

4

Templates

In the last chapter, you saw how to take a well-formed HTML document and turn it into a stylesheet by adding the XSLT elements `<xsl:value-of>` and `<xsl:for-each>` to pick out information from a source XML document and produce an HTML result. The stylesheets that we looked at were **simplified stylesheets**. Simplified stylesheets are good as a starting point when you're creating a stylesheet, and they can be all you need in some cases. However, to utilize the more sophisticated functionality of XSLT, you need to use full stylesheets.

In this chapter, we'll take the simplified stylesheet that we developed during the last chapter and turn it into a full stylesheet. I'll also introduce you to **templates** as a way of breaking up your code and look in a bit more detail at how XSLT processors construct a result from some source XML. You'll learn:

- ❑ What full stylesheets look like
- ❑ How the XSLT processor navigates the source document to create a result
- ❑ How to break up your code into separate templates
- ❑ How templates help with document-oriented and unpredictable XML
- ❑ How to create tables of contents in your pages using template modes

XSLT Stylesheet Structure

The simplified stylesheets that we used in the last chapter are a specialized form of stylesheet that make a good starting point when we're creating an XSLT stylesheet. Simplified stylesheets aren't all that common in larger applications because they're fairly restricted in what they can do, especially with document-oriented XML.

Technically, simplified stylesheets are defined in the XSLT Recommendation in terms of how they map on to full stylesheets. In the last chapter, we developed the following simplified stylesheet (`HelloWorld.xsl`) to take the Hello World XML document (`HelloWorld.xml`) and convert it to HTML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="1.0">
  <head><title>Hello World Example</title></head>
  <body>
    <p>
      <xsl:value-of select="/greeting" />
    </p>
  </body>
</html>
```

The equivalent full stylesheet for the simplified stylesheet looks very similar. The content of the simplified stylesheet is wrapped in two elements – `<xsl:template>` and `<xsl:stylesheet>` – to create `HelloWorld2.xsl`. The `<xsl:stylesheet>` element takes the `version` attribute and the XSLT namespace declaration instead of the `<html>` element, giving the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head><title>Hello World Example</title></head>
      <body>
        <p>
          <xsl:value-of select="/greeting" />
        </p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

In the next couple of sections, we'll look at what these new XSLT elements do.

Stylesheet Document Elements

The document element of a full stylesheet is `<xsl:stylesheet>` (a `<stylesheet>` element in the namespace `http://www.w3.org/1999/XSL/Transform` – as usual I'm using the prefix `xsl` here but you could use whatever you liked as long as it's associated with the XSLT namespace with a namespace declaration). Like the document element in simplified stylesheets, the `<xsl:stylesheet>` element needs to declare the XSLT namespace and give the version of XSLT that's used in the stylesheet with a `version` attribute. This time, though, the `version` attribute doesn't need to be qualified with the `xsl` prefix because it already lives on an element in the XSLT namespace, so the processor knows it's part of XSLT.

You can also use `<xsl:transform>` as the document element in a full stylesheet, rather than `<xsl:stylesheet>`. There is no difference in functionality between the two document elements – they each use exactly the same attributes and do exactly the same thing. Some people prefer to use `<xsl:transform>` when doing transformations that aren't producing presentation-oriented formats such as XSL-FO or XHTML. Personally, I use `<xsl:stylesheet>` all the time.

Defining Templates

Inside the `<xsl:stylesheet>` document element, XSLT stylesheets are made up of a number of templates, each of which matches a particular part of the source XML document and processes it whatever way you define. The templates are rules that define how a particular part of the source XML document maps on to the result that you want. Thus a full stylesheet has a structure that's quite similar to the structure of CSS stylesheets – a set of rules that match different elements and describe how they should be presented.

There are some very fundamental differences between CSS stylesheets and XSLT stylesheets, though. First, while CSS always processes all the elements in a document, you can use XSLT to pick and choose which elements to display. Second, while multiple rules can be applied to style a particular element in CSS, only one template can be applied at a time in XSLT. Third, XSLT templates can match a lot of things that CSS templates can't, such as attributes and comments.

Templates are defined using the `<xsl:template>` element. The `match` attribute on `<xsl:template>` indicates which parts of the source document should be processed with the particular template and the content of the `<xsl:template>` element dictates what is done with that particular part of the source document. You can use literal result elements, `<xsl:value-of>`, and `<xsl:for-each>` inside a template in exactly the same way as you do within a simplified stylesheet to generate some output.

A full XSLT stylesheet has an `<xsl:stylesheet>` document element, which contains a number of `<xsl:template>` elements, each of which defines the processing that should be carried out on a particular part of the source XML.

Try It Out – Converting a Simplified Stylesheet to a Full Stylesheet

In this section, we'll convert the simplified stylesheet `TVGuide.xml` that we created in the last chapter into a full stylesheet and test that the full stylesheet gives exactly the same result for `TVGuide.xml` as the simplified stylesheet did.

The simplified stylesheet `TVGuide.xml` looks as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="1.0">
<head>
  <title>TV Guide</title>
  <link rel="stylesheet" href="TVGuide.css" />
  <script type="text/javascript">
    function toggle(element) {
      if (element.style.display == 'none') {
        element.style.display = 'block';
      } else {
        element.style.display = 'none';
      }
    }
  </script>
```

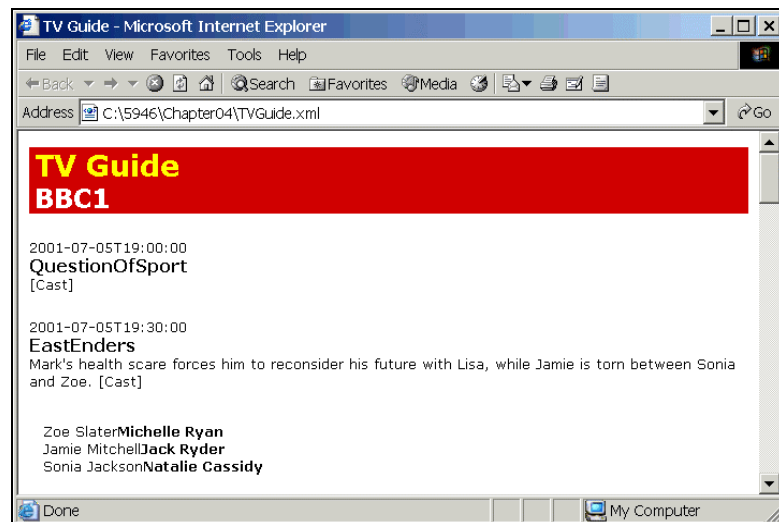
```

</head>

<body>
  <h1>TV Guide</h1>
  <xsl:for-each select="/TVGuide/Channel">
    <h2 class="channel"><xsl:value-of select="Name" /></h2>
    <xsl:for-each select="Program">
      <div>
        <p>
          <span class="date"><xsl:value-of select="Start" /></span><br />
          <span class="title"><xsl:value-of select="Series" /></span><br />
          <xsl:value-of select="Description" />
          <span onclick="toggle({Series}Cast);">[Cast]</span>
        </p>
        <div id="{Series}Cast" style="display: none;">
          <ul class="castlist">
            <xsl:for-each select="CastList/CastMember">
              <li>
                <span class="character">
                  <xsl:value-of select="Character" />
                </span>
                <span class="actor">
                  <xsl:value-of select="Actor" />
                </span>
              </li>
            </xsl:for-each>
          </ul>
        </div>
      </div>
    </xsl:for-each>
  </xsl:for-each>
</body>
</html>

