context and meaning.

A History of XML and SGML

XML is a product of the Generalized Markup Language (**GML**) that IBM created in the 1960s. GML was created to provide a standard way to mark up textual documents for portability across different platforms. This proved to be a difficult task, but after nearly 20 years of research, IBM's work was adopted by the International Organization for Standards (or **ISO**—nope, not **IOS**, I don't know why.) ISO adopted IBM's GML and called it **SGML**, or Standard Generalized Markup Language. SGML was the basis for complex documentation systems. Unfortunately SGML is a little *too* complex for general use (kinda funny, isn't it?).

HTML was created in the early 1990s as a subset of SGML. It was designed to be a simple markup language to facilitate the transfer of web-based documents. But HTML's simplicity can be limiting. Consider the different HTML that is sometimes required to display similar content on different browsers.

In 1996, XML was created to resolve the problems of HTML and the complexities of SGML. XML's strength was in its organization. Document structure and presentation were finally separated, while the benefits of standard markup procedure were maintained.

How is XML Being Used?

O'Reilly School supports the use of XML, and all of our courses are written in XML. Separating the document structure from its presentation is important for us. Since our tutorials are written in XML, our course development staff doesn't have to worry about formatting course content for the web, print, or any other particular technology. The presentation is left up to design professionals. Our authors don't have to spend time adding unnecessary tags to their documents. And design professionals don't have to spend time changing multiple individual documents so they have a uniform appearance. If we decide to change the look of the HTML representation of our tutorials, we only have to edit *one* XSL file. Or, if we decide to make our tutorials available in plain text format, we only have to create *one* XSL file. Good stuff!

Other companies have found powerful ways to use XML too. The internet has demonstrated that open technologies and formats are much more useful than proprietary formats. A file created on a word processor that saves documents in XML format can be read by another word processor that reads XML files. No conversion is necessary. That same file could also be opened and modified by a database system.

Finally, XML can be used as an intermediary between users. For example, chemists can use XML to share information with each other in a simple (yet powerful) format. Or banks can distribute financial information to their customers in an XML format that all programs can read.

Parsing XML

There are several technologies available for parsing and transforming XML. C, C++, Perl, JavaScript, Python, Java programs, Java servlets, PHP, and ASP can all work with XML. The focus of this course is *not* on these other technologies though, it's on XML itself. Our course provides all of the tools you'll need to use XML and XSL with our chosen XML and XSL software.

Technologies come and go quickly, so we won't try to compare everything on the market, but feel free to check out some of these great resources:

- xml.com
- XML FAQ
- Cafe con Leche XML News and Resources

A Basic XML Document

Now that you know a bit about the history of XML, let's see it in action!

WARNING <?xml version="1.0"?> must be the first code typed into your XML file. No white space can be present before this text.

Be sure you're using XML syntax and type the code below into CodeRunner as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<HelloWorld>
   <Message>This is my first XML document!</Message>
</HelloWorld>
```

When you finish, click the **Check Syntax** button:

If you typed everything correctly, you will see **No errors found.**.

Congratulations! You just created your first XML document!

So, what does this document *do*? By itself, this little example doesn't do anything. Remember, XML documents are intended to be read by computers and humans. You could email this file to your friend and she would be able to understand it, or you could write a program in nearly any language (like Java or Python) to read the file and access its contents, without a lot of messy programming.

Differences Between XML and HTML

Since HTML is a markup language like XML, they have many similarities, but there are a few key differences between HTML and XML too.

Here are some fundamental differences you should be aware of:

- With HTML, small errors in syntax are often ignored.
- HTML has only pre-defined tags, whereas XML tags are created by the author.
- Documents can be structured logically in XML (the author chooses the appropriate structure), while HTML has a pre-defined "head" and "body" type structure.
- XML isn't always useful on its own. Translating it to different forms (such as HTML) is one of its great powers.

XML has different rules from HTML because XML was created to serve a different purpose. Most of the differences between HTML and XML syntax serve to make parsing XML documents faster and easier. No mistakes are allowed in XML!

These differences should make sense to you. If they don't, please take a look at the HTML course again.

Common Mistakes

Before we get too deep into XML, you should be aware of some common pitfalls out there waiting for new XML programmers. As an experienced HTML programmer, you're probably used to HTML's more flexible syntax structure. If you're converting existing HTML to XML (defining some HTML tags in your XML specification), you may see some problems.

White Space

XML treats white space much differently from HTML. In HTML, white space (spaces, newlines, tabs, and other "white" characters) is pretty much ignored. This is not the case in XML. Every character is important! Let me say that again. *Every character is important!* (Okay, now that I've said that, depending on your application, it is possible that white space may not matter. If you are using XML to output to HTML only, there is little need to worry about this, but for strictly XML applications, it **is** important.)

Closing Tags

All tags must be closed in XML. So a tag like **<br>** by itself is incorrect in XML. To remedy this situation, you either use **<br/>** or **<br></br>**. These two methods of closing tags are different. You'll learn more about that when we study DTDs later in the course.

Nesting Tags

All XML tags must be nested correctly. In HTML, nesting is not always important. Change your XML so it looks like this:

CODE TO TYPE:

```
<?xml version="1.0"?>
<div><span style="font-weight:bold" id="text">This is some text.</div></span>
```

Check Syntax Check the syntax.

This code is incorrect because the **span** tag is not closed before the **div** tag. Structure is extremely important in XML (so important that your document will not be processed if the structure is incorrect!) Let's modify the code and make it work:

CODE TO TYPE:

```
<?xml version="1.0"?>
<div><span style="font-weight:bold" id="text">This is some text.~~</div>~~</span></div>
```

Check Syntax Check the syntax again; it should parse correctly now.

Root Element

In addition to nesting elements correctly, XML requires a root element, which contains all the other elements. Edit your XML as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<div><span style="font-weight:bold" id="text">This is some text.</span></div>
<div><span style="color:red" id="text2">This is red text.</span></div>
```

Check Syntax Check the syntax.

This code is incorrect because there is no containing element. Try adding **html** tags as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<html>
<div><span style="font-weight:bold" id="text">This is some text.</span></div>
<div><span style="color:red" id="text2">This is red text.</span></div>
</html>
```

Check Syntax Check the syntax again; it should parse correctly now.

Capitalization

Another drastic difference between HTML and XML is capitalization. In HTML, browsers tolerate tags in upper or lower case, or even a combination of the two. This is not the case with XML. Letter case is important in XML.

WARNING XML is case-sensitive, so <SPAN> and <span> are different!

Quoting Attributes

Another difference between HTML and XML may be in the syntax of your inline style attributes. Change your XML as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<span style=font-weight:bold id="text">This is some text.</span>
```

Check Syntax Check the syntax.

You may be used to typing a line this way, but it is incorrect syntax in XML. The style attributes must have quotation marks around them. Go ahead and fix the syntax. Modify your code as shown below:

CODE TO TYPE:

```
<?xml version="1.0"?>
<span style="font-weight:bold" id="text">This is some text.</span>
```

Check the syntax again.

Okay, so now that you're warmed up, let's keep going! You're on your way to appreciating all the power that is XML. See you at the next lesson!

Your First XML Document

In this lesson, we'll construct a basic XML file for storing phone book information. You might use this phonebook to contain information about family and friends. You could send your phonebook to your family via e-mail, and they could add their own entries to the file. An XML file is a powerful format to use for your phonebook.

Phonebook XML

The information we want to store in this XML file is:

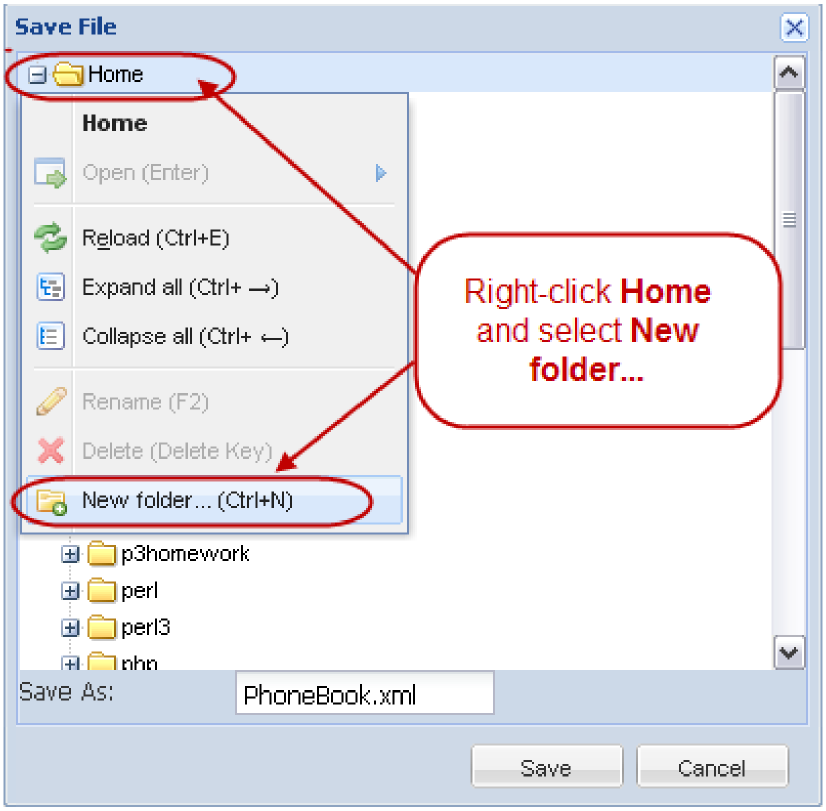
- First Name
- Last Name
- Phone Number

How can we implement this in XML? First, we have to define a set of tags we want to use to keep track of our information. These tags are similar to the tags used in HTML because both HTML and XML are derived from SGML. But unlike HTML, XML has no predefined tags. Tags in XML are also referred to as *elements*.

For our example, we'll use these elements:

- <First>
- <Last>
- <Phone>
- <PhoneBook>
- <Listing>

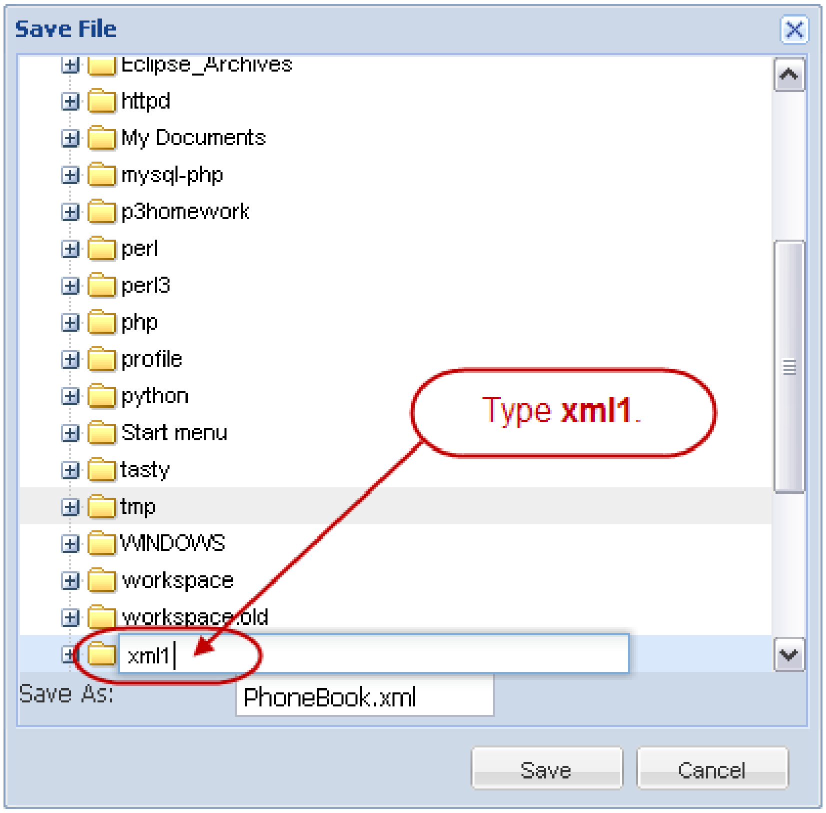
Let's get started on your first program! First, we'll create a folder to keep all of our XML stuff organized. In the left panel of your CodeRunner window, right-click **Home**, and select **New folder...** as shown:



Right-click **Home** and select **New folder...**

Save As: PhoneBook.xml

Type **xml1** for the folder name, and press **Enter**:



Now let's create an XML file! Make sure you're in XML mode in the editor. This XML file will mark up information that would go into a phonebook. Type everything exactly as it appears in the box here:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
    <Listing>
       <First>Alex</First>
       <Last>Chilton</Last>
       <Phone>1-800-123-4567</Phone>
    </Listing>
</PhoneBook>
```

Click the **Save** button in the CodeRunner Editor toolbar, select the **/xml1** folder we just created, and type the filename **PhoneBook.xml**. So what have we created? Even if you knew nothing about XML, you might guess that this file contains a phone book and one listing for Alex Chilton, who somehow got himself an 800 number that is exceedingly easy to remember.

The first line with **<?xml version="1.0"?>** indicates that this is an XML file. The second line with **<!DOCTYPE PhoneBook>** specifies the *document type* of the file we are creating (we decided this is a document of type PhoneBook). You will learn more about DOCTYPE later, but for now remember that your DOCTYPE must match the first set of tags in your XML document. In this case, the !DOCTYPE is PhoneBook. <PhoneBook> is the first tag of our document. The other lines in the XML document define your phonebook and the data it tracks. In this example, we are tracking First name, Last name, and Phone number. How would you add another listing?

Before we go any further, we'll want to include a few notes. When you write XML (or any other document or program) you should comment often. In this course, you'll *always* comment your files! Ultimately it saves time and improves the quality of your work. Comments in XML are similar to comments in HTML, structured like **<!-- this is a comment -->** (note that no closing tag is needed in this case). Add a comment as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
    <!-- One listing for each person -->
    <Listing>
       <First>Alex</First>
       <Last>Chilton</Last>
       <Phone>1-800-123-4567</Phone>
    </Listing>
</PhoneBook>
```

Save it. We won't include the comment in our following examples in the interest of brevity, but you should use comments liberally throughout your code.

Note

Remember:

- XML is case-sensitive. <stuff> and <STUFF> are *not* the same!
- All XML tags must be closed. This means for every <STUFF> a matching </STUFF> must exist. <STUFF/> is an example of an single tag that is closed.
- <?xml version="1.0"?> must be the first thing in your XML file. No white space can be present before this text.

Now let's set a style sheet for our XML file. Edit **PhoneBook.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xsl" type="text/xsl"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
    <Listing>
       <First>Alex</First>
       <Last>Chilton</Last>
       <Phone>1-800-123-4567</Phone>
    </Listing>
</PhoneBook>
```

Save it. This code links our XML file to an *XSL file* we've provided. An XSL file is a style sheet for your XML file. (In later lessons, we'll show you how to make XSL files on your own. For now we'll just do it for you to get where we want to go in this lesson.)

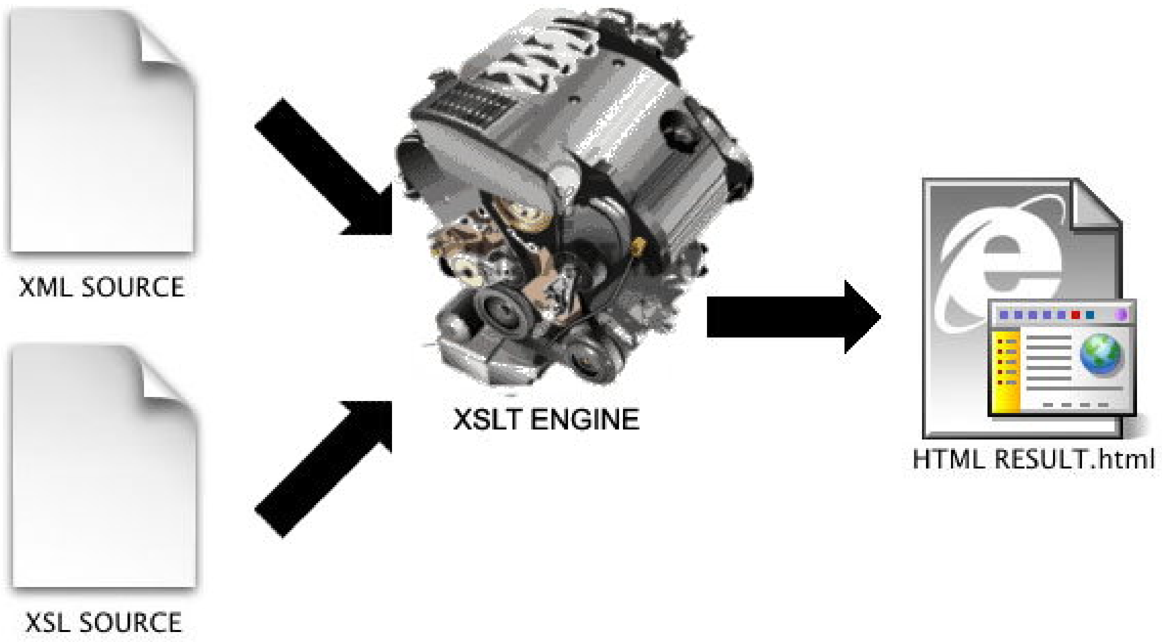
XSL is a language that describes a set of transformations that are made on a source XML file. We are using XSL to "preview" our XML as it might be represented in HTML. In other words, this XSL file contains the rules for turning our XML file into HTML.

Locate the **Check Syntax**, **Validate**, and **Translate** buttons in the toolbar. To translate this page, click the **Translate** button (Translate). You'll see your phone book file represented in HTML. Click on the **Source** tab in the pop-up window and you'll see the HTML code. In a later lesson, you'll learn how to change the HTML representation of your XML file.

Note

If you don't see the HTML representation of your phonebook, and/or see an error message instead, there may be something wrong with your XML file. Check to make sure you typed it in correctly, without spelling errors, and with all tags properly closed.

What does the Translate button do? It takes the source XML file and a set of translation instructions (the XSL file) and outputs the result. Here is a graphical representation of the translation process:



When you click **Translate**, both files are sent to the OST *XML Translator*, which checks the files and then combines them to give HTML output. There are different parsers for different types of output. In this introductory course, most of our output will be HTML.

Check Syntax Check the syntax and make note of the result. The message you see verifies whether or not your document translated correctly and has the correct syntax.

With this XML document working, let's add another listing. Modify **PhoneBook.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xsl" typ
e="text/xsl"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
    <Listing>
       <First>Alex</First>
       <Last>Chilton</Last>
       <Phone>1-800-123-4567</Phone>
    </Listing>
    <Listing>
       <First>Laura</First>
       <Last>Chilton</Last>
       <Phone>1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

Save it, then Translate translate it and check out what happens. You'll see two phonebook entries—one for Alex Chilton and one for Laura Chilton.

We will use this file as a starting point in future lessons and objectives, so you'll want to save a copy. If you want to keep a separate copy of the file as it is in this lesson, use the **Save As** button () and give it a different name. Be sure to include the xml extension when you're saving your xml files. Then, reopen the original **PhoneBook.xml** from the **/xml1** folder, and try adding more entries into your phonebook.

Now let's change our XSL instructions so our resulting HTML document will look different. Modify **PhoneBook.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook2.xsl" ty
pe="text/xsl"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
    <Listing>
       <First>Alex</First>
       <Last>Chilton</Last>
       <Phone>1-800-123-4567</Phone>
    </Listing>
    <Listing>
       <First>Laura</First>
       <Last>Chilton</Last>
       <Phone>1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

💾 and Translate again. It's our phonebook again, but it really looks different now! Soon you'll learn how to translate your XML file into different representations, then you'll really get a feel for the power of XML! Save this file as **PhoneBook02.xml**.

Let's make some more changes. Perhaps Alex Chilton has three phone numbers—one for work, one for home, and one for his cell phone. How can we add more numbers to your phonebook in a way that makes sense? Type the code below as shown to see one possible solution:

CODE TO TYPE:

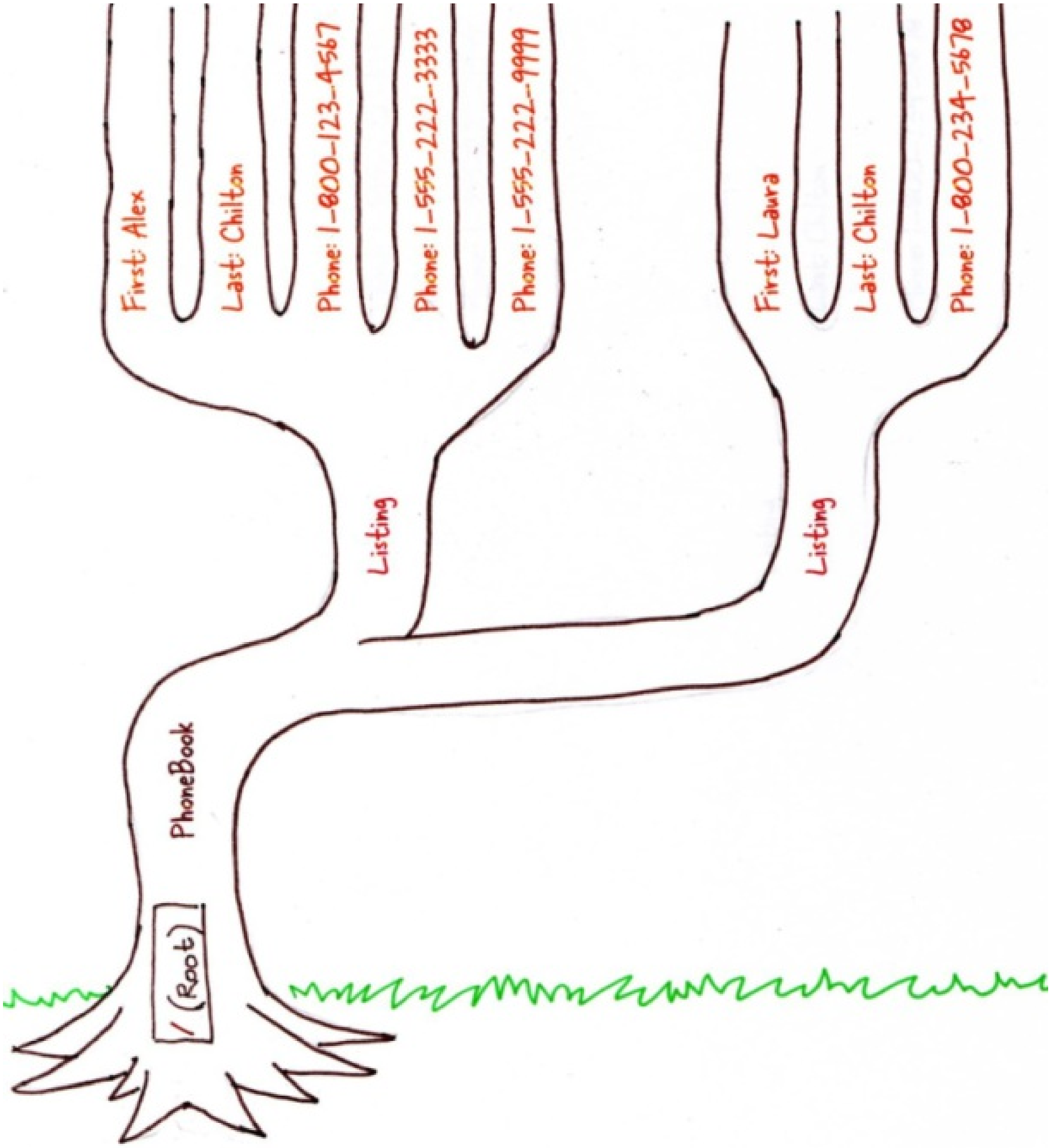
```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook2.xsl" ty
pe="text/xsl"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
    <Listing>
       <First>Alex</First>
       <Last>Chilton</Last>
       <Phone>1-800-123-4567</Phone>
       <Phone>1-555-222-3333</Phone>
       <Phone>1-555-222-9999</Phone>
    </Listing>
    <Listing>
       <First>Laura</First>
       <Last>Chilton</Last>
       <Phone>1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

💾 and Translate again. You should get some interesting results! XML is one powerful language!

Of course, you'd want to know which of these phone numbers was Alex's cell phone and which was his home phone too. We'll get to that in just a bit. For now, let's make sure you understand how an XML file is structured.

The Tree Structure of XML

As you might have noticed, XML documents form kind of a tree. You're probably familiar with the way directories and files are organized on your computer—also in a tree-like structure. Here's our XML, represented by a tree with no closing tags:



(Well, *kinda* like a tree!) As you can see, "/" is the "root"; from that springs the PhoneBook trunk, and from that, there are two Listings and each Listing has First, Last, and Phone branches. Inside First is the first name, inside Last is the last name, and inside each Phone is a phone number.

In this example, Listing is a *child* element of PhoneBook. Similarly, PhoneBook is the *parent* element of Listing. First, Last, and Phone are all *children* of Listing and *grandchildren* of PhoneBook. PhoneBook is also the *grandparent* of First, Last, and Phone. The terminology we use to describe the relationships between elements in XML is the same as that used in genealogy.

Elements and Attributes

What is an Element?

An *element* is a tag in our XML document, such as **<First>**. Anything inside this particular tag should be interpreted as the first name.

Note

Some XML references use the word *node* to refer to elements.

The *root* element is the first element that occurs in an XML document. In this case, the PhoneBook element is the root element.

Here is our basic phonebook XML file:

OBSERVE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook2.
xsl" type="text/xsl"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
    <Listing>
       <First>Alex</First>
       <Last>Chilton</Last>
       <Phone>1-800-123-4567</Phone>
    </Listing>
    <Listing>
       <First>Laura</First>
       <Last>Chilton</Last>
       <Phone>1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

You can interpret the example above like this:

- The XML Document is a **PhoneBook** (as is determined by the PhoneBook tag). All text between these tags should be a part of the phonebook.
- The phonebook consists of a series of zero or more **Listings**.
- Each **Listing** has several components, such as the **First** and **Last** names.
- Each **Listing** also has a **Phone** element, as you might guess, to specify the phone number.

