

Introduction to XSLT

Transform XML data from one format to another with Extensible Stylesheet Language Transformations (XSLT)

Skill Level: Introductory

[Nicholas Chase \(ibmqquestions@nicholaschase.com\)](mailto:ibmqquestions@nicholaschase.com)

Freelance writer
Backstop Media

23 Jan 2007

The need to transform XML is so common that Extensible Stylesheet Language Transformations (XSLT) is considered one of the basic XML specifications. This tutorial explains how to create XSLT stylesheets. It also covers the basics of XPath, which enables you to select specific parts of an XML document. Finally, it gives you a look at some of the more advanced capabilities that XSLT offers.

Section 1. Before you start

This tutorial is for developers who want to use Extensible Stylesheet Language Transformations (XSLT) to transform the XML data into other forms without the need for to program in Java™ or other languages.

This tutorial assumes that you're familiar with XML, but does not get into any of the esoteric aspects of the language, so a basic understanding is sufficient. Except for small section on extensions, you will not need to have any particular programming language under your belt. Even then, the concepts are straightforward and apply to any programming language, as long as it is supported by the processor.

What is this tutorial about?

This tutorial is about Extensible Stylesheet Language Transformations (XSLT). XSLT is one of the fundamental XML-related specifications, enabling you to easily transform your XML data from one form to another.

In this tutorial, you'll learn the following:

- The basics of XSLT
- Using simple templates
- Propagating data
- Controlling spaces
- The basics of XPath
- XPath functions
- Looping and conditional statements
- Importing and including other stylesheets
- Extending XSLT
- XSLT variables

Section 2. Getting started

Suppose for a moment that you have a set of data in XML. You probably chose to store it that way in part because of the flexibility of XML. You know that you can use it on multiple platforms and with virtually any programming language. But sometimes, you need to change the form of that XML. For example, you might need to convert the data to another XML format to store in a different system, or to a different form for presentation or some other use.

For example, you might have data in XML that you wish to display on the Web, which means converting it to HTML. You could manually go through the document and make the changes, of course, or you might consider loading the XML into a DOM and then building the output document manually.

Fortunately, you don't have to do that. There is an easier way.

What is XSLT?

Extensible Stylesheet Language Transformations (XSLT) provides a way to automatically transform XML data from one format to another. The target form is usually another XML document, but it doesn't have to be; you can transform XML data into virtually anything simply by creating a XSLT stylesheet and processing the data. If you want to make changes to the output, you simply make changes to the stylesheet and process the XML again. This has the additional advantage of giving power to nonprogrammers, such as designers, who can edit the stylesheet and affect the results.

Let's look at an example.

What you're going to accomplish

In this tutorial, you will take an XML document and transform it into an XHTML document that you can display as a Web page. The input data is a simple file of recipes (see Listing 1).

Listing 1. The basic data

```
<recipes>
  <recipe>
    <name>Gush'gosh</name>
    <ingredients>

<ingredient><qty>1</qty><unit>pound</unit>
<food>hamburger</food></ingredient>

<ingredient><qty>1</qty><unit>pound</unit>
<food>elbow macaroni</food></ingredient>

<ingredient><qty>2</qty><unit>cups</unit>
<food>brown sugar</food></ingredient>
  <ingredient><qty>1</qty><unit>bag</unit>
<food>chopped onions</food></ingredient>

<ingredient><qty>1</qty><unit>teaspoon</unit>
<food>dried dill</food></ingredient>
    </ingredients>
    <instructions>
      <instruction>Brown the hamburger.</instruction>
      <instruction>Add onions and cook until
transparent.</instruction>
      <instruction>Add brown sugar and dill.</instruction>
      <instruction>Cook and drain pasta.</instruction>
      <instruction>Combine meat and pasta.</instruction>
    </instructions>
  </recipe>

  <recipe>
    <name>A balanced breakfast</name>
    <ingredients>
      <ingredient><qty>1</qty><unit>cup</unit>
<food>cereal</food></ingredient>

<ingredient><qty>1</qty><unit>glass</unit>
<food>orange juice</food></ingredient>
```

```

<ingredient><qty>1</qty><unit>cup</unit>
<food>milk</food></ingredient>

<ingredient><qty>2</qty><unit>slices</unit>
<food>toast</food></ingredient>
  </ingredients>
  <instructions>
    <instruction>Combine cereal and milk in
bowl.</instruction>
    <instruction>Add all ingredients to table.</instruction>
  </instructions>
</recipe>

</recipes>

```

Editor's note: These recipes are examples only, as interpreted by the author. The correct recipe for Gush'gosh (as provided by his wife, who actually does the cooking) consists of 1 lb. hamburger, 1 lb. elbow macaroni, 1/2 cup brown sugar, 1 small bag (about 10 oz) chopped onions, 1 teaspoon dried dill and 1 small can of tomato paste, which is added with the brown sugar.

This is, of course, a fairly simple example so you don't get bogged down in details of the data, but your XML data can be anything, from logging activity to financials.

The goal is to transform this data into an XHTML page that displays the recipes individually and formats their ingredients and instructions (see Listing 2).

Listing 2. The output

```

<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/TR/xhtml1/strict">
<head><title>Recipe</title></head>
<body>
<h2>Gush'gosh</h2>
<h3>Ingredients:</h3>
<p>
  1 pound hamburger<br/>
  1 pound elbow macaroni<br/>
  2 cups brown sugar<br/>
  1 bag chopped onions<br/>
  1 teaspoon dried dill<br/>
</p>
<h3>Directions:</h3>
<ol>
  <li>Brown the hamburger.</li>
  <li>Add onions and cook until transparent.</li>
  <li>Add brown sugar and dill.</li>
  <li>Cook and drain pasta.</li>
  <li>Combine meat and pasta.</li>
</ol>

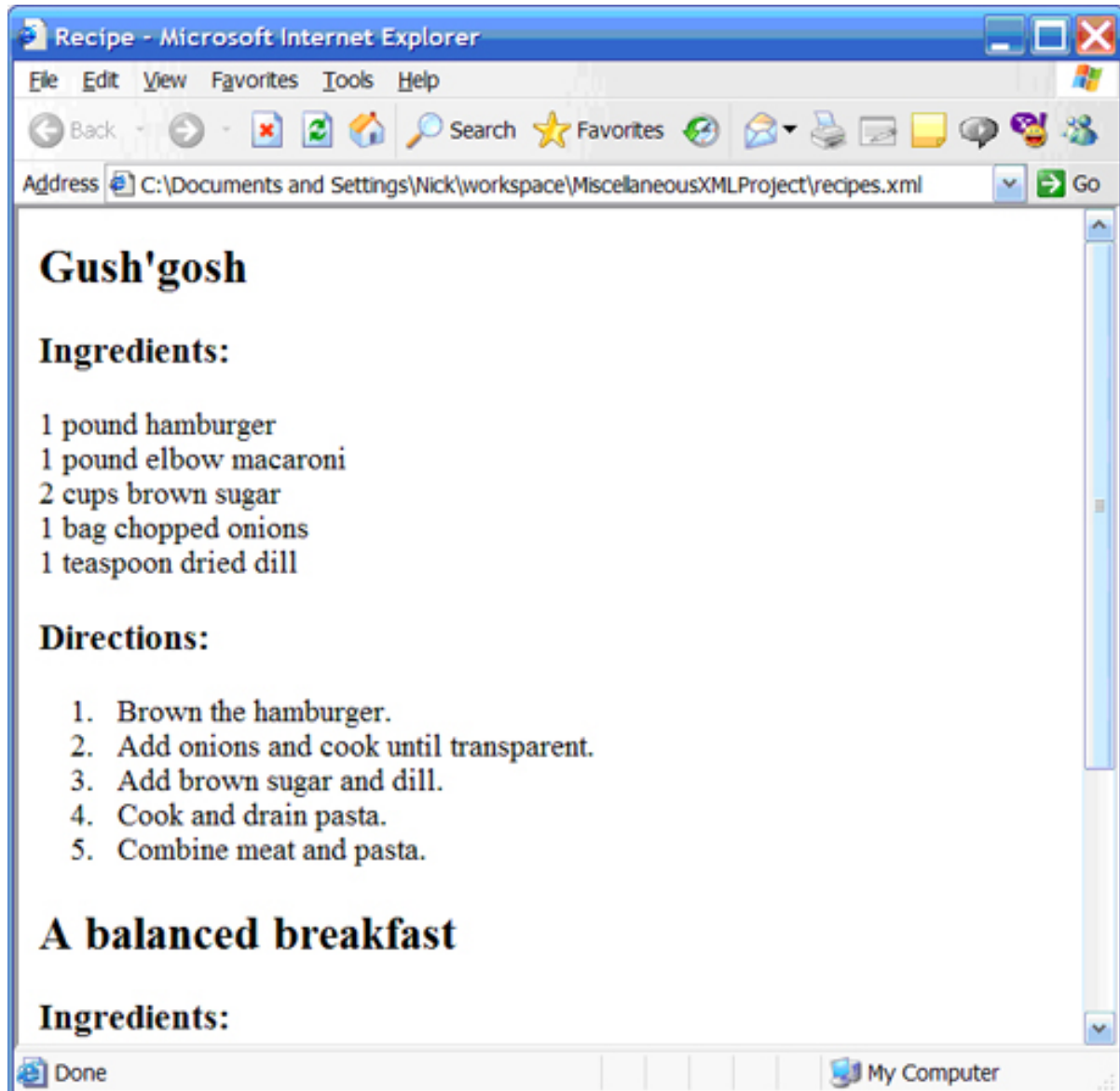
<h2>A balanced breakfast</h2>
<h3>Ingredients:</h3>
<p>
  1 cup cereal<br/>
  1 glass orange juice<br/>
  1 cup milk<br/>
  2 slices toast<br/>
</p>
<h3>Directions:</h3>
<ol>

```

```
</ol>  
</body>  
</html>  
  
<li>Combine cereal and milk in bowl.</li>  
<li>Add all ingredients to table.</li>
```

You can then display this result in the browser, as shown in Figure 1.

Figure 1. Result displayed in the browser



As mentioned, the final destination can be any format, not just XHTML, or even XML.

Let's start with the basic transformation.

The basic stylesheet

The most basic of stylesheets is simply an XML document that includes XSLT output (see Listing 3).

Listing 3. The most basic stylesheet

```
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns="http://www.w3.org/TR/xhtml1/strict">
  <head>
    <title>Recipe</title>
  </head>
  <body>
    <h2><xsl:value-of
  select="/recipes/recipe/name"/></h2>
    <h3>Ingredients:</h3>
    <p><xsl:value-of
  select="/recipes/recipe/ingredients"/></p>
    <h3>Directions:</h3>
    <p><xsl:value-of
  select="/recipes/recipe/instructions"/></p>
  </body>
</html>
```

Notice the use of the `xsl:` namespace. Adding this namespace tells the processor which elements are related to processing and which elements should be simply output. The `value-of` elements tell the processor to insert a particular piece of data in that location. Which particular piece is determined by the content of the `select` attribute.

The `select` attribute consists of an XPath expression. XPath will be discussed in detail in [More advanced XPath](#), but here you can see that the `name`, `ingredients`, and `instructions` elements are being accessed by stepping through the hierarchy of the document. You start with the root element, `/recipes`, and work your way down from there.