```

When you use this stylesheet with `TVGuide.xml`, you get the following display:



To create a full stylesheet from this simplified stylesheet, you need to do the following:

- ☐ Add an `<xsl:template>` element whose `match` attribute has the value `/` around the `<html>` element
- ☐ Add an `<xsl:stylesheet>` element around the new `<xsl:template>` element
- ☐ Move the XSLT namespace declaration from the `<html>` element to the `<xsl:stylesheet>` element
- ☐ Remove the `xsl:version` attribute from the `<html>` element and add an equivalent `version` attribute on the `<xsl:stylesheet>` element

The result of these four steps is `TVGuide2.xsl`, with the following outline:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

</xsl:stylesheet>
```

Now run the transformation with `TVGuide.xml`, this time with the full stylesheet `TVGuide2.xsl`. Do this by amending the `xml-stylesheet` processing instruction in `TVGuide.xml` as follows:

```
<?xml-stylesheet type="text/xsl" href="TVGuide2.xsl"?>
```

You should see exactly the same result as you had before.

The Node Tree

Before we start looking at templates in detail, we first need to look at an XML document in the way that an XSLT processor does. When an XSLT processor reads in a document, it generates a representation of the XML as a **node tree**. As you might expect from its name, the node tree is a bunch of **nodes** arranged in a tree. Nodes are a general term for the components of an XML document, such as:

- ☐ element nodes
- ☐ attribute nodes
- ☐ text nodes
- ☐ comment nodes
- ☐ processing instruction nodes

The nodes are arranged in a tree such that the tree forms a new branch for every node contained in an element. The relationships between the nodes in the tree are described in terms of familial relationships, so the nodes that an element contains are called its **children** and an element node is its children's **parent**. Similarly, all the children of an element node are **siblings**, and you can also talk about the **descendents** of an element node or a node's **ancestors**.

At the very top of the node tree is the **root node** (for some reason node trees grow down rather than up). You can think of the root node as being equivalent to the XML document itself. The root node's children are the document element and any comments or processing instructions that live outside the document element.

Attribute nodes are a bit special because attributes are not contained in elements in the same way as other elements or text, but they are still associated with particular elements. The element that an attribute is associated with is still known as its parent, but attributes are not their parent element's children, just its attributes.

Note that comments and processing instructions are nodes and part of the node tree, so you need to take them into account if you count nodes or iterate over them. As we'll see in Chapter 7, text nodes that consist purely of whitespace might also be part of the node tree, but you have some control over which are and which aren't.

The view of XML as a node tree is a very natural view because of the way that XML is structured, with elements nesting inside each other. In XML, the relationship between an element and its contents is a one-to-many relationship – each element can only have one parent – which fits the pattern of a tree structure. Processing XML as a tree of nodes is also useful because it means you can focus down on a particular branch of the tree (the content of a particular element) very easily. Other models of XML documents, such as the Document Object Model (DOM) and the XML Infoset, also view XML documents as tree structures, although the models are just slightly different from the node tree that XSLT uses.

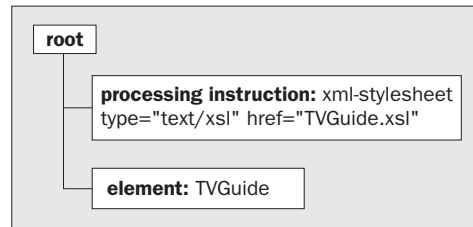
You can find out more about the DOM at <http://www.w3.org/DOM/Activity.html> and more about the XML Infoset at <http://www.w3.org/TR/xml-infoset/>.

XSLT processors treat documents as a node tree in which the contents of an element are represented as its children. Every node in a node tree descends from the root node.

Having a picture of the node tree can be very useful because it lets you view the XML document in the same way as the XSLT processor does. Here's a simplified version of the XML that we're using to hold the information in our TV guide:

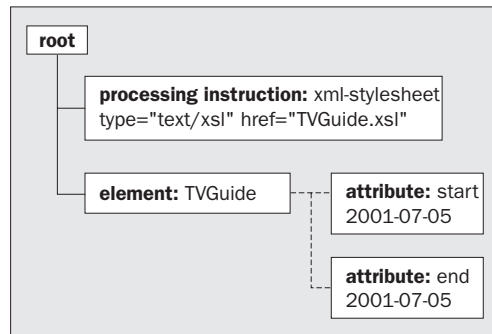
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="TVGuide.xsl"?>
<TVGuide start="2001-07-05" end="2001-07-05">
  <Channel>
    <Name>BBC1</Name>
    ...
    <Program>
      <Start>2001-07-05T19:30:00</Start>
      <Duration>PT30M</Duration>
      <Series>EastEnders</Series>
      ...
    </Program>
    ...
  </Channel>
  ...
</TVGuide>
```

Every node tree starts with the root node, so that's the starting point for our diagram. The root node of the tree is like the document itself. The XML document has two nodes at the top level, the `xml-stylesheet` processing instruction and the document element – the `<TVGuide>` element. The document element is the top-most element in the node tree, but other things (like comments and processing instructions) can occur at the same level. We can also draw the children of the root node in, as follows:

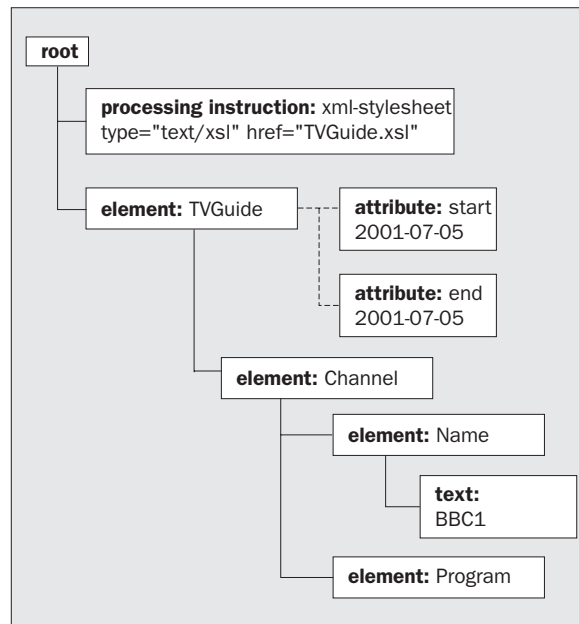


An XSLT processor doesn't see the XML declaration (the first line of an XML file). The information held in the XML declaration relates to how the XML document has been stored, which the XSLT processor doesn't care about. Also, note that the pseudo-attributes in the `xml-stylesheet` processing instruction aren't nodes (unlike proper attributes), they're just part of the value of the processing instruction.

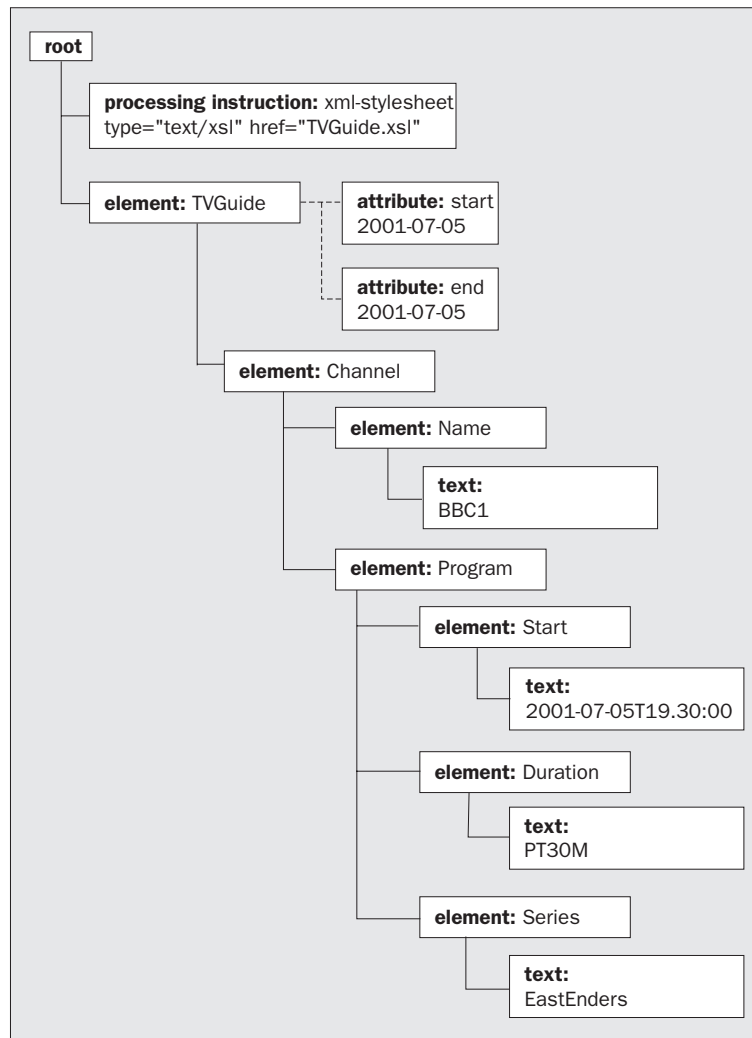
Now, the `<TVGuide>` element has a couple of attributes: `start` and `end`. These attributes shouldn't be added to the tree in the same way as the children of the `<TVGuide>` would be, so we'll place them off to one side and use a different kind of line for them, as follows:



Now let's look at the content of the `<TVGuide>` element. The `<TVGuide>` element contains a `<Channel>` element, which in turn contains a `<Name>` element and a `<Program>` element. The `<Name>` element contains some text. Pieces of text are represented as separate nodes in the node tree, so the `<Name>` element node contains a text node, as shown here:



The child elements of the `<Program>` element are treated in the same way – they each have a single text node as a child:



The rest of the tree follows a similar pattern: element nodes having either other elements or text nodes as children. It's often very useful to keep a picture of the node tree of the source document that you're working with close at hand, to help you work out what nodes you're selecting and processing.

Many XSLT editors help with this by providing a simple tree view on a source document. Alternatively, you can use a stand-alone tool such as Mike Brown's Pretty XML Tree Viewer (<http://skew.org/xml/stylesheets/treeview/html/>).

XSLT Processing Model

Templates in stylesheets each match particular nodes in the node tree. The `match` attribute on a template tells the XSLT processor what kind of nodes they match. In a full stylesheet, you tell the XSLT processor what nodes you want to be processed (using an `<xsl:apply-templates>` instruction, as you'll see later) and the XSLT processor goes through the nodes you've selected one by one trying to find a matching template for each in turn. When it looks for a template, it searches the whole stylesheet, so it doesn't matter where the template is within that stylesheet.

When the XSLT processor finds a matching template, it uses the content of that template to process the matching node and generate some output. The content of the template might include instructions that tell the processor to apply templates to a particular set of nodes, in which case it goes through those nodes finding and processing matching templates, and so on.

In this section, we'll look at the implications of this processing model and the kinds of changes that we can make to our stylesheet to take advantage of templates.

XSLT processing involves telling the processor to apply templates to some nodes in the source document. The XSLT processor locates a template that matches the node, and processes its content to generate a result. The location of the template within the stylesheet doesn't matter.

The Starting Template

This process of applying templates to nodes in the node tree has to start somewhere. In the majority of cases, the input to the stylesheet is an XML document, and the processor starts at the top of the node tree, on the root node. After building the node tree, the XSLT processor takes the root node and tries to find a template that matches it. If the XSLT processor finds one, it processes the content of that template to generate the output.

If you're running a stylesheet from code, then you can make the stylesheet start from a node other than the root node if you want. This is useful if you want to only process a section of a larger document, for example. In these cases, you need templates that match the nodes that you use as the source of the transformation – a template that matches the root node will only be used if you tell the processor to process the root node.