This document has a unique hierarchy. How would the document be changed if the elements were mixed up? Compare this example to the last one:

OBSERVE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook2.
xsl" type="text/xsl"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
<Listing>
 <Last>Chilton</Last>
 <First>Alex</First>
 <Phone>1-800-123-4567</Phone>
</Listing>
<Listing>
    <Last>Chilton</Last>
    <First>Laura</First>
    <Phone>1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

As it turns out, there is *no significant difference* between this document and the previous one. Type it in, try it out, experiment! The HTML output will still look the same. Try mixing up all of the elements to see what happens.

Mixing *some* elements will change the meaning of your XML document. The next document has the same tags as the previous documents, but it has a different meaning.

OBSERVE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook2.
xsl" type="text/xsl"?>
<!DOCTYPE Listing>
<Listing>
    <PhoneBook>
        <Last>Chilton</Last>
        <First>Alex</First>
        <Phone>1-800-123-4567</Phone>
    </PhoneBook>
    <PhoneBook>
         <Last>Chilton</Last>
         <First>Laura</First>
         <Phone>1-800-234-5678</Phone>
     </PhoneBook>
</Listing>
```

How is this document different? Reading through the document would suggest that the Listing has a PhoneBook element. The document also suggests the PhoneBook has elements of Last, First, and Phone. This *could* be correct, but it isn't the intended "meaning" of our phone book.

Okay, so how does this document look when we try to parse it in the same way as our other XML documents? Modify **PhoneBook.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
    <?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBo
ok2.xsl" type="text/xsl"?>
<!DOCTYPE Listing>
<Listing>
    <PhoneBook>
        <Last>Chilton</Last>
        <First>Alex</First>
        <Phone>1-800-123-4567</Phone>
    </PhoneBook>
    <PhoneBook>
         <Last>Chilton</Last>
         <First>Laura</First>
         <Phone>1-800-234-5678</Phone>
    </PhoneBook>
</Listing>
```

💾 and Check Syntax it.

Now Translate. The XML file you create might make perfect sense to you, but not to the rest of the world.

Undo these last changes before proceeding! You can use **Ctrl+Z** (on PCs) or **Command+Z** (on Macs) in CodeRunner to undo previous changes, one at a time.

What is an Attribute?

Sometimes we need more detailed information than an element can supply. We may want to modify our information using *attributes*. Using attributes allows us to organize our information logically, and keep our code neat so it can be understood by others more easily. Because XML attributes modify elements, attributes cannot perform their magic without *elements*.

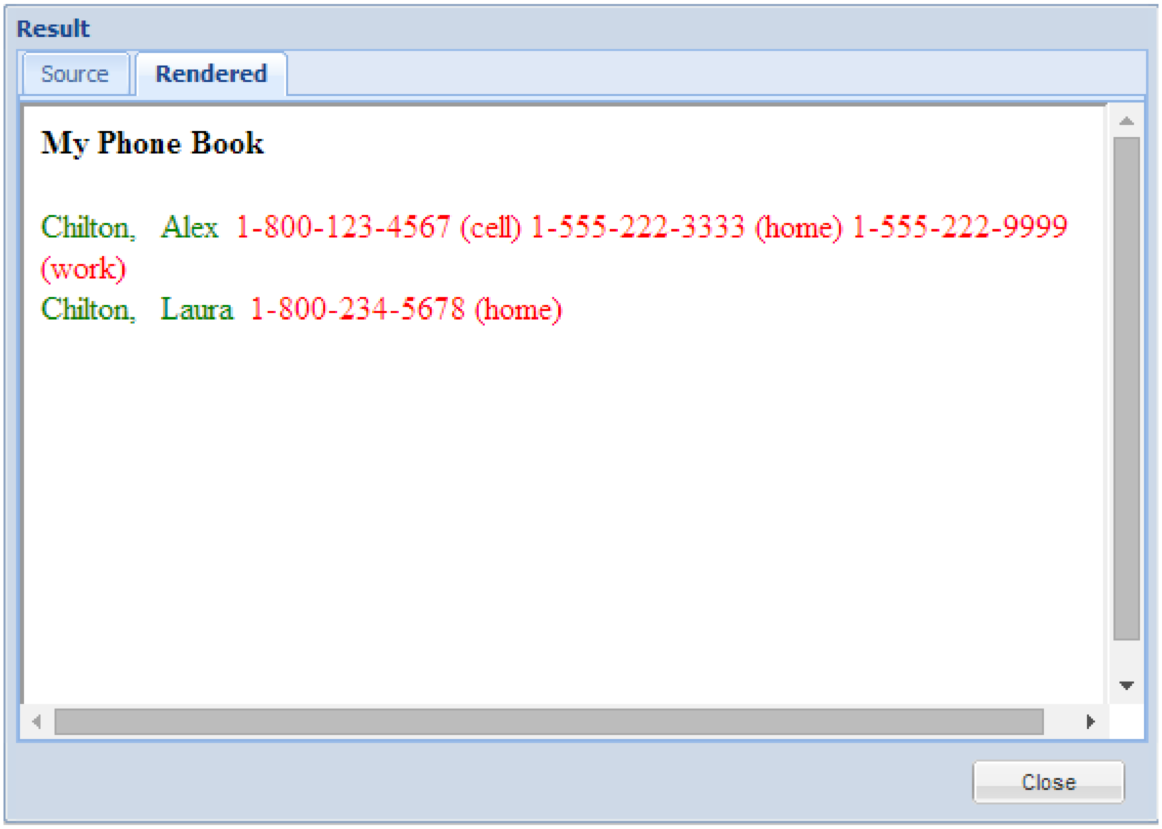
Let's say we are working with our XML-based phone book. Many of our friends have multiple phone numbers for multiple phones: office, home, and cell. We *could* create new elements for each of these phones, such as <HOME_Phone>, <CELL_Phone>, and so on. This would make it clear to anybody which data is specified by the tags. But Attributes are a more elegant solution to the problem. Instead of specifying a unique tag for the home, cell, and work phone numbers, we'll specify this in the Phone tag itself. Make sure you specify the new stylesheet—**PhoneBook3.xsl**. Modify **PhoneBook.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook3.xsl" type="text/xsl"?>
<!DOCTYPE PhoneBook>
<PhoneBook>
    <Listing>
        <First>Alex</First>
        <Last>Chilton</Last>
        <Phone Type="cell">1-800-123-4567</Phone>
        <Phone Type="home">1-800-123-4568</Phone>
        <Phone Type="work">1-800-123-4569</Phone>
    </Listing>
    <Listing>
         <Last>Chilton</Last>
         <First>Laura</First>
         <Phone Type="home">1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

So what does this new XML document specify? It appears to be the same phonebook we've been working with all along. The new part of the document is the **Type** attribute specified in the Phone element.

💾 and Translate it:



My Phone Book

Chilton, Alex 1-800-123-4567 (cell) 1-555-222-3333 (home) 1-555-222-9999 (work)
Chilton, Laura 1-800-234-5678 (home)

Pretty cool, huh? Go forth and experiment by specifying other Phone Types.

Now that you have a feel for using attributes, go ahead and create some of your own. Change your XML document to include more phone numbers, perhaps multiple office numbers, fax numbers, school numbers, whatever you want. Preview your XML to see how this changes the HTML representation of your document. When you are previewing your XML, keep in mind that *no* changes have been made to our XSL document specifying the transformation from XML to HTML. You'll learn how we accomplish that shortly...

When To Use Elements and Attributes

We could try to define the specific times and places to use elements and attributes, but in XML programming, as is often true in life, there is no single correct formula. It's a given that your document should include at least one element, or your document would be empty. You could have your whole XML document consist of a single element with multiple attributes, but this would be pretty messy and difficult to read. Instead. a balance is required.

It's been our experience that using elements with attributes interspersed sparingly is the best way to create XML. The most important consideration is readability. Design your XML documents so they can be read by anybody—even those unfortunates who haven't taken this course—and so readers can follow the hierarchy of your documents.

If you cannot give your XML document to somebody else and have them identify its content (or at least offer a good guess), then you probably need to rethink your choices.

Finding and Fixing Syntax Errors

In keeping with OST's philosophy of learning by doing, mangling, fixing, and getting plain dirty with programs, I ask you now to go ahead and break the PhoneBook XML file. Of course you have misgivings about thrashing your own beautiful code, so click the New File button (🗋) and try out this pre-broken XML file:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xsl" typ
e="text/xsl"?>
<!DOCTYPE PhoneBook>
 <PhoneBook>
    <Listing>
       <First>Alex</First>
       <Last>Chilton</Last>
    </Listing>
  <Phone>1-800-123-4567
    </Listing>
    <Listing>
       <First>Laura<First>
       <Last>Chilton</Last>
       </Phone>1-800-234-5678</Phone>
   </Listing>
</PhoneBook>
```

Click Check Syntax to see what's wrong with the file. You'll notice that it is horribly broken. This might happen if you're typing an XML file quickly. A sample error message may look like this (only without the color):

OBSERVE:

```
Your document has a syntax error.
Fatal Error: Opening and ending tag mismatch: Phone line 9 and Listing on line 10 colum
n 15
Fatal Error: Opening and ending tag mismatch: First line 12 and Phone on line 14 column
 16
Fatal Error: Opening and ending tag mismatch: First line 12 and Phone on line 14 column
 38

Line
    1 <?xml version="1.0"?>
    2 <?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xs
l" type="text/xsl"?>
    3 <!DOCTYPE PhoneBook>
    4  <PhoneBook>
    5     <Listing>
    6        <First>Alex</First>
    7        <Last>Chilton</Last>
    8     </Listing>
    9   <Phone>1-800-123-4567
   10     </Listing>
   11     <Listing>
   .
   .
   .
```

This means that your XML file has an error. The processor found the first error at **lines 9 and 10**. The error found at that line is **Opening and ending tag mismatch**. Also listed is your document, with the line highlighted where the error occurs. In this case it is **</Listing>**.

To correct this problem, go to line 9 of your XML file. You'll notice that we never closed your **Phone** tag on the previous line. Go ahead and fix that.

Note The error messages generated by each XML/XSLT program will be different, but they should all give you a line number and error message.

You may also see this message if you try to use characters such as less-than < or greater-than > in your element content. Since these characters are used to define tags in XML, you really can't use them in the body of your XML document without some special consideration. A possible way around this will be discussed in a future lesson.

XML Vocabulary

Before you go any further in this course, here's some vocabulary common to XML and XSL that you need to know:

XML	e**X**tensible **M**arkup **L**anguage
XSL	e**X**tensible **S**tylesheet **L**anguage
DTD	**D**ocument **T**ype **D**efinition. A set of rules that specifies the structure of XML documents. DTDs are not XML documents. Instead, DTDs have a different structure. You'll see more about DTDs in the next lesson.
XSD, or Schema	Like a DTD, a set of rules that specifies the structure of XML documents. A schema is an XML document itself. You'll learn more about schemas in future lessons.
Element	A tag in our XML document. An example of an HTML element is **<IMG>**, where the text in **green** is an element.
Attribute	Additional values added to elements. An example of an HTML attribute (with element) is <IMG **SRC="mypic.jpg"**>, where the text in **red** is the attribute.
Entities	A method of search and replace. In XML, entities all begin with an ampersand (&) and end with a semicolon (;). You define your entity and the text that is replaced by it. If you define an entity for **&company;** with the text "AwesomeCorpUSA," occurrences of **&company;** in your XML document will be replaced with "AwesomeCorpUSA." We'll talk about this in greater detail in a future lesson.

XML Tools

One of the best qualities of XML is that it doesn't require *any* tools for humans to begin using it. XML itself is readable and understandable by humans. Editing XML doesn't require anything more than a simple text editor. This isn't how you'll ultimately use XML, but for now it's a good way to practice.

Programming applications that use XML are beyond the scope of this course, but understanding the concepts that lead up to XML applications is important. In this course we use CodeRunner as our text editor. **Check Syntax**, **Validate**, and **Translate** are three important tools that are available for us to use with XML in CodeRunner.

Non-Validating Parsers

In CodeRunner, **Check Syntax** takes your XML document and runs it through an XML parser in a non-validating mode. This means that your XML is not checked against any DTD or schema. (A DTD is one way to specify a set of rules your XML document must follow; a schema is another. You'll learn more about those later.) As long as your XML is *well-formed*, it will parse correctly. How might this be useful? If you are absolutely sure that your XML file is structured correctly, your application may execute faster if it doesn't have to check your file against its DTD or schema.

Validating Parsers

In CodeRunner, **Validate** takes your XML document and runs it through an XML parser in validating mode. Your XML is checked against its DTD or schema. Your document must be well-formed *and* valid for it to parse correctly. If you are not already certain that your XML files are structured correctly, you can use Validate to check them to make sure that they are.

Translating

In CodeRunner, **Translate** takes your XML document *and* an XSL document and runs them through an XSLT engine. The XSLT engines require two inputs: one XML document and one XSL document. XSLT engines provide one output: the result of the XSL document's translations applied to the XML document. XSLT engines must use an XML parser to read the XML and XSL documents. CodeRunner performs a non-validating parse of your XML document and the XSL document when you Translate.

XML parsers and translators are used in various ways. If you're knowledgeable in a language such as Java, you may know how to create your own "Personal Information Manager" program to keep your address book—stored as an XML file. As long as you know how to write a text file in Java, you can store your data as XML.

Reading these XML files back in, though, is *not* easy. Instead of spending time and energy programming a way to read XML files, why not use a pre-written Java parser? Many Java parsers have been written specifically to be included in Java programs. Now you can use the functions that the parser provides to read your XML file. Problem solved!

Similarly, if you want to add XSLT functionality to your program, there are many engines that have been written specifically to be included in other applications. Another problem solved!

We are all about solving problems and saving time. And for this course, you should also save your work, because you will continue to use your PhoneBook file throughout the coming lessons. See you in the next one!

Using DTDs (Document Type Definitions)

What is a Document Type Definition?

A **D**ocument **T**ype **D**efinition, or **DTD**, is a set of rules you define for your XML documents. Why would this be important, you ask? Well, let's say you're using XML to exchange financial information between your bank and your home computer. If the format of your XML document isn't consistent throughout, your savings account data could be interpreted as your checking account data, causing all kinds of confusion. DTD enables us to define the components of an XML document and put them where we want them to be. But wait, there's more! DTDs can also be used to define *entities*, or common elements for XML documents.

Here's what DTDs do:

- Define all elements and attributes of an XML document.
- Define the order in which elements and attributes can occur.
- Define all entities that can be used in an XML document.
- Define the document type.

Declaring Use of a DTD in Your XML file

DTDs can be stored in your XML document or in a separate file. If your DTD is going to be located inside your XML file, it must be at the *top* of your file (after the <?xml version="1.0"?> declaration). The beginning of an internal DTD looks like this:

OBSERVE:

```
<!DOCTYPE mydoctype [
...
...
...
]>
```

More likely, you will store your DTD outside your XML document. The beginning of an XML file that has an external DTD looks like this:

OBSERVE:

```
<!DOCTYPE mydoctype SYSTEM
        "http://www.mysite.com/dtds/mydoctype.dtd">
```

Let's break down this code:

mydoctype is the document type. If your XML document will hold personal banking information, your doctype might be **financial** or **banking**.

SYSTEM is used if your DTD is a separate file from your XML document. You can define your DTDs to be either **PUBLIC** or **SYSTEM**, depending on your application. PUBLIC means that all are free to use the DTD. This may be the case if your DTD is defining the XML standard for financial exchange between banks. If you are using your own DTD, you'll use SYSTEM. (For this course, your DTDs will be internal, so you won't have to worry about this.)

http://www.mysite.com/dtds/mydoctype.dtd indicates the location (or URL) of the DTD file if your DTD is a separate file.

Here's an example of an XML document with an internal DTD:

OBSERVE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xsl" type="text/xsl"?>
<!DOCTYPE PhoneBook [
<!ELEMENT PhoneBook (Listing+)>
<!ELEMENT Listing (First,Last,Phone+)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Phone (#PCDATA)>
<!ATTLIST Phone
    Type    CDATA    #REQUIRED>
]>
<PhoneBook>
    <Listing>
        <First>Alex</First>
        <Last>Chilton</Last>
        <Phone>1-800-123-4567</Phone>
    </Listing>
    <Listing>
        <First>Laura</First>
        <Last>Chilton</Last>
        <Phone>1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

Here's an example of an external DTD:

OBSERVE:

```
<!ELEMENT PhoneBook (Listing+)>
<!ELEMENT Listing (First,Last,Phone+)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Phone (#PCDATA)>
<!ATTLIST Phone
    Type    CDATA    #REQUIRED>
```

External DTDs usually have a **.dtd** extension. You can save them anywhere as long as you specify the proper location of your DTD in your XML file.

Internal, External, or Neither?

So, which type of DTD should you use? Well, that really depends on you and your needs. Use the DTD type that's more convenient for you and makes your XML document more clear to read and easier to understand.

For instance, if your XML document will be used frequently as a standard format for a series of textbooks, why not include a pre-defined DTD? If your XML document is unique and basic, like here in your personal phonebook, you may want to keep it simple and include the DTD with your XML document.

On occasion you might find yourself using a non-validating XML processor. This means that the DTD for your XML document will be ignored. The XML processor will check your XML document to make sure it follows the rules of XML, but it *won't* check to see if it follows the DTD.

DTD: The Specifics

Okay, let's try an example! We will specify an internal DTD for our phone book file. Elements of the DTD are the same as XML elements. An element declaration in XML looks like this:

OBSERVE:

```
<!ELEMENT First (#PCDATA)>
```

What does this declaration mean? **First** is the name of the element that will be in the XML document. **(#PCDATA)** specifies that the content of the First element will be PCDATA, or *parsed character data*, which is text that will be parsed by an XML parser.

(*CDATA*, or *character data*, is text that will not be parsed by an XML parser.)

You can specify as many elements as you need. Sometimes you may want to specify the order in which elements will be accessed as well. Using the phone book XML document, we may want to specify that the First name appears before the Last name. The DTD would look like this:

OBSERVE:

```
<!ELEMENT Listing (First,Last)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
```

Now in our **Listing** tag, two elements must appear: First and Last (in that order).

Now, modify your **PhoneBook.xml** file as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xsl" typ
e="text/xsl"?>
<!DOCTYPE PhoneBook~~>~~ [
<!ELEMENT PhoneBook (Listing+)>
<!ELEMENT Listing (First,Last,Phone+)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Phone (#PCDATA)>
<!ATTLIST Phone
    Type    CDATA    #REQUIRED>
]>
<PhoneBook>
    <Listing>
        <First>Alex</First>
        <Last>Chilton</Last>
        <Phone Type="cell">1-800-123-4567</Phone>
        <Phone Type="home">1-800-123-4568</Phone>
        <Phone Type="work">1-800-123-4569</Phone>
    </Listing>
    <Listing>
        <First>Laura</First>
        <Last>Chilton</Last>
        <Phone>1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

💾 and Validate ⚙ it. You'll see this text:

OBSERVE:

```
Error: Element Phone does not carry attribute Type on line 22
```

What does this mean? Your XML document was compared to its internal DTD (the very same one we just created). The DTD specifies that the element Phone must have an attribute of **Type**, *always*. A Phone element in our XML file doesn't have a **Type** attribute, so our XML parser gave us an error. Go ahead and add the **Type** attribute and set it to "home." Validate the document once again; that error will go away.

Now try removing the Phone elements completely. Take out the code in ~~red~~ as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xsl" type="text/xsl"?>
<!DOCTYPE PhoneBook [
<!ELEMENT PhoneBook (Listing+)>
<!ELEMENT Listing (First,Last,Phone+)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Phone (#PCDATA)>
<!ATTLIST Phone
    Type    CDATA    #REQUIRED>
]>
<PhoneBook>
    <Listing>
      <First>Alex</First>
      <Last>Chilton</Last>
      ~~<Phone Type="cell">1-800-123-4567</Phone>~~
      ~~<Phone Type="home">1-800-123-4568</Phone>~~
      ~~<Phone Type="work">1-800-123-4569</Phone>~~
    </Listing>
    <Listing>
      <First>Laura</First>
      <Last>Chilton</Last>
      <Phone Type="home">1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

💾 and Validate it again. You'll get this error:

OBSERVE:

```
Error: Element Listing content does not follow the DTD, expecting (First , Last , Phone+), got (First Last ) on line 12
```

This error means that the DTD requires at least one Phone element to occur after the Last element. Since none exist, an error is raised. Restore the deleted phone numbers to correct the error.

Experiment with this concept for a while. Change your DTD and XML file and deliberately break the validation. Note the errors that are generated. Below you will find more information on the syntax of DTDs. Try them all!

Symbol	Description	Example
+	The element must occur one or more times.	<!ELEMENT PhoneBook (Listing+)>
()	Defines a set of elements. Used with other modifiers.	<!ELEMENT Listing (First,Last)>
,	Defines the order that the elements must occur.	<!ELEMENT Listing (First,Last)>
|	Defines a set of elements, of which one must be used.	<!ELEMENT Listing (Phone|First|Last)>
?	The element can occur zero or one times.	<!ELEMENT Listing (First,Last?)>
*	The element (or set) must occur zero or more times.	<!ELEMENT Listing (Phone*)>

Entities

Sometimes you'll find yourself using the same text over and over again in your documents. You might want to include your name, company name, or a copyright notice in every document. Fortunately, you don't have to cut and paste this text over and over—just declare your own *entity* in your DTD:

OBSERVE:

```
<!ENTITY Company "O'Reilly Media">
```

This entity declaration states that **Company** contains "O'Reilly Media." If you are familiar with C or C++, this is similar to the #DEFINE syntax. Once defined, your entity will be available like any other system entity.

To use your declared entity, include the entity reference preceded by an ampersand (&) and followed by a semi-colon:

OBSERVE:

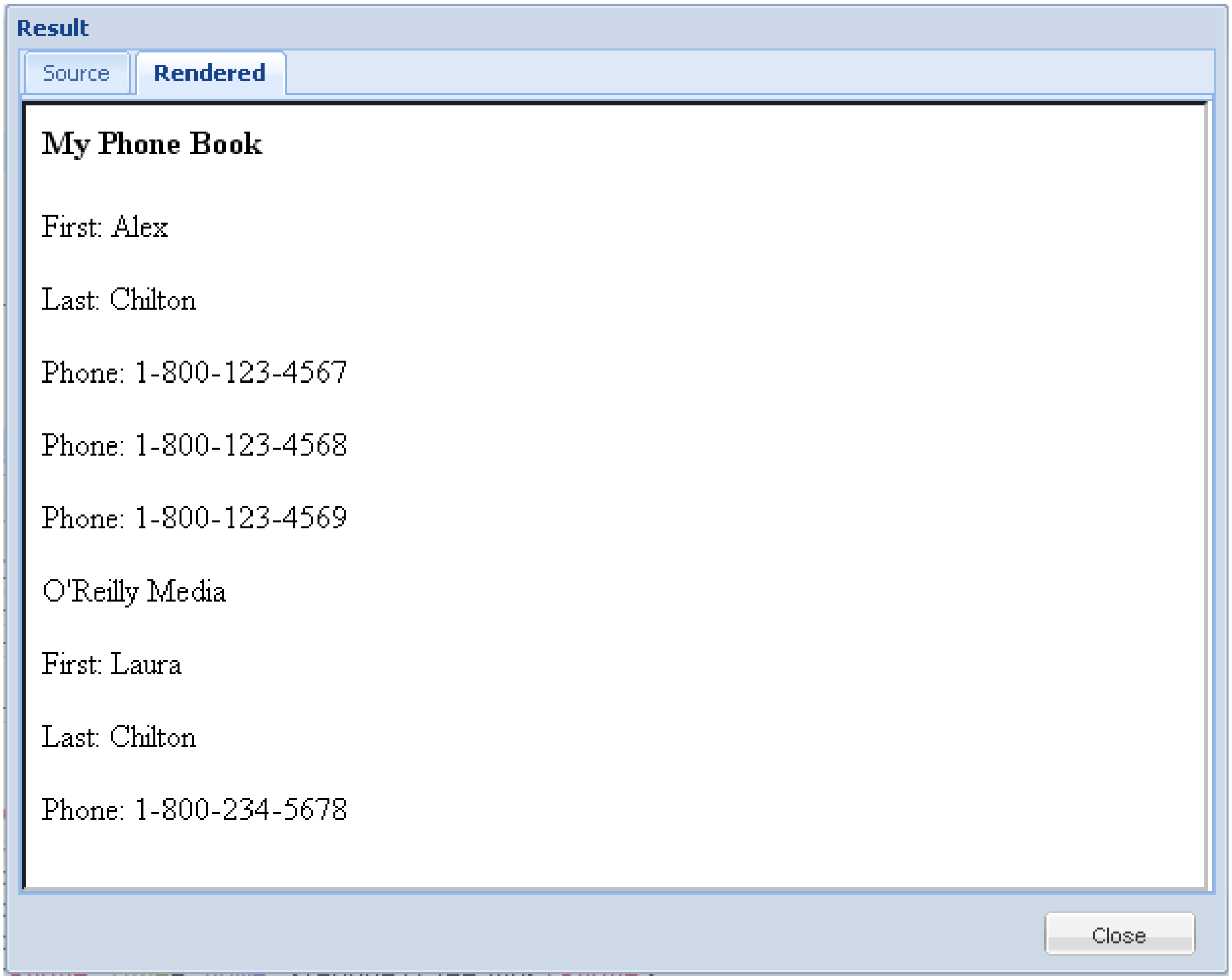
```
<Company>&Company;</Company>
```

Try it in your document. Modify **PhoneBook.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xsl" typ
e="text/xsl"?>
<!DOCTYPE PhoneBook [
<!ELEMENT PhoneBook (Listing+)>
<!ELEMENT Listing (First,Last,Phone+,Company?)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Phone (#PCDATA)>
<!ATTLIST Phone
    Type    CDATA    #REQUIRED>
<!ELEMENT Company (#PCDATA)>
<!ENTITY Company "O'Reilly Media">
]>
<PhoneBook>
    <Listing>
      <First>Alex</First>
      <Last>Chilton</Last>
      <Phone Type="cell">1-800-123-4567</Phone>
      <Phone Type="home">1-800-123-4568</Phone>
      <Phone Type="work">1-800-123-4569</Phone>
      <Company>&Company;</Company>
    </Listing>
    <Listing>
        <First>Laura</First>
        <Last>Chilton</Last>
        <Phone Type="home">1-800-234-5678</Phone>
    </Listing>
</PhoneBook>
```

💾 and Translate ⚙ it. If you typed everything correctly, you'll see this:



My Phone Book

First: Alex

Last: Chilton

Phone: 1-800-123-4567

Phone: 1-800-123-4568

Phone: 1-800-123-4569

O'Reilly Media

First: Laura

Last: Chilton

Phone: 1-800-234-5678

&Company; was replaced with **O'Reilly Media**

There are five entities predefined in XML. All XML processors are required to support these entities:

Entity name	Replacement text
lt	The less than sign (<)
gt	The greater than sign (>)
amp	The ampersand (&)
apos	The single quote or apostrophe (')
quot	The double quote (")

ANY and EMPTY

Sometimes you'll find that your element should be empty. Other times you may not care if elements are contained within other elements (kind of like the <P> in HTML). You can define these elements in your DTD. Check this out:

OBSERVE:

```
<?xml version="1.0"?>
<!ELEMENT PhoneBook ANY>
<!ELEMENT Marked EMPTY>
```

In this example, **ANY** means that any other element can occur in between <PhoneBook> and </PhoneBook>, and **EMPTY** can be used to denote when an element must be empty.