How to perform the transformation

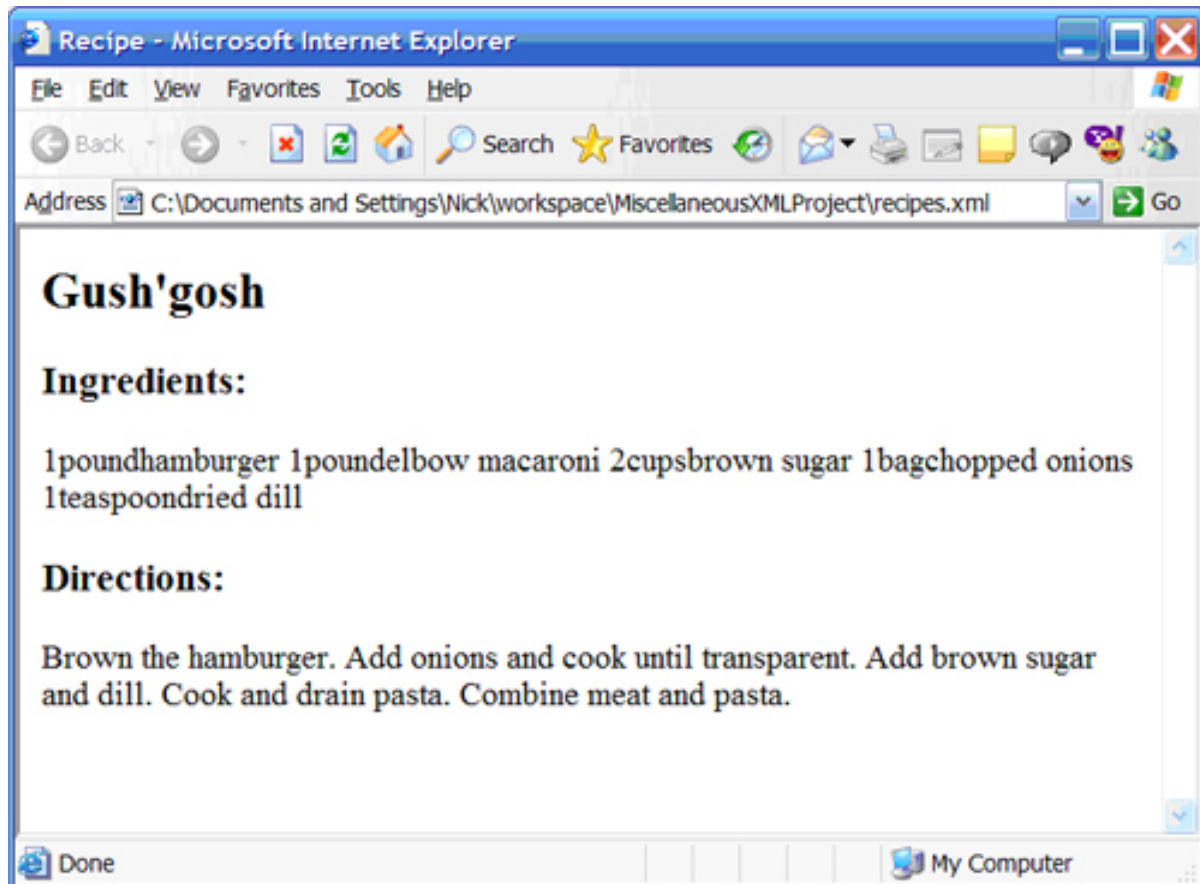
The simplest way to perform an XML transformation is to add the `xml-stylesheet` directive to the XML and display it in the browser (see Listing 4).

Listing 4. Adding the xml-stylesheet processing instruction to the XML

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="basicstylesheet.xsl" version="1.0"
?>
<recipes>
  <recipe>
    <name>Gush 'gosh</name>
  ...
```

This processing instruction tells the browser to retrieve the stylesheet located at `basicstylesheet.xsl` and use it to transform the XML data and output the results. If you then open the XML document in Microsoft® Internet Explorer®, you will see a result similar to Figure 2.

Figure 2. Retrieving the stylesheet and transforming the XML data



Now, this isn't exactly what you want, but if you do a view/source in the browser, all you'll see is the original XML. To see the results of the actual transformation, you need to perform the transformation and create an output file. You can do this from the command line using Java code with the following command (see Listing 5).

Listing 5. Transforming the document from the command line

```
java org.apache.xalan.xslt.Process -IN recipes.xml -XSL basicstylesheet.xsl -out result.html
```

If you get a `ClassNotFoundException`, you might need to download Apache Xalan (see "Get products and technologies" in [Resources](#)) and add the included JAR files to your classpath.

Once you perform the transformation shown in Listing 5, you will see that the

result.html file contains the following code shown in Listing 6.

Listing 6. The results

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/TR/xhtml1/strict">
<head><title>Recipe</title></head>
<body>
<h2>Gush'gosh</h2>
<h3>Ingredients:</h3>
<p>
    1poundhamburger
    1poundelbow macaroni
    2cupsbrown sugar
    1bagchopped onions
    1teaspoondried dill
  </p>
<h3>Directions:</h3>
<p>
    Brown the hamburger.
    Add onions and cook until transparent.
    Add brown sugar and dill.
    Cook and drain pasta.
    Combine meat and pasta.
  </p>
</body></html>
```

I added a little bit of spacing for readability, but there are a couple of things to notice here. First, [Listing 6](#) displays information for only one recipe. Second, the ingredients were crammed together without any spaces. Neither of these is what you want. Fortunately, you can create more specific templates to output the data in the form in which you want it.

Section 3. Adding the templates

A transformation doesn't do you any good unless it leaves the data in the form in which you want it. To do that, you'll need to know how to use templates, which you'll learn in this section.

Creating templates

Most stylesheets do not use the very simple form you just saw in the previous section. Instead, they are broken down into a series of templates, each of which is applied to a particular type of data. Let's start by converting the stylesheet into this form (see Listing 7).

Listing 7. The reworked stylesheet

```

        <xsl:stylesheet
version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:template match="/">
<html>
  <head>
    <title>Recipe</title>
  </head>
  <body>
    <h2><xsl:value-of select="/recipes/recipe/name"/></h2>
    <h3>Ingredients:</h3>
    <p><xsl:value-of
select="/recipes/recipe/ingredients"/></p>
    <h3>Directions:</h3>
    <p><xsl:value-of
select="/recipes/recipe/instructions"/></p>
  </body>
</html>
</xsl:template>

        </xsl:stylesheet>

```

All of the information here is the same as it was before, except that the processor looks at the stylesheet and starts with the template set to match the document root, as specified by the `match` attribute here. It then outputs that template, including any values, just as it did before. If you execute the transformation now, you should see results exactly like those shown in [Listing 6](#).

But that's not what you want. Instead, you want to be able to format the ingredients and instructions. To do that, you can create separate templates for those elements and tell the stylesheet to include them (see [Listing 8](#)).

Listing 8. Creating additional templates

```

<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:template match="/">
<html>
  <head>
    <title>Recipe</title>
  </head>
  <body>
    <h2><xsl:value-of select="/recipes/recipe/name"/></h2>
    <h3>Ingredients:</h3>
    <p><xsl:apply-templates
select="/recipes/recipe/ingredients"/></p>
    <h3>Directions:</h3>
    <p><xsl:apply-templates
select="/recipes/recipe/instructions"/></p>
  </body>
</html>
</xsl:template>

<xsl:template match="ingredients">

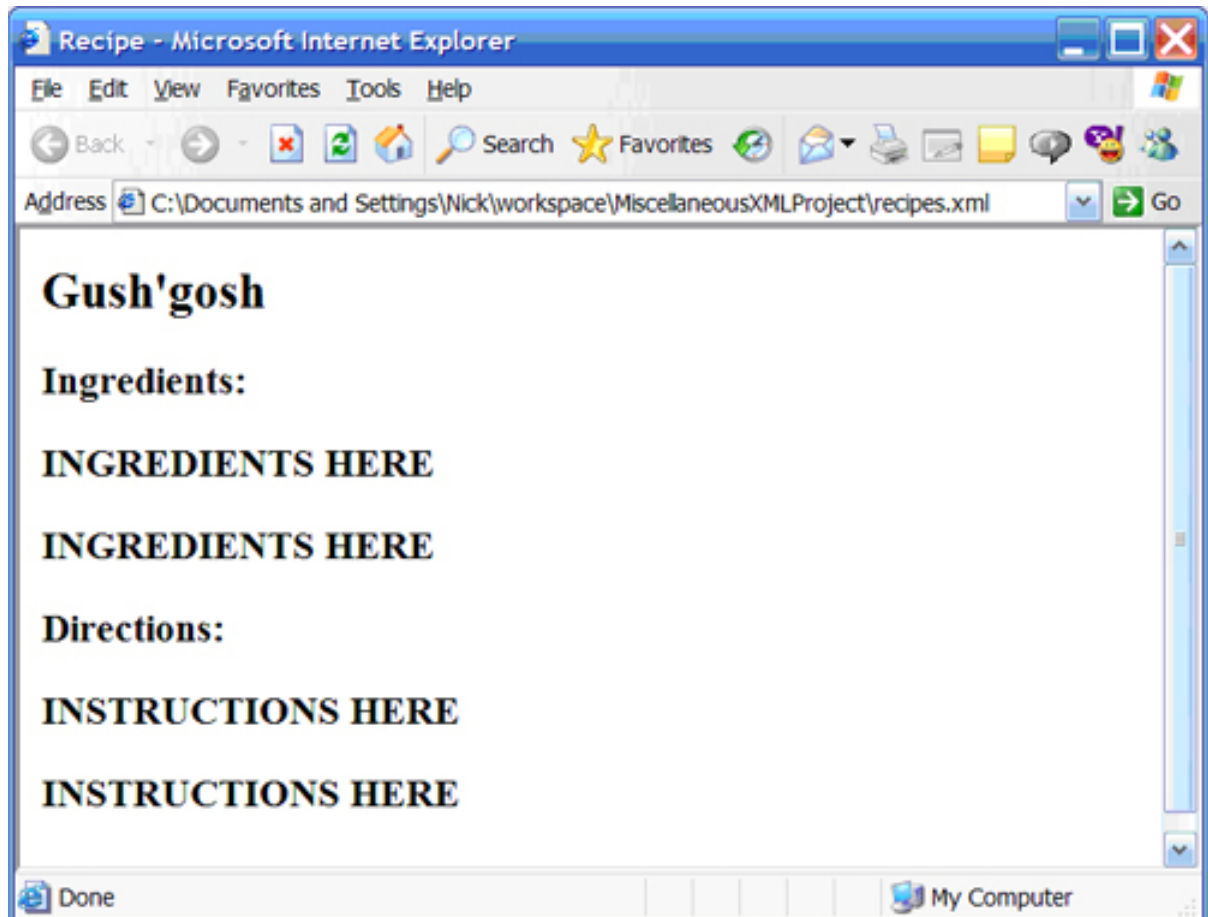
```

```
<h3>INGREDIENTS HERE</h3>
</xsl:template>

<xsl:template match="instructions">
  <h3>INSTRUCTIONS HERE</h3>
</xsl:template>
</xsl:stylesheet>
```

Notice that rather than simply outputting the `value-of` element, you are now telling the stylesheet to apply any applicable templates to the ingredients and instructions elements. You then created separate templates for these elements, specifying them in the `match` attribute. The processor gets to the `apply-templates` element and selects any ingredients elements in the document. It then looks for an ingredients template, and when it finds it, it outputs that template. It does the same for the instructions element. The result looks like that shown in Figure 3.

Figure 3. Applying applicable templates to the ingredients and instructions elements



Well, this is a little closer, in that at least now you know that there are two recipes, but obviously you don't want to combine the ingredients for all the recipes and the directions for all of the recipes. Fortunately, you can solve this problem by organizing your templates a little better.

Propagating templates

To organize your data a little better, pay attention to how you break out and propagate templates. XSLT was designed to enable you to process information in an iterative manner. For example, you can break information out by recipe, and then format the instructions and ingredients (see Listing 9).