A template that matches the root node has a `match` attribute with a value of `/`, one that looks like:

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

If you're familiar with programming, you can think of this template as analogous to the `main()` method on a class. Whatever happens, when you process a document with the stylesheet, the XSLT processor will process the contents of this template, so it gives you top-level control over what the stylesheet does.

If you look back at the full stylesheet that you created based on the simplified stylesheet from the last chapter, you'll see that it contains only one template, which matches the root node. The stylesheet works because the XSLT processor always activates that template.

A template that matches the root node acts as high-level control over the result of the stylesheet.

Matching Elements with Templates

At the moment, we use `<xsl:for-each>` to tell the XSLT processor to go through the `<Channel>` elements one by one. That's one place where we could use templates instead. In this section, we'll look at how we can replace this `<xsl:for-each>` with a separate template and an `<xsl:apply-templates>` instruction.

First, we need a template that tells the XSLT processor what to do when it's told to process a `<Channel>` element. You can match an element with a template by giving the name of the element in the `match` attribute of the `<xsl:template>` element. So the following template will match `<Channel>` elements and process them to generate the same result as is currently generated inside `<xsl:for-each>`:

```
<xsl:template match="Channel">
  <h2 class="channel"><xsl:value-of select="Name" /></h2>
  <xsl:for-each select="Program">
    <div>
      <p>
        <span class="date"><xsl:value-of select="Start" /></span><br />
        <span class="title"><xsl:value-of select="Series" /></span><br />
        <xsl:value-of select="Description" />
        <span onclick="toggle({Series}Cast);">[Cast]</span>
      </p>
      <div id="{Series}Cast" style="display: none;">
        <ul class="castlist">
          <xsl:for-each select="CastList/CastMember">
            <li>
              <span class="character">
                <xsl:value-of select="Character" />
              </span>
              <span class="actor">
                <xsl:value-of select="Actor" />
              </span>
            </li>
          </xsl:for-each>
        </ul>
      </div>
    </div>
  </xsl:for-each>
</xsl:template>
```

If a template's `match` attribute gives the name of an element, the XSLT processor will use that template for elements with that name.

You can put this template wherever you like at the top level of the stylesheet (at the same level as the other `<xsl:template>` elements). The XSLT processor will find it and use it whenever it needs to process a `<Channel>` element. In `TVGuide3.xsl`, I've put this template after the template that matches the root node, so the top level of the stylesheet looks like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

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

</xsl:stylesheet>
```

I usually order the templates in my stylesheets starting from the template matching the root node and working down the levels of the node tree. But you are free to arrange your templates however you like.

However, the XSLT processor will never use this template unless you tell the processor to process (apply templates to) a `<Channel>` element. You instruct the XSLT processor to apply templates to some nodes using the `<xsl:apply-templates>` instruction. Like `<xsl:for-each>`, the result of the `<xsl:apply-templates>` instruction gets inserted into the result of the transformation at the point where you use the instruction, so while the location of `<xsl:template>` doesn't matter, the positioning of `<xsl:apply-templates>` is very important.

The `<xsl:apply-templates>` instruction has a `select` attribute, which tells the XSLT processor which nodes to apply templates to. The `select` attribute on `<xsl:apply-templates>` works in the same way as the `select` attribute on `<xsl:for-each>` – you give a path that points to the nodes to which you want to apply templates.

We want the result to appear at the same place as the result of the original `<xsl:for-each>`, and we already know the path to those nodes because we're already using it on the `<xsl:for-each>`. So you can apply templates instead by replacing the `<xsl:for-each>` with an `<xsl:apply-templates>` element that has exactly the same `select` attribute. Doing this in `TVGuide4.xsl` gives a template matching the root node as follows:

```
<xsl:template match="/">
  <html>
    <head>
      <title>TV Guide</title>
      ...
    </head>
  </html>
</xsl:template>
```

```

    </head>
    <body>
      <h1>TV Guide</h1>
      <xsl:apply-templates select="/TVGuide/Channel" />
    </body>
  </html>
</xsl:template>

```

Using templates instead of `<xsl:for-each>` breaks up the stylesheet into manageable chunks in a similar way to functions and methods in standard programming languages. One advantage of this is that you don't have XSLT with lots and lots of indentations, which can really help with readability! A bigger advantage is that you can use the same template for similar nodes in different places or on the same node multiple times, as we'll see later in the chapter. On the down side, the stylesheet no longer looks as similar to the HTML document that we're producing as it used to, and to change the result of the stylesheet, you may have to navigate between multiple templates.

You can replace an `<xsl:for-each>` element with a template holding its contents and an `<xsl:apply-templates>` element selecting the nodes you want to process.

Try It Out – Replacing `<xsl:for-each>` with Templates

We've still used `<xsl:for-each>` in a couple of other places within `TVGuide4.xsl`, so let's convert those instances in the same way as we've done with the one that iterated over `<Channel>` elements.

The next `<xsl:for-each>` is where we iterate over the `<Program>` elements, which is not within the template that matches `<Channel>` elements. We can convert it by first replacing the `<xsl:for-each>` with an `<xsl:apply-templates>` element that has the same value for its `select` attribute:

```

<xsl:template match="Channel">
  <h2 class="channel"><xsl:value-of select="Name" /></h2>
  <xsl:apply-templates select="Program" />
</xsl:template>

```

and then creating a template that matches `<Program>` elements. The new template has the same contents as the old `<xsl:for-each>` did:

```

<xsl:template match="Program">
  <div>
    <p>
      <span class="date"><xsl:value-of select="Start" /></span><br />
      <span class="title"><xsl:value-of select="Series" /></span><br />
      <xsl:value-of select="Description" />
      <span onclick="toggle({Series}Cast);">[Cast]</span>
    </p>
    <div id="{Series}Cast" style="display: none;">
      <ul class="castlist">
        <xsl:for-each select="CastList/CastMember">
          <li>

```

```

        <span class="character">
            <xsl:value-of select="Character" />
        </span>
        <span class="actor">
            <xsl:value-of select="Actor" />
        </span>
    </li>
</xsl:for-each>
</ul>
</div>
</div>
</xsl:template>

```

This template also contains an `<xsl:for-each>`, one that iterates over the `<CastMember>` element children of `<CastList>` elements. Again, we can replace this `<xsl:for-each>` with an `<xsl:apply-templates>`:

```

<xsl:template match="Program">
    <div>
        <p>
            <span class="date"><xsl:value-of select="Start" /></span><br />
            <span class="title"><xsl:value-of select="Series" /></span><br />
            <xsl:value-of select="Description" />
            <span onclick="toggle({Series}Cast);">[Cast]</span>
        </p>
        <div id="{Series}Cast" style="display: none;">
            <ul class="castlist">
                <xsl:apply-templates select="CastList/CastMember" />
            </ul>
        </div>
    </div>
</xsl:template>

```

and create a separate template that deals with giving output for `<CastMember>` elements:

```

<xsl:template match="CastMember">
    <li>
        <span class="character"><xsl:value-of select="Character" /></span>
        <span class="actor"><xsl:value-of select="Actor" /></span>
    </li>
</xsl:template>

```

We now have four templates in `TVGuide5.xsl`, matching:

- ☐ The root node
- ☐ `<Channel>` elements
- ☐ `<Program>` elements
- ☐ `<CastMember>` elements

If you run `TVGuide5.xsl` with `TVGuide.xml`, you should get exactly the same result as the original, simplified stylesheet. Splitting up the processing into separate templates hasn't changed the result of the transformation.

The Built-in Templates

We've seen that when you apply templates to a node, the XSLT processor tries to find the template that matches that node. But what happens when there isn't a template that matches a node? For example, if we applied templates to the `<Name>` child of the `<Channel>` element, as follows, but didn't have a template to match the `<Name>` element:

```
<xsl:template match="Channel">
  <h2 class="channel"><xsl:apply-templates select="Name" /></h2>
  <xsl:apply-templates select="Program" />
</xsl:template>
```

When the XSLT processor can't find a template to match the node that it's been told to process, it uses a **built-in template**. If you find that the result of your stylesheet includes text you didn't expect, the chances are that it's due to the built-in templates. Just because there isn't a template for a particular node that doesn't mean that it's not processed.

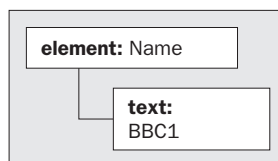
For elements, the built-in template is as follows:

```
<xsl:template match="*">
  <xsl:apply-templates />
</xsl:template>
```

This template uses two bits of syntax that we haven't seen before:

- ❑ The `match` attribute of the template takes the value `*`. Templates with a match pattern of `*` match all elements.
- ❑ The `<xsl:apply-templates>` element doesn't have a `select` attribute. If you use `<xsl:apply-templates>` without a `select` attribute, the XSLT processor collects all the children of the current node (which is the node that the template matches) and applies templates to them.

To see the effect of this, take another look at the part of the node tree containing the `<Name>` element:



The `<Name>` element has only one child node, a text node with the value `BBC1`. When you tell the XSLT processor to apply templates to the `<Name>` element, it will use the built-in template for elements, and hence apply templates to the text node.

Now, again, we don't have a template that matches text nodes in our stylesheet, so the processor uses a built-in template. The built-in template for text nodes is:

```
<xsl:template match="text()">
  <xsl:value-of select="." />
</xsl:template>
```

Again, this template uses a couple of new bits of syntax:

- ❑ The `match` attribute of the `<xsl:template>` element takes the value `text()`. Templates with a match pattern of `text()` match text nodes.
- ❑ The `select` attribute of the `<xsl:value-of>` element takes the value `.`. The path `.` selects the context node, so `<xsl:value-of select="." />` gives the value of the context node, in this case the text node.

In combination, these two built-in templates mean that if you apply templates to an element, but don't have a template for that element (or any elements it contains), then you'll get the value of the text held within the element. So applying templates to the `<Name>` element means that you get the value BBC1 in the result.

If a processor can't find a template that matches a node, it uses the built-in template for that node type. In effect, these give the value of the elements to which you apply templates.

Try It Out – Using the Built-in Templates

There are quite a few places in our stylesheet where we want to just get the value of an element. Rather than using `<xsl:value-of>` to get these values, we could apply templates to the elements and let the built-in templates do their work to give us the element values.

We've already done this with the template for `<Channel>` elements, to get the name of the channel:

```
<xsl:template match="Channel">
  <h2 class="channel"><xsl:apply-templates select="Name" /></h2>
  <xsl:apply-templates select="Program" />
</xsl:template>
```

We can also replace the `<xsl:value-of>` instructions in the template for `<Program>` elements to get the values of the `<Start>`, `<Series>`, and `<Description>` elements:

```
<xsl:template match="Program">
  <div>
    <p>
      <span class="date"><xsl:apply-templates select="Start" /></span>
      <br />
      <span class="title"><xsl:apply-templates select="Series" /></span>
      <br />
      <xsl:apply-templates select="Description" />
    </p>
  </div>
</xsl:template>
```

```
<span onclick="toggle({Series}Cast);">[Cast]</span>
</p>
<div id="{Series}Cast" style="display: none;">
  <ul class="castlist">
    <xsl:apply-templates select="CastList/CastMember" />
  </ul>
</div>
</div>
</xsl:template>
```

and in the template for `<CastMember>` elements, to get the values of the `<Character>` and `<Actor>` elements:

```
<xsl:template match="CastMember">
  <li>
    <span class="character">
      <xsl:apply-templates select="Character" />
    </span>
    <span class="actor">
      <xsl:apply-templates select="Actor" />
    </span>
  </li>
</xsl:template>
```

Both the `<Character>` and `<Actor>` elements actually contain `<Name>` elements giving the name of the character or actor. With the built-in templates, you get the value of the text held in these `<Name>` elements.

The stylesheet `TVGuide6.xsl` still contains the same number of templates, but applies templates to all the elements that it processes rather than simply getting their values. If you run `TVGuide6.xsl` with `TVGuide.xml` you should get exactly the same result as you did before. Changing the `<xsl:value-of>` elements to `<xsl:apply-templates>` elements hasn't altered the result of the stylesheet.

Extending Stylesheets

As we've seen in the previous section, the effect of the built-in templates is that if you apply templates to an element, you get all the text that's contained in the element, at any level. This is the same as what you get when you use `<xsl:value-of>` and select the element. In other words, if you don't have a template that matches `<Program>` elements or any of their descendants then the following instructions give you exactly the same result:

```
<xsl:apply-templates select="Program" />
<xsl:value-of select="Program" />
```

There are pluses and minuses to using `<xsl:apply-templates>` rather than `<xsl:value-of>`. The biggest downside is that it's less efficient to use `<xsl:apply-templates>` because it forces the XSLT processor to search through the stylesheet for templates that match the node rather than directly giving the value of the node. For this reason, I would generally only apply templates to elements that I know could contain other elements, and not to text nodes, attributes, or elements that I know only have text content.

On the plus side, using `<xsl:apply-templates>` makes it a lot easier to change the format of the value that you get from an element, just by adding a template for that element. For example, if I wanted to change the way I give the name of the series so that it's given as a link to a page on that series instead, I can add a template that matches the `<Series>` element and generates an `<a>` element in the result giving a link to the page.