Let's try this out. Edit **PhoneBook.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook.xsl" typ
e="text/xsl"?>
<!DOCTYPE PhoneBook [
<!ELEMENT PhoneBook (Listing+)>
<!ELEMENT Listing (First,Last,Phone+,~~Company?~~Marked)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Phone (#PCDATA)>
<!ELEMENT Marked EMPTY>
<!ATTLIST Phone
    Type    CDATA    #REQUIRED>
~~<!ELEMENT Company (#PCDATA)>~~
]>
<PhoneBook>
    <Listing>
      <First>Alex</First>
      <Last>Chilton</Last>
      <Phone Type="cell">1-800-123-4567</Phone>
      <Phone Type="home">1-800-123-4568</Phone>
      <Phone Type="work">1-800-123-4569</Phone>
      <Marked>yes</Marked>
      ~~<Company>&Company;</Company>~~
    </Listing>
    <Listing>
      <First>Laura</First>
      <Last>Chilton</Last>
      <Phone Type="work">1-800-234-5678</Phone>
      <Marked/>
    </Listing>
</PhoneBook>
```

This document has a DTD that supports a new element: **Marked**. The element is allowed after the Phone element, and must be empty. 💾 and Validate it. You'll get an error like:

OBSERVE:

```
Error: Element Marked was declared EMPTY this one has content on line 21
```

This is a long way of saying that the element must be empty. You can fix the problem by removing the "yes" as we did with Laura:

OBSERVE:

```
<Marked/>
```

or

OBSERVE:

```
<Marked>~~yes~~</Marked>
```

What is a Content Model?

Now that you've learned the basics of DTD construction and the right way to define elements, attributes, and entities, you can use them to define your *Content Model*. A content model provides the framework for data to flow into, as it is entered into your XML document, thus keeping the data organized and correctly formatted.

Suppose you are creating an XML document to store contact information for your friends. You want to include both home and work addresses as part of this information. You could use this:

OBSERVE:

```
<Address Type="HOME">
    123 Sesame Street, Madison WI 53704
</Address>

<Address Type="WORK">
    236 Main Street, Madison WI 53704
</Address>
```

Or you could use this:

OBSERVE:

```
<Address Type="home">
    <Street>123 Sesame Street</Street>
    <City>Madison</City>
    <State>WI</State>
    <Zip>53704</Zip>
</Address>

<Address Type="work">
    <Street>236 Main Street</Street>
    <City>Madison</City>
    <State>WI</State>
    <Zip>53704</Zip>
</Address>
```

Through careful definition of your elements and attributes, you cause the XML documents that use your DTD to have a specific structure. If you create an XML document that doesn't follow the rules you've specified in your DTD, the XML application you are using at the time will refuse to work.

This is important in situations like our online banking example. You want your home financial software to accept XML data for all of your transactions, and you want to make sure the data is formatted correctly. If your software checks the incoming XML data against its DTD, incorrectly formatted XML files will not be processed. This is how it should be—you don't want an incorrectly formatted XML document to mess up all of your financial information!

Alright, you're looking good so far. Keep it up and see you in the next lesson!

Introduction to Schemas

What is a Schema?

In the last lesson we looked at DTDs, and how a DTD could be used to specify the elements and attributes that can occur in an XML document. These rules determine whether an XML document is valid.

Schemas also define a set of rules for XML documents, but are much more powerful than DTDs. DTDs can be included within an XML file or they can be external. Schemas are always external files. Schemas:

- are written in XML, so existing XML tools can be used to determine whether a schema is valid.
- support data types, so you can be sure that data types in your XML file are handled properly.
- are extensible, so they can handle future enhancements gracefully.

If schemas are so great, why do we need DTDs? Great question! As a casual XML user you may not need a schema to define the rules for your XML file. A simple internal DTD may be sufficient.

We are nearly ready to create our own schema, but before we do, we need to discuss **namespaces**.

What is a Namespace?

Namespaces are used in XML files to avoid naming collisions. Suppose you want to combine your phonebook XML file with your friend's, but your phonebook looks like this:

OBSERVE:

```
<Phone Type="home">
    312-555-1212
</Phone>
```

Meanwhile, your friend's phonebook XML file has **<Phone>** elements that contain other elements, like this:

OBSERVE:

```
<Phone>
    <Type>home</Type>
    <Number>312-555-1212</Number>
</Phone>
```

You can merge these two representations by using namespaces. Type the code below into a new XML file as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE CombinedPhoneBook>
<CombinedPhoneBook>
<my:Phone Type="home">
    312-555-1212
</my:Phone>
<other:Phone>
    <other:Type>home</other:Type>
    <other:Number>312-555-1212</other:Number>
</other:Phone>
</CombinedPhoneBook>
```

Save it in your **/xml1** folder as **AddressBook.xml**.

OBSERVE:

```
<?xml version="1.0"?>
<!DOCTYPE CombinedPhoneBook>
<CombinedPhoneBook>
<my:Phone Type="home">
    312-555-1212
</my:Phone>
<other:Phone>
    <other:Type>home</other:Type>
    <other:Number>312-555-1212</other:Number>
</other:Phone>
</CombinedPhoneBook>
```

This document has two namespaces with two prefixes: **my** and **other**. This document isn't exactly right, though, because we haven't specified the namespaces allowed in our document.

Check Syntax and note the errors:

OBSERVE:

```
Error: Namespace prefix my on Phone is not defined on line 3 column 15
Error: Namespace prefix other on Phone is not defined on line 6 column 13
Error: Namespace prefix other on Type is not defined on line 7 column 16
Error: Namespace prefix other on Number is not defined on line 8 column 18
```

To specify the namespaces allowed in our document, we need to modify our XML file's root element—in this case, **CombinedPhoneBook**. Change the document as shown below:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE CombinedPhoneBook>
<CombinedPhoneBook
xmlns:my="http://oreillyschool.com/myPhoneBook/"
xmlns:other="http://oreillyschool.com/otherPhoneBook/">
<my:Phone Type="home">
    312-555-1212
</my:Phone>
<other:Phone>
    <other:Type>home</other:Type>
    <other:Number>312-555-1212</other:Number>
</other:Phone>
</CombinedPhoneBook>
```

Save it and Check Syntax. No errors! Let's take a closer look at the new code:

OBSERVE:

```
xmlns:my="http://oreillyschool.com/myPhoneBook/"
xmlns:other="http://oreillyschool.com/otherPhoneBook/"
```

Here we used **xmlns** to define the **myPhoneBook** and **otherPhoneBook** namespaces, with URLs of **http://oreillyschool.com/myPhoneBook/** and **http://oreillyschool.com/otherPhoneBook/**. The URLs uniquely identify the namespace, and are not actually used by XML parsers. Instead, they are often used to direct humans to a web page describing the namespace.

This next concept is important, so let me drive it home:

- **my** is the prefix we are using for the **http://oreillyschool.com/myPhoneBook/** namespace
- **other** is the prefix we are using for the **http://oreillyschool.com/otherPhoneBook/** namespace.

If our XML document only contains elements and attributes from a single namespace, we can forget about the prefixes and still use a namespace by declaring a *default namespace*:

OBSERVE:

```
<?xml version="1.0"?>
<!DOCTYPE myPhoneBook>
<myPhoneBook xmlns="http://oreillyschool.com/myPhoneBook/">
<Phone Type="home">
    312-555-1212
</Phone>
</myPhoneBook>
```

This document still uses the same namespace as before (the *myPhoneBook* namespace), but we have omitted the prefix from the body of the document.

Schemas are just another XML document, but they are written using a specific namespace, as you will see shortly.

Your First Schema

Our little phonebook from the last lesson is fine for storing a limited amount of personal information for a limited number of contacts. But we can come up with a better structure that will let us store even more details about our friends, family, and business contacts.

Most people have multiple phone numbers, addresses, email addresses, instant message user names, and websites. Sometimes this information can be grouped into "home" and "work" categories. It would also be handy to store other details about our contacts such as birthdays, names of family members, and even random notes like "Alex is allergic to shellfish."

We'll store this information in an address book, which will eventually look something like this:

OBSERVE:

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

    <Contact>
        <FirstName>Alex</FirstName>
        <MiddleName>M.</MiddleName>
        <LastName>Chilton</LastName>
        <Birthday>1950-12-28</Birthday>

        <Group Name="Work">
            <Address>
                <Street>123 4th Street Suite 505</Street>
                <City>San Fransisco</City>
                <State>CA</State>
                <PostalCode>94101</PostalCode>
                <Country>US</Country>
            </Address>
            <Phone>555-1212</Phone>
            <IM Service="AOL">achil</IM>
            <Email>achil@bigcorp.us</Email>
            <WebSite>http://bigcorp.com.us/</WebSite>
        </Group>

        <Group Name="Home">
            <Phone>555-9152</Phone>
            <IM Service="Google">achil101</IM>
        </Group>

    </Contact>
</AddressBook>
```

We'll start with a straightforward and basic file. Clear your **AddressBook.xml** file, and type in the code below as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
</AddressBook>
```

In this new XML file, we've defined the namespace used by schemas. The text prefix of the namespace we are including within our document is **xsi**.

The file name of the XML schema itself is specified by the **xsi:noNamespaceSchemaLocation** attribute. The file name of our schema will be **AddressBook.xsd**.

Note Make sure you type file names exactly as you see them in the lesson. Case is important. If you save your files in different directories, you need to update your file names accordingly.

Check Syntax ⚙ Check the syntax. If everything is okay, you'll see **No errors found.**

Schemas are a type of XML file, so we'll continue to use the XML syntax in CodeRunner to author our file. Click **New File** (🗋), and type the code below as shown:

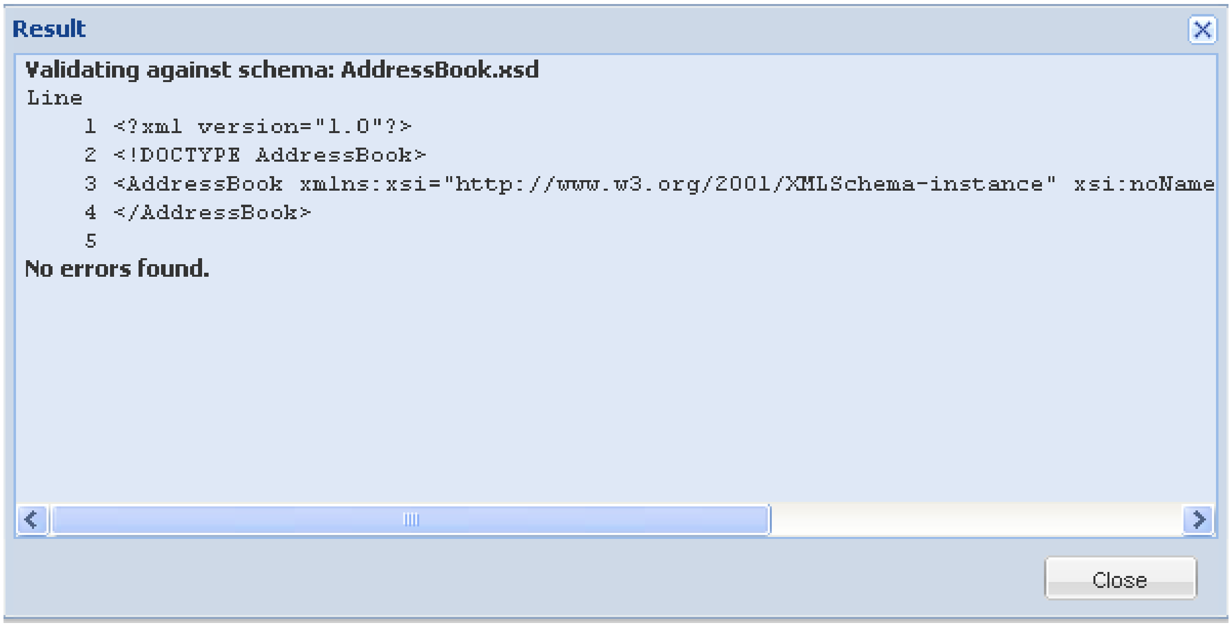
CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook">
  </xs:element>

</xs:schema>
```

💾 Save it in your **/xml1** folder as **AddressBook.xsd**. Here we used the **xs:element** element to define one allowable element in our XML file: an element named **AddressBook**.

Check Syntax ⚙ Check the syntax. If everything is okay, you will see **No errors found.**

Next, switch back to your **AddressBook.xml** file, and click Validate ⚙. If everything in your XML file and schema is correct, you'll see the following:



```
Validating against schema: AddressBook.xsd
Line
    1 <?xml version="1.0"?>
    2 <!DOCTYPE AddressBook>
    3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noName
    4 </AddressBook>
    5
No errors found.
```

Congratulations! You have created your first schema!

Now let's specify which child elements that AddressBook can contain. Add the code below to your AddressBook.xsd schema as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook">
    <xs:complexType>

    </xs:complexType>
  </xs:element>

</xs:schema>
```

Save it. We just added a **complexType** element, which marks AddressBook as a *complex element*. Complex elements can:

- be empty.
- contain only elements.
- contain only text.
- contain elements and text.

Note We'll discuss complex types and other data types in great detail in the next lesson.

The next step is to define the order in which we want child elements to occur using the **xs:sequence** element. AddressBook will have only one allowable child element named **Contact**, but that child element can occur as many times as needed.

Add the following code to your schema:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook">
    <xs:complexType>
      <xs:sequence>

      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

💾 Save it. With ordering in place, we can now set Contact as the child element of AddressBook. To do this, we'll use the **xs:element** element, along with some new bits. Add the code below to your schema:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

💾 Save it. Let's take a closer look:

OBSERVE:

```
        <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact"/>
```

In this code, we specify that our phonebook doesn't need **Contact** elements using **minOccurs="0"**, and that it can have as many contacts as we want by using **maxOccurs="unbounded"**.

💾 Save your schema again because it's time to take it for a test drive!

Switch back to your xml file and click Validate. If your schema and XML files are correct, you'll see **No errors found.** Excellent!

Now try adding a few Contact elements to your **AddressBook.xml** file. Type the code below as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
    <Contact>Alex Chilton</Contact>
    <Contact>Laura Chilton</Contact>
</AddressBook>
```

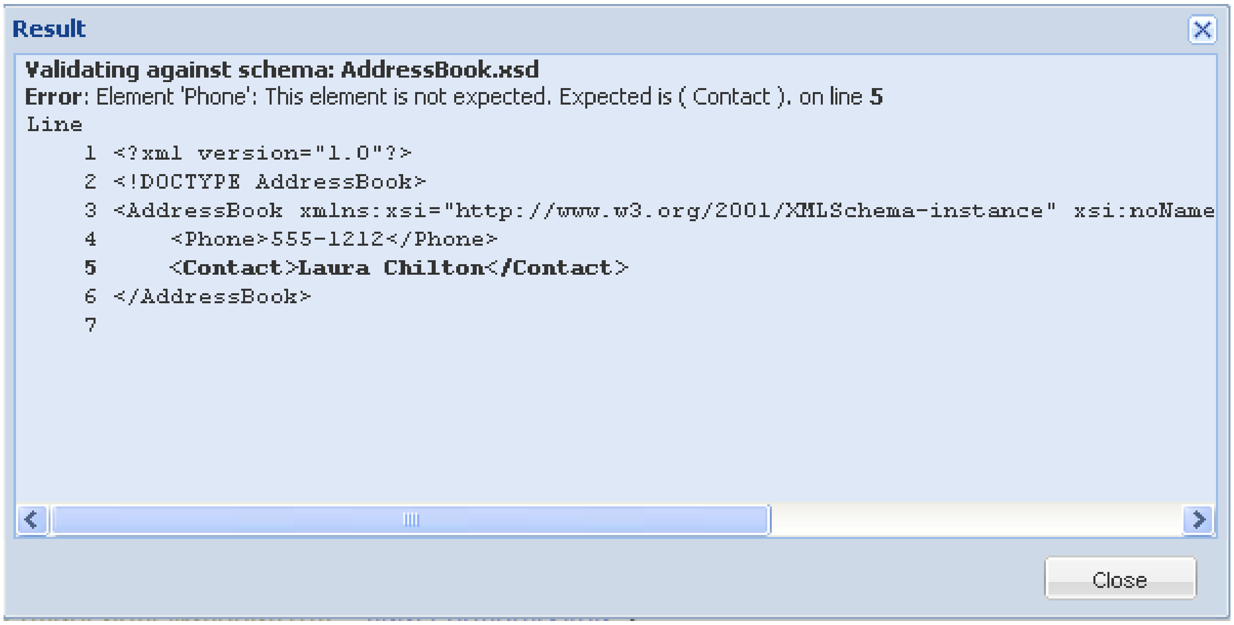
💾 and Validate. If your schema and XML files are correct, you'll see **No errors found.**

Let's introduce an error to the XML file, just to make sure the schema is working correctly. Change the first Contact element to a Phone element, as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
    <Phone>555-1212</Phone>
    <Contact>Laura Chilton</Contact>
</AddressBook>
```

💾 and Validate. Looks like our schema caught this error:



Validating against schema: AddressBook.xsd
Error: Element 'Phone': This element is not expected. Expected is (Contact). on line **5**

```
Line
    1 <?xml version="1.0"?>
    2 <!DOCTYPE AddressBook>
    3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noName
    4     <Phone>555-1212</Phone>
    5     <Contact>Laura Chilton</Contact>
    6 </AddressBook>
    7
```

Experiment with the XML document—try adding other elements or attributes to see what sort of error messages are returned. Remember the schema is not complete yet, so not everything will return an error.

When you finish experimenting, restore your **AddressBook.xml** to the last good version.

Excellent work so far! In this lesson we've only touched on the basics of schemas, but in the next lesson, we'll dive deeper into schemas and look at data types. See you there!

Data Types

Welcome back! In the last lesson, we had our first experience working with schemas. In this lesson, we will continue to add features to our Address Book schema.

Data Types

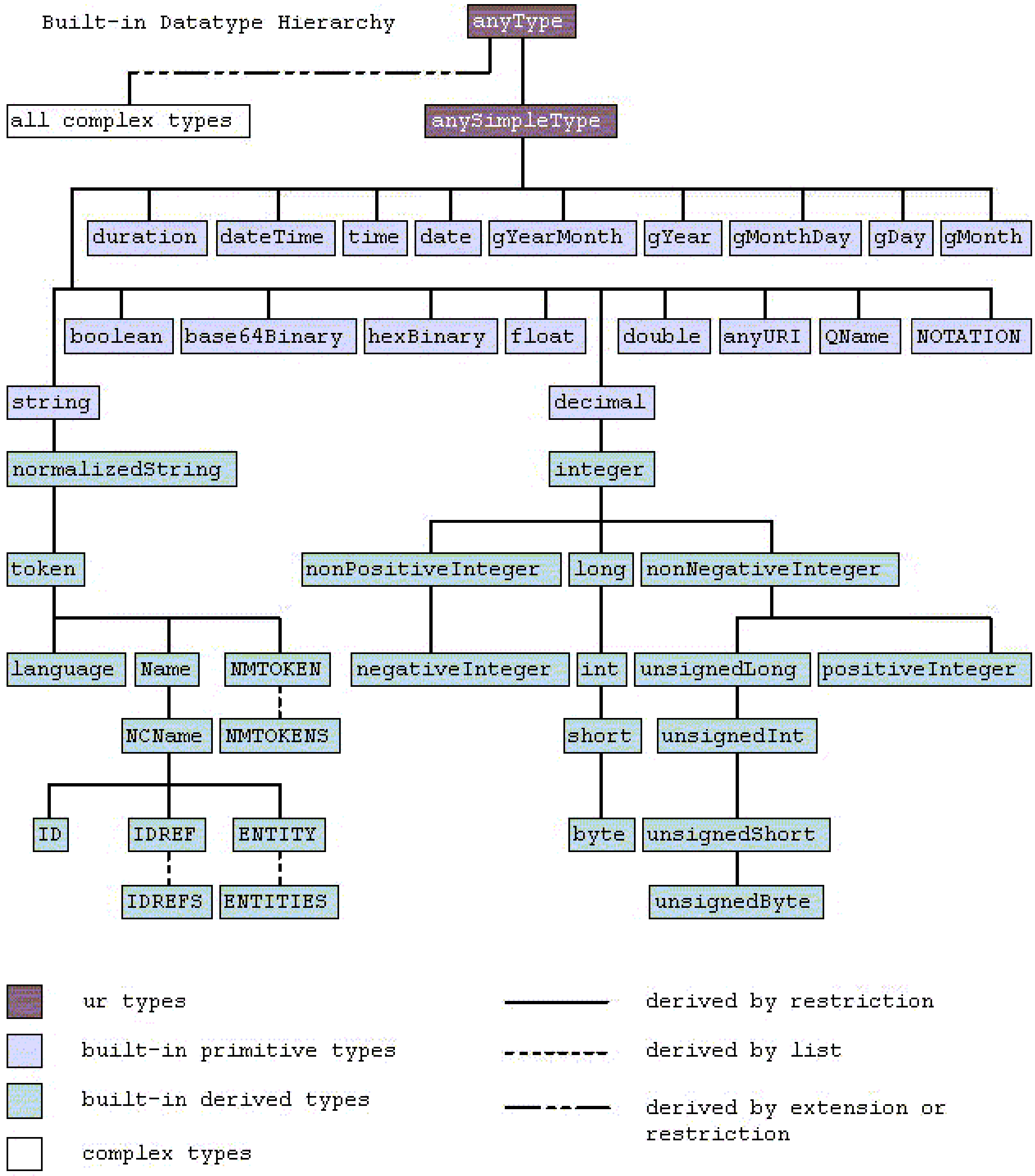
In the programming world, data types are like rules for allowed values. Here are some examples of data types that are common in many different programming languages:

Data Type	Sample Values
integer	-1 0 2 20195 -195039
decimal	10.59 0.00 -25.95
string	This is a sample string. #2 sample

Data types are important. You would be confused if you logged into your bank's web site and found your account balance to be a string like **O'Reilly Media** as opposed to a value like **$3590.85**.

The XML Schema definition describes several data types. Like XML itself, data types form a hierarchy. At the root of the hierarchy is *anyType*, which literally means "any type." Unless you specify otherwise, the data type for elements and attributes is assumed to be **anyType**.

The hierarchy looks like this:



For more information on built-in data types, visit w3.org.

There are two children of anyType: complex types, and *anySimpleType*. In XML, simple types cannot have attributes or child elements; complex types can.

Take a look at the integer data type above. It has additional specific child types, such as *nonPositiveInteger* and *positiveInteger*. Types can be extended to be more specific (for example, positive integers between 5 and 10). We'll work on this in a future lesson.

There are many different simple types defined—including lots of different types of integers, strings, and dates. Here are a few of the more important ones:

Data Type	Sample Values
	-1

xs:integer	0 2 20195 -195039
xs:decimal	10.59 0.00 -25.95
xs:string	This is a sample string. #2 sample
xs:date	2000-01-01

We will use both complex and simple types to expand our Address Book schema.

Complex Data Types

Before we get back to our schema, let's take another look at the address book we're building:

OBSERVE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">

    <Contact>
        <FirstName>Alex</FirstName>
        <MiddleName>M.</MiddleName>
        <LastName>Chilton</LastName>
        <Birthday>1950-12-28</Birthday>

        <Group Name="Work">
            <Address>
                <Street>123 4th Street Suite 505</Street>
                <City>San Fransisco</City>
                <State>CA</State>
                <PostalCode>94101</PostalCode>
                <Country>US</Country>
            </Address>
            <Phone>555-1212</Phone>
            <IM Service="AOL">achil</IM>
            <Email>achil@bigcorp.us</Email>
            <WebSite>http://bigcorp.com.us/</WebSite>
        </Group>

        <Group Name="Home">
            <Phone>555-9152</Phone>
            <IM Service="Google">achil101</IM>
        </Group>

    </Contact>
</AddressBook>
```

Our schema isn't finished yet, so our address book isn't quite that big. Open **AddressBook.xml** from the last lesson. It should look like this:

OBSERVE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
  <Contact>Alex Chilton</Contact>
  <Contact>Laura Chilton</Contact>
</AddressBook>
```

Next, open the **AddressBook.xsd** file from the last lesson. The schema should look like this:

OBSERVE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Currently, our AddressBook element has a definition for one data type—an unnamed complex type that specifies one element named Contact. In the last lesson, we learned that it's possible for complex elements to:

- be empty.
- contain only elements.
- contain only text.
- contain elements and text.

Then we used a **sequence** element as an *order* indicator. Possible order indicators are:

- **sequence**: child elements must occur in a specific order, and can occur as few or as many times as necessary.
- **all**: child elements can appear in any order, but each element can only occur once.
- **choice**: one child element or another can occur.

Contact can occur as many times as we want it to or not at all, because we set **minOccurs** to 0 and **maxOccurs** to unbounded (in this context, "unbounded" means "as many times as we need.") **minOccurs** and **maxOccurs** are known as *occurrence* indicators, because they specify the exact number of times an element can occur in a document.

This unnamed complex type is specific to the AddressBook element, and cannot be referenced or used anywhere else.

As an alternative to this setup, we could define an "Address Book" complex data type elsewhere in our XML file. Change your **AddressBook.xsd** schema as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook">
    ~~<xs:complexType>~~
      ~~<xs:sequence>~~
        ~~<xs:element minOccurs="0" maxOccurs="unbounded" name="Contact"/>~~
      ~~</xs:sequence>~~
    ~~</xs:complexType>~~
  </xs:element>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Here we defined a new data type named **AddressBookType**. We'll make one more change to our schema; we need to specify this data type for the AddressBook element. We can also shorten the **xs:element**:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Validate Validate your XML document against this updated schema. If you typed everything correctly, you'll see **No errors found.**

Now that we've defined AddressBookType, we'll define ContactType to support these elements:

- FirstName
- MiddleName
- LastName
- Birthday

There are several restrictions we need to place on this data. Contacts must be allowed only one first name, middle name, last name, and birthday. Birthdays should also be a valid date, as opposed to values like "Yes" or "Monday."

Let's start making changes! Edit your schema as shown below:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="ContactType"
/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

This change sets the data type on the Contact element to be ContactType. We still need to define this complex type.

Remember, ContactType is a complex data type because it will only contain other elements. So we'll use xs:complexType and xs:sequence in our schema. Add the code below to your schema as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="ContactType"
/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ContactType">
    <xs:sequence>

    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Next, we'll define the first four elements allowed inside of Contact, with appropriate rules to specify how many times each element can occur. Edit your schema as shown below:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="ContactType"
/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName"/>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Now we're getting somewhere! Let's test this schema. Save your schema, then switch back to your **AddressBook.xml** file, remove the existing names, and add a first and last name, as shown below:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
  ~~<Contact>Alex Chilton</Contact>~~
~~  <Contact>Laura Chilton</Contact>~~
  <Contact>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
  </Contact>
</AddressBook>
```

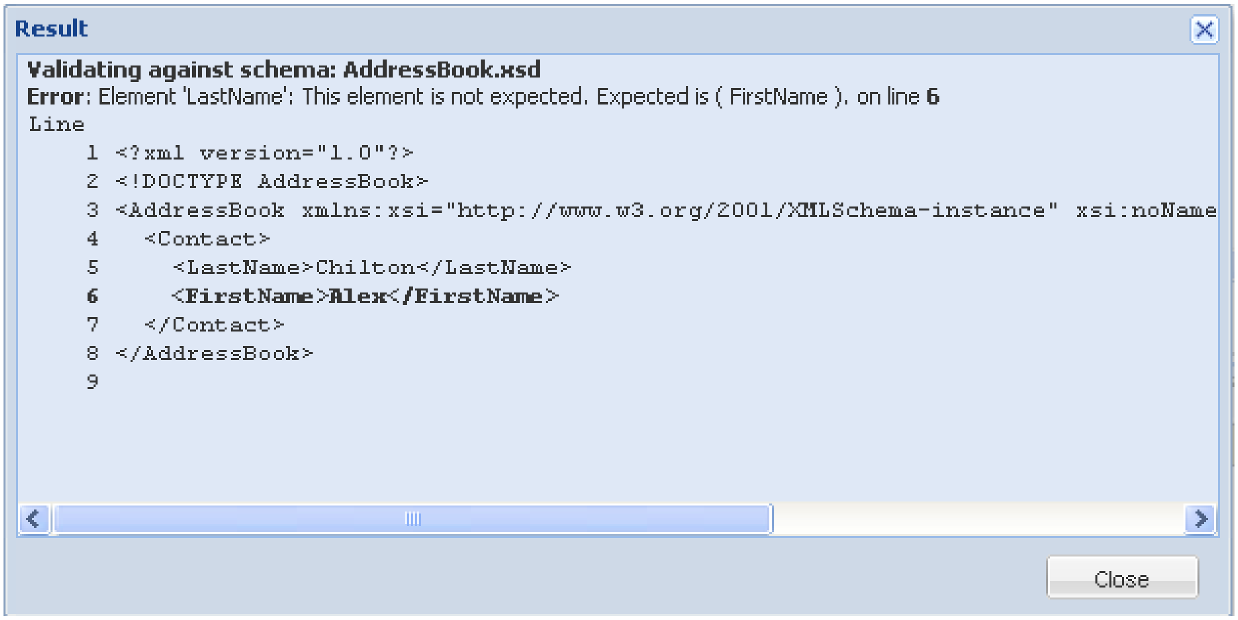
Validate Validate your XML. You'll see **No errors found.**

What happens if you change the order of the elements inside of Contact? Try it! Modify **AddressBook.xml** as shown below:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
  <Contact>
    <LastName>Chilton</LastName>
    <FirstName>Alex</FirstName>
    ~~<LastName>Chilton</LastName>~~
  </Contact>
</AddressBook>
```

Validate Validate your XML. It looks like the file has errors!



Validating against schema: AddressBook.xsd
Error: Element 'LastName': This element is not expected. Expected is (FirstName). on line **6**

```
Line
     1 <?xml version="1.0"?>
     2 <!DOCTYPE AddressBook>
     3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noName
     4   <Contact>
     5     <LastName>Chilton</LastName>
     6     <FirstName>Alex</FirstName>
     7   </Contact>
     8 </AddressBook>
     9
```

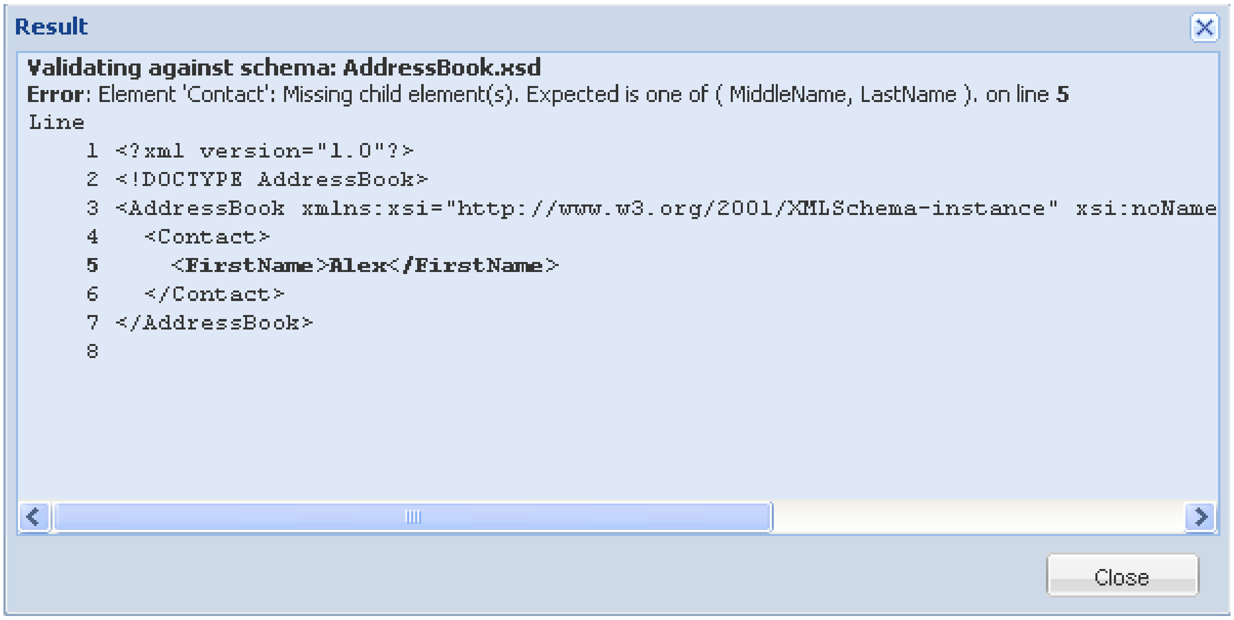
Order is important in our XML document because we specified a **sequence** in the schema:

OBSERVE:

```
.
.
.
  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName"/>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday"/>
    </xs:sequence>
  </xs:complexType>
.
.
.
```

Elements inside of **Contact** must occur in the following order: **FirstName**, **MiddleName**, **LastName**, and **Birthday**. **MiddleName** and **Birthday** do not have to occur at all, but **FirstName** and **LastName** must occur. All elements can only occur once.

Now delete the **LastName** from your XML document and **Validate**. You'll see an error like this:



Validating against schema: AddressBook.xsd
Error: Element 'Contact': Missing child element(s). Expected is one of (MiddleName, LastName). on line **5**

```
Line
    1 <?xml version="1.0"?>
    2 <!DOCTYPE AddressBook>
    3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noName
    4   <Contact>
    5     <FirstName>Alex</FirstName>
    6   </Contact>
    7 </AddressBook>
    8
```

Add the **LastName** back after **FirstName** to make this error go away.

Simple Data Types

To complete the definition of the elements in our schema, we need to specify the simple data type (or allowed values) for each element in our **ContactType**.

Simple data types cannot have attributes or child elements. This works well for elements like **FirstName** and **Birthday**—we don't want those elements to have children or any attributes.

In the chart at the beginning of this lesson, we saw a few simple data types that schemas can use. Here are a few:

Data Type	Schema Type	Sample Values
Integer	xs:integer	5, 10, -50 or 29059259
Decimal	xs:decimal	5.29, 10.90, -50.000001 or 29059259.6
String	xs:string	Any character, including punctuation and spaces!
Date	xs:date	2010-05-01

Let's start by picking an appropriate string data type for the name elements.

String Types

Strings often contain *white space*: spaces, empty lines, and tabs. As you might remember from an earlier lesson, *white space in XML is important*. Consider this short example of XML:

OBSERVE:

```
<name>
 John Smith
</name>
<address>
150 N. Michigan
Suite 500
Chicago, IL 60601
</address>
```

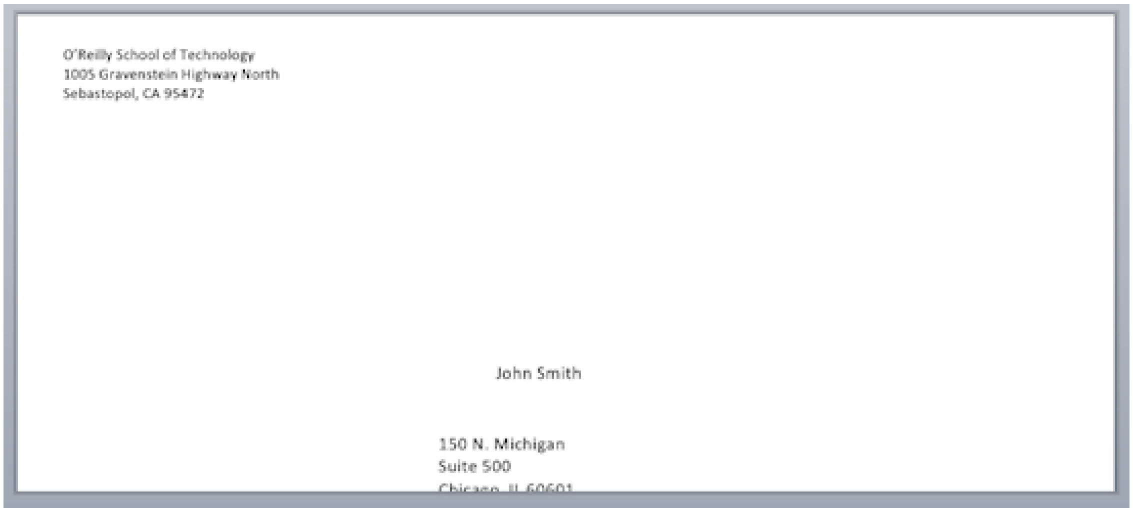
When an XML program reads this element, it might see this:

OBSERVE:

```
<name>\n\tJohn Smith\n</name><address>\n150 N. Michigan\nSuite 500\nChicago, IL
60601\n</address>
```

In our example, new lines were replaced by the special sequence **\n** and tabs were replaced by **\t**. You may not have noticed this before, but the beginning and the end of the name and address elements had newline characters.

We humans don't care about newline or tab characters. We are able to read over the address and interpret everything correctly. Those newlines and tabs do matter to programs, though. If a program read those new elements in order to print an envelope, the results might look something like this:



O'Reilly School of Technology
1005 Gravenstein Highway North
Sebastopol, CA 95472

John Smith

150 N. Michigan
Suite 500
Chicago, IL 60601

The name is indented due to the tab, and the city is cut off due to the extra newlines.

XML schemas let you combat this problem by using data types. These data types don't restrict the types of characters used in your elements and attributes, instead they tell your XML parser how to deal with the whitespace that may occur in your document.

There are three string data types allowed in schemas:

- **xs:string**—any characters, including any type of white space
- **xs:normalizedString**—characters, with tabs and newlines replaced by spaces
- **xs:token**—characters, with tabs, newlines, and multiple spaces removed

The best data type for the **FirstName**, **MiddleName**, and **LastName** elements is **xs:token**, because we don't want or need any tabs or newlines in our contacts' names.

Let's update our schema to include this data type. Change **AddressBook.xsd** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="Conta
ctType"/>
    </xs:sequence>
  </xs:complexType>


  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xs:token"/
>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Dates

Now the **Birthday** element needs a data type. Most people don't care about the time someone was born, just the date, so we'll use the **xs:date** data type. Update your schema so it looks like this:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="Conta
ctType"/>
    </xs:sequence>
  </xs:complexType>


  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xs:token"/
>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Dates must be in a specific format: **YYYY-MM-DD** where **YYYY** is the four-digit year (like **2012**), **MM** is the two-digit month (like **10**), and **DD** is the two-digit day (like **16**). A valid date for February 5, 2013 would be **2013-02-05**.

Save your schema, then switch back to **AddressBook.xml** and add a **Birthday** element:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
  <Contact>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Birthday>1950-12-28</Birthday>
  </Contact>
</AddressBook>
```

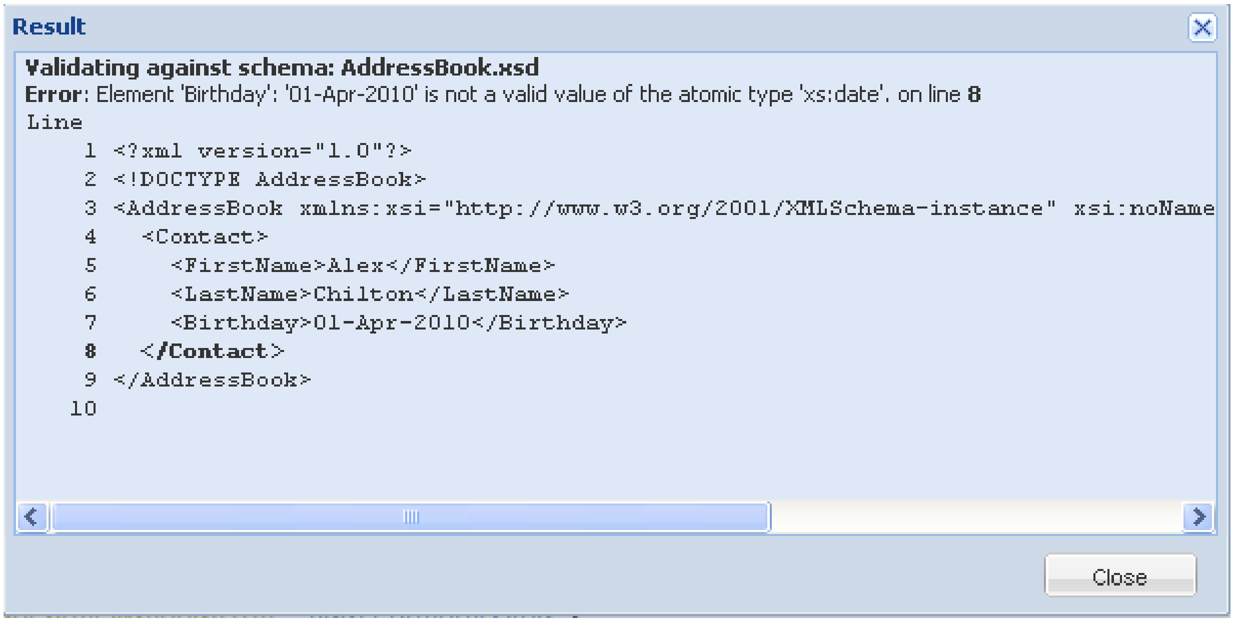
💾 and Validate —you should see **No errors found.**

What happens if you enter an invalid date? Try it. Change **AddressBook.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
    <Contact>
       <FirstName>Alex</FirstName>
       <LastName>Chilton</LastName>
       <Birthday>01-Apr-2010</Birthday>
    </Contact>
</AddressBook>
```

💾 and Validate . That date format isn't valid, so you'll see an error like this:



Validating against schema: AddressBook.xsd
Error: Element 'Birthday': '01-Apr-2010' is not a valid value of the atomic type 'xs:date'. on line **8**

```
Line
     1 <?xml version="1.0"?>
     2 <!DOCTYPE AddressBook>
     3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noName
     4   <Contact>
     5     <FirstName>Alex</FirstName>
     6     <LastName>Chilton</LastName>
     7     <Birthday>01-Apr-2010</Birthday>
     8   </Contact>
     9 </AddressBook>
    10
```

Our schema protects us from entering data in inconsistent form. Switch the birthday back before you continue.

Attributes

Now that we've seen how basic data types work, and we have basic contact information started in our Address Book, let's implement the next element: **Group**. This element will let us group our home and work contact information together.

In **AddressBook.xml**, add the Group element as shown below:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
  <Contact>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Birthday>1950-12-28</Birthday>
    <Group Name="Work">

    </Group>
  </Contact>
</AddressBook>
```

Switch back to **AddressBook.xsd** and update the **Contact** element:

CODE TO TYPE:

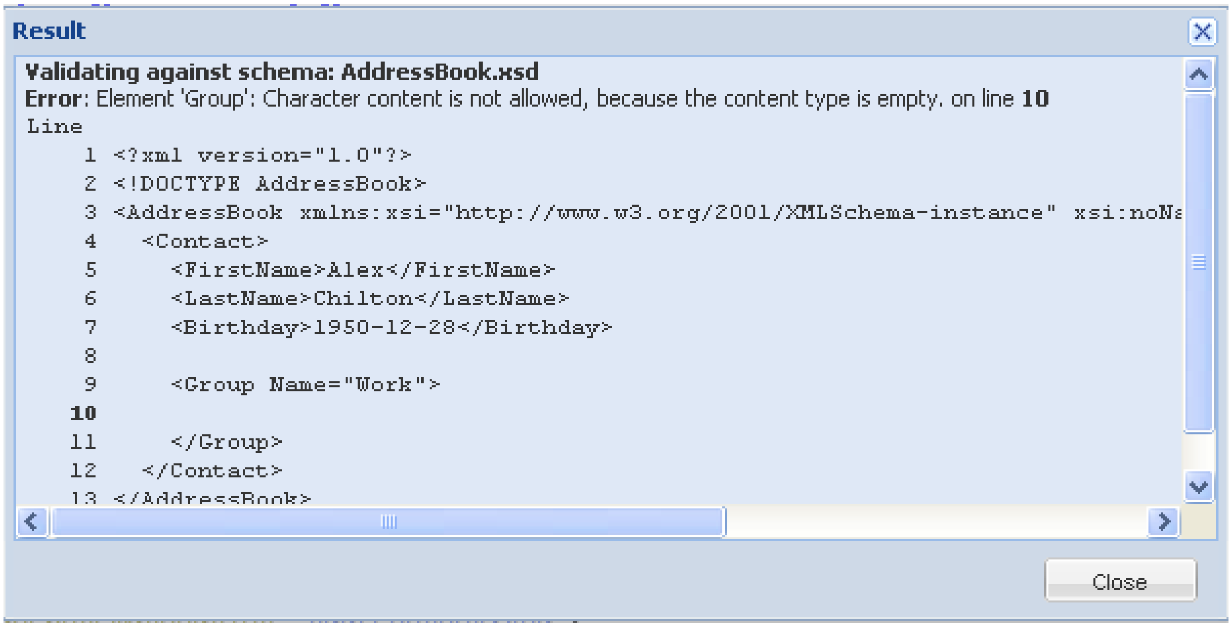
```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="ContactType"
/>
    </xs:sequence>
  </xs:complexType>

   <xs:complexType name="ContactType">
     <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="xs:token"/>
        <xs:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xs:token"/>
        <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
        <xs:element minOccurs="0" maxOccurs="1" name="Birthday" type="xs:date"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="Group" type="GroupType"/>
     </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Next we need to define our **GroupType** complex type. The syntax for adding an attribute is nearly the same as the syntax for adding an element. In your schema, add a new complex type named **GroupType** with a single **Name** attribute, as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="ContactType"
/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xs:token"/>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday" type="xs:date"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Group" type="GroupType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="GroupType">
    <xs:attribute name="Name" type="xs:token"/>
  </xs:complexType>

</xs:schema>
```

Save your schema, then switch back to your XML document and Validate. You'll see an error:



Result

Validating against schema: AddressBook.xsd
Error: Element 'Group': Character content is not allowed, because the content type is empty. on line **10**

```
Line
     1 <?xml version="1.0"?>
     2 <!DOCTYPE AddressBook>
     3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNa
     4   <Contact>
     5     <FirstName>Alex</FirstName>
     6     <LastName>Chilton</LastName>
     7     <Birthday>1950-12-28</Birthday>
     8
     9     <Group Name="Work">
    10
    11     </Group>
    12   </Contact>
    13 </AddressBook>
```

The attribute content is defined (xs:token) but we also need to define the content for the element itself.

Extensions

To define the content allowed in the Group element, we need to add an extension to our definition of GroupType. Modify **AddressBook.xsd** as shown

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="ContactType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xs:token"/>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday" type="xs:date"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Group" type="GroupType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="GroupType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="Name" type="xs:token"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

</xs:schema>
```

We are declaring that GroupType has simpleContent, which means only character data, and we've set the type to xs:string, which will allow characters, line feeds, carriage returns and tabs.

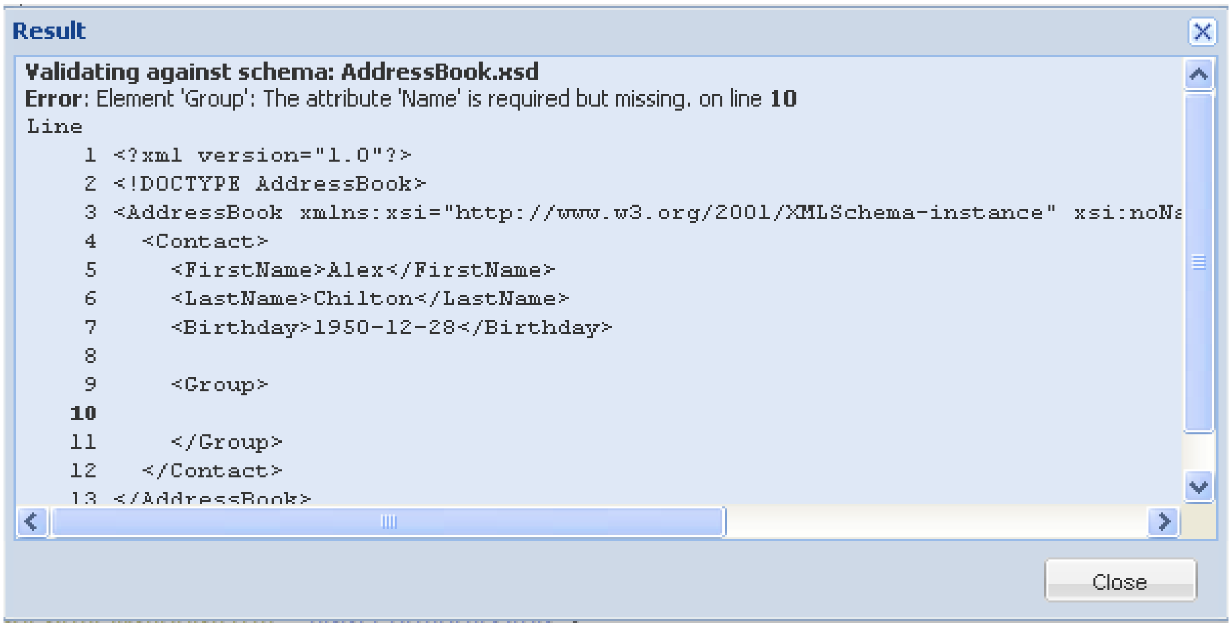
Save your schema, then switch back to your XML document and Validate. If you typed everything correctly, you will see **No errors found.**

So, what happens if you remove the **Name** attribute from your **Group** element? Try it. You should still see **No errors found.** By default, attributes are optional. We can change this in our schema, though. Switch back to **AddressBook.xsd** and modify it as shown (we'll omit some lines here for brevity's sake):

CODE TO TYPE:

```
...
  <xs:complexType name="GroupType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="Name" type="xs:token" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
...
```

Save your schema, switch back to your XML document and remove the attribute, and then, Validate. This time you'll see an error:



Validating against schema: AddressBook.xsd
Error: Element 'Group': The attribute 'Name' is required but missing. on line **10**

```
Line
    1 <?xml version="1.0"?>
    2 <!DOCTYPE AddressBook>
    3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNa
    4   <Contact>
    5     <FirstName>Alex</FirstName>
    6     <LastName>Chilton</LastName>
    7     <Birthday>1950-12-28</Birthday>
    8
    9     <Group>
   10
   11     </Group>
   12   </Contact>
   13 </AddressBook>
```

use can have one of three values: optional (the default), required, or prohibited. Specifying **use="prohibited"** will disallow the use of the attribute on the element.

Phew! We've covered a lot of material in this lesson! In the next lesson we'll expand our schema by adding more checks and restrictions. See you soon!

Restrictions

Welcome back! In the last lesson we learned about data types. Data types let us apply rules to our XML documents, like requiring the **Birthday** element in our address book to be a valid date, in a specific format. Data types are the initial way we specify these kinds of rules. There are additional tools we can use to control our XML documents.

Restrictions

Before we add restrictions to our schema, let's make things more interesting and add a few more elements and attributes. Open your address book XML file and your schema from the last lesson and let's add to our address book. Start a new file, or change your address book so it looks like this:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AddressBook>
<AddressBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AddressBook.xsd">
  <Contact>
    <FirstName>Alex</FirstName>
    <MiddleName>M.</MiddleName>
    <LastName>Chilton</LastName>
    <Birthday>1950-12-28</Birthday>
    <Group Name="Work">
      <Address>
        <Street>123 4th Street Suite 505</Street>
        <City>San Francisco</City>
        <State>CA</State>
        <PostalCode>94101</PostalCode>
        <Country>US</Country>
      </Address>
      <Phone>555-1212</Phone>
      <IM Service="AOL">achil</IM>
      <Email>achil@bigcorp.us</Email>
      <WebSite>http://bigcorp.us/</WebSite>
    </Group>
  </Contact>
</AddressBook>
```

Save it, and open **AddressBook.xsd** and modify it as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="AddressBook" type="AddressBookType"/>

  <xs:complexType name="AddressBookType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Contact" type="ContactType"
/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xs:token"/>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday" type="xs:date"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Group" type="GroupType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="GroupType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Address" type="AddressType"
/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Phone" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="IM" type="IMType"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Email" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="WebSite" type="xs:token"/>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:token" use="required"/>
  </xs:complexType>

  <xs:complexType name="AddressType">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Street" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="City" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="State" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="PostalCode" type="xs:token"
/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Country" type="xs:token"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="IMType">
     <xs:simpleContent>
        <xs:extension base="xs:token">
           <xs:attribute name="Service" type="xs:token" use="required"/>
        </xs:extension>
     </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

Notice that we removed the **xs:simpleContent** and **xs:extension** statements from the **GroupType** definition. That's because we've added child elements to **Group**. We'll create rules for those elements next. Where did we have to add **xs:simpleContent** and **xs:extension** statements? Why?