Listing 9. Breaking out the recipes

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/xhtml1/strict">

  <xsl:template match="/">
    <html>
      <head>
        <title>Recipe</title>
      </head>
      <body>
        <xsl:apply-templates select="/recipes/recipe"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="recipe">

    <h2><xsl:value-of select="./name"/></h2>
    <h3>Ingredients:</h3>
    <p><xsl:apply-templates select="./ingredients"/></p>
    <h3>Directions:</h3>
    <p><xsl:apply-templates
select="./instructions"/></p>

  </xsl:template>

  <xsl:template match="ingredients">

    <h3>INGREDIENTS HERE</h3>

  </xsl:template>

  <xsl:template match="instructions">

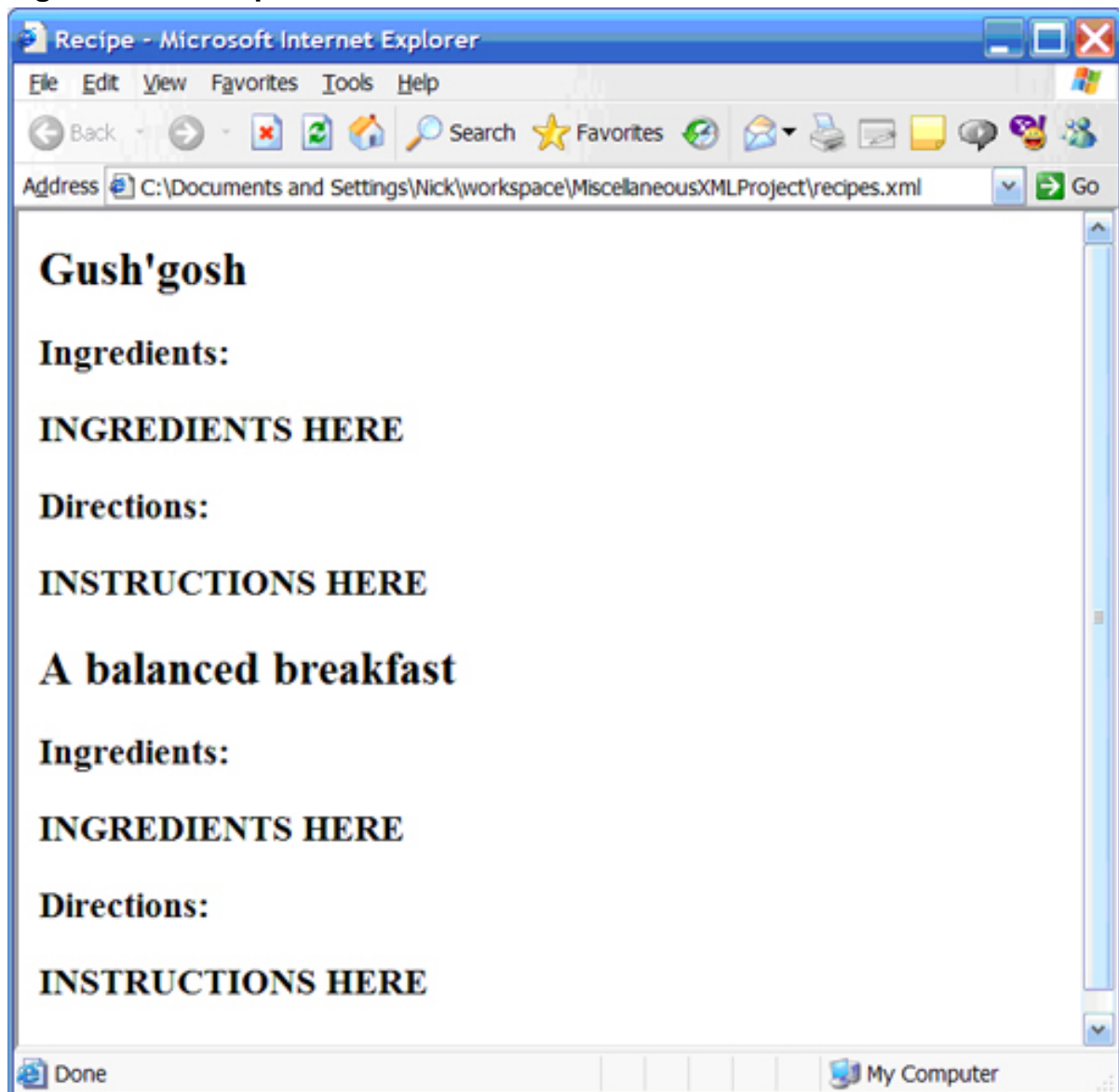
    <h3>INSTRUCTIONS HERE</h3>

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

In this case, the stylesheet outputs the basic page (the HTML) and then loops through each recipe, outputting the name, ingredients, and instructions for each.

Again, XPath will be examined in [More advanced XPath](#), but in this case, the `recipe` element becomes the "context node", so your `select` attributes are relative to that node, just as a file might be relative to a particular directory in a file system. The result looks like that shown in Figure 4.

Figure 4. The recipe element becomes the context node



All right, the format is closer to what you want, but you still need to display the actual information. To do that, modify the instructions and ingredients templates (see Listing 10).

Listing 10. Dealing with the ingredients and instructions

```

...
<xsl:template match="recipe">
    <h2><xsl:value-of select="./name"/></h2>
    <h3>Ingredients:</h3>
    <p><xsl:apply-templates select="./ingredients"/></p>
    <h3>Directions:</h3>
    <ol><xsl:apply-templates
select="./instructions"/></ol>
</xsl:template>

<xsl:template match="ingredients/ingredient">
    <xsl:value-of select="./qty"/> <xsl:value-of
select="./unit"/> <xsl:value-of select="./food"/><br />
</xsl:template>

<xsl:template match="instructions/instruction">
    <li><xsl:value-of select="."/></li>
</xsl:template>
</xsl:stylesheet>

```

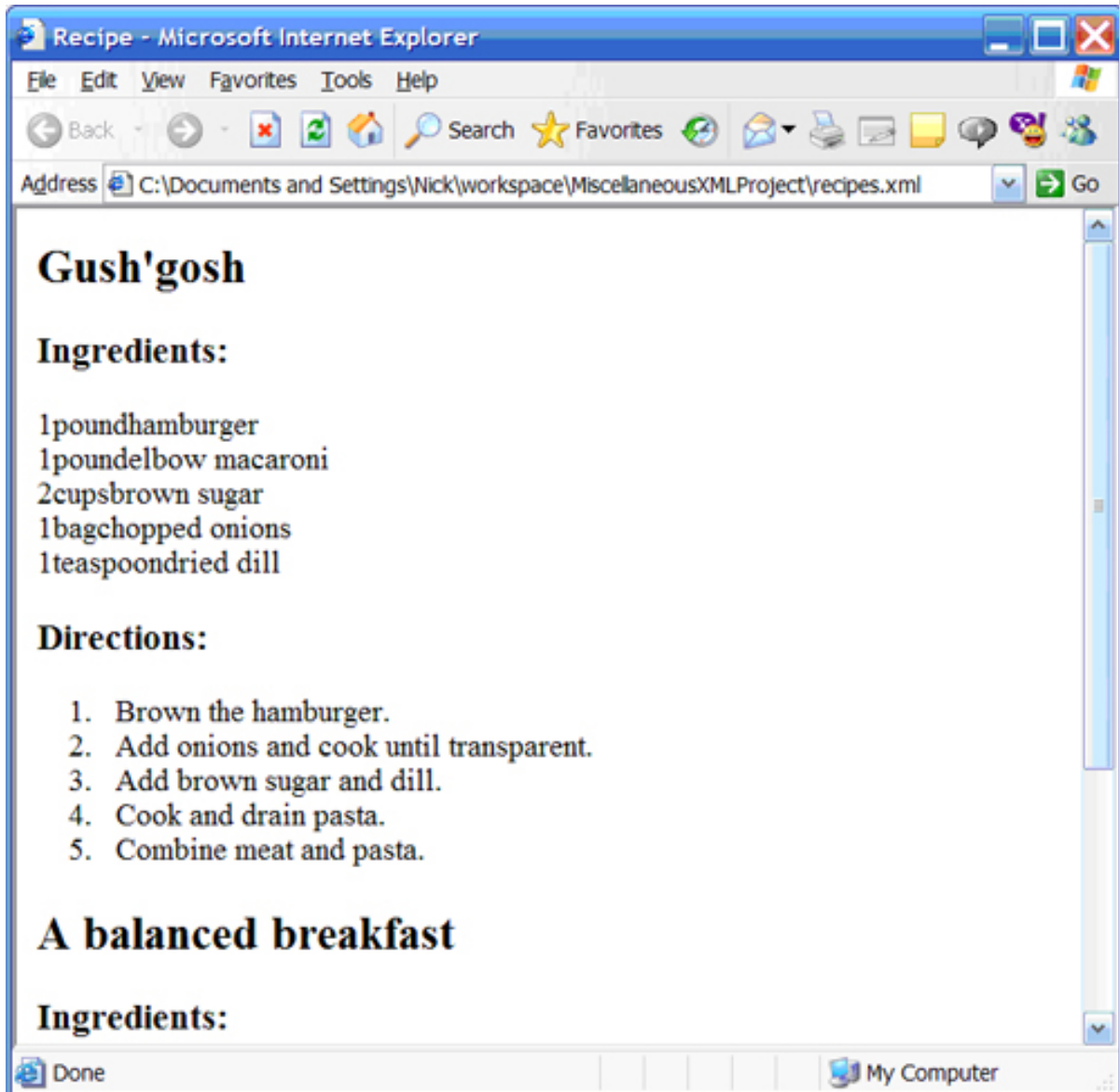
Something to notice here when you propagate templates: You told the processor to apply any applicable templates to the `ingredients` element, but you don't have a template specifically for that element. If you apply templates to an element and there are no templates to apply, the data simply will not appear. That's not the case here.

Instead, you took advantage of the fact that when you told the processor to apply any applicable templates to the `ingredients` element, it checks not only for the `ingredients` element, but also for any children of the `ingredients` element. This is how it finds the `ingredient` template, in which you output the quantity, unit, food, and a line break.

You did the same for the instructions, formatting them as list items. Notice that you created the actual ordered list in the main recipe template, and then sent the items off for individual processing.

The result looks like that shown in Figure 5.

Figure 5. Creating the ordered list in the main recipe template



The format is definitely closer to what you want now. But if you look at the output, you can see that the spacing problems in the ingredients still exists (see Listing 11).

Listing 11. The raw output

```
<?xml version="1.0" encoding="UTF-8"?>
<html
xmlns="http://www.w3.org/TR/xhtml1/strict"><head><title>Recipe
</title></head><body><h2>Gush'gosh</h2><h3>
Ingredients:</h3><p>
  1poundhamburger<br/>
  1poundelbow macaroni<br/>
  2cupsbrown sugar<br/>
  1bagchopped onions<br/>
  1teaspoondried dill<br/>
</p><h3>Directions:</h3><ol>
```

```

    <li>Brown the hamburger.</li>
    <li>Add onions and cook until transparent.</li>
    ...

```

Adding spaces

Now why would this happen, when you included spaces in the stylesheet? Shouldn't they appear in the output? Actually, not necessarily. There are ways to tell the stylesheet to preserve whitespace -- which will be examined in [Looping and importing](#) -- but in some cases it is simpler to explicitly add text to your output (see Listing 12).

Listing 12. Adding text

```

...
    <xsl:template match="ingredients/ingredient">
        <xsl:value-of select="./qty"/><xsl:text>
</xsl:text>
        <xsl:value-of select="./unit"/><xsl:text>
</xsl:text>
        <xsl:value-of select="./food"/><br />
    </xsl:template>
...

```

This takes care of the missing spaces in the data. You can also use the `text` element to add any arbitrary text to the template. (Remember, that's text, not elements such as the line break.) The result is the output as you want to see it (see Listing 13).