```
<xsl:template match="Series">
  <a href="{ }.html">
    <xsl:value-of select="." />
  </a>
</xsl:template>
```

This uses an attribute value template (which we met in the last chapter) to give the URL for the page, using the value of the context node (the `<Series>` element) plus the string `'.html'`. For example, the element `<Series>EastEnders</Series>` will result in a link to `EastEnders.html`.

This flexibility is also very useful when the XML format that you're converting from isn't finalized. If we added `<Description>` elements to the `<Character>` and `<Actor>` elements, as in `TVGuide2.xml` in the code download, then we could update the stylesheet to cope with the change by adding templates for these elements so that they only generate the value of the `<Name>` rather than including the description. We do this in `TVGuide7.xsl`, which contains:

```
<xsl:template match="Actor">
  <xsl:apply-templates select="Name" />
</xsl:template>

<xsl:template match="Character">
  <xsl:apply-templates select="Name" />
</xsl:template>
```

If you didn't add these templates, then applying templates to the `<Actor>` or `<Character>` elements would result in text containing both the name and description of the actor or character, concatenated.

Using templates allows you to extend your stylesheet more easily than you can if you use `<xsl:for-each>` and `<xsl:value-of>`.

Templates as Mapping Rules

We've been a little formulaic in our conversion from the simplified stylesheet to the full stylesheet that we have now – we take every `<xsl:for-each>` and `<xsl:value-of>` and turn it into an `<xsl:apply-templates>`. The introduction of templates has really just been a way of modularizing the code.

But there's another way of thinking about templates that can make your stylesheets more elegant and more extensible, and that's to consider them as a mapping rule. Each template describes how to map a particular node in the source XML on to a result that you're after. In this section, we'll first look at using templates with mixed content, and at how using templates as mapping rules is particularly useful when processing document-oriented XML. Then, we'll go on to look at how we could apply the same kind of technique to our stylesheet and the impact of doing so.

Processing Document-Oriented XML

Let's first look at the problem of generating output from document-oriented XML. Document-oriented XML arises when you take a paragraph of text and mark up particular words and phrases within that paragraph, giving mixed content – elements and text intermingled. This is in contrast to data-oriented XML, which is concerned with storing data and usually results in element-only and text-only content.

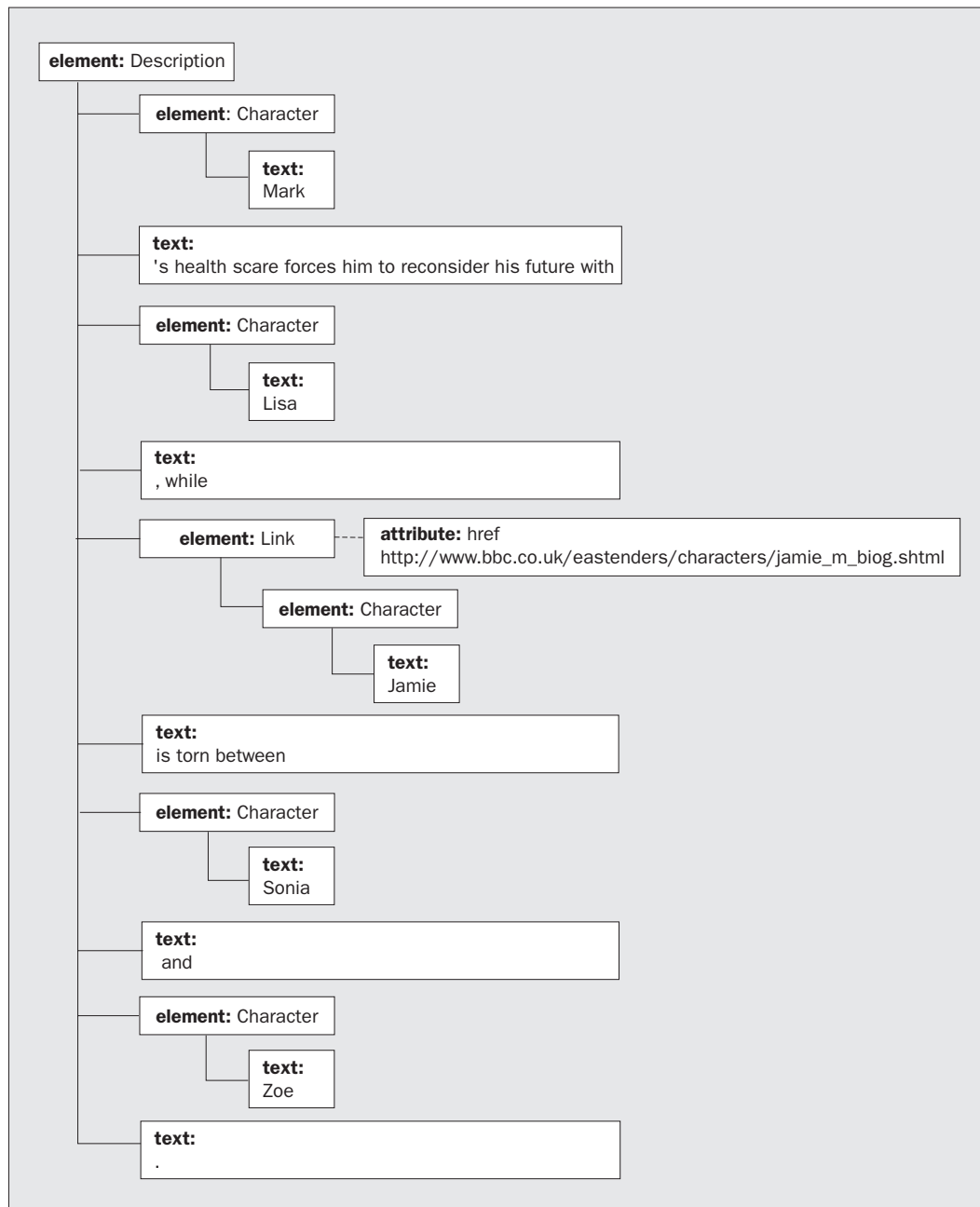
Take another look at the XML structure that we're using to store information about TV programs:

```
<Program>
  <Start>2001-07-05T19:30:00</Start>
  <Duration>PT30M</Duration>
  <Series>EastEnders</Series>
  <Title></Title>
  <Description>
    Mark's health scare forces him to reconsider his future with Lisa,
    while Jamie is torn between Sonia and Zoe.
  </Description>
  ...
</Program>
```

While most of the XML structure is oriented around providing data, the description of the TV program is more document-oriented. You could imagine wanting to add a bit of document-oriented markup to the `<Description>` element, perhaps a link to the EastEnders biography for Jamie and highlights around the names of the characters:

```
<Description>
  <Character>Mark</Character>'s health scare forces him to reconsider his
  future with <Character>Lisa</Character>, while
  <Link
    href="http://www.bbc.co.uk/eastenders/characters/jamie_m_biog.shtml">
    <Character>Jamie</Character>
  </Link> is torn between <Character>Sonia</Character> and
  <Character>Zoe</Character>.
</Description>
```

The `<Description>` element now holds mixed content. Looking at the node tree representation of that piece of XML makes this clearer:



We want the elements that we use in the description to be transformed into HTML elements instead. In the HTML version of the page, we want to use `<span>` elements around the character names, and transform the `<Link>` elements into `<a>` elements, to give:

```
<p>
  <span class="date">2001-07-05T19:30:00</span><br />
  <span class="title">EastEnders</span><br />
  <span class="character">Mark</span>'s health scare forces him to
  reconsider his future with <span class="character">Lisa</span>, while
  <a href="http://www.bbc.co.uk/eastenders/characters/jamie_m_biog.shtml">
    <span class="character">Jamie</span></a> is torn between
    <span class="character">Sonia</span> and
    <span class="character">Zoe</span>.
  <span onclick="toggle(EastEndersCast);">[Cast]</span>
</p>
```

If you imagine processing the content of the `<Description>` element with `<xsl:for-each>`, you'll see that you run into problems. We haven't looked at how to yet, but it's possible to iterate over all the nodes that are children of the `<Description>` element, and test what kind of node they are to decide what to do with them. But even if you did that, you still need to take account of nested elements, so you'd get very long and very deep conditional processing to cover all levels of nesting.

However, with templates it's a lot easier. You can create a template for each kind of element that you know can occur in the content of the `<Description>` element, which describes how to map between that element and the result that you want. In this example, there are two elements, `<Character>` and `<Link>`, so you need a template for each. Within the `<span>` and `<a>` elements that these templates create, you apply templates to the content of the `<Character>` or `<Link>` element to account for possible nested elements:

```
<xsl:template match="Character">
  <span class="character">
    <xsl:apply-templates />
  </span>
</xsl:template>

<xsl:template match="Link">
  <a href="{@href}">
    <xsl:apply-templates />
  </a>
</xsl:template>
```

Whenever you add a new type of element that you might include in the description, you can add a new template that describes how to map that on to HTML.

Templates are particularly suited to processing document-oriented XML. Each template acts as a mapping rule from source to result.

Try It Out – Creating Presentation Rules

We can add support for lots of different elements that we want to be able to use within the `<Description>` element. Highlighting character names and providing links to other web sites is useful, but you might also want to add elements for emphasis, foreign words, names of directors, series, channels, films, and so on – different elements for the different types of words and phrases that can appear in descriptions of TV programs and series.

We'll add just a few of these elements in TVGuide2.xml, to create TVGuide3.xml, which looks as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<TVGuide start="2001-07-05" end="2001-07-05">

<Channel>
  <Name>BBC1</Name>
  ...
  <Program rating="5" flag="favorite">
    <Start>2001-07-05T19:30:00</Start>
    <Duration>PT30M</Duration>
    <Series>EastEnders</Series>
    <Title></Title>
    <Description>
      <Character>Mark</Character>'s health scare forces him to reconsider
      his future with <Character>Lisa</Character>, while
      <Link
        href="http://www.bbc.co.uk/eastenders/characters/jamie_m_biog.shtml">
        <Character>Jamie</Character>
      </Link> is torn between <Character>Sonia</Character> and
      <Character>Zoe</Character>.
    </Description>
    <CastList>
      <CastMember>
        <Character>
          <Name>Zoe Slater</Name>
          <Description>
            The youngest Slater girl, <Character>Zoe</Character> really
            makes the most of the fact she's the baby of the family.
          </Description>
        </Character>
        <Actor>
          <Name>Michelle Ryan</Name>
          <Description>
            For more details, see
            <Link href="http://www.ajmanagement.co.uk/michelle-ryan.htm">
              <Actor>Michelle Ryan</Actor>'s Agency
            </Link>.
          </Description>
        </Actor>
      </CastMember>
      <CastMember>
        <Character>
          <Name>Jamie Mitchell</Name>
          <Description>
            Jamie's a bit of a heartthrob (who could resist that
            little-boy-lost look?) but until <Character>Janine
            Butcher</Character> came along he'd steered clear of girls.
          </Description>
        </Character>
        <Actor>
          <Name>Jack Ryder</Name>
```

```
        <Description>
          Won Best Newcomer for <Character>Jamie Mitchell</Character>
          in the 1999 TV awards.
        </Description>
      </Actor>
    </CastMember>
  </CastMember>
  ...
  </CastMember>
</CastList>
...
</Program>
...
</Channel>
...
</TVGuide>
```

Try using `TVGuide6.xsl` with `TVGuide3.xml`, which uses `<xsl:apply-templates>` to apply templates to the `<Description>` element. There aren't any templates for `<Character>` or `<Link>` elements, so the built-in templates are used instead. The result of the transformation of the `<Description>` element looks just the same as before, because the built-in templates automatically show any text within an element.

We want a new version of the stylesheet (`TVGuide8.xsl`), which generates HTML where the words and phrases in the description that we've picked out with `<Character>` and `<Link>` elements are displayed and behave slightly differently from the rest of the text. Links should *be* links, for example, and character names should be slightly larger than the surrounding text.

To make the marked-up text display and act differently, we need to introduce templates for these new elements: one for the `<Link>` element, to create a hypertext link with an HTML `<a>` element:

```
<xsl:template match="Link">
  <a href="{@href}">
    <xsl:apply-templates />
  </a>
</xsl:template>
```

and one for the `<Character>` element, to create a `<span>` element with a class of `character` around the character names:

```
<xsl:template match="Character">
  <span class="character">
    <xsl:apply-templates />
  </span>
</xsl:template>
```

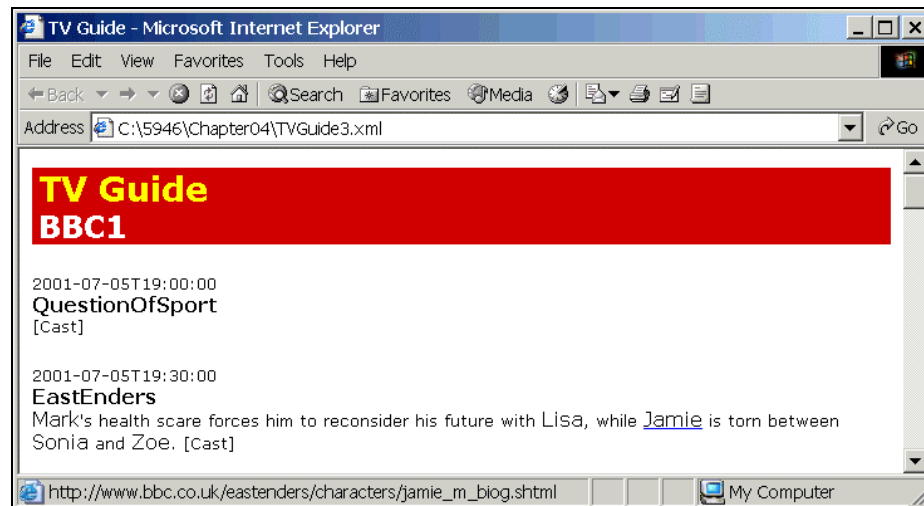
To make the character names slightly bigger, we'll add a rule to `TVGuide.css`, to create `TVGuide2.css`, which contains:


```
.character {
    font-size: larger;
}
```

Putting the final touches on `TVGuide8.xsl`, we need the HTML that it creates to point to `TVGuide2.css` rather than `TVGuide.css`, so the `<link>` element generated in the template matching the root node needs to be altered slightly:

```
<xsl:template match="/">
  <html>
    <head>
      <title>TV Guide</title>
      <link rel="stylesheet" href="TVGuide2.css" />
      ...
    </head>
    ...
  </html>
</xsl:template>
```

Having made this final change, transform `TVGuide3.xml` with `TVGuide8.xsl`. The result of the transformation should look something like the following:



The character names are slightly larger than the rest of the text, and clicking on the word "Jamie" takes you to the EastEnders site with Jamie Mitchell's biography. Feel free to add your own elements to that description, and add your own templates matching them to present the document-oriented XML.

Context-Dependent Processing

We've now introduced a number of elements into our XML structure that we're actually using elsewhere in different ways. For example, we use the `<Character>` element to indicate the name of a character in a description, and to hold information about a character within the `<CastMember>` element. The template makes no distinction between these two uses of the `<Character>` element, and does the same thing for each, making a `<span>` element with a character class:

```
<xsl:template match="Character">
  <span class="character">
    <xsl:apply-templates />
  </span>
</xsl:template>
```

At the moment, we're also creating a `<span>` element when we create the cast list, in the template matching the `<CastMember>` element:

```
<xsl:template match="CastMember">
  <li>
    <span class="character">
      <xsl:apply-templates select="Character" />
    </span>
    <span class="actor">
      <xsl:apply-templates select="Actor" />
    </span>
  </li>
</xsl:template>
```

This means we end up with two `<span>` elements around the names of the characters in the cast list. We can get rid of the superfluous `<span>` element either by reverting back to using `<xsl:value-of>` to get the name of the character, or by removing the `<span>` element in the template that matches `<CastMember>` elements (and the same applies for the `<Actor>` elements as well, since we might name actors in a description):

```
<xsl:template match="CastMember">
  <li>
    <xsl:apply-templates select="Character" />
    <xsl:apply-templates select="Actor" />
  </li>
</xsl:template>
```

So now we have the same template being used to process `<Character>` elements in different contexts. However, one of the extensions that we made earlier in this chapter was to have the `<Character>` and `<Actor>` elements within `<CastMember>` actually give both a name and a description of the character. When we take this into account, we have a problem because we don't want the `<span>` element to contain both the name and the description of the character. We need a different template for the `<Character>` and `<Actor>` elements when they are children of `<CastMember>` elements (ones that just apply templates to the `<Name>` element child of the `<Character>` or `<Actor>` element). The new templates for the `<Character>` elements that occur in `<CastMember>` elements need to look like:

```

<xsl:template match="Character">
  <span class="character">
    <xsl:apply-templates select="Name" />
  </span>
</xsl:template>

```

But we can't use this template with the `<Character>` elements that occur in `<Description>` elements because they don't contain `<Name>` elements. If we use this template with those elements, the `<span>` elements won't have any content.

So we need some way of having different templates for the different contexts in which these elements are allowed. We can do this by changing the value of the `match` attribute of `<xsl:template>`, and this is where we need to use patterns, which are why they are introduced next.

Patterns

So far we've seen four kinds of values for the `match` attribute of `<xsl:template>`:

- ❑ `/` matches the root node
- ❑ `*` matches any element
- ❑ `text()` matches text nodes
- ❑ the name of an element matches that element

These values are all examples of **patterns**. An XSLT processor uses the pattern specified in the `match` attribute of `<xsl:template>` to work out whether it can use a template to process a node to which you've told it to apply templates. In our case, we need one pattern to match `<Character>` elements that are children of `<CastMember>` elements, and another pattern to match `<Character>` elements that appear at any level within `<Description>` elements. In both cases, we're checking the context in which the `<Character>` element appears. The two patterns we need are:

- ❑ `CastMember/Character` to match `<Character>` elements that occur within `<CastMember>` elements
- ❑ `Description//Character` to match `<Character>` elements that occur nested to any level within `<Description>` elements

These types of patterns are technically known as **location path patterns**. As you can see, location path patterns look a lot like the location paths that we use to select nodes to process in `select` attributes, and it can be easy to get confused between the two. You use location paths to select nodes; they point from the current node to a set of other nodes in the tree, stepping down from element to child. You use location path patterns to match nodes; they test whether a particular node has particular ancestors, looking up the node tree to work out the context of the node.

A location path pattern is made up of a number of **step patterns**, separated by either `/` or `//`. If the separator is a `/`, then the pattern tests a parent-child relationship. For example, the pattern `Description/Character` matches `<Character>` elements whose *immediate parent* is a `<Description>` element. If