Set of Values

The address book XML file we've created has a new element, the **IM**:

OBSERVE:

```
<IM Service="AOL">achil</IM>
```

The **IM** element in our address book has only one attribute: **Service**. Currently, any text is allowed for **Service**—it could be:

- AOL
- aim
- AIM
- Google
- Jabber
- My Own IM Service

Don't forget though, **aim** and **AIM** have different meanings for a computer, because case is usually important.

Suppose we decide to restrict allowed values for **Service** to a few of the most popular IM Services:

- AIM
- Facebook
- Google
- ICQ
- MSN
- Skype
- Yahoo

This restriction would make it easier for us to handle text, because then we wouldn't have to worry about which records contain **aim**, **AIM**, **aIM**, or any other case variations.

We'll specify these values in our schema by creating our own data type that has a *restriction*. We'll call this data type **IMServiceType**.

There are a couple of terms that the XML Schema specification uses in conjunction with data types and restrictions. You're likely to encounter these terms as you work with XML schemas and programs, so you'll want to be familiar with them:

- A **value space** is the set of values for a given data type. For our **IMServiceType**, the set of values would include AIM, Facebook, Google, ICQ, MSN, Skype, and Yahoo.
- A **facet** is a single defining aspect of a value space. Facets are like the building blocks of a data type—they describe the data type and its allowed values. Facets come in two types:
  - **fundamental** facets let you specify details such as whether the data type is numeric, or whether it contains a certain number of elements.
  - **constraining (or non-fundamental)** facets let you specify the size values are allowed to be, whether they can include white space, or whether values are members in a set.

In order to restrict our IMServiceType data type to a set of values, we'll use an **enumeration** constraining facet. Let's start by defining our IMServiceType data type. Type the code below in your **AddressBook.xsd** as shown:

CODE TO TYPE:

```
...
  <xs:complexType name="IMType">
    <xs:simpleContent>
      <xs:extension base="xs:token">
        <xs:attribute name="Service" type="IMServiceType" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:simpleType name="IMServiceType">
    <xs:restriction base="xs:token">
      <xs:enumeration value="AIM"/>
      <xs:enumeration value="Google"/>
      <xs:enumeration value="MSN"/>
      <xs:enumeration value="Yahoo"/>
      <xs:enumeration value="Skype"/>
      <xs:enumeration value="ICQ"/>
      <xs:enumeration value="Facebook"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

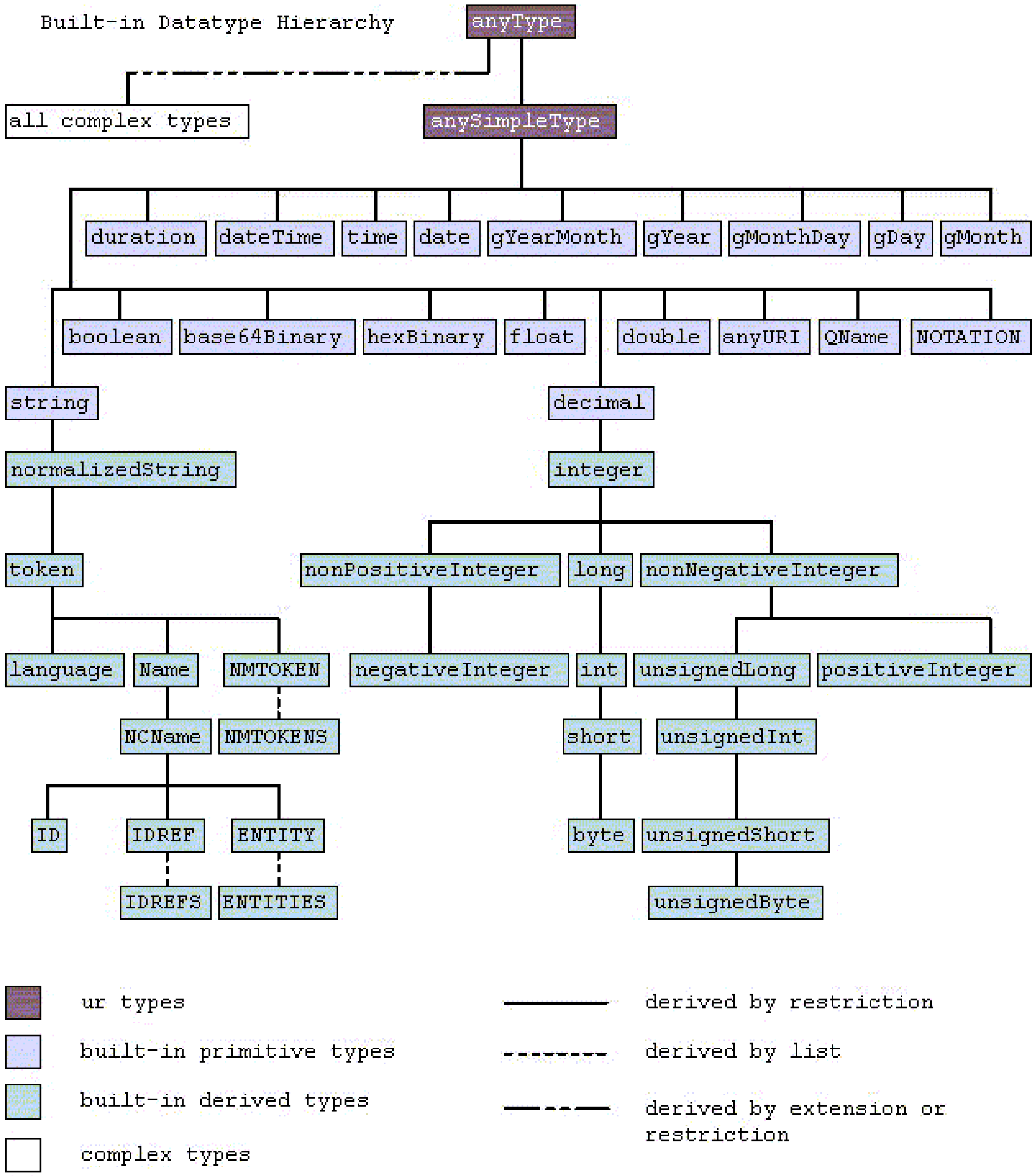
Let's take a look at the new code:

OBSERVE:

```
  <xs:simpleType name="IMServiceType">
    <xs:restriction base="xs:token">
      <xs:enumeration value="AIM"/>
      <xs:enumeration value="Google"/>
      <xs:enumeration value="MSN"/>
      <xs:enumeration value="Yahoo"/>
      <xs:enumeration value="Skype"/>
      <xs:enumeration value="ICQ"/>
      <xs:enumeration value="Facebook"/>
    </xs:restriction>
  </xs:simpleType>
```

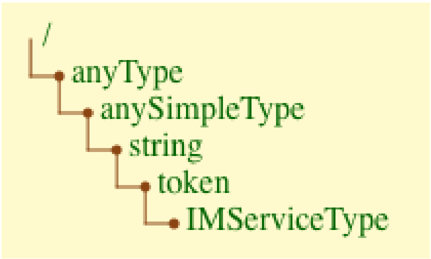
With this addition to our schema, we've defined a new **simple type** named **IMServiceType**. It has a **restriction** that is based on **xs:token**, and its allowed values are **enumerated**.

Take a look back at the type hierarchy:



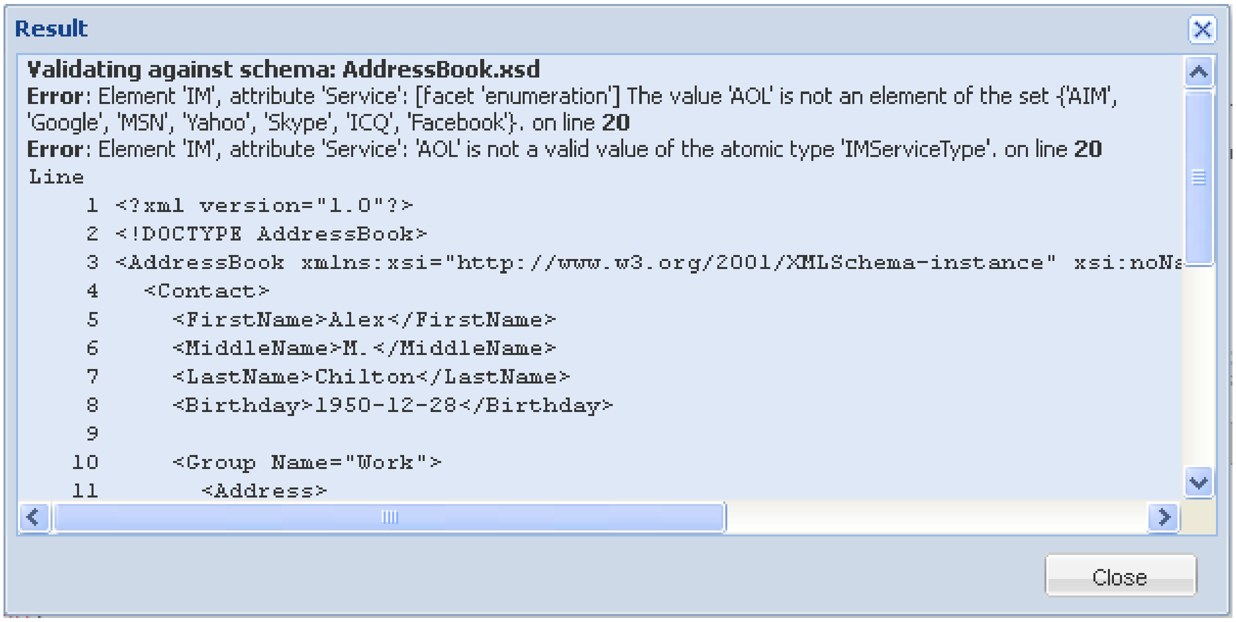
Our new **IMServiceType** is an extension of the **token** simple type, which is itself an extension of the **string** simple type, which is an extension of the **anySimpleType** simple type, which is an extension of **anyType**.

Graphically, this extension might look like:



Since **IMServiceType** extends (or *inherits* from) **token**, its values must also follow **token**'s rules. This inheritance is powerful! Instead of specifying every rule for your data types, you only need to add rules to an existing set of rules.

Switch back to **AddressBook.xml** and Validate. You'll see an error like this:



Validating against schema: AddressBook.xsd
Error: Element 'IM', attribute 'Service': [facet 'enumeration'] The value 'AOL' is not an element of the set {'AIM', 'Google', 'MSN', 'Yahoo', 'Skype', 'ICQ', 'Facebook'}. on line **20**
Error: Element 'IM', attribute 'Service': 'AOL' is not a valid value of the atomic type 'IMServiceType'. on line **20**

```
Line
     1 <?xml version="1.0"?>
     2 <!DOCTYPE AddressBook>
     3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNa
     4   <Contact>
     5     <FirstName>Alex</FirstName>
     6     <MiddleName>M.</MiddleName>
     7     <LastName>Chilton</LastName>
     8     <Birthday>1950-12-28</Birthday>
     9
    10     <Group Name="Work">
    11       <Address>
```

This error tells us that the current value of the **Service** attribute—AOL—is not allowed. Change it to **AIM** and validate again. This time you'll see **No errors found.**

Range of Values

Suppose we wanted to restrict the allowed values for **Birthday**. After all, we don't have any friends who were born in the 1800s, or after the current year. Enumerating the allowed values works well for the **Service** attribute on the **IM** element, but it wouldn't work as well for the **Birthday** element. Instead, we can specify a limit on a range of values.

To add this restriction, we need to add a new simple data type. Switch back to your schema and scroll to the bottom. Add the code below as shown:

CODE TO TYPE:

```
...
  <xs:simpleType name="BirthdayType">
    <xs:restriction base="xs:date">
      <xs:minInclusive value="1900-01-01"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Take a look at the code we added:

OBSERVE:

```
...
  <xs:simpleType name="BirthdayType">
    <xs:restriction base="xs:date">
      <xs:minInclusive value="1900-01-01"/>
    </xs:restriction>
  </xs:simpleType>
```

Once again, we defined our simple data type, named **BirthdayType**. It **extends the date** data type, and has one rule: the minimum date is **January 1st, 1900**.

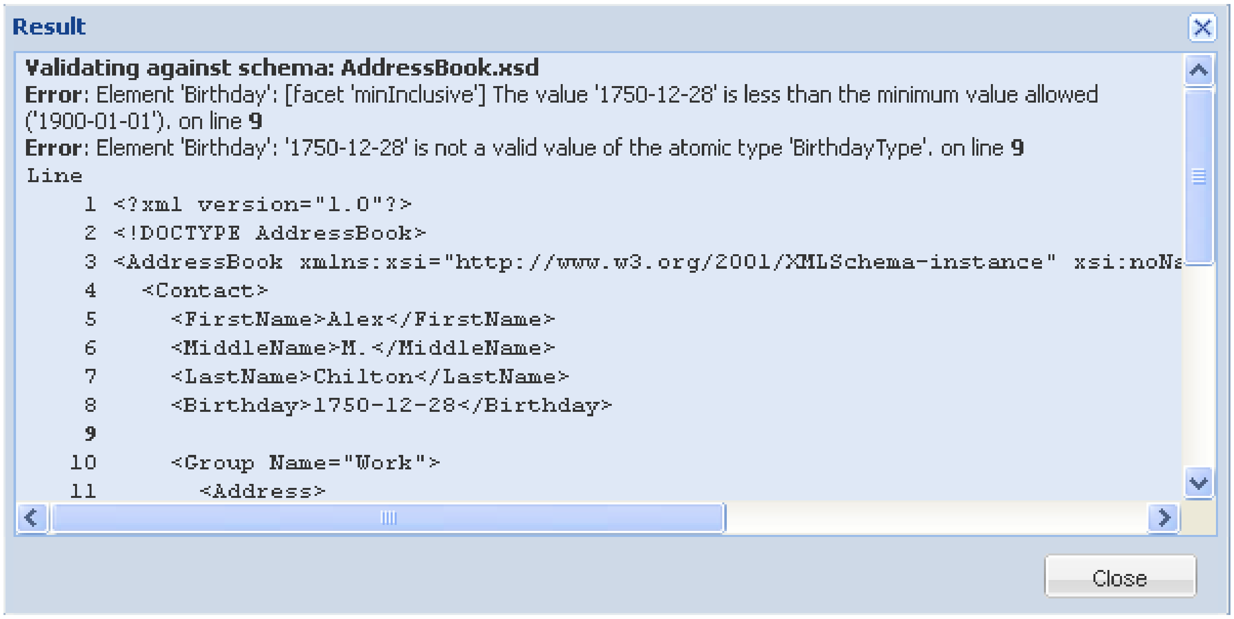
Now, to take this change "live," we need to update the data type specified in our **Birthday** element. Modify the code as shown:

CODE TO TYPE:

```
...
  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xs:token"/>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday" type="BirthdayType"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Group" type="GroupType"/>
    </xs:sequence>
  </xs:complexType>
...lines omitted...
```

💾 and switch back to **AddressBook.xml** and Validate.

You will see **No errors found**—because Alex Chilton's birthday is December 28th, 1950, above the minimum. Try setting it to December 28th, 1750, then Validate again. You'll see an error like this:



Validating against schema: AddressBook.xsd
Error: Element 'Birthday': [facet 'minInclusive'] The value '1750-12-28' is less than the minimum value allowed ('1900-01-01'). on line **9**
Error: Element 'Birthday': '1750-12-28' is not a valid value of the atomic type 'BirthdayType'. on line **9**

```
Line
    1 <?xml version="1.0"?>
    2 <!DOCTYPE AddressBook>
    3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNa
    4   <Contact>
    5     <FirstName>Alex</FirstName>
    6     <MiddleName>M.</MiddleName>
    7     <LastName>Chilton</LastName>
    8     <Birthday>1750-12-28</Birthday>
    9
   10     <Group Name="Work">
   11       <Address>
```

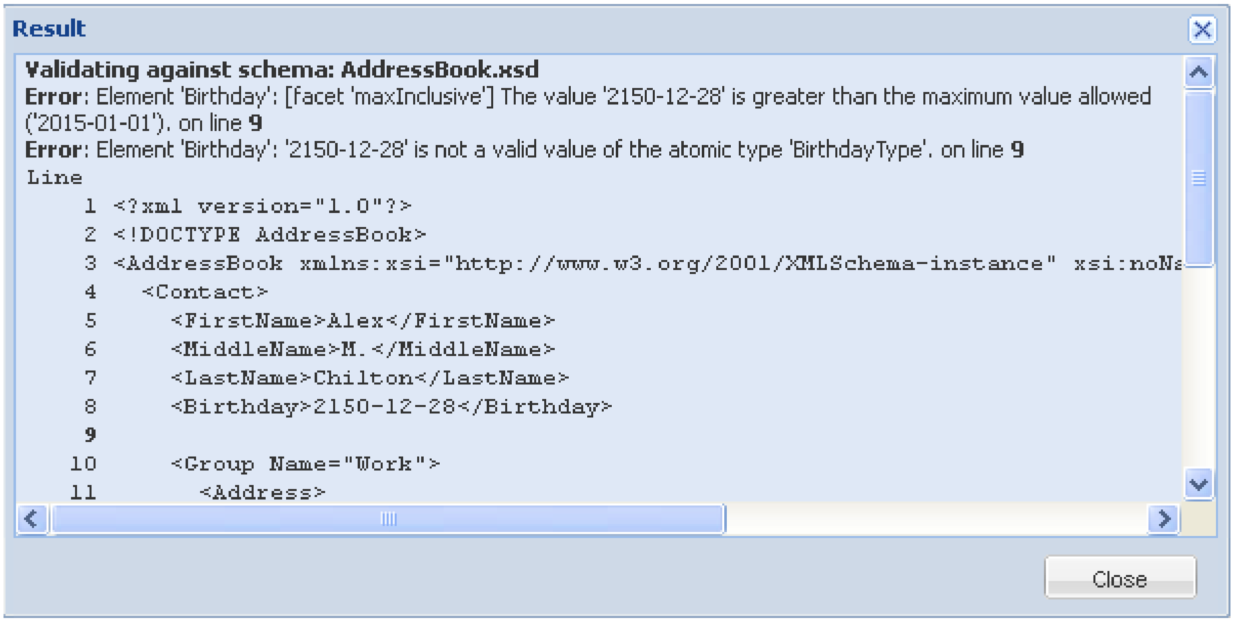
This error tells you that the value you provided for **Birthday** is not correct because it violates the minValue *facet* (*restriction*) as specified in the schema.

We also want to limit the maximum value for birthdays. We can do this by adding a **maxInclusive** element to our restriction as shown:

CODE TO TYPE:

```
...
  <xs:simpleType name="BirthdayType">
    <xs:restriction base="xs:date">
      <xs:minInclusive value="1900-01-01"/>
      <xs:maxInclusive value="2015-01-01"/>
    </xs:restriction>
  </xs:simpleType>
```

💾 Switch back to your XML document, and change the **Birthday** to a large value, like **2150-12-28**. You'll see this error message:



Validating against schema: AddressBook.xsd
Error: Element 'Birthday': [facet 'maxInclusive'] The value '2150-12-28' is greater than the maximum value allowed ('2015-01-01'). on line **9**
Error: Element 'Birthday': '2150-12-28' is not a valid value of the atomic type 'BirthdayType'. on line **9**

```
Line
    1 <?xml version="1.0"?>
    2 <!DOCTYPE AddressBook>
    3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNa
    4   <Contact>
    5     <FirstName>Alex</FirstName>
    6     <MiddleName>M.</MiddleName>
    7     <LastName>Chilton</LastName>
    8     <Birthday>2150-12-28</Birthday>
    9
   10     <Group Name="Work">
   11       <Address>
```

Switch Alex Chilton's birthday back to a normal value.

Length

Eventually we might want to import our address book into a database. Most databases require you to specify a maximum size for fields like **FirstName**. Since most of our friends have short names, we'll make the **FirstName** field in our database only 50 characters long.

This restriction can live outside of the database as well when we place a length restriction on our XML schema. Add a new simple type to **AddressBook.xsd**, named **NameType**, as shown:

CODE TO TYPE:

```
...
  <xs:simpleType name="NameType">
    <xs:restriction base="xs:token">
      <xs:maxLength value="50"/>
    </xs:restriction>
  </xs:simpleType>
```

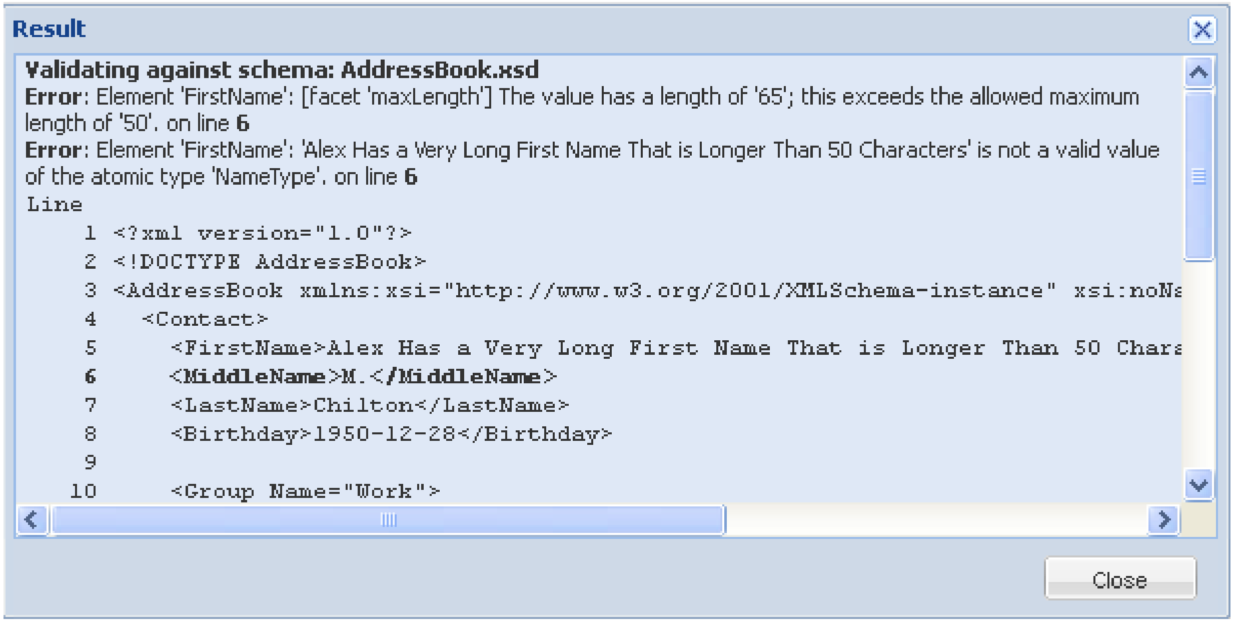
Also, update the data type on the FirstName element as shown:

CODE TO TYPE:

```
...
  <xs:complexType name="ContactType">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="NameType"/>
      <xs:element minOccurs="0" maxOccurs="1" name="MiddleName" type="xs:token"/
>
      <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
      <xs:element minOccurs="0" maxOccurs="1" name="Birthday" type="xs:date"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Group" type="GroupTy
pe"/>
     </xs:sequence>
   </xs:complexType>
...
```

💾 and switch back to **AddressBook.xml** and Validate. Since the first name is short, you'll see the standard **No errors found** message.