Listing 13. the final output

```

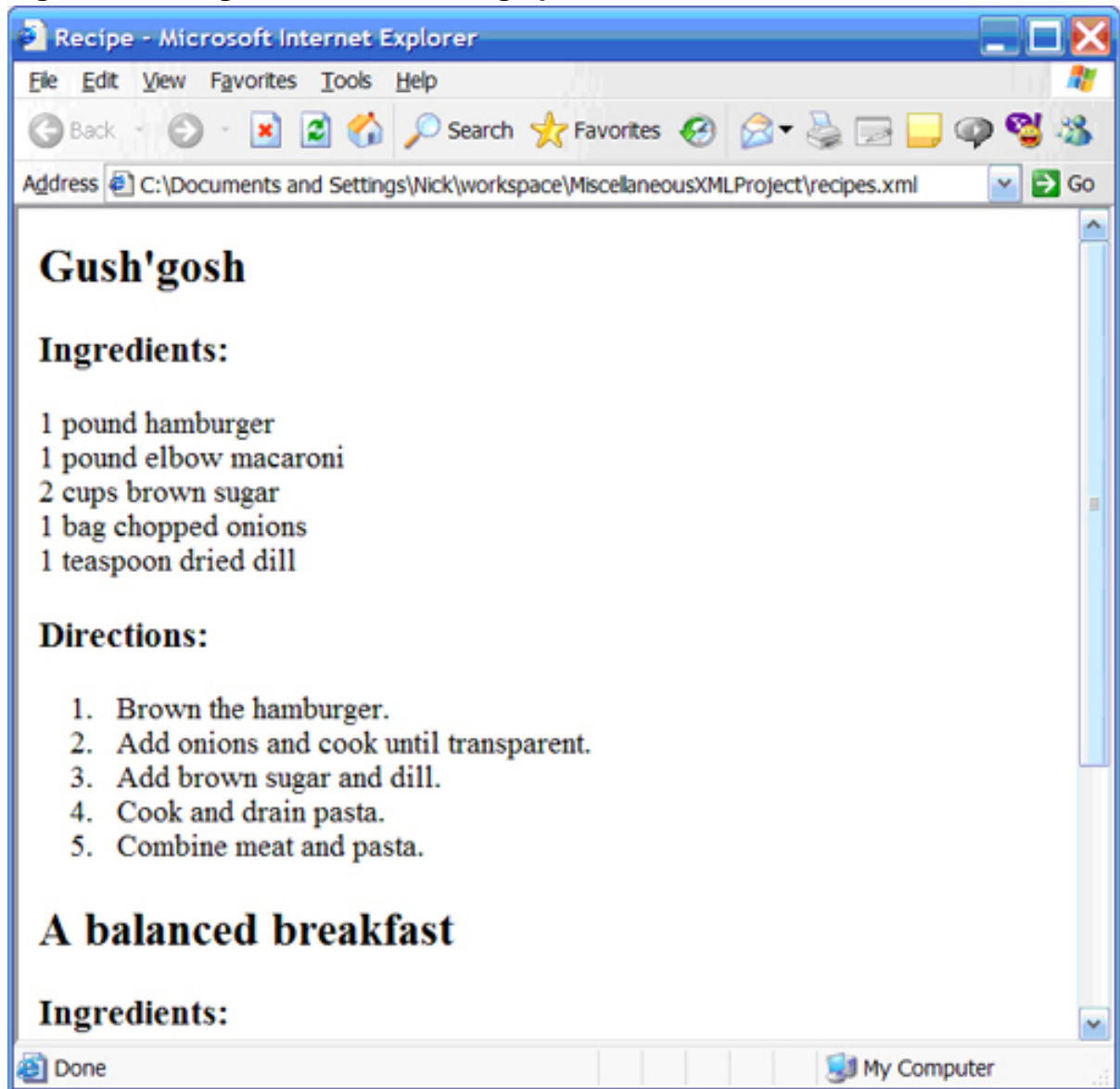
<?xml version="1.0" encoding="UTF-8"?>
<html
xmlns="http://www.w3.org/TR/xhtml1/strict"><head><title>Recipe
</title></head><body><h2>Gush'gosh</h2><h3>
Ingredients:</h3><p>
    1 pound hamburger<br/>
    1 pound elbow macaroni<br/>
    2 cups brown sugar<br/>
    1 bag chopped onions<br/>
    1 teaspoon dried dill<br/>
</p><h3>Directions:</h3><ol>
    <li>Brown the hamburger.</li>
    <li>Add onions and cook until transparent.</li>
    <li>Add brown sugar and dill.</li>
    <li>Cook and drain pasta.</li>
    <li>Combine meat and pasta.</li>
</ol><h2>A balanced
breakfast</h2><h3>Ingredients:</h3><p>
    1 cup cereal<br/>
    1 glass orange juice<br/>
    1 cup milk<br/>
    2 slices toast<br/>

```

```
</p><h3>Directions:</h3><ol>  
  <li>Combine cereal and milk in bowl.</li>  
  <li>Add all ingredients to table.</li>  
</ol></body></html>
```

You can see the results in Figure 6.

Figure 6. Taking care of the missing spaces in the data



Next, you'll learn how to use XPath to add specific information to the page.

Section 4. Basic XPath

The ability to transform data in the way in which you would like to see it requires an understanding of XML Path Language, or XPath, which enables you to control which data propagates and/or displays. This section explains the concepts behind XPath and shows you how to create basic expressions.

What is XPath?

Now, you might notice that one very important part of making the stylesheet work is the ability to select a specific part of the document. For example, if you want to display the instructions, you need to know how to reference them. In XSLT, you reference this information through the use of an XPath expression.

An XPath expression can select a single node or a set of nodes, or it can return a single value based on one or more nodes in the document. So far, you've dealt with fairly simple XPath expressions, which select a node or nodes by walking down the hierarchy. XPath provides several ways to specify groups of nodes based on relationships such as parent-child or ancestor-descendant. In this tutorial, you'll look at these relationships, called axes.

You'll also look at some of XPath's more powerful functions, and you'll look at *predicates*, which are essentially conditional statements that you can add to an expressions. For example, in [Setting the context](#) you saw how to select all of a document's recipe elements; predicates enable you to select only a specific element based on specific criteria.

Finally, you'll look at functions, which take that power one step further by enabling much of the same types of logic you might write into a procedural program.

Let's start by looking at an expression's context.

Setting the context

The first step in understanding XPath is to understand that the results you get will be largely determined by the current context node. You can think of the context node as a sort of "you are here" sign, from which you move off in various directions according to the XPath expression. For example, consider this simple stylesheet (see Listing 14).

Listing 14. Showing the context

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```

xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:template match="/">

<xsl:copy-of select="." />

</xsl:template>

</xsl:stylesheet>

```

Here is a new XSL element, `copy-of`. Where `value-of` outputs the text content of an element, `copy-of` does exactly what it says, and outputs the actual node referenced by the `select` attribute.

In this case, the `select` attribute has one of the simplest XPath expressions possible. A single period (`.`) refers to the context node, just as it would in a file system when you want to refer to "the current directory, whatever that is." So if you performed this transformation, you get a result of (see Listing 15).

Listing 15. The simplest transformation

```

<recipes>
<recipe recipeId="1">
<name>Gush'gosh</name>
<ingredients>

<ingredient><qty>1</qty><unit>pound</unit>
<food>hamburger </food></ingredient>

<ingredient><qty>1</qty><unit>pound</unit>
<food>elbow macaroni</food></ingredient>

<ingredient><qty>2</qty><unit>cups</unit>
<food>brown sugar</food></ingredient>

<ingredient><qty>1</qty><unit>bag</unit>
<food>chopped onions</food></ingredient>

<ingredient><qty>1</qty><unit>teaspoon</unit>
<food>dried dill</food></ingredient>
</ingredients>
<instructions>
<instruction>Brown the hamburger.</instruction>
<instruction>Add onions and cook until transparent.</instruction>
<instruction>Add brown sugar and dill.</instruction>
<instruction>Cook and drain pasta.</instruction>
<instruction>Combine meat and pasta.</instruction>
</instructions>
</recipe>
...
</recipes>

```

You'll notice that this is simply the document repeated to you; that's because the context node, as specified by the template's `match` attribute is the document root, or `/`. If you change the context node, you can change the output. For example, you can set the context node to be the first recipe (see Listing 16).

Listing 16. Moving the context node

```

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">

  <xsl:template match="/recipes/recipe">

    <xsl:copy-of select="." />

  </xsl:template>

</xsl:stylesheet>

```

Now if you run the transformation, you can see the difference (see Listing 17).

Listing 17. The results

```

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

<recipe recipeId="1">
  <name>Gush'gosh</name>
  <ingredients>

    <ingredient><qty>1</qty><unit>pound</unit>
    <food>hamburger</food></ingredient>

    <ingredient><qty>1</qty><unit>pound</unit>
    <food>elbow macaroni</food></ingredient>

    <ingredient><qty>2</qty><unit>cups</unit>
    <food>brown sugar</food></ingredient>

    <ingredient><qty>1</qty><unit>bag</unit>
    <food>chopped onions</food></ingredient>

    <ingredient><qty>1</qty><unit>teaspoon</unit>
    <food>dried dill</food></ingredient>
  </ingredients>
  <instructions>
    <instruction>Brown the hamburger.</instruction>
    <instruction>Add onions and cook until transparent.</instruction>
    <instruction>Add brown sugar and dill.</instruction>
    <instruction>Cook and drain pasta.</instruction>
    <instruction>Combine meat and pasta.</instruction>
  </instructions>
</recipe>

<recipe recipeId="2">
  <name>A balanced breakfast</name>
  <ingredients>

    <ingredient><qty>1</qty><unit>cup</unit>
    <food>cereal</food></ingredient>

    <ingredient><qty>1</qty><unit>glass</unit>
    <food>orange juice</food></ingredient>

    <ingredient><qty>1</qty><unit>cup</unit>
    <food>milk</food></ingredient>

```

```

<ingredient><qty>2</qty><unit>slices</unit>
<food>toast</food></ingredient>
</ingredients>
<instructions>
<instruction>Combine cereal and milk in bowl.</instruction>
<instruction>Add all ingredients to table.</instruction>
</instructions>
</recipe>

```

Because you moved the context node, the output changed. This becomes important when you want to select a node relative to the context node. For example you might want to select just the title of the recipe (see Listing 18).

Listing 18. Selecting just the title

```

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:template match="/recipes/recipe">

<xsl:copy-of select="./name" />

</xsl:template>

</xsl:stylesheet>

```

Here you're selecting the `name` element that is one level down from the current context node, so the results will be (see Listing 19).

Listing 19. The results

```

<?xml version="1.0" encoding="UTF-8"?>
<name>Gush'gosh</name>
<name>A balanced breakfast</name>

```

You'll examine specifying nodes by number later, but for the moment notice that you can move the context node to the second recipe (see Listing 20).

Listing 20. Moving the context node once more

```

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:template match="/recipes/recipe">
</xsl:template>

<xsl:template match="/recipes/recipe[2]">
<xsl:copy-of select="./name" />

```

```
</xsl:template>

</xsl:stylesheet>
```

(Don't worry about the extra template; it's to keep the text of the other recipe from appearing.)

You can see that the resulting transformation shows this change of context (see Listing 21).

Listing 21. The results

```
<?xml version="1.0" encoding="UTF-8"?>
<name>A balanced breakfast</name>
```

Understanding axes

Now that you know where you're starting from, it's useful to know where you can go. So far, you've used very simple XPath statements that are much like the hierarchy in a filesystem, but XPath provides you with much more power than that. For example, while you selected only children, you also have the option to look for parents of a particular node, as well as descendants or ancestors.

To start with, let's talk about notation. You used the simplified, abbreviated form of asking for children. To use the "long form" version, you specify expressions such as `child::name` instead of `./name`

In both cases, you specify any node that is a child of the context node, and is also a `name` element. You can also chain these together just as you did before, as in `/child::recipes/child::recipe` instead of `/recipes/recipe`.

Descendants

Now, you might wonder why to bother with the long form, when the short form is so much easier. Well, you wouldn't, if selecting children were all you could do. As it happens, however, you can also use this notation to select other relationships. For example, the XPath expression `descendant::instruction` selects all instruction elements that are descendants of the context node, and not just children. Again, you can combine instructions. For example, you can select all of the instructions for the second recipe: `/recipes/recipe[2]/descendant::instruction`.

One variation on the descendant axis is the descendant-or-self axis, which selects the requested node from all of the context nodes descendants, but also looks at the context node itself. For example, the expression `descendant-or-self::instructions` selects all of the instructions nodes at or below the context node. A common abbreviation for this axis is the double slash, `//`.

That means that the expressions: `/recipes/recipe[2]//instructions` and `//instructions` select all of the instructions for the second recipe, and all of the instructions in the document, respectively. This second example is very common, and handy when you want to simply select all elements of a particular type within the document.

Attributes

Another common task you will run into is the need to select an attribute for a particular element. For example, you might want to select the `recipeId` for a particular recipe. The expression `/recipes/recipe/attribute::recipeId` selects the `recipeId` attribute for all of the `recipe` elements. This axis also has an abbreviated form, so you can write it as: `/recipes/recipe/@recipeId`.

Parents

So far everything you looked at brings you *down* the hierarchical tree, but you also have the option to go *up* by selecting the parent of a particular node. For example, suppose the context node was one of the directions, but you wanted to output the `recipeId` for the current recipe. You could do that as follows:

```
./parent::node()/parent::node()/@recipeId.
```

This expression starts at the current node, the instruction, then moves to the node that is that node's parent, the instructions element, and then to THAT node's parent, recipe, and then to the appropriate attribute. Of course, you might be more familiar with the abbreviated form: `../..../@recipeId`.

Now let's look at setting some conditions.

Section 5. More advanced XPath

Most of the time, you can do what you need with just the techniques already covered. It's not unusual, however, to run into a situation which requires that you to be more specific. This section shows you how to use predicates to select nodes based on specific criteria, and introduces you to some of the functions built into XPath.

Using predicates

Often, you want not just any node, but a specific node based on specific conditions. You saw an example of that earlier, when you used the expression

`/recipes/recipe[2]//instructions`. That's really the abbreviated version of `/recipes/recipe[position() = 2]//instructions` and what it means is that you're asking the XPath processor to go through each `recipes` element (of which there is, of course, only one), and for each `recipes` element go through each `recipe` element. For each `recipe` element, check to see if the expression `position() = 2` is true. (In other words, is this the second recipe in the list?) If that statement, called the predicate, is true, the processor uses that node and moves on, returning any instructions.

You can do a variety of things with predicates. For example, you might return only recipes that have a `name`: `/recipes/recipe[name]`. This expression is only testing for the existence of a `name` element that is a child of the `recipe` element. You can also look for specific values. For example, you can return only the recipe named "A balanced breakfast": `//recipe[name="A balanced breakfast"]`.

Note that the predicate only tells the processor whether to return the actual node, so in this case, what is returned is the `recipe` element and not the name. On the other hand, you can tell the processor to return only the name of the first recipe with either of these two expressions (see Listing 22).

Listing 22. Returning only the names of the first recipe

```
//recipe[@recipeId='1']/name
//name[parent::recipe[@recipeId='1']]
```

In the case of the first expression, you first select all of the `recipe` elements, then return only the one that has a `recipeId` attribute of 1. Once you find that node, you move to its child node called `name`, and return that. In the case of the second expression, you find all of the `name` elements, then select only the one that has a parent with a `recipeId` attribute of 1. In either case, you get the same output (see Listing 23).

Listing 23. Output

```
<?xml version="1.0" encoding="UTF-8"?>
<name>Gush'gosh</name>
```

Functions

XPath also provides a number of different functions. Some of them relate to the nodes themselves, such as those that look at position, some of them manipulate strings, some relate to numbers, such as sums, and some relate to boolean values.

Nodeset functions

The nodeset-related functions help you to do things like choose a particular node based on position. For example, you can specifically request the last recipe: `//recipe[last()]`. The expression selects all of the `recipe` elements, and then returns only the last one. You can also use functions on their own, as opposed to as part of a predicate. For example, you can specifically request the count of recipe elements: `count(//recipe)`.

You've already looked at the `position()` function and how that works. Other nodeset-related functions include `id()`, `local-name()`, `namespace-uri()`, and `name()`.

String functions

Most of the string functions are for manipulating strings rather than testing them, with the exception of the `contains()` function. The `contains()` function can tell you whether a particular string is part of a larger whole. For example, you can return only the nodes that contain a particular string, such as: `//recipe[contains(name, 'breakfast')]`.

This expression returns the recipe element that has the string "breakfast" in its name element.

The `substring()` function enables you to select a specific range of characters from a string. For example, the expression: `substring(//recipe[2]/name, 1, 5)` returns `A bal`.

The first argument is the complete string, the second is the position of the first character, and the third is the length of the string.

Other string functions include `concat()`, `substring-before()`, `substring-after()`, `starts-with()`, and `string-length()`.

Numeric functions

Numeric functions include the `number()` function, which converts a value into a numeric value so that other functions can act on it. Number functions also include `sum()`, `floor()`, `ceiling()`, and `round()`. For example, you can find the sum of all of the `recipeld` values with the expression: `sum(//recipe/@recipeId)`.

True, there's not much reason to do such a calculation, but it's the only numeric value in the sample document.

The `floor()` function finds the greatest integer that is less than or equal to the supplied value, while the `ceiling()` function goes in the other direction. The `round()` performs in the traditional way (see Listing 24).

Listing 24. Results of numeric functions

```
floor(42.7) = 42
ceiling(42.7) = 43
round(42.7) = 43
```

Boolean functions

Boolean functions are most useful when you get into conditional expressions, which you'll look at in [Conditional processing](#). Perhaps the most useful is the `not()` function, which can be used to tell if a particular node does not exist. For example, the expression `//recipe[contains(name, 'breakfast')]` returns every recipe that has the string "breakfast" in its `name` element. But what if you wanted all of the recipes that were not for breakfast? You could use the expression:

```
//recipe[not(contains(name, 'breakfast'))].
```

Other boolean functions include `true()` and `false()`, which return constants, and `boolean()`, which converts a value into a boolean so it can be used as a test value.

Section 6. Looping and importing

Now look at two important aspects of using XSLT stylesheets: creating loops and importing external stylesheets.

Looping

XSLT can take a little getting used to in that it is a functional language rather than a procedural language. In other words, you typically don't explicitly control the way in which it carries out the instructions you give it. Exceptions do exist, however. You have, for example, the ability to perform both looping and conditional operations. Let's start with looping.

In the example so far, you have used XSLT's built in template propagation to apply styling to particular elements. In cases, this is fine. However, in certain situations in which you have complex XML files, or complex requirements, you might find it easier to explicitly apply information (see Listing 25).

Listing 25. Applying styles directly using looping

```
<xsl:template match="recipe">
  <h2><xsl:value-of select="./name"/></h2>
  <h3>Ingredients:</h3>
  <p>
```

```
        <xsl:for-each select="./ingredients/ingredient">
</xsl:text>    <xsl:value-of select="./qty"/><xsl:text>
</xsl:text>    <xsl:value-of select="./unit"/><xsl:text>
</xsl:text>    <xsl:value-of select="./food"/><br />
        </xsl:for-each>

</p>

<h3>Directions:</h3>
<ol>

        <xsl:for-each select="./instructions/instruction">
<li><xsl:value-of select="."/></li>
        </xsl:for-each>

</ol>

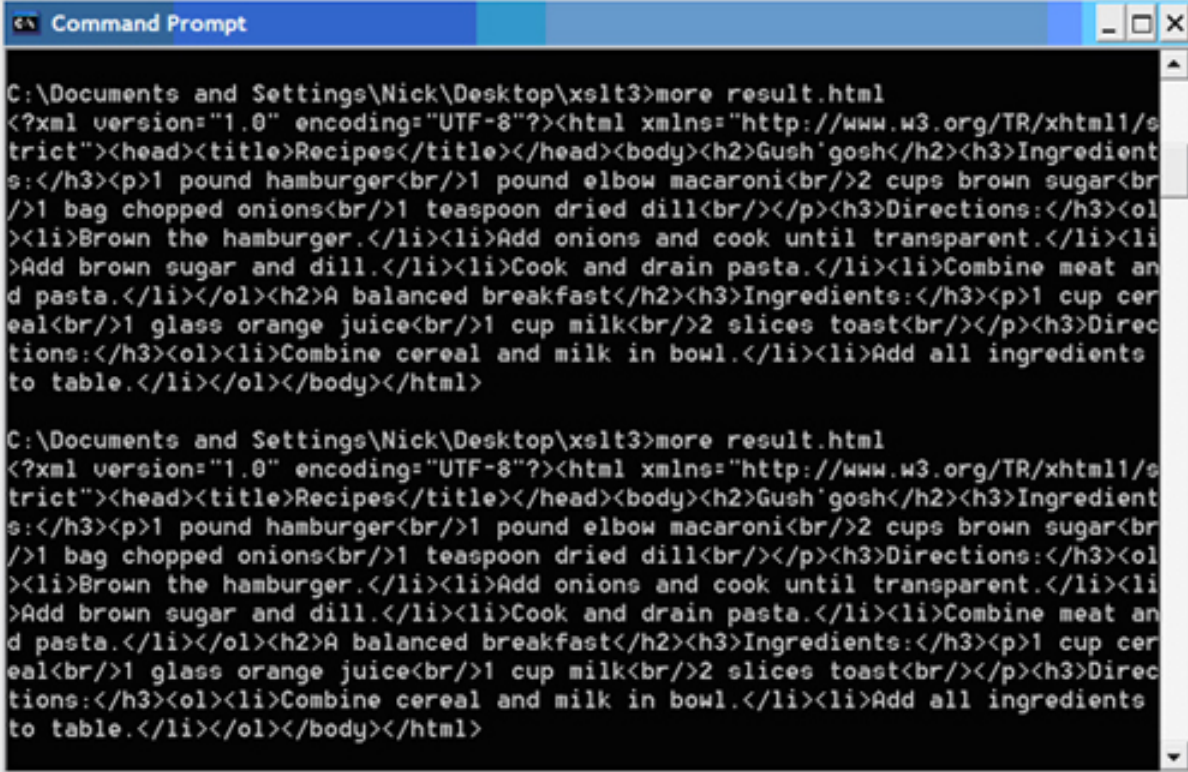
</xsl:template>

</xsl:stylesheet>
```

This looping construction is very similar to the `for-each` structure for which it is named. Like its namesake, each instance of the loop carries with it the next value in a list. In Java programming that might be the next value in an array; here, it is the next node in the set of nodes returned by the XPath statement in the `select` attribute. What that means is that the first time you execute the first loop, the context node is the first `ingredient` element. This enables you to select the `qty`, `unit`, and `food` children of that element and add them to the document just as was done before with the template. This is the same with the instructions, except that they're simply output directly.

The results are identical to those obtained by propagating through templates -- almost. Each template gets added to the document as an individual line, but since you are essentially processing this information as one template, you lose a lot of the spacing you saw before (see Figure 7).

Figure 7. The results



```

C:\Documents and Settings\Nick\Desktop\xslt3>more result.html
<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/strict"><head><title>Recipes</title></head><body><h2>Gush'gosh</h2><h3>Ingredients:</h3><p>1 pound hamburger<br/>1 pound elbow macaroni<br/>2 cups brown sugar<br/>1 bag chopped onions<br/>1 teaspoon dried dill<br/></p><h3>Directions:</h3><ol><li>Brown the hamburger.</li><li>Add onions and cook until transparent.</li><li>Add brown sugar and dill.</li><li>Cook and drain pasta.</li><li>Combine meat and pasta.</li></ol><h2>A balanced breakfast</h2><h3>Ingredients:</h3><p>1 cup cereal<br/>1 glass orange juice<br/>1 cup milk<br/>2 slices toast<br/></p><h3>Directions:</h3><ol><li>Combine cereal and milk in bowl.</li><li>Add all ingredients to table.</li></ol></body></html>