Now change the **FirstName** to something more than 50 characters long—like **Alex Has a Very Long First Name That is Longer Than 50 Characters**. Validate your document—now you'll see a new error:



Validating against schema: AddressBook.xsd
Error: Element 'FirstName': [facet 'maxLength'] The value has a length of '65'; this exceeds the allowed maximum length of '50'. on line **6**
Error: Element 'FirstName': 'Alex Has a Very Long First Name That is Longer Than 50 Characters' is not a valid value of the atomic type 'NameType'. on line **6**

```
Line
    1 <?xml version="1.0"?>
    2 <!DOCTYPE AddressBook>
    3 <AddressBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNa
    4   <Contact>
    5     <FirstName>Alex Has a Very Long First Name That is Longer Than 50 Chara
    6     <MiddleName>M.</MiddleName>
    7     <LastName>Chilton</LastName>
    8     <Birthday>1950-12-28</Birthday>
    9
   10     <Group Name="Work">
```

If your application requires something different, you can also use **minLength** to force your value to have a minimum length, or just **length** if your value must be an exact number of characters. Change Alex's name back to "Alex" before you continue!

Pattern Values

Another way we can restrict the content of an element is to set a pattern value. You could use this to ensure that the values for the **State** element are two uppercase letters. We'll need to add a new simple type and then define a pattern value for that type. Edit **AddressBook.xsd** as shown:

CODE TO TYPE:

```
...
  <xs:simpleType name="StateAbbreviation">
    <xs:restriction base="xs:token">
      <xs:pattern value="([A-Z]{2})"/>
    </xs:restriction>
  </xs:simpleType>
```

Next, update the State element as shown:

CODE TO TYPE:

```
...lines omitted...
 <xs:complexType name="AddressType">
   <xs:sequence>
     <xs:element minOccurs="0" maxOccurs="unbounded" name="Street" type="xs:token"/>
     <xs:element minOccurs="0" maxOccurs="unbounded" name="City" type="xs:token"/>
     <xs:element minOccurs="0" maxOccurs="unbounded" name="State" type="StateAbbreviation"/>
     <xs:element minOccurs="0" maxOccurs="unbounded" name="PostalCode" type="xs:token"/>
     <xs:element minOccurs="0" maxOccurs="unbounded" name="Country" type="xs:token"/>
   </xs:sequence>
 </xs:complexType>
...
```

and switch back to **AddressBook.xml** and Validate.

Now change **State** from **CA** to **Ca**, or something other than two capital letters. You should get a new error message. Try a variety of different patterns. Be sure to set the state back to CA before you proceed.

What if we wanted to restrict pattern values for a complex element, like **IM**? Change **AddressBook.xsd** as shown:

CODE TO TYPE:

```
...
  <xs:complexType name="IMType">
    <xs:simpleContent>
      <xs:extension base="IMValues">
        <xs:attribute name="Service" type="IMServiceType" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
```

Now we just need to add a definition of IMValues. Let's set it so that only lowercase letters are allowed.

CODE TO TYPE:

```
...lines omitted...
  <xs:simpleType name="IMValues">
    <xs:restriction base="xs:token">
      <xs:pattern value="([a-z]*)"/>
    </xs:restriction>
  </xs:simpleType>
```

💾, switch back to **AddressBook.xml**, and try various values for the IM Service name, like **jChilton**, and Validate ⚙, so you are familiar with the kinds of errors this will produce.

We covered a lot in this lesson! In the next lesson we'll learn about different ways to structure our schemas to make the process of writing and using them more efficient. See you there!

Schema Structure

Welcome back! We've already learned a lot about schemas. In this lesson we'll review what we've learned, and then discuss different ways to structure XML schemas.

Names and Refs

In earlier lessons, we added various data types for our elements and attributes:

- AddressBookType
- ContactType
- GroupType
- AddressType
- IMType
- IMServiceType

Defining new data types isn't the only way to specify rules in schemas. In fact, you don't have to define any data types in your schema. Instead we could define all of our simple attributes, then refer to them later.

Let's see how that would work in this little example. In XML, create a new file as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
  </Person>
</Friends>
```

This small XML file stores a list of friends and phone numbers. Save it in your **/xml1** folder as **Friends.xml**.

Now let's set up our schema. Switch to a new file, and add the skeleton schema code as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

</xs:schema>
```

Save this file in your **/xml1** folder as **Friends.xsd**.

Next, we'll define every simple element and attribute used in our XML document. Our root element—**Friends**—is not a simple element because it contains child elements. **Person** and **Phone** aren't simple elements either. That leaves **FirstName**, **LastName**, and **Type**. Add definitions for those items to the schema as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="FirstName" type="xs:token"/>
  <xs:element name="LastName" type="xs:token"/>

  <xs:attribute name="Type" type="xs:token"/>

</xs:schema>
```

Save it, switch back to your XML file, and Validate. At this point, the schema is incomplete, so you'll get an error message like **Error: Element 'Friends': No matching global declaration available. Line: 4**. Switch back to your schema.

Next, we need to define our **Friends** element, a complex type. Add the code below to your schema as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="FirstName" type="xs:token"/>
  <xs:element name="LastName" type="xs:token"/>

  <xs:attribute name="Type" type="xs:token"/>

  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>

      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Instead of defining a separate data type for the **Friends** element, we'll define it "inline." This code is similar to code we presented in the Introduction to Schemas. **Person** is the only child element of **Friends**, so we'll add it to our schema as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="FirstName" type="xs:token"/>
  <xs:element name="LastName" type="xs:token"/>

  <xs:attribute name="Type" type="xs:token"/>

  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>

            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Now we can define the two child elements of **Person**. Type the code below as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="FirstName" type="xs:token"/>
  <xs:element name="LastName" type="xs:token"/>

  <xs:attribute name="Type" type="xs:token"/>

  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1" maxOccurs="1" ref="FirstName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="LastName"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Here we specify that **FirstName** and **LastName** are allowable child elements of **Person**. The **ref** (short for reference) specifies that the full definitions of **FirstName** and **LastName** occur elsewhere in the schema—in this case, they are at the beginning of the file.

Now we can add the related code for the **Phone** element and attribute. This time we'll add a new element definition at the beginning of our file and use a **ref**:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="FirstName" type="xs:token"/>
  <xs:element name="LastName" type="xs:token"/>

  <xs:attribute name="Type" type="xs:token"/>

  <xs:element name="Phone">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:token">
          <xs:attribute ref="Type" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1" maxOccurs="1" ref="FirstName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="LastName"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Finally, we're able to add the **Phone** element to the rules for **Person**. Type the code below as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="FirstName" type="xs:token"/>
  <xs:element name="LastName" type="xs:token"/>

  <xs:attribute name="Type" type="xs:token"/>

  <xs:element name="Phone">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:token">
          <xs:attribute ref="Type" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1" maxOccurs="1" ref="FirstName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="LastName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="Phone"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Save it, switch back to your XML file, and Validate. You should see **No errors found.**

Building a Schema

At this point, you may wonder when to use your own data types and when to use ref. How should you structure your schema? Well, that depends.

Good schemas (along with good computer programs, web pages, and even books) are defined by the ease with which you can understand and maintain them. Ultimately, it is up to you as the schema author to decide how to code your schema.

Take a look back at the definition for **Friends:**

OBSERVE:

```
  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1" maxOccurs="1" ref="FirstName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="LastName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="Phone"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
   </xs:complexType>
 </xs:element>
```

This element definition is getting pretty complex, even though its related XML is relatively short. Here's how it would look if everything was defined inline:

OBSERVE:

```
  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1" maxOccurs="1" name="FirstName" type="xs:token"/
>
              <xs:element minOccurs="1" maxOccurs="1" name="LastName" type="xs:token"/>
              <xs:element minOccurs="1" maxOccurs="1" name="Phone">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:token">
                      <xs:attribute name="Type" use="required" type="xs:token"/>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

This element definition works exactly the same way as the prior definition, but it's much longer. If you find your schema is getting unwieldy, you can always *refactor* your code—rewrite it to make it more efficient and easier to understand.

Substitutions

Let's go back to our little **Friends.xml** file:

OBSERVE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
  </Person>
</Friends>
```

Suppose we have many Spanish-speaking friends who also want to use this XML structure, but might prefer to use **Amigos** instead of **Friends**. We can enable them to do that. Change **Friends.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ~~Friends~~Amigos>
<~~Friends~~Amigos
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
  </Person>
</~~Friends~~Amigos>
```

💾 and Validate ⚙. You'll see a message like **Error: Element 'Amigos': No matching global declaration available for the validation root. on line 4**. This sort of difference can be accounted for in a schema by using *substitutions*. Switch to your schema and add this line:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="FirstName" type="xs:token"/>
  <xs:element name="LastName" type="xs:token"/>

  <xs:attribute name="Type" type="xs:token"/>

  <xs:element name="Phone">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:token">
          <xs:attribute ref="Type" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1" maxOccurs="1" ref="FirstName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="LastName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="Phone"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Amigos" substitutionGroup="Friends"/>

</xs:schema>
```

We defined a new element named **Amigos** that is a substitute for **Friends**. To test it, save your schema, switch back to your XML file, and validate. You'll see "No errors found."

Substitution can be pretty handy, but it does have limitations. Substitution doesn't work for attributes, and it only works for *global* elements. In our schema, **FirstName**, **LastName**, **Phone**, and **Friends** are all global elements, because each related **xs:element** is a direct child of **xs:schema**.

Any

Our friend list is pretty small; we can only specify each friend's first and last name and phone number. Suppose we want to allow any additional elements under **Person**. How might we do this?

Switch to your XML document and add a new element as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Amigos>
<Amigos
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
    <IM>Alex1092</IM>
  </Person>
</Amigos>
```

Validate your document; you see a message like **Error: Element 'IM': This element is not expected. on line 9**.

To get around this problem without actually listing all of the allowed child elements of Person, we can use the **any** element. Switch to your schema and modify it as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="FirstName" type="xs:token"/>
  <xs:element name="LastName" type="xs:token"/>

  <xs:attribute name="Type" type="xs:token"/>

  <xs:element name="Phone">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:token">
          <xs:attribute ref="Type" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name="Friends">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1" maxOccurs="1" ref="FirstName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="LastName"/>
              <xs:element minOccurs="1" maxOccurs="1" ref="Phone"/>
              <xs:any minOccurs="0" processContents="skip"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Amigos" substitutionGroup="Friends"/>

</xs:schema>
```

Save it, switch back to your XML file, and Validate. You should see **No errors found.**

The **processContents="skip"** attribute on **any** directs the XML parser to skip validation of elements, essentially allowing any elements or attributes to be placed in this part of the XML document. Other values for **processContents** are:

- **strict:** any element can be used, but the element must be specified in the schema for the required namespace.
- **lax:** like strict, but no error will occur if the schema cannot be found.

By default, **strict** is used.

We've covered a lot in this lesson! In the next lesson, we'll shift gears and discuss transformations using XSL. See you there!

Basic XSL

What is XSL?

So far we've only used XSL behind the scenes to show how XML can be translated into HTML to view in your web browser. You know enough about XML now to begin experimenting in XSL.

To reiterate, XSL stands for e**X**tensible **S**tylesheet **L**anguage. In English, that means XSL is an XML document that contains instructions for taking a source XML file and transforming it into something else. This "something else" could be HTML, XML, text, or *all* of those. XSL is one of the most important uses of XML—it allows a single source XML document to be transformed in many different ways. In still other words, XSL *is* an XML document, so we must follow all of XML's rules when creating XSL files.

Getting the Most Out of XSL

XSL can be used to transform XML, but what else can it do? We can use XSL to add some sorting capability so our XML phone book information will be returned in alphabetical order. This is especially useful when we have several people accessing our PhoneBook.xml document. People who use our phone book file may add phone book listings at the top of the file, the bottom of the file, or they may even try to alphabetize the list themselves. Instead of depending on users to sort the information themselves, we can use an XSL document to translate the source XML phonebook into an alphabetically ordered phonebook (in HTML or XML).

This is only one of many possible uses for XML. Try a Google search for Uses For XSL to find more.

Using the Translate Function in CodeRunner

CodeRunner has a powerful Translate function. Let's take a minute and go over how it works. Open your **Friends.xml** file and modify the code as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/introxml/xsl/PhoneBook4.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
     <FirstName>Alex</FirstName>
     <LastName>Chilton</LastName>
     <Phone Type="Cell">555-1212</Phone>
     <IM>Alex1092</IM>
  </Person>
</Friends>
```

Linking XSL to your XML Source

You used xml-stylesheet statements in your xml files back in your first XML document, so the new code probably looks familiar. When working with XML and XSL documents, you need to specify which XSL document will be used to translate your XML document by default. The xml-stylesheet statement serves that purpose. Here's the code we might use to link XSL inside your XML document:

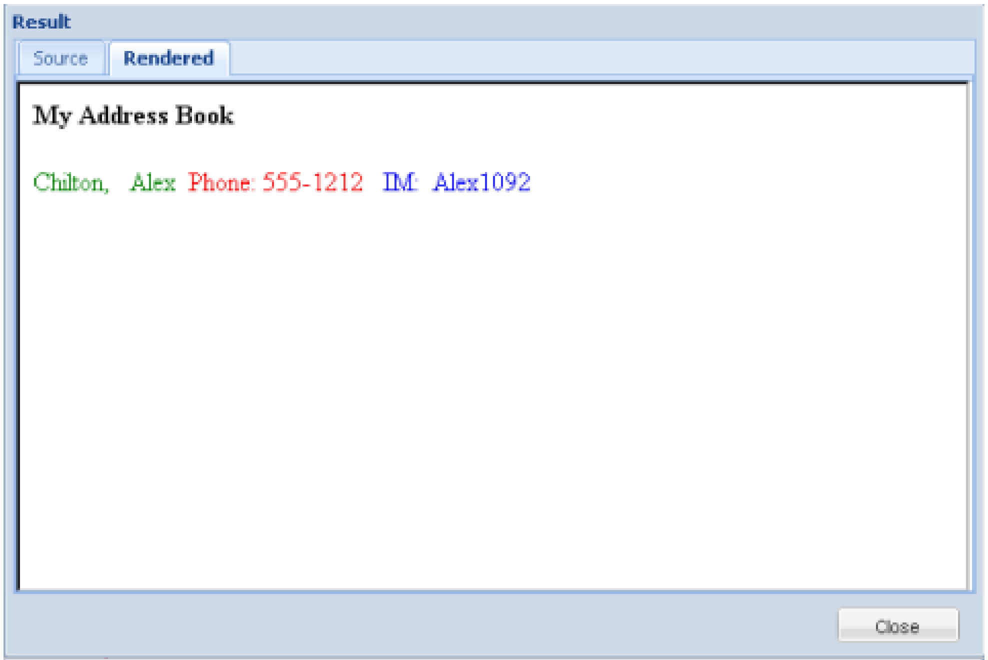
OBSERVE:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="default.xsl"?>
.
.
.
```

In this example, we added a reference to the XSL file **default.xsl**, which makes that our default XSL file.

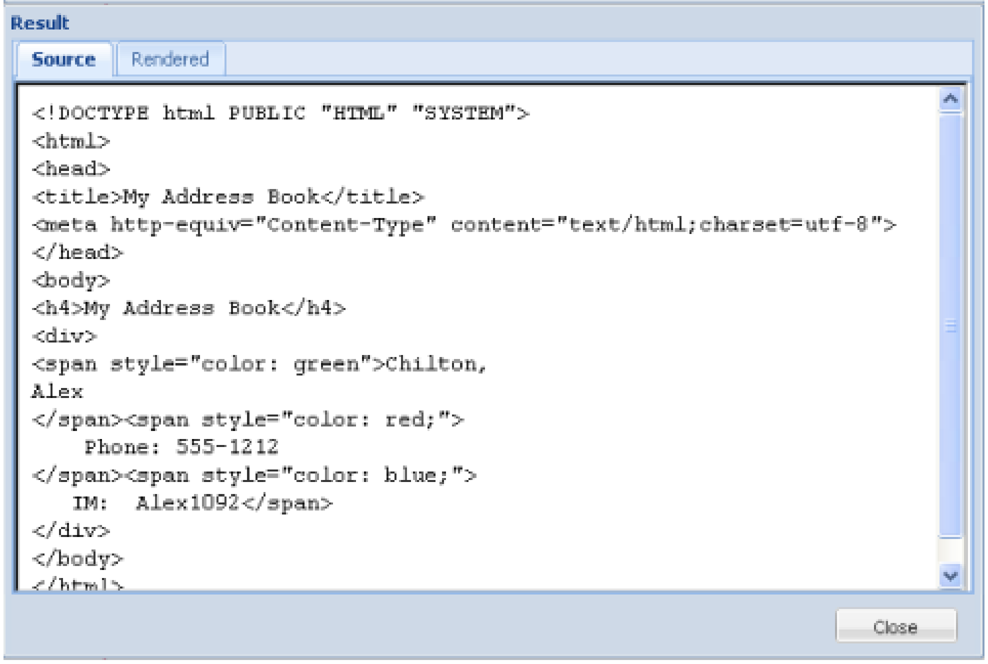
If you were to open your XML file in a web browser, it would automatically perform the translation using the default.xsl style sheet and the results would be displayed on your screen.

Now, click Translate to translate your XML file. Your XML document is translated and the results displayed in a new window. You'll see an example of how the page will display in the Rendered tab and the raw code the browser is working with in the Source tab:



My Address Book

Chilton, Alex Phone: 555-1212 IM: Alex1092



```
<!DOCTYPE html PUBLIC "HTML" "SYSTEM">
<html>
<head>
<title>My Address Book</title>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
</head>
<body>
<h4>My Address Book</h4>
<div>
<span style="color: green">Chilton,
Alex
</span><span style="color: red;">
    Phone: 555-1212
</span><span style="color: blue;">
   IM:  Alex1092</span>
</div>
</body>
</html>
```

Layout of an XSL Document and Templates

Creating XSL files is similar to creating XML files. Let's get started by taking a look at the text that is required for your XSL document:

OBSERVE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

</xsl:stylesheet>
```

What do these lines mean? Well, **<?xml version="1.0" ?>** refers to the version of xml being used, and the standards that are being applied to the document. **xsl:stylesheet version="1.0"** indicates the version of the style sheet definition you're using (similar to the version of XML you are claiming to follow). **xmlns:xsl="http://www.w3.org/1999/XSL/Transform"** declares the name space your XSL document is using. Finally, **xsl:output method="html"/** tells our translator that we are outputting HTML text as opposed to plain text. **/xsl:stylesheet** is the style sheet's closing tag. Make sure you include this closing tag when you editing your files in CodeRunner! It's easy to forget.

More About xsl:output

The **xsl:output** tag isn't actually required by all processors, but it's a good idea to include it anyway. Even if your processor ignores the tag, a human who tries to read your XSL document will not.

So what does this tag do? It tells the processor (and human readers) the "target" of your translation. There are three major types of **xsl:output** targets:

- <xsl:output method="**html**">
- <xsl:output method="**text**">
- <xsl:output method="**xml**">

Choosing between the XML and HTML tags may seem excessive at times, since both XML and HTML are actually text files, but some extra magic may occur, depending on which of these output types you choose. For instance, if your XSL document is going to convert your XML phone book to an HTML file and you set your output type to "html,", your processor may add some extra information to the HTML document to make it easier for web browsers to process. Your processor may also try to indent the resulting document if the output type is HTML.

If you are translating your document to XML, you'll want to indicate your DOCTYPE or schema as well. If your DTD is external, you can add a **doctype-system** attribute to your xsl:output element. To specify your external DTD for your XML output method, you'd do this:

OBSERVE:

```
<xsl:output method="xml" doctype-system="http://www.oreillyschool.com/dtds/mydtd.dtd" />
```

To specify your schema for your XML output method, you'd do this:

OBSERVE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
    <Friends2 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends2.xsd">

    </Friends2>
</xsl:template>
</xsl:stylesheet>
```

xsl:template

Now we can get to work on our basic XSL document. To practice. we'll apply XSL to our existing XML data. Let's work with our address book and get it to print out HELLO every time it hits an entry. Create an XSL document by typing the code below in CodeRunner as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
    HELLO
</xsl:template>
</xsl:stylesheet>
```

Save this file in your **/xml1** folder as **AddressBook01.xsl**; then, change **Friends.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="AddressBook01.xsl" type="text/xsl"?>
<!DOCTYPE Amigos>
<Amigos
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
     <FirstName>Alex</FirstName>
     <LastName>Chilton</LastName>
     <Phone Type="Cell">555-1212</Phone>
     <IM>Alex1092</IM>
  </Person>
</Amigos>
```

Translate Translate the document. You should see a "HELLO" on the screen. Once you get that example working, try adding another **Person** to your XML document. Translate the document again, and see what happens. Surprised?

The <**xsl:template match="/"**> tag defines a set of instructions for the *root element* of your XML source file. (Remember, the *root* element is the first element that occurs in an XML document. In this case, the **Friends** element is the root element.) When the root element is matched against this template, HELLO is output. Only one HELLO is ever output, because there is only one root element.

xsl:apply-templates

<xsl:apply-templates> is one of the most frequently used tags in XSL. When translation software encounters a tag in your XML file that matches the tag defined by your template, it will "execute" the XSL instructions located within the xsl:template tags of your XSL document. If you place your own XML tags or text inside the template, they will be copied to the resulting document. You define a set of instructions within xsl:template tags, and they are executed for the tags that you have designated in your source XML document. This is known in the XML world as *push-processing*, because your XML document is "pushed" through the instructions. The XSLT processor "pushes" the source tree nodes through the stylesheet, which has template rules to handle various kinds of nodes as they come through. In the next lesson, you will learn about *pull-processing* with <xsl:for-each>.

Here is an example of <xsl:apply-template select=""> usage:

Example	What it does
<xsl:apply-templates select="//*"/>	Applies templates to all elements in the document.
<xsl:apply-templates />	Applies templates to all children of the current element.
<xsl:apply-templates select="Phone"/>	Applies templates to Phone elements that are children of the current element.

Note You can use **<xsl:apply-templates />** inside of a template. When the template is applied to an element, all text inside the element is copied to the resulting document.

Let's try printing out **XML IS FUN** when you encounter a <Person> tag. Modify **AddressBook01.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
    HELLO
    <xsl:apply-templates select="//Person"/>
</xsl:template>
<xsl:template match="Person">
    XML IS FUN
</xsl:template>
</xsl:stylesheet>
```

Save this in your **/xml1** folder as **AddressBook02.xsl**, make the appropriate change to your xml file, and then Translate. You should see one instance of "XML IS FUN" created for each Person tag in your XML document. Try adding more elements to the XML document and translate again.

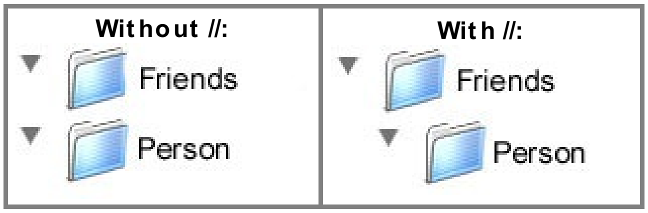
Let's go over this code in detail.

OBSERVE:

```
<xsl:template match="/">
    HELLO
    <xsl:apply-templates select="//Person"/>
</xsl:template>
<xsl:template match="Person">
    XML IS FUN
</xsl:template>
```

You may remember what **xsl:template match="/"** does, but what does **<xsl:apply-templates select="//Person"/>** mean? When the XSLT processor is chugging through your style sheet, it will process the root element first. From there, it encounters the instruction **<xsl:apply-templates select="//Person"/>**. This prompts the processor to look for the **template** to match "Person". It finds all **Person** elements that are children of **Friends** (because of the //) and applies the template to it.

Try replacing **<xsl:apply-templates select="//Person"/>** with **<xsl:apply-templates select="Person"/>**. Instead of matching Person elements below the root element, this instruction will match **Person** elements at the root level. In a "File System" view, the difference might look like this:



Note When working with large or complex XML documents, it's helpful to draw a quick sketch of your XML document and its layout, so you can keep track of the location of the elements in your document.

I think that's enough for now! Spend some time experimenting with your xml and xsl files, and I'll see you in the next lesson.

Advanced XSL

Translating the Address Book to XHTML

You have a basic understanding of how XSLT works. Now we're going to build an XSL file that will translate your address book into an XHTML representation. Let's start with a basic XSL document that will print out only the basic XHTML stuff: the **<html>, <head>, and <body>** elements. Our goal is to create this complete XHTML document.

Let's get started with your address book XML file. Change Amigos back to Friends and add another Person to the file, as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="AddressBook03.xsl" type="text/xsl"?>
<!DOCTYPE ~~Amigos~~Friends>
<~~Amigos~~Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
    <IM>Alex1092</IM>
  </Person>
  <Person>
    <FirstName>Laura</FirstName>
    <LastName>Chilton</LastName>
    <Phone>555-5678</Phone>
    <IM>LaurethSulfate</IM>
  </Person>

</~~Amigos~~Friends>
```

You'll recognize our standard address book. Now let's match the root element of our XML document and start creating the basic XHTML. Start a new file and type the code below as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>

  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Save it in your **/xml1** folder as **AddressBook03.xsl**. Switch to **Friends.xml** and Translate it. Nothing is rendered, but if you look at the **Source** of the translation, you'll see the beginnings of our XHTML document. In the previous lesson, we matched the root element and the output was "HELLO." This XSL is similar, only the output consists of some XHTML.

This is a great beginning. Now let's consider our address book's child element, **Person**. We need to add a template to match our **Person** element and then add the appropriate **xsl:apply-templates**. The template can go anywhere in our style sheet; we'll put it at the bottom this time. We want our data to be displayed within the body tags in our resulting XHTML document, so we'll place the **xsl:apply-templates** within them. Modify the XSL file as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>
    <xsl:apply-templates select="//Person" />
  </body>
</html>
</xsl:template>
<xsl:template match="Person">
person
</xsl:template>
</xsl:stylesheet>
```

💾 Save the XSL file and translate the XML document again. This time you should see "person person"—one instance of the word "person" is created for each **Person** element in our XML document.