C:\Documents and Settings\Nick\Desktop\xslt3>more result.html
<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/strict"><head><title>Recipes</title></head><body><h2>Gush'gosh</h2><h3>Ingredients:</h3><p>1 pound hamburger<br/>1 pound elbow macaroni<br/>2 cups brown sugar<br/>1 bag chopped onions<br/>1 teaspoon dried dill<br/></p><h3>Directions:</h3><ol><li>Brown the hamburger.</li><li>Add onions and cook until transparent.</li><li>Add brown sugar and dill.</li><li>Cook and drain pasta.</li><li>Combine meat and pasta.</li></ol><h2>A balanced breakfast</h2><h3>Ingredients:</h3><p>1 cup cereal<br/>1 glass orange juice<br/>1 cup milk<br/>2 slices toast<br/></p><h3>Directions:</h3><ol><li>Combine cereal and milk in bowl.</li><li>Add all ingredients to table.</li></ol></body></html>

```

This matters in some applications of XML, but because you are using HTML, it really doesn't matter. But you will need to keep this in mind when deciding which method to use.

Including and importing stylesheets

Another variation on the stylesheet involves its structure. So far, all of your information has been in a single stylesheet. In some cases, however, you might want to break it out into individual pieces. This modularity can improve maintainability, as well as provide flexibility to use different stylesheets for different purposes. For example, you can create two individual stylesheets, one for the ingredients (see Listing 26).

Listing 26. The ingredients.xsl file

```

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:template match="ingredients/ingredient">

<xsl:value-of select="./qty"/><xsl:text> </xsl:text>
<xsl:value-of select="./unit"/><xsl:text> </xsl:text>
<xsl:value-of select="./food"/><br />

</xsl:template>

```

```
</xsl:stylesheet>
```

You can also create a stylesheet for the instructions (see Listing 27).

Listing 27. The instructions.xsl file

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:template match="instructions/instruction">
<li><xsl:value-of select="."/></li>
</xsl:template>
</xsl:stylesheet>
```

These templates are identical to the ones from the actual stylesheet. You can add them to the stylesheet by including them (see Listing 28).

Listing 28. Including the stylesheets

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">

    <xsl:include href="ingredients.xsl" />
    <xsl:include href="instructions.xsl" />

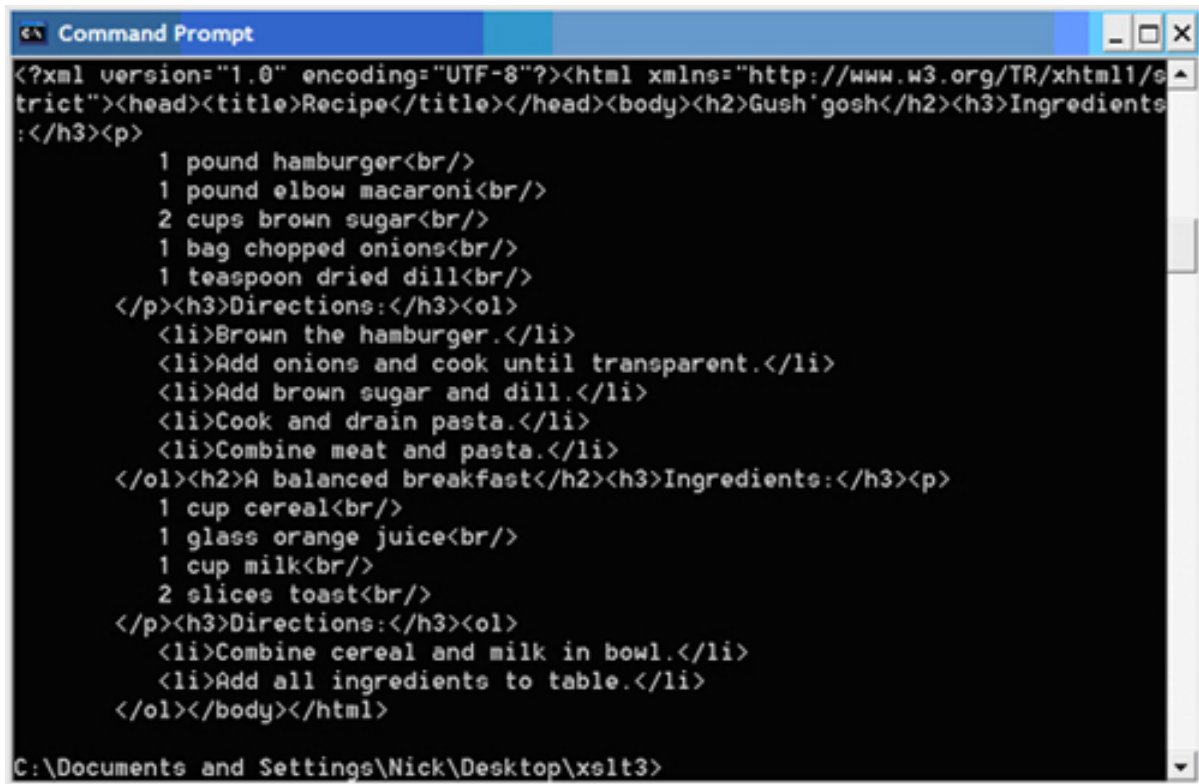
<xsl:template match="/">
...
</xsl:template>

<xsl:template match="recipe">

<h2><xsl:value-of select="./name" /></h2>
<h3>Ingredients:</h3>
<p><xsl:apply-templates select="./ingredients" /></p>
<h3>Directions:</h3>
<ol><xsl:apply-templates
select="./instructions" /></ol>

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

While you added the stylesheets to the same directory as the main stylesheet, you don't have to; the `href` attribute can contain any accessible URL. Notice that you send the processor in search of templates for the `ingredients` and `instructions` elements, for which none are seen in this file. If, however, you process the stylesheet, the results are precisely those you would have seen in the additional templates had they been included directly, rather than through the `include` element (see Figure 8).

Figure 8. The results


```

C:\Documents and Settings\Nick\Desktop\xslt3>
<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/strict"><head><title>Recipe</title></head><body><h2>Gush' gosh</h2><h3>Ingredients</h3><p>
  1 pound hamburger<br/>
  1 pound elbow macaroni<br/>
  2 cups brown sugar<br/>
  1 bag chopped onions<br/>
  1 teaspoon dried dill<br/>
</p><h3>Directions:</h3><ol>
  <li>Brown the hamburger.</li>
  <li>Add onions and cook until transparent.</li>
  <li>Add brown sugar and dill.</li>
  <li>Cook and drain pasta.</li>
  <li>Combine meat and pasta.</li>
</ol><h2>A balanced breakfast</h2><h3>Ingredients:</h3><p>
  1 cup cereal<br/>
  1 glass orange juice<br/>
  1 cup milk<br/>
  2 slices toast<br/>
</p><h3>Directions:</h3><ol>
  <li>Combine cereal and milk in bowl.</li>
  <li>Add all ingredients to table.</li>
</ol></body></html>

```

The include element provides the same effect as actually adding the content to the stylesheet at that point. Semantically, they are identical. On the other hand, you have a second option for including information: the import.

XSLT enables you to import a stylesheet at the top of the file. Why at the top? Because the purpose of importing a stylesheet is to give yourself the option to override any templates that are part of the import. For example, you can import the ingredients template and override it (see Listing 29).

Listing 29. Importing style sheets

```

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict">

  <xsl:import href="ingredients.xsl" />
  <xsl:import href="instructions.xsl" />

  <xsl:template match="/">
  ...
  </xsl:template>

  <xsl:template match="recipe">
  ...
  </xsl:template>

  <xsl:template match="ingredients">

```

```

        <ul><xsl:apply-templates select="./ingredient" /></ul>
    </xsl:template>

    <xsl:template match="ingredient">
        <li><xsl:value-of select="./qty"/><xsl:text>
</xsl:text>
        <xsl:value-of select="./unit"/>
        <xsl:text> </xsl:text><xsl:value-of
select="./food"/></li>

    </xsl:template>

</xsl:stylesheet>

```

Here you actually replaced a single template with two templates, which override those in the imported stylesheet (see Figure 9).

Figure 9. The results

```

C:\Documents and Settings\Nick\Desktop\xslt3>java org.apache.xalan.xslt.Process
-IN recipe.xml -XSL recipe.xsl -OUT result.html

C:\Documents and Settings\Nick\Desktop\xslt3>more result.html
<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/s
trict"><head><title>Recipe</title></head><body><h2>Gush'gosh</h2><h3>Ingredients
:</h3><p><ul><li>1 pound hamburger</li><li>1 pound elbow macaroni</li><li>2 cups
brown sugar</li><li>1 bag chopped onions</li><li>1 teaspoon dried dill</li></ul
></p><h3>Directions:</h3><ol>
    <li>Brown the hamburger.</li>
    <li>Add onions and cook until transparent.</li>
    <li>Add brown sugar and dill.</li>
    <li>Cook and drain pasta.</li>
    <li>Combine meat and pasta.</li>
</ol><h2>A balanced breakfast</h2><h3>Ingredients:</h3><p><ul><li>1 cup c
ereal</li><li>1 glass orange juice</li><li>1 cup milk</li><li>2 slices toast</li
></ul></p><h3>Directions:</h3><ol>
    <li>Combine cereal and milk in bowl.</li>
    <li>Add all ingredients to table.</li>
</ol></body></html>

C:\Documents and Settings\Nick\Desktop\xslt3>_

```

Note that because all of the residence rules, you could have achieved the same thing with an `include`. However, XSLT enables you to use the `priority` attribute to determine which template has greater weight when processing imports.

Extending XSLT

You have looked at ways to make XSLT more like programming. What about adding actual programming to it? Let's look at adding Java functionality to an XSLT stylesheet.

First note that while the extensions mechanism is part of the XSLT Recommendation, the actual implementation you're looking at here is specific to the Xalan XSLT processor. The concepts are likely similar for other processors, but you will have to check your documentation for specifics.

Next you'll create an extension that enables you to scale a recipe for more than one serving.

Extension elements

Extending XSLT requires different techniques. The first is the use of `extension` elements. An extension element is an element in a namespace specifically designated to correspond to a Java class. Consider, for example, this extension element (see Listing 30).

Listing 30. Using an extension element

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/xhtml1/strict"

  xmlns:scaler =
  "com.backstop.RecipeScaler"
  extension-element-prefixes="scaler">

  <xsl:template
    match="/" >
    <html>
    <head>
    <title>Recipes</title>
    </head>
    <body>

      <scaler:scaleMessage
      servings="2" />

    <xsl:apply-templates
      select="/recipes/recipe"/>
    </body>
    </html>
  </xsl:template>
  ...
```

You have created a namespace that corresponds to the `com.backstop.RecipeScaler` class, which includes a static method called `scaleMessage` (see Listing 31).

Listing 31. The RecipeScaler class

```
package
com.backstop;

public
class
RecipeScaler {

    public
    static String
    scaleMessage (
    org.apache.xalan.extensions.XSLProcessorContext
    context,
    org.w3c.dom.Element
    thisElement){

        return
        "This recipe has
        been scaled by a
        factor of " +
        thisElement.getAttribute("servings")
        + ".";

    }

}
```

When the processor gets to the element, it sees the `scaler:` namespace prefix and knows that it is designated as an extension element prefix, so it knows the class designated in a namespace definition. The method it calls responds to the local name of the element, `scaleMessage`. The method itself receives two arguments, one of which you actually use. The context parameter refers to the processor context, which enables you to look at elements on the `extension` element, but you are going to keep a simple line focusing on only the second argument, the `extension` element itself. Because you receive that element as a parameter to the method, you can extract the value of any attributes added to that element, such as `servings`, in this case. The text returned by the method gets added to the output in place of the `extension` element.

This means that if you process the stylesheet, you get results shown in Figure 10.

Figure 10. The results of the extension element

```

C:\Documents and Settings\Nick\Desktop\xslt3>
<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/s
trict"><head><title>Recipes</title></head><body>This recipe has been scaled by a
factor of 2.<h2>Gush'gosh</h2><h3>Ingredients:</h3><p>
  1 pound hamburger<br/>
  1 pound elbow macaroni<br/>
  2 cups brown sugar<br/>
  1 bag chopped onions<br/>
  1 teaspoon dried dill<br/>
</p><h3>Directions:</h3><ol>
  <li>Brown the hamburger.</li>
  <li>Add onions and cook until transparent.</li>
  <li>Add brown sugar and dill.</li>
  <li>Cook and drain pasta.</li>
  <li>Combine meat and pasta.</li>
</ol><h2>A balanced breakfast</h2><h3>Ingredients:</h3><p>
  1 cup cereal<br/>
  1 glass orange juice<br/>
  1 cup milk<br/>
  2 slices toast<br/>
</p><h3>Directions:</h3><ol>
  <li>Combine cereal and milk in bowl.</li>
  <li>Add all ingredients to table.</li>
</ol></body></html>

```

Extension elements can be quite useful, if a little bit difficult to use.

Extension functions

Another way to add functionality through the stylesheet is to use extension functions, which are a bit simpler to implement than extension elements. For example, you can create a function that multiplies the quantity of ingredient and the number of servings (see Listing 32).

Listing 32. The scaleIngredient() method

```

package
com.backstop;

public
class
RecipeScaler {

    public
    static String
    scaleMessage (
    org.apache.xalan.extensions.XSLProcessorContext
    context,
    org.w3c.dom.Element
    thisElement){

        return
        "This recipe has
        been scaled by a
        factor of " +

```

```
thisElement.getAttribute("servings")
+ ".";
    }

    public
    static int
    scaleIngredient(int
    servings, int
    original){

        return
        (servings *
        original);
    }
}
```

Adding this function to the stylesheet is similar to adding an extension element, in that you already have a namespace map to the class (see Listing 33).

Listing 33. Adding the extension function

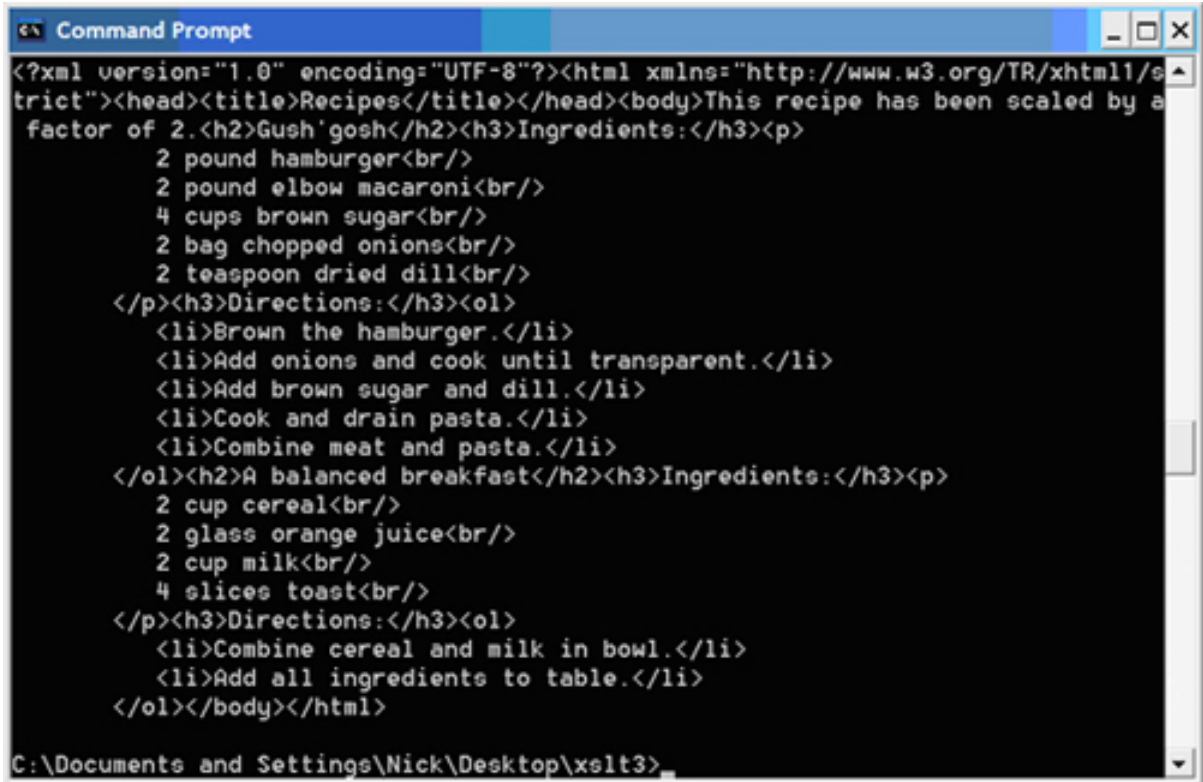
```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict"
xmlns:scaler = "com.backstop.RecipeScaler"
extension-element-prefixes="scaler">
...
<xsl:template match="ingredients/ingredient">

<xsl:value-of select="scaler:scaleIngredient(2, ./qty)"/>
<xsl:text> </xsl:text><xsl:value-of select="./unit"/>
<xsl:text> </xsl:text><xsl:value-of select="./food"/><br
/>

</xsl:template>
...
```

Notice that the function call includes the namespace prefix. As before, the processor sees the prefix and knows it needs to execute a call to the `RecipeScaler` class. The results are that the ingredients are multiplied by two (see Figure 11).

Figure 11. Using the extension function



```

<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/strict"><head><title>Recipes</title></head><body>This recipe has been scaled by a factor of 2.<h2>Gush'gosh</h2><h3>Ingredients:</h3><p>
  2 pound hamburger<br/>
  2 pound elbow macaroni<br/>
  4 cups brown sugar<br/>
  2 bag chopped onions<br/>
  2 teaspoon dried dill<br/>
</p><h3>Directions:</h3><ol>
  <li>Brown the hamburger.</li>
  <li>Add onions and cook until transparent.</li>
  <li>Add brown sugar and dill.</li>
  <li>Cook and drain pasta.</li>
  <li>Combine meat and pasta.</li>
</ol><h2>A balanced breakfast</h2><h3>Ingredients:</h3><p>
  2 cup cereal<br/>
  2 glass orange juice<br/>
  2 cup milk<br/>
  4 slices toast<br/>
</p><h3>Directions:</h3><ol>
  <li>Combine cereal and milk in bowl.</li>
  <li>Add all ingredients to table.</li>
</ol></body></html>

```

But while this works, it's not very maintainable. Now look at a way to make that easier.

Section 7. Programming XSLT

Before I wrap things up, you need to know about two aspects of XSLT that give you some of the capabilities you expect from a regular program language.

XSLT variables

It's nice that you have a way to execute a function, but you wound up with a constant buried in the midst of the stylesheet. Wouldn't it be better if you could set a variable at the top of the page? Of course it would (see Listing 34).

Listing 34. Setting a variable

```

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/xhtml1/strict"

```

```
xmlns:scaler = "com.backstop.RecipeScaler"
extension-element-prefixes="scaler">

<xsl:variable name="numberOfServings" select="3" />

<xsl:template match="/">
<html>
<head>
<title>Recipes</title>
</head>
<body>

<scaler:scaleMessage servings="$numberOfServings" />

<xsl:apply-templates select="/recipes/recipe"/>
</body>
</html>
</xsl:template>

<xsl:template match="recipe">
...
</xsl:template>

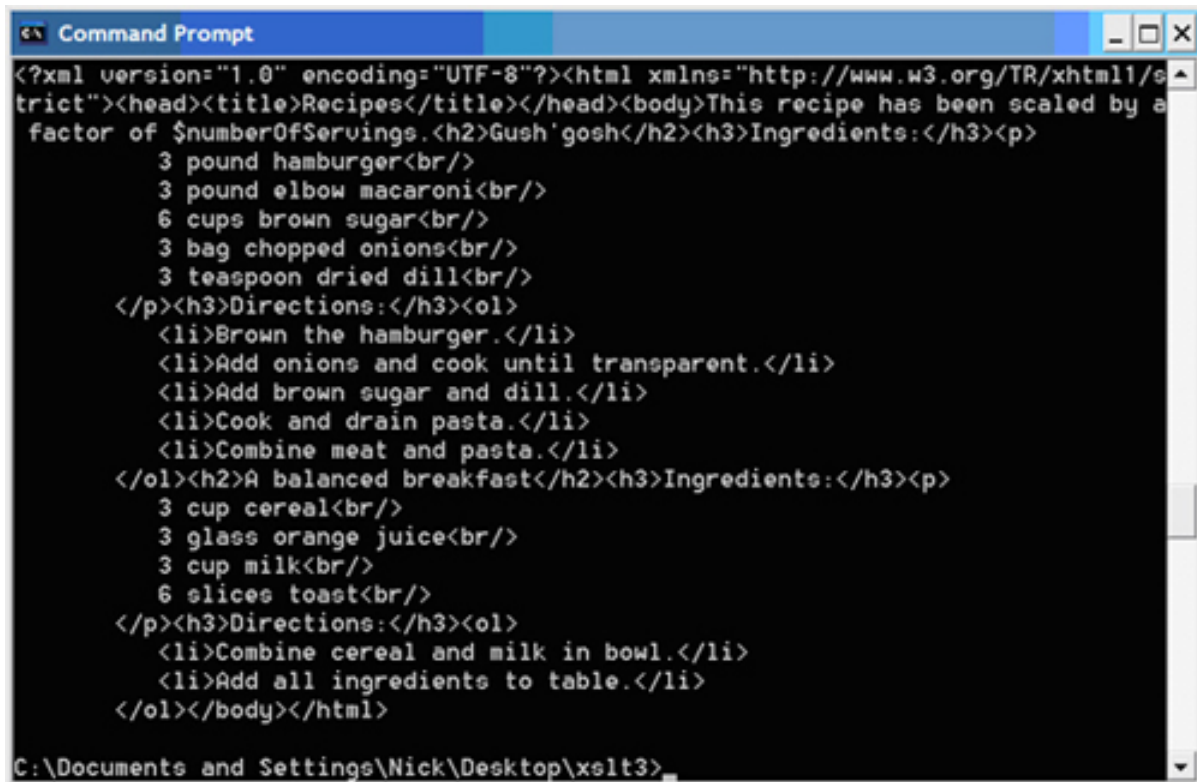
<xsl:template match="ingredients/ingredient">

<xsl:value-of
select="scaler:scaleIngredient($numberOfServings,
./qty)"/>
<xsl:text> </xsl:text><xsl:value-of select="./unit"/>
<xsl:text> </xsl:text><xsl:value-of select="./food"/><br
/>

</xsl:template>
...
```

XSLT enables you to create a variable, and that reference uses the "dollar sign" (\$) notation you see in the listing. If you process the stylesheet, you'll see two effects (see Figure 12).

Figure 12. Using a variable