Now let's talk about Person's children: **FirstName**, **LastName**, **Phone**, and **IM**. We need to add new templates to correspond to these elements. We can add them anywhere in our document. Let's just add them to the bottom. We want to apply the templates for these elements inside our Person element, because they are children of Person. Let's just work with **FirstName**, **LastName**, and **IM** for now. We'll add **Phone** in a minute. Modify **AddressBook03.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>
  <xsl:apply-templates select="//Person" />
  </body>
</html>
</xsl:template>
<xsl:template match="Person">
  <xsl:apply-templates select="FirstName" />
  <xsl:apply-templates select="LastName" />
  <xsl:apply-templates select="IM" />
</xsl:template>
<xsl:template match="FirstName">
first
</xsl:template>
<xsl:template match="LastName">
last
</xsl:template>
<xsl:template match="IM">
im
</xsl:template>
</xsl:stylesheet>
```

Nothing new here. 💾 and Translate again. Instead of seeing "person person," you see "first last im first last im." Now our foundation is in place.

In our previous lab we learned that we can print out the contents of an element by using the **xsl:apply-templates** instruction. Let's do that with our document. We'll replace the "first," "last," and "im" text in our XSL with the new

xsl:apply-templates instruction. Modify **AddressBook03.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>
    <xsl:apply-templates select="//Person" />
  </body>
</html>
</xsl:template>
<xsl:template match="Person">
  <xsl:apply-templates select="FirstName" />
  <xsl:apply-templates select="LastName" />
  <xsl:apply-templates select="IM" />
</xsl:template>
<xsl:template match="FirstName">
  <xsl:apply-templates />
</xsl:template>
<xsl:template match="LastName">
  <xsl:apply-templates />
</xsl:template>
<xsl:template match="IM">
  <xsl:apply-templates />
</xsl:template>
</xsl:stylesheet>
```

💾 and Translate again. You see "AlexChiltonAlex1092LauraChiltonLaurethSulfate." This document is ugly! Fortunately, we can fix that. We'll encapsulate the **<xsl:apply-template>** inside some XHTML by putting the output in <span> tags and adding some css styles to make our text look like the example here. While we're at it, we'll add line breaks after **LastName** and **IM** to provide some space.

All of our changes need to be put inside the template for **Person**. Make sure to close your **br** tags correctly. Modify **AddressBook03.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>
    <xsl:apply-templates select="//Person" />
  </body>
</html>
</xsl:template>
<xsl:template match="Person">
  <span style="color:green;"><xsl:apply-templates select="FirstName" />

  <xsl:apply-templates select="LastName" /></span><br/>
  <span style="color:red; font-size:large;"><xsl:apply-templates select="IM" /></span>
    <br/>
</xsl:template>
<xsl:template match="FirstName">
  <xsl:apply-templates />
</xsl:template>
<xsl:template match="LastName">
  <xsl:apply-templates />
</xsl:template>
<xsl:template match="IM">
  (<xsl:apply-templates />)
</xsl:template>
</xsl:stylesheet>
```

💾 and Translate again. This is the best-looking address book we've created so far. Experiment! Move the **spans** and **brs** from the **Person** template to the **IM** template. You'll see similar results. Try adding some other css—perhaps font-weight or background-color, or something else to jazz up this document a bit. How would you add spaces between the names?

This method of translating to XHTML is time-consuming, but not particularly difficult. Still, there must be an easier way! What about attributes? Our **Phone** element has a **Type** attribute. If we wanted that in the output, how would we write the XSL?

xsl:value-of

Fortunately, in XSL, there's an instruction that outputs the value of an element OR an attribute: **xsl:value-of**. We can use it to replace the **xsl:apply-templates** for FirstName, LastName, and IM, as well as their corresponding templates. If we replace **xsl:apply-templates** with **xsl:value-of**, the output will be the value of an element or attribute.

Modify your XSL document as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
  <html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>
    <xsl:apply-templates select="//Person" />
  </body>
  </html>
</xsl:template>

<xsl:template match="Person">
  <span style="color:green;"><xsl:value-of select="FirstName"/>
  <xsl:value-of select="LastName"/></span><br/>
  <span style="color:red; font-size:large">(<xsl:value-of select="IM"/>)</span> <br/>
</xsl:template>
</xsl:stylesheet>
```

This text is much shorter and more clear! 💾 and Translate again. The result is exactly the same as before.

Finally, let's add support to print out the **Type** attribute of the **Phone** element.

Modify your XSL document as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
  <html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>
    <xsl:apply-templates select="//Person" />
  </body>
  </html>
</xsl:template>

<xsl:template match="Person">
  <span style="color:green;"><xsl:value-of select="FirstName"/>
  <xsl:value-of select="LastName"/></span><br/>
  <span style="color:red; font-size:large;">(<xsl:value-of select="IM"/>) </span><br/>
  <xsl:value-of select="Phone/@Type"/>:
    <xsl:value-of select="Phone"/>
    <br/>
</xsl:template>
</xsl:stylesheet>
```

Most of the changes will be familiar to you now, except maybe the Phone/@Type. We are interested in **Phone**'s **Type** attribute, but not **Person**'s. The @ symbol indicates that we are interested in the *attribute* **Type**, not the element. Had we entered **Phone/Type** above instead, we would have gotten the value of **Phone**'s **Type** element (which doesn't exist). Had we included only **@Type**, we would have gotten the value of **Person**'s **Type** attribute (which doesn't exist).

💾 and Translate again. Did you notice what happens when there's no **Type** attribute? Suppose we decided that if there is no **Type** attribute, the **Phone** would automatically be **Home**? How can we accomplish that?

xsl:if

We'll start with an **if statement**.

If you know almost any programming language, you're familiar with the **if statement**. If not, fear not! An **if statement** tests to find out whether a condition is true or false. The result of that test determines whether a subsequent action will take place in your program. We can use the **if statement** to determine whether the Type attribute exists, and if so, whether it has a value. If the Type attribute exists and is not empty, we will output it.

Basic Syntax for xsl:if

So how do you use **xsl:if**? Let's take a look:

OBSERVE:

```
<xsl:if test="boolean">
     TRUE
</xsl:if>
```

A boolean (true or false) test is performed on the text within the **test attribute**. If the result of that test is true, everything in **blue** is evaluated. Unlike in other languages, this construct does not allow for an **ELSE statement**. Instead, you must use **xsl:choose** (you'll see this in action shortly).

Syntax for Test

xsl:if requires its **TEST** attribute to be boolean. Many things can go into the **TEST** attribute—comparisons between numbers, strings and values of elements, or attributes themselves. Here are some examples of **TEST** elements and what they do:

Test Condition	Description
@Type='Home'	If the Type attribute is equal to Home, this expression is TRUE.
Type='Cell'	If the Type element is equal to Cell, this expression is TRUE.

xsl:if with our Address Book

Remember what happened when we translated our address book with the last XSL file when there was no **Type** attribute specified? We can fix this by using <xsl:if>. Modify **AddressBook03.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
  <html>
    <head>
      <title>My Address Book</title>
    </head>
    <body>
      <xsl:apply-templates select="//Person" />
    </body>
  </html>
</xsl:template>

<xsl:template match="Person">
  <span style="color:green;"><xsl:value-of select="FirstName"/>
  <xsl:value-of select="LastName"/></span><br/>
  <span style="color:red; font-size:large;">(<xsl:value-of select="IM"/>) </span
><br/>
  <xsl:if test="boolean(Phone/@Type)">
    <xsl:value-of select="Phone/@Type" />:
  </xsl:if>
<xsl:value-of select="Phone"/>
<br/>
</xsl:template>
</xsl:stylesheet>
```

💾 and Translate.

Using **<xsl:if>** works to a point, but it doesn't print out the default value of **Home** when no **Type** attribute is present. In that case, it doesn't print anything. Since <xsl:if> doesn't offer an "else" clause, we cannot use it to print **Home** when no **Type** attribute is present. We'll have to use **<xsl:choose>** instead.

xsl:choose

The **xsl:choose** construct works like this:

OBSERVE:

```
<xsl:choose>
  <xsl:when test="boolean 1">
    statement a
  </xsl:when>
  <xsl:when test="boolean 2">
    statement b
  </xsl:when>
  <xsl:otherwise>
    statement c
  </xsl:otherwise>
</xsl:choose>
```

In this example, the first test is performed. If it is TRUE, then **statement a** is executed. Otherwise, the next test is performed (if it exists). If that test is TRUE, then **statement b** is executed. Otherwise (as you may have guessed), **statement c** is executed.

Note When using **xsl:choose**, you can have as many **xsl:when** sections as you need.

Edit your **AddressBook03.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>
    <xsl:apply-templates select="//Person" />
  </body>
</html>
</xsl:template>

<xsl:template match="Person">
  <span style="color:green;"><xsl:value-of select="FirstName"/>
  <xsl:value-of select="LastName"/></span><br/>
  <span style="color:red; font-size:large;">(<xsl:value-of select="IM"/>) </span><br/>
  <xsl:choose>
    <xsl:when test="boolean(Phone/@Type)">
      <xsl:value-of select="Phone/@Type" />:
    </xsl:when>
    <xsl:otherwise>
       Home:
    </xsl:otherwise>
  </xsl:choose>
<xsl:value-of select="Phone"/>
<br/>
</xsl:template>
</xsl:stylesheet>
```

We added the extra **xsl:when** clause required if the **Type** attribute is not set.

Now add another **xsl:when** section to output (Home) when the **Type** attribute exists, but is empty.

xsl:for-each

In XSL, there are two different ways to complete the same processing task. Instead of push-processing, some people prefer *pull-processing*. Pull-processing can be accomplished via the **xsl:for-each** tag. This is similar to the "for loop" constructs in many other languages.

In *push-processing*, your XML elements are essentially pushed through the templates as they are matched. In *pull-processing*, your XML elements are matched first and then the templates are applied to them, one at a time. Let me give you a visual representation of these processes. Say you're doing laundry and you want to fold your SOCKS and iron your SHIRTS. If your laundry style is *push* processing, you'll grab clothes at random. If you encounter a SOCK, you fold it immediately, and if you encounter a SHIRT you iron it immediately. If your laundry style is *pull* processing, you grab all of your SOCKs first and fold them, then grab all of your SHIRTs and iron them.

Let's try using **xsl:for-each**. Begin by converting **AddressBook03.xsl** from push-processing to pull-processing, as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>My Address Book</title>
  </head>
  <body>
    ~~<xsl:apply-templates select="//Person" />~~
    <xsl:for-each select="//Person">
      <span style="color:green;"><xsl:value-of select="FirstName"/>
      <xsl:value-of select="LastName"/></span><br/>
      <span style="color:red; font-size:large;">(<xsl:value-of select="IM"/>) </span><br/>
      <xsl:choose>
        <xsl:when test="boolean(Phone/@Type)">
          <xsl:value-of select="Phone/@Type" />:
        </xsl:when>
        <xsl:otherwise>
          Home:
        </xsl:otherwise>
      </xsl:choose>
      <xsl:value-of select="Phone"/>
      <br/>
    </xsl:for-each>
  </body>
</html>
</xsl:template>
~~<xsl:template match="Person">~~
~~  <span style="color:green;"><xsl:value-of select="FirstName"/>~~
~~  <xsl:value-of select="LastName"/></span><br/>~~
~~  <span style="color:red; font-size:large;">(<xsl:value-of select="IM"/>) </span><br/>~~
~~  <xsl:choose>~~
~~    <xsl:when test="boolean(Phone/@Type)">~~
~~      <xsl:value-of select="Phone/@Type" />:~~
~~    </xsl:when>~~
~~    <xsl:otherwise>~~
~~      Home:~~
~~    </xsl:otherwise>~~
~~  </xsl:choose>~~
~~<xsl:value-of select="Phone"/>~~
~~<br/>~~
~~</xsl:template>~~
</xsl:stylesheet>
```

Now, 💾 and Translate. This translation performs identically to the previous translations.

So what happened with our XSL exactly? Let's break down the new code. You may remember the template to match the root element. We output some common XHTML, then we encounter **<xsl:for-each select="//Person">**. This line selects *every* Person element in our XML document. One by one, those elements are processed using the instructions within that indented block. The **</xsl:for-each>** closes the xsl:for-each tag properly. Next there are some XHTML closing tags, template closing tags, and finally the style sheet closing tag.

Experiment with **xsl:for-each**. When you feel like you've got a handle on this concept, save this XSL file.

template-match or for-each?

So, now that you've learned about each of these tags, which one is better to use in which situation? There is no correct answer. You're the boss—use the method you like better, the one you find clear and easier and to read. Always strive for simplicity and readability.

XPath

In this lesson, we'll examine XPath, the **XML Path Language**.

What is XPath?

XPath is the language that lets you navigate an XML document and extract elements and attributes from that document. We actually used XPath in the last lessons when we discussed XSLT—the XPath consisted of the bits of code we used to select elements and attributes.

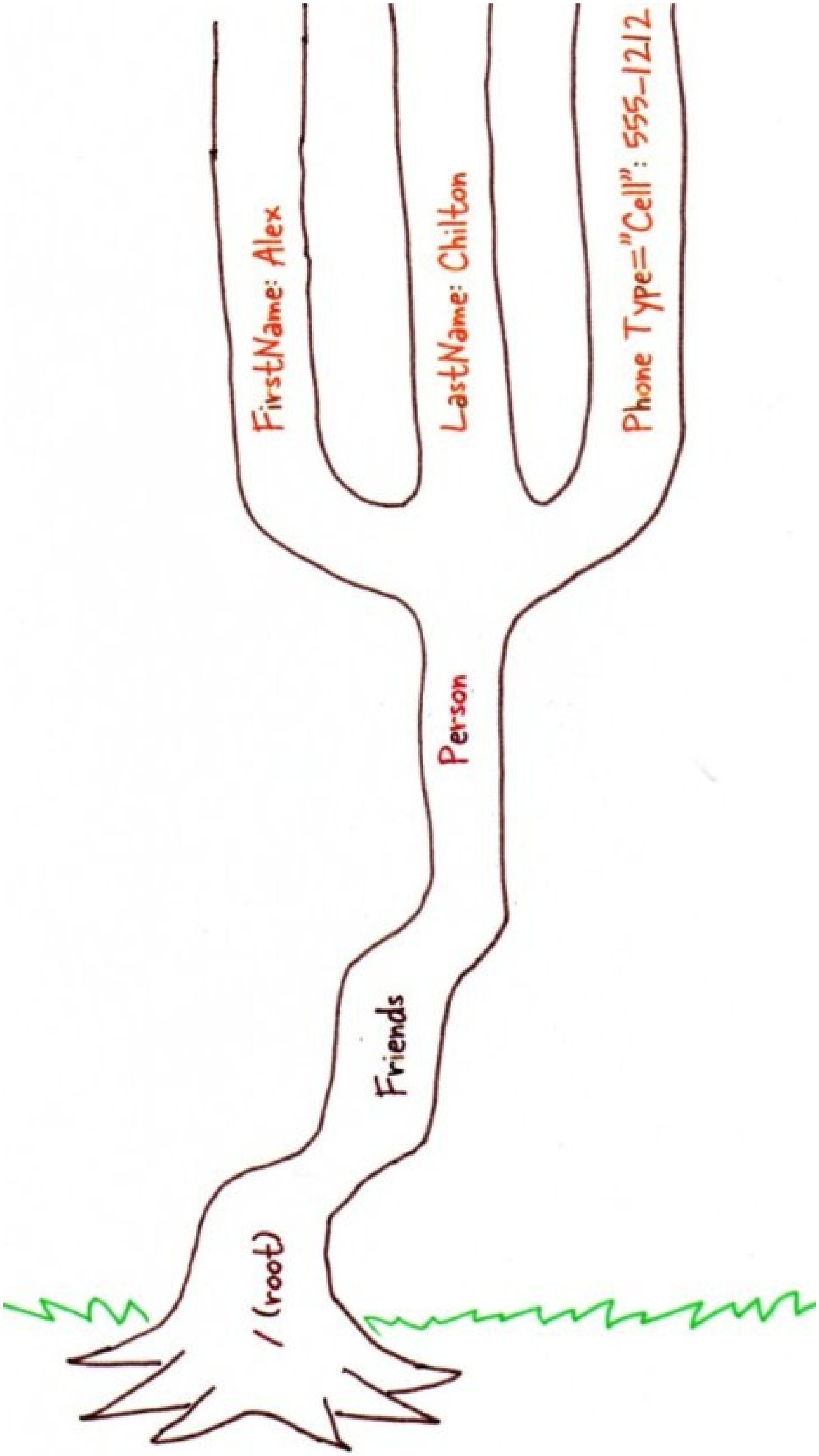
Let's take a look at the XML file we created back in the Schema Structure lesson for **Friends.xml**:

OBSERVE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
     <FirstName>Alex</FirstName>
     <LastName>Chilton</LastName>
     <Phone Type="Cell">555-1212</Phone>
  </Person>
</Friends>
```

In XPath terminology, an XML document forms a tree of *nodes*. (Remember back in Your First XML Document, we discussed the tree structure of XML.) A *node* is any element, attribute, or bit of text in a document.

In tree form, the document above looks like this:



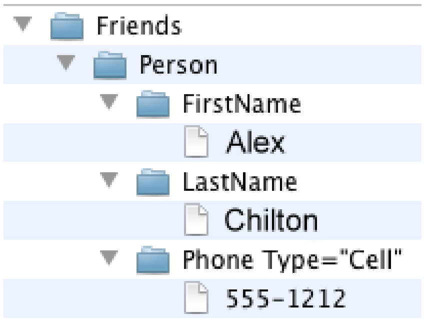
The **/** indicates the root of the XML document. At the root is the **Friends** node, which has one child node of **Person**, which has child nodes of **FirstName**, **LastName**, and **Phone**.

The **FirstName** node has a child text node with a value of **Alex**.

The **FirstName**, **LastName** and **Phone** nodes are all siblings. The **Phone** node has a child attribute node named **Type**.

FirstName has a parent node of **Person**, and a grandparent node of **Friends**.

Compare this to a directory on your computer. You might have a folder for **Friends**, which contains a folder for **Person**, which contains folders for **FirstName**, **LastName**, and **Phone**:



If you are familiar with Unix, Linux, or Macintosh computers, you could write a path to the first name like this:

OBSERVE:

```
//Friends/Person/FirstName/Alex
```

In the Windows world, that same path would look like this:

OBSERVE:

```
C:\Friends\Person\FirstName\Alex
```

Just like we use paths on our computers to specify locations of files, we use XPath to specify locations of XML elements and attributes that interest us. XPath isn't just for XSL—it's also useful in any application that reads or writes XML.

Remember the short XSL file we created in Basic XSL? Here it is:

OBSERVE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
    HELLO
</xsl:template>
</xsl:stylesheet>
```

You may not have realized it, but we used *XPath* in our **<xsl:template match="/">** element. The forward slash **/** is the XPath expression for the root node of an XML file.

Using XPath to Navigate an XML Document

Create a new XML file and copy the code below (just this once, you can copy and paste it):

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FriendsXPath.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
  </Person>
</Friends>
```

💾 it in your **/xml1** folder as **FriendsX.xml**. Suppose we want to retrieve the first name of a person in our friends list. How can we make that happen? In a new XSL file, type the code below as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
    <xsl:value-of select="/Friends/Person/FirstName"/>
</xsl:template>

</xsl:stylesheet>
```

💾 it in your **/xml1** folder as **FriendsXPath.xsl**, switch back to **FriendsX.xml**, and Translate. Sure enough, you'll see the first name:

OBSERVE:

```
Alex
```

Our XSL file displayed the resulting node of the XPath expression of **/Friends/Person/FirstName**.

So, what if our XML document has multiple people in it? Modify **FriendsX.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FriendsXPath.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
  </Person>
 <Person>
    <FirstName>Jim</FirstName>
    <LastName>Dandy</LastName>
    <Phone Type="Cell">555-1313</Phone>
  </Person>
</Friends>
```

💾 and Translate again. You'll see something familiar:

OBSERVE:

```
Alex
```

What happened to Jim?

Our XSLT engine only displays the first result, even though there are multiple **<FirstName>** nodes in our document. In other words, our XPath expression resulted in a *node set* instead of a single node. A *node set* is a list of nodes.

To get around this, we can use **<xsl:for-each>** to iterate over each node in the node set. Modify **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person/FirstName">
    First Name: <xsl:value-of select="."/><br/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate again. This time, you see both names:

OBSERVE:

```
First Name: Alex
First Name: Jim
```

In this XSL file, we looped over the results of our XPath expression—**/Friends/Person/FirstName**. The XPath expression in the **value-of** element is a single period (.) and is used to indicate "current node." In this context, the current node is each **<FirstName>** as processed by **<xsl:for-each>**.

If you have used the **ls** command on a Unix or Mac computer, or the **dir** command on a Windows machine, you've probably seen output that looks like this:

OBSERVE:

```
cold:~$ ls -al /Friends/Person/FirstName/
total 0
drwxr-xr-x  3 certjosh  certjosh  102 Jun 22 14:22 .
drwxr-xr-x  6 certjosh  certjosh  204 Jun 22 14:23 ..
-rw-r--r--  1 certjosh  certjosh    0 Jun 22 14:22 Alex
cold:~$
```

OBSERVE:

```
C:\dir Friends\Person\FirstName
 Volume in drive C has no label.
 Volume Serial Number is EC14-6B6E

 Directory of C:\Friends\Person\FirstName

06/21/2010  01:22 PM    <DIR>          .
06/21/2010  01:22 PM    <DIR>          ..
09/01/2009  06:14 AM    <DIR>          Alex
               1 File(s)              0 bytes
               7 Dir(s)   2,632,855,552 bytes free

C:\>
```

Both of these listings have special entries which are nearly identical in the XPath world:

	Entry	Directory Meaning	XPath Meaning
Single period	.	Current directory	Current node
Two periods	..	Parent of current directory	Parent of current node

Armed with this new information, let's experiment and change **FriendsXPath.xsl** slightly as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person/FirstName">
    First Name: <xsl:value-of select="."/><br/>
    Phone: <xsl:value-of select="../Phone"/><br/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate. Now you see the phone number associated with each person:

OBSERVE:

```
First Name: Alex
Phone: 555-1212
First Name: Jim
Phone: 555-1313
```

There is one more distinction we should make with XPath expressions. An XPath expression can be *absolute* or *relative*. When the expression is absolute, the expression is a full path that points to a specific place in the XML tree, *independent* of the current location. When the expression is relative, it is *dependent* on the current location.

Absolute expressions always begin with the root node (in our case, **Friends**) or a forward slash (/).

Relative expressions start with the descendant of the root node.

Our XSL document has three XPath expressions:

1. **/** - the root node
2. **/Friends/Person/FirstName** - an *absolute* path
3. **../Phone** - a *relative* path

The second XPath expression is absolute, because it begins with a forward slash. It always results in a list of **FirstName** nodes.

The third XPath expression is relative, and is *dependent* on position. In our example, we use it in conjunction with the second XPath expression. Those two expressions together can be expressed this way:

/Friends/Person/FirstName/../Phone

...which shortens to:

/Friends/Person/Phone

In English we'd say, "given a FirstName node, find its parent, than find that parent's child Phone element."

Predicates

As we said earlier, a node set is a list of nodes that is the result of an XPath expression. Because it is a list, we can use a special notation to pick out items from the list. Let's try it. Modify **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person[2]/FirstName">
    First Name: <xsl:value-of select="."/><br/>
    Phone: <xsl:value-of select="../Phone"/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate. This time you will see only Jim:

OBSERVE:

```
First Name: Jim
Phone: 555-1313
```

We used a *Predicate* (the square brackets and number [2]) to select the second person from our XML document. XPath uses one-based indexing, meaning that locations elements begin at position one. If you are familiar with languages such as C or Perl, you may notice the structural difference—those languages typically use zero-based indexing instead of one-based indexing.

XPath also has a special function that lets us pick out the last node instead of hard coding a number. Try it:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person[last()]/FirstName">
    First Name: <xsl:value-of select="."/><br/>
    Phone: <xsl:value-of select="../Phone"/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Add another person to **FriendsX.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FriendsXPath.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
  </Person>
 <Person>
    <FirstName>Jim</FirstName>
    <LastName>Dandy</LastName>
    <Phone Type="Cell">555-1313</Phone>
  </Person>
  <Person>
    <FirstName>Neko</FirstName>
    <LastName>Case</LastName>
    <Phone Type="Cell">555-2323</Phone>
  </Person>
</Friends>
```

💾 and Translate again. You see the last person in the list:

OBSERVE:

```
First Name: Neko
Phone: 555-2323
```

We covered a lot in this lesson! In the next lesson we'll discuss additional ways to use XPath to navigate through an XML document. See you there!

Advanced XPath

Welcome back! In the last lesson we learned how to use XPath to navigate an XML document. In this lesson, we'll explore several of the functions available in XPath, and learn additional ways to traverse documents.

Functions

Let's get right to work. Open **FriendsX.xml** from the last lesson and remove the last person we added:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FriendsXPath.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
  </Person>
  <Person>
    <FirstName>Jim</FirstName>
    <LastName>Dandy</LastName>
    <Phone Type="Cell">555-1313</Phone>
  </Person>
  ~~<Person>~~
~~    <FirstName>Neko</FirstName>~~
~~    <LastName>Case</LastName>~~
~~    <Phone Type="Cell">555-2323</Phone>~~
~~  </Person>~~
</Friends>
```

Now, suppose you want to know how many people (**Person** nodes) exist in our document. You can find out with XPath using the **count** function.

Let's give that a try. Open your **FriendsXPath.xsl** from the last lesson and modify it as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:value-of select="count(//Person)"/>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate your XML document. You should get a result of **2**. The **//Person** XPath translates in English to "any occurrence of the **Person** element in the document."

So, what else can we do with **count**? Let's say we want to see the names of the elements that have two children. Modify **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="//*[count(*)=2]">
    <xsl:value-of select="name()"/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate. You'll see this:

OBSERVE:

```
Friends
```

This is the only element in our document that has exactly two children. Great, but how did our XPath expression work? Let's break it down:

OBSERVE:

```
//*[count(*)=2]
```

The first characters—//*—literally mean "any element in the document."

When we discussed predicates earlier, we learned that the square brackets **[ ]** indicate that we are using a predicate to pick nodes from a node set. In our current example, instead of picking a specific index like 1 or 2, our predicate selects **nodes with two children** with the expression **count(*)=2**.

Putting this together in English, the expression **//*[count(*)=2]** would read, "select all elements from the document that have exactly two child elements."

We used one additional function here—**name()**. This function returns the name of the element.

We're just getting started. We can have a lot more fun with XPath! How about selecting elements whose names contain "Name?" Sure, why not? Update **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="//*[contains(name(),'Name')]">
    Element name: <xsl:value-of select="name()"/><br/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate:

OBSERVE:

```
Element name: FirstName
Element name: LastName
Element name: FirstName
Element name: LastName
```

Once again we used a predicate to pick specific nodes from our node set of "all elements." This time we used the **contains** function to make sure that the result of **name()** contained the word **Name**.

Now suppose we want to select all elements with names that do *not* contain the word **Name**. This is possible too. Modify **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="//*[not( contains(name(),'Name' )) ]">
    Element name: <xsl:value-of select="name()"/><br/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate your document. You'll see all of the elements you didn't see before:

OBSERVE:

```
Element name: Friends
Element name: Person
Element name: Phone
Element name: Person
Element name: Phone
```

Here we used the **not()** function to negate the meaning of the enclosed predicate expression.

Sometimes the questions we want answered aren't so straightforward, though. What if we want to combine our prior questions so we can see elements with two children, as well as elements with names that contain the word "Name"? XPath has you covered! We can use the pipe (**|**) to combine two XPath expressions. Modify **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="//*[count(*)=2]  | //*[contains(name(),'Name') ]">
    Element name: <xsl:value-of select="name()"/><br/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate:

OBSERVE:

```
Element name: Friends
Element name: FirstName
Element name: LastName
Element name: FirstName
Element name: LastName
```

Axes

The functions we've seen so far allow us to write some pretty complex expressions, but they won't answer every question. Take a look at our XML document:

OBSERVE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FriendsXPath.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
     <FirstName>Alex</FirstName>
     <LastName>Chilton</LastName>
     <Phone Type="Cell">555-1212</Phone>
  </Person>
 <Person>
     <FirstName>Jim</FirstName>
     <LastName>Dandy</LastName>
     <Phone Type="Cell">555-1313</Phone>
  </Person>
</Friends>
```

XML documents form "family trees," complete with parents, children, and siblings. That means that the two **Person** elements in our document are siblings. If we wanted to access these two elements as we might other siblings, how could we navigate to a sibling using an XPath expression? We'd do it using an *axis*. In XPath, an *axis* defines a node set, relative to the current node. Let's get started. Write an XPath expression using a predicate to pick the very first **Person** node. Edit **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person[1]">
    The first person's first name: <xsl:value-of select="FirstName"/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate. Here we used an XPath expression with predicate **/Friends/Person[1]** to select the first **Person** element, and then displayed that Person's **FirstName**, which is Alex.

Now that we have selected the first element, how can we select the sibling? We'll use the **following-sibling** axis. Modify **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person[1]">
    The first person's first name: <xsl:value-of select="FirstName"/><br/>
    The second person's first name: <xsl:value-of select="following-sibling::*/FirstName"/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate ⚙:

OBSERVE:

```
The first person's first name: Alex
The second person's first name: Jim
```

The expression works, using this syntax:

OBSERVE:

```
axis::nodetest[optional predicate]
```

Our specific expression is:

OBSERVE:

```
following-sibling::*/FirstName
```

Our **axis** is **following-sibling**—which refers to all sibling elements that occur after the current element (a better name might have been *following-siblings*). The **nodetest** is ***/FirstName**—which means "all elements with a FirstName child element." We are not using a predicate, so that section is omitted from our expression.

What do you think would happen if we had more than one sibling following Alex in our xml document? Let's give it a try. Modify **FriendsX.xml** as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FriendsXPath.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
    <FirstName>Alex</FirstName>
    <LastName>Chilton</LastName>
    <Phone Type="Cell">555-1212</Phone>
  </Person>
  <Person>
    <FirstName>Jim</FirstName>
    <LastName>Dandy</LastName>
    <Phone Type="Cell">555-1313</Phone>
  </Person>
  <Person>
    <FirstName>Janie</FirstName>
    <LastName>Jones</LastName>
    <Phone Type="Cell">555-1414</Phone>
  </Person>
</Friends>
```

💾 and Translate ⚙:

OBSERVE:

```
The first person's first name: Alex
The second person's first name: Jim
```

What's going on? Although **following-sibling** refers to all sibling elements that occur after the current element, **xsl:value-of** only returns the value of the first following sibling. To get all the names of the following siblings, layer another for-each statement in your xsl file:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person[1]">
    The first person's first name: <xsl:value-of select="FirstName"/><br/>
    <xsl:for-each select="following-sibling::*">
      The following sibling: <xsl:value-of select="FirstName"/>
      <br/>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate:

OBSERVE:

```
The first person's first name: Alex
The following sibling: Jim
The following sibling: Janie
```

You might wonder why we used this **following-sibling** axis instead of using a double period (..) like we did in the last lesson. I'm glad you're thinking about stuff like that! Let's try it—modify **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person[1]">
    The first person's first name: <xsl:value-of select="FirstName"/><br/>
    ~~<xsl:for-each select="following-sibling::*">~~
      The following sibling: <xsl:value-of select="../Person/FirstName"/>
    ~~</xsl:for-each>~~
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate. The results aren't quite what we're looking for:

OBSERVE:

```
The first person's first name: Alex
The following sibling: Alex
```

Alright, then, let's use a predicate! Modify **FriendsXPath.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
  <xsl:for-each select="/Friends/Person[1]">
    The first person's first name: <xsl:value-of select="FirstName"/><br/>
    The following sibling: <xsl:value-of select="../Person[2]/FirstName"/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

💾 and Translate. You'll see better results:

OBSERVE:

```
The first person's first name: Alex
The following sibling: Jim
```

If you are feeling uneasy about this last expression—good! Hard-coding that **2** in the predicate may seem like a good idea now, but with more than two **People** elements in your document, you'd have to add another line with the predicate [3] to see Janie!

In programming, there are often many different ways to accomplish the same goal. Clear, easy-to-understand code is always the best option. While many aspects of axes can be duplicated using "normal" XPath Expressions, axes are usually more easily understood. **following-sibling** is much clearer than **../Person[*n*]**!

Other Axes

following-sibling is not the only axis in XPath—in fact there are several:

Axis	Nodes selected from *current node*.
ancestor	All ancestors, including parent and grandparent.
ancestor-or-self	All ancestors, including parent and grandparent **and the current node**.
descendant	All children, grandchildren, great grandchildren, etc.
descendant-or-self	All children, grandchildren, great grandchildren, etc **and the current node**.
following	Everything **after** the closing tag of current node.
preceding	Everything **before** the opening tag of current node.
following-sibling	All siblings after current node.
preceding-sibling	All siblings before the current node.
child	All children.
parent	Parent of current node.
self	Current node.
attribute	All attributes.
namespace	All namespace nodes of current node.

There's a lot to digest in this lesson. In the next lesson we'll switch gears back to XSLT and discuss some recent updates in the newer XSLT 2.0 standard. See you there!

The future of XML

XSLT 2.0 / XPath 2.0

In previous lessons we learned to use XSLT to transform our XML documents into other representations. We also covered XSLT Version 1.0—the first release of the XSLT specification.

XSLT 1.0 (and the related XPath 1.0) work really well, but they have their shortcomings. W3C, the group that designs XSLT, addressed some of these issues in the version 2.0 of XSLT and XPath.

Version 2.0 has been released for a while now, but as of 2010 it has not been widely adopted. Major languages such as PHP and C#/VB.Net do not offer native XSLT 2.0 processors. (There is a 2.0 processor available for .Net and Java, though: Saxon) Similarly, major web browsers such as Firefox and Internet Explorer offer XSLT 1.0 functionality, but do not offer XSLT 2.0 yet.

So, how can you tell if an XSL file uses version 2.0? Take a look at these few lines of code:

OBSERVE:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="2.0">
...rest of style sheet...
```

For version 2.0 we use two namespaces: one for **XSL**, and one for **schema**s. Finally, we set the **version** to **2.0**.

Changes Found in Version 2.0

Data Types

One major change in version 2.0 is the inclusion of XML schema's data types. In version 1.0, there were four data types:

- Strings
- Booleans
- Node sets
- Numbers

In version 2.0, there are many additional data types, such as:

- xs:integer
- xs:decimal
- xs:string
- xs:date

Data types are seriously useful. One of the best examples of this can be found when handling date and time data. Suppose you have two dates with times that refer to a library book:

OBSERVE:

```
<book checkOut="2010-05-25T13:45:01" return="2010-06-12T08:18:18"/>
```

Now let's say that you want to display the number of days that the book was checked out. If you're using XSLT 1.0, you'd need to parse the dates and times manually and then perform date arithmetic to get the information you need.

But if you're fortunate enough to be using XSLT 2.0, you can use date functions to retrieve your information fast! Let's give this a try. Create a new XML file as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<book checkOut="2010-05-25T13:45:01" return="2010-06-12T08:18:18"/>
```

Save it in your **/xml1** folder as **book.xml**. Next, create a new XSL file as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="2.0">

<xsl:output method="text" />

<xsl:template match="book">
The book was checked out for:
  <xsl:value-of select="xs:dateTime(@return) - xs:dateTime(@checkOut)" />
days.
</xsl:template>

</xsl:stylesheet>
```

Save it in your **/xml1** folder as **book.xsl**. Right now, CodeRunner supports only XSLT 1.0. In order to take advantage of XSLT 2.0 features, we can use Saxon from the Unix command line. Switch to a terminal by clicking the **Terminal** icon ( ). You may be asked to accept the security certificate for MindTerm, the ssh client we'll be using in this course. Go ahead and accept it. Then you'll be logged in to the OST server.

The server is named **cold**. All OST students have shells on this server. The shell is where Unix commands are executed on the server. The commands you execute on your shell will not affect any other shell. **cold:~/xml1$** is what's known as a "command prompt." If you see **cold:~/xml1$**, you're ready to execute Unix commands. (To log out of the terminal when you finished, type **exit** at the command prompt.) Type this code at the Unix prompt:

INTERACTIVE SESSION:

```
cold:~/xml1$ cd xml1
cold1:~/xml1$ saxon9he.sh -xsl:book.xsl book.xml
```

If you typed everything correctly, you'll see this:

OBSERVE:

```
The book was checked out for:
P17DT18H33M17S
days
```

This text represents the *duration* of time between the check out and return dates—in this case, 17 days, 18 hours, 33 minutes, and 17 seconds. If we want to see only the number of days, we can use the **days-from-duration** function. Change **book.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="2.0">

<xsl:output method="text" />

<xsl:template match="book">
The book was checked out for:
  <xsl:value-of select="days-from-duration(xs:dateTime(@return) - xs:dateTime(@c
heckOut))" />
days.
</xsl:template>

</xsl:stylesheet>
```

Run saxon again in the Terminal window (you can repeat the command by pressing the up arrow). You'll see this:

OBSERVE:

```
The book was checked out for:
17
days
```

Output

In Basic XSL, we discussed the **<xsl:output** element, and mentioned three allowed method attributes: **html**, **text**, and **xml**. In 2.0, there's an additional allowed attribute value: **xhtml**.

In XSLT 1.0, exactly one result document was created from an input XML file and input XSL file. This is no longer the case. Using the **xsl:result-document** directive, you can create as many output documents as you want.

Using this new tool, we could write an XSL file that creates an "index" file with our friends' names, and individual files for each friend. Let's see how this can be done.

Open your **FriendsX.xml** file, or create a new file with the following contents:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FriendsXPath.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person>
     <FirstName>Alex</FirstName>
     <LastName>Chilton</LastName>
     <Phone Type="Cell">555-1212</Phone>
  </Person>
 <Person>
     <FirstName>Jim</FirstName>
     <LastName>Dandy</LastName>
     <Phone Type="Cell">555-1313</Phone>
  </Person>
 <Person>
     <FirstName>Janie</FirstName>
     <LastName>Jones</LastName>
     <Phone Type="Cell">555-1414</Phone>
  </Person>
</Friends>
```

Next, make a new XSL file with the following contents:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="2.0">

<xsl:output name="my-index-format" method="xhtml" indent="yes"
            doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
            doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"/>

<xsl:template match="/">
<xsl:result-document href="FriendsIndex.html" format="my-index-format">

</xsl:result-document>
</xsl:template>

</xsl:stylesheet>
```

Save it in your **/xml1** folder as **MyFriends.xsl**.

OBSERVE:

```
<xsl:output name="my-index-format" method="xhtml" indent="yes"
            doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
            doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"/>

<xsl:template match="/">
<xsl:result-document href="FriendsIndex.html" format="my-index-format">

</xsl:result-document>
```

Here we used the new **xhtml output method** in the **xsl:output** element. We named our output **my-index-format**—and referenced that output in our **xsl:result-document** element.

With these lines in place, we can add the required XHTML elements to our index. We'll use **xsl:for-each** to list our **Person** elements. Edit **MyFriends.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
 <xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="2.0">

<xsl:output name="my-index-format" method="xhtml" indent="yes"
            doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
            doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"/>

<xsl:template match="/">
<xsl:result-document href="FriendsIndex.html" format="my-index-format">
  <html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>My Friends</title></head>
    <body style="background-color: white;">
      <h1>List of Friends</h1>
      <ol>
        <xsl:for-each select="//Person">
          <li><a href="friend-{position()}.html"><xsl:value-of select="concat(Fi
rstName, ' ', LastName)"/> </a></li>
        </xsl:for-each>
      </ol>
    </body>
  </html>

</xsl:result-document>
</xsl:template>

</xsl:stylesheet>
```

Let's examine the new code:

OBSERVE:

```
<li><a href="friend-{position()}.html"><xsl:value-of select="concat(FirstName, '
 ', LastName)"/> </a></li>
```

This **for-each** looks familiar, except for a few changes. First, we use the **position()** function inside curly braces {} to output the position number of the current **Person** element. Alex Chilton will be assigned "1" and Jim Dandy will be assigned "2." The braces are required around any function in an attribute.

Next, we use the function **concat** to combine the text of the **FirstName** element, a space, and the text of the **LastName** element.

Save all of your files, switch to a terminal, and type this command:

INTERACTIVE SESSION:

```
cold:~/xml1$ saxon9he.sh -xsl:MyFriends.xsl FriendsX.xml
```

Our output is directed to a file named **FriendsIndex.html**, so you won't see any output in the terminal session, unless there is a problem with your transformation.

You can use the Unix **cat** command (or browse in the CodeRunner File Browser panel to your **/xml1** folder where the **FriendsIndex.html** file is created) to view the file:

INTERACTIVE SESSION:

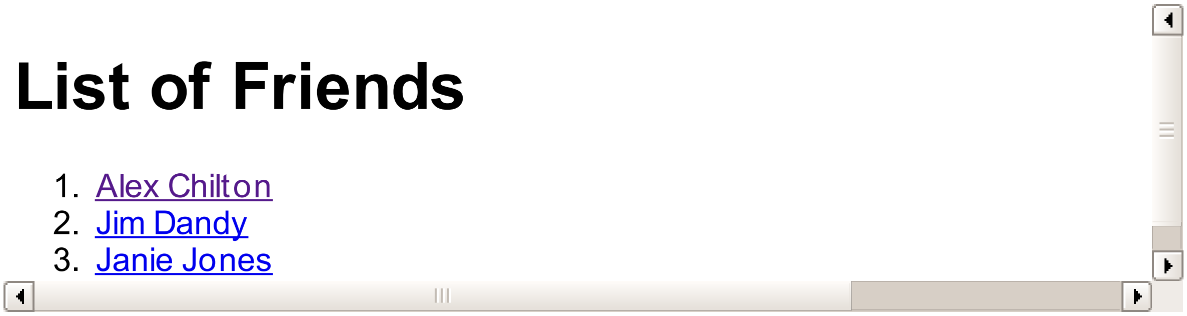
```
cold:~/xml1$ cat FriendsIndex.html
```

You'll see this:

OBSERVE:

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xht
ml1-strict.dtd">
<html xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://www.w3.org/1999/
xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>My Friends</title>
  </head>
  <body style="background-color: white;">
    <h1>List of Friends</h1>
    <ol>
      <li><a href="friend-1.html">Alex Chilton</a></li>
      <li><a href="friend-2.html">Jim Dandy</a></li>
      <li><a href="friend-2.html">Janie Jones</a></li>
    </ol>
  </body>
</html>
```

If you preview your **FriendsIndex.html** file in a web browser (either by opening and previewing it in CodeRunner, or by directing your browser to **http://*yourOSTlogin*.oreillystudent.com/xml1/FriendsIndex.html**, where *yourOSTlogin* is the name you use to log in to OST), you'll see this:



List of Friends

1. Alex Chilton
2. Jim Dandy
3. Janie Jones

Now that our index file is in place, let's add code to create the individual friend files. This time we'll use short HTML to make these files, instead of XHTML. Modify **MyFriends.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="2.0">

<xsl:output name="my-index-format" method="xhtml" indent="yes"
            doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
            doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"/>

<xsl:output method="html" indent="yes" name="html"/>

<xsl:template match="/">
<xsl:result-document href="FriendsIndex.html" format="my-index-format">
  <html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>My Friends</title></head>
    <body style="background-color: white;">
      <h1>List of Friends</h1>
      <ol>
        <xsl:for-each select="//Person">
          <li><a href="friend-{position()}.html"><xsl:value-of select="concat(Fi
rstName, ' ', LastName)"/> </a></li>
        </xsl:for-each>
      </ol>
    </body>
  </html>

</xsl:result-document>

<xsl:for-each select="//Person">
  <xsl:result-document href="friend-{position()}.html" format="html">
  <html><body style="background-color: white;">
    <h1><xsl:value-of select="concat(FirstName, ' ', LastName)"/> </h1>
    <b><xsl:value-of select="Phone/@Type"/></b> phone: <xsl:value-of select="Pho
ne"/>
  </body></html>
  </xsl:result-document>
</xsl:for-each>

</xsl:template>

</xsl:stylesheet>
```

We used the same code—**friend-{position()}.html**—to link our index file to our detail file. Save all of your files, switch back to the terminal, and type this command again:

INTERACTIVE SESSION:

```
cold:~/xml1$ saxon9he.sh -xsl:MyFriends.xsl Friends.xml
```

Our output is directed to four files:

- FriendsIndex.html
- friend-1.html
- friend-2.html
- friend-3.html

Use the unix **cat** command to view one of the files:

INTERACTIVE SESSION:

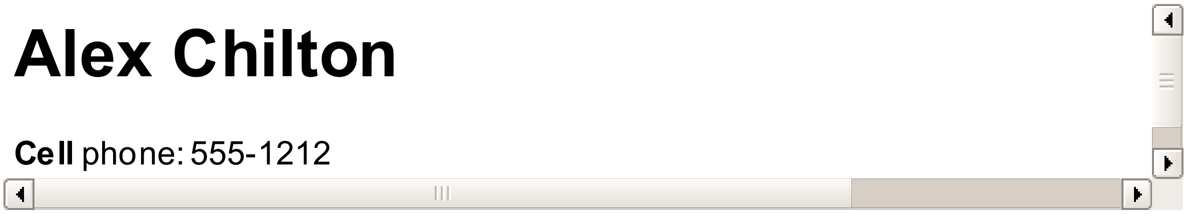
```
cold:~/xml1$ cat friend-1.html
```

You'll see this:

OBSERVE:

```
<html xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <body style="background-color: white;">
      <h1>Alex Chilton</h1><b>Cell</b> phone: 555-1212
   </body>
</html>
```

If you preview the **FriendsIndex.html** file in your browser and click the **Alex Chilton** link, you'll see this:



Alex Chilton

Cell phone: 555-1212

The other links work too—excellent!

Grouping

Another new feature in XSLT 2.0 is *grouping*. Suppose you wanted to extend your friends list and group people into two buckets: family friends and work friends. Then you want to sort your list to display the members of both buckets.

In XSLT 1.0, this is quite the chore—it's possible, but the solution is not straightfoward. XSLT 2.0 offers a cleaner, more elegant solution.

Let's give it a try. Add a new person to **FriendsX.xml**, and then add a **group** attribute, as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FriendsXPath.xsl" type="text/xsl"?>
<!DOCTYPE Friends>
<Friends
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Friends.xsd">
  <Person Group="Family">
     <FirstName>Alex</FirstName>
     <LastName>Chilton</LastName>
     <Phone Type="Cell">555-1212</Phone>
     </Person>
  <Person Group="Family">
     <FirstName>Jim</FirstName>
     <LastName>Dandy</LastName>
     <Phone Type="Cell">555-1313</Phone>
  </Person>
  <Person Group="Family">
     <FirstName>Janie</FirstName>
     <LastName>Jones</LastName>
     <Phone Type="Cell">555-1414</Phone>
  </Person>
  <Person Group="Work">
     <FirstName>Ana</FirstName>
     <LastName>Ng</LastName>
     <Phone Type="Cell">555-1515</Phone>
  </Person>
</Friends>
```

Next, start a new XSL file as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="2.0">

<xsl:output method="text"/>

<xsl:template match="/">

  <xsl:for-each-group select="//Person" group-by="@Group">
    Group: <xsl:value-of select="current-grouping-key()"/>


  </xsl:for-each-group>

</xsl:template>

</xsl:stylesheet>
```

Save it in your **/xml1** folder as **Group.xsl**. Here we use the new **for-each-group** element, select all **Person** elements, and group by the value of the **Group** attribute. Each distinct value of the **Group** attribute is returned by the **current-grouping-key** function.

Run the **xslt** command as shown:

INTERACTIVE SESSION:

```
cold:~/xml1$ saxon9he.sh -xsl:Group.xsl FriendsX.xml
```

You'll see the two groups we defined:

OBSERVE:

```
Group: Family
Group: Work
```

Next, we'll loop over the values contained in each group. Modify **Group.xsl** as shown:

CODE TO TYPE:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="2.0">

<xsl:output method="text"/>

<xsl:template match="/">

  <xsl:for-each-group select="//Person" group-by="@Group">
    Group: <xsl:value-of select="current-grouping-key()"/>

     <xsl:for-each select="current-group()">
       Person: <xsl:value-of select="concat(FirstName, ' ', LastName)"/>
     </xsl:for-each>

  </xsl:for-each-group>

</xsl:template>

</xsl:stylesheet>
```

Here we added another **for-each** loop. The **current-group** function returns the values contained in each group. In other words, our style-sheet loops over each group, and then loops over the people in each group.

Save your files, and then in your terminal, run the **xslt** command again as shown:

INTERACTIVE SESSION:

```
cold:~/xml1$ saxon9he.sh -xsl:Group.xsl FriendsX.xml
```

You will see your friends, divided according to group (and independent of their order in the XML file):

OBSERVE:

```
 Group: Family
Person: Alex Chilton
Person: Jim Dandy
Person: Janie Jones
Group: Work
Person: Ana Ng
```

If you're familiar with SQL databases, you might notice a similarity to the SQL **GROUP BY** syntax.

XSLT 2.0 has lots of cool new features that will help you be a much more efficient and happy programmer!

In the next lesson you'll find a description of your final project. Can you believe it? You're almost done with the course! Excellent work so far.

Final Project

Welcome to the final lesson and project for your Introduction to XML course! When beginning any XML project, remember that XML and XSL are flexible technologies. There are lots of different ways to structure your data in XML, just as there are many different ways to create the same document using XSL. As an XML programmer, you'll want to make your XML and XSL as clear as possible for humans to understand. Choose your elements and attributes carefully, and craft a master layout for your documents. This layout will be protected by your schemas, and your XML files will be made more useful through translations to HTML, XML, or other types of files. Have fun with this final project!

Personal Information Manager

Throughout the course lessons, you learned how to use XML and XSL to create meaningful applications. Now you're ready to create your final XML application of: a personal information manager, or **PIM**.

A PIM keeps track of your contact information in an clear, manageable format. Typical entries include:

- First Name
- Last Name
- Home Phone
- Work Phone
- Cell Phone
- Address
- Email Address

Depending on your data, there may be many more or slightly different entries. For your final project, you'll define a format for your PIM information to be stored in XML. You'll create XSL documents to translate your source XML into HTML, and create a schema.

Storing the Data in XML

The first task at hand is to define how your information will be stored. Your project will need to contain at least seven defined elements and three attributes. Start the project by defining your schema. Your schema must be as specific as possible to strictly enforce your specifications. Be sure to use appropriate data types.

It's up to you to define your own XML markup for your PIM. Make sure your tags can be understood clearly by others! Also, be sure to validate your XML document against your schema—you don't want to turn in an invalid XML phonebook!

Output to HTML

If you maintain your own website, you'll find this part of the final project especially useful. You're going to use your knowledge of HTML, XML, and XSL to create an **XSL 2.0** file to generate *two* meaningful HTML representations of your phonebook.

You're required to use a table for your objective—so you'll likely have to use the **xsl:for-each** construct.

You also need to demonstrate your knowledge of XPath and XSLT 2.0 features.

That's it. After your final project is complete, you'll have demonstrated your knowledge of XML and XSL. Save your files as an enduring testament to your knowledge and expertise. Then go forth and create XML and XSL with confidence! You've earned it. Thanks for all of your hard work, and we hope to see you in another course soon!