```

<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/strict"><head><title>Recipes</title></head><body>This recipe has been scaled by a factor of $numberOfServings.<h2>Gush'gosh</h2><h3>Ingredients:</h3><p>
  3 pound hamburger<br/>
  3 pound elbow macaroni<br/>
  6 cups brown sugar<br/>
  3 bag chopped onions<br/>
  3 teaspoon dried dill<br/>
</p><h3>Directions:</h3><ol>
  <li>Brown the hamburger.</li>
  <li>Add onions and cook until transparent.</li>
  <li>Add brown sugar and dill.</li>
  <li>Cook and drain pasta.</li>
  <li>Combine meat and pasta.</li>
</ol><h2>A balanced breakfast</h2><h3>Ingredients:</h3><p>
  3 cup cereal<br/>
  3 glass orange juice<br/>
  3 cup milk<br/>
  6 slices toast<br/>
</p><h3>Directions:</h3><ol>
  <li>Combine cereal and milk in bowl.</li>
  <li>Add all ingredients to table.</li>
</ol></body></html>

```

Notice that the ingredients were multiplied by the number of servings, as expected. However, if you look more carefully, you will see that the extension element did not process properly, taking a variable as a string rather than as the value of the variable itself. This is not a bug; the specification does not require the processor to do anything at all to attribute values before processing the extension element. So you'll have to find another way around this problem.

Conditional processing

The first thing that you can do is to use conditional processing so that you only display the message if it's needed in the first place. For example, see Listing 35.

Listing 35. Using an if element

```

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/xhtml1/strict"
  xmlns:scaler = "com.backstop.RecipeScaler"
  extension-element-prefixes="scaler">

  <xsl:variable name="numberOfServings" select="1" />

  <xsl:template match="/">

  <html>
  <head>

```

```

<title>Recipes</title>
</head>
<body>

<xsl:if test="$numberOfServings > 1">

<scaler:scaleMessage servings="3" />

</xsl:if>

<xsl:apply-templates select="/recipes/recipe"/>
</body>
</html>
</xsl:template>
...

```

The contents of the `if` element specified by the `test` attribute evaluates to true. If not, as in this case, the output does not appear at all (see Figure 13).

Figure 13. The results of the if statement

```

C:\Documents and Settings\Nick\Desktop\xslt3>java org.apache.xalan.xslt.Process
-IN recipe.xml -XSL recipe.xsl -OUT result.html

C:\Documents and Settings\Nick\Desktop\xslt3>more result.html
<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/s
trict"><head><title>Recipes</title></head><body><h2>Gush'gosh</h2><h3>Ingredient
s:</h3><p>1 pound hamburger<br/>1 pound elbow macaroni<br/>2 cups brown sugar<br
/>1 bag chopped onions<br/>1 teaspoon dried dill<br/></p><h3>Directions:</h3><ol
><li>Brown the hamburger.</li><li>Add onions and cook until transparent.</li><li>
Add brown sugar and dill.</li><li>Cook and drain pasta.</li><li>Combine meat an
d pasta.</li></ol><h2>A balanced breakfast</h2><h3>Ingredients:</h3><p>1 cup cer
eal<br/>1 glass orange juice<br/>1 cup milk<br/>2 slices toast<br/></p><h3>Direc
tions:</h3><ol><li>Combine cereal and milk in bowl.</li><li>Add all ingredients
to table.</li></ol></body></html>

C:\Documents and Settings\Nick\Desktop\xslt3>

```

The way it is written, it doesn't make much sense; if the value is greater than one, the extension element will display with a value of "3." You are better off with a variety of choices (see Listing 36).

Listing 36. The choose element

```

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

```

```
xmlns="http://www.w3.org/TR/xhtml1/strict"
xmlns:scaler = "com.backstop.RecipeScaler"
extension-element-prefixes="scaler">

<xsl:variable name="numberOfServings" select="3" />

<xsl:template match="/">

<html>
<head>
<title>Recipes</title>
</head>
<body>

    <xsl:choose>
    <xsl:when test="$numberOfServings < 1">
    Recipes have been scaled by an invalid number.
    </xsl:when>
    <xsl:when test="$numberOfServings = 1">
    </xsl:when>
    <xsl:when test="$numberOfServings = 2">
    <scaler:scaleMessage servings="2" />
    </xsl:when>
    <xsl:when test="$numberOfServings = 3">

    <scaler:scaleMessage servings="3" />

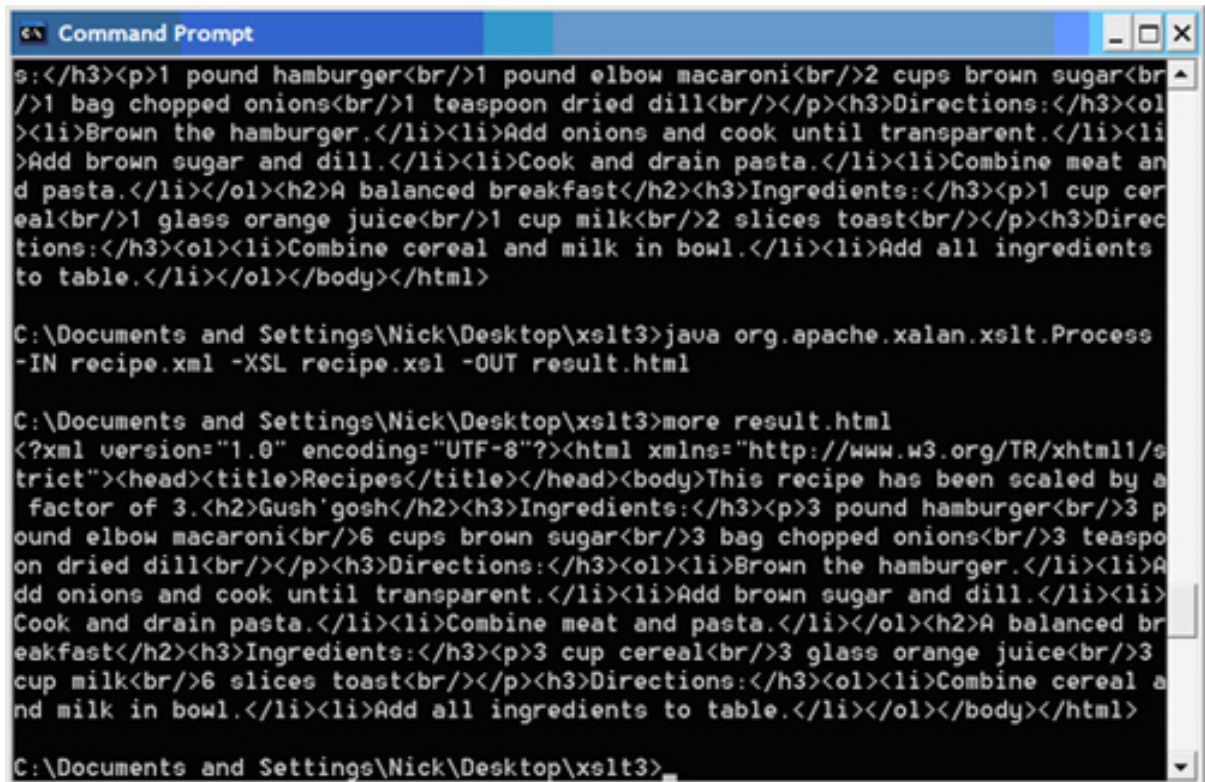
    </xsl:when>
    <xsl:otherwise>
    Recipes have been scaled for multiple portions.
    </xsl:otherwise>
    </xsl:choose>

    <xsl:apply-templates select="/recipes/recipe"/>
    </body>
</html>
</xsl:template>
...
```

In this case, you have a combination of the `if-then-else` and the `case` statement from traditional programming languages. The `choose` element acts as a container, but the `when` element displays its contents only if its `test` attribute evaluates to true. Finally, if none of the `when` elements evaluate to true, the processor displays the contents of the `otherwise` element.

The result is as you might expect (see Figure 14).

Figure 14. The result



```

C:\Documents and Settings\Nick\Desktop\xslt3>java org.apache.xalan.xslt.Process
-IN recipe.xml -XSL recipe.xsl -OUT result.html

C:\Documents and Settings\Nick\Desktop\xslt3>more result.html
<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/TR/xhtml1/s
trict"><head><title>Recipes</title></head><body>This recipe has been scaled by a
factor of 3.<h2>Gush'gosh</h2><h3>Ingredients:</h3><p>3 pound hamburger<br/>3 p
ound elbow macaroni<br/>6 cups brown sugar<br/>3 bag chopped onions<br/>3 teasp
on dried dill<br/></p><h3>Directions:</h3><ol><li>Brown the hamburger.</li><li>A
dd onions and cook until transparent.</li><li>Add brown sugar and dill.</li><li>
Cook and drain pasta.</li><li>Combine meat and pasta.</li></ol><h2>A balanced br
eakfast</h2><h3>Ingredients:</h3><p>3 cup cereal<br/>3 glass orange juice<br/>3
cup milk<br/>6 slices toast<br/></p><h3>Directions:</h3><ol><li>Combine cereal a
nd milk in bowl.</li><li>Add all ingredients to table.</li></ol></body></html>

C:\Documents and Settings\Nick\Desktop\xslt3>_

```

And now you have the flexibility to specifically display more quantities, or to adapt for other cases.

Section 8. Summary

Wrap up

This tutorial took you from minimal knowledge about XSL Transformations to the creation of fairly complex stylesheets. First you learned about the basics of stylesheets, and then about XPath expressions, one of the foundations of XSLT. The last part of this tutorial examined some of the more complicated aspects of using XSLT stylesheets, that is, their variables, conditional processing, and extensions. As a result, you should have enough knowledge to do virtually anything with XSLT stylesheets -- or at least to know what you need to find out if you get stuck.

Resources

Learn

- [XSLT 1.0 recommendation](#): Read the spec maintained by the World Wide Web Consortium.
- [XPath 1.0 recommendation](#): Read the current version or look ahead to the [XPath 2.0 recommendation](#).
- [An XSLT style sheet and an XML dictionary approach to internationalization](#) (Laura Menke, developerWorks, April 2001): Get an example of how XSLT can be useful in real-world problems.
- [How an XSLT processor works](#) (Benoît Marchal, developerWorks, March 2004): Learn more of the theory behind an XSLT processor.
- [Planning to upgrade XSLT 1.0 to 2.0, Part 1: Improvements in XSLT](#) (David Marston and Joanne Tong, developerWorks, October 2006): Compare XSLT 1.0, which is the version currently in use, with what is coming in XSLT 2.0.
- [Using XPath with the IBM Workplace data access tool](#) (Mark Wallace, developerWorks, April 2005): Create a database application (including a form, data definition, and grid) using XML Path Language (XPath) expressions with the IBM data access tool.
- [Practical data binding: XPath as data binding tool, Part 1](#) (Brett McLaughlin, developerWorks, November 2005): Learn to select portions of an XML document with XPath.
- [The Java XPath API](#) (Elliott Rusty Harold, developerWorks, July 2006): Learn about querying XML from Java programs.
- [Working XML: Get started with XPath 2.0](#) (Benoît Marchal, developerWorks, May 2006): Get a good look at what's coming in the new data model with increased power and efficiency of the language.
- [Simple Xalan extension functions](#) (Elliott Rusty Harold, developerWorks, November 2006): Get another look at Simple Xalan extension functions in Java code.
- [Tip: Call JavaScript from an XSLT style sheet](#) (Nicholas Chase, developerWorks, April 2002). Add functionality to your style sheets when you call JavaScript from an XSLT style sheet, rather than Java code.
- [IBM XML certification](#): Find out how you can become an IBM-Certified Developer in XML and related technologies.
- [XML technical library](#): See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.

- [developerWorks technical events and webcasts](#): Stay current with technology in these sessions.
- [developerWorks XML zone](#): Explore hundreds of articles and tutorials about XML.
- The [technology bookstore](#): Browse for books on these and other technical topics.

Get products and technologies

- [Apache Xalan](#): Download Xalan and add the included JAR files to your classpath to follow the tutorial examples.
- [IBM trial software](#): Build your next development project with trial software available for download directly from developerWorks.

Discuss

- [XML zone discussion forums](#): Participate in any of several XML-centered forums.
- [developerWorks blogs](#): Get involved in the developerWorks community.

About the author

Nicholas Chase

Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the Chief Technology Officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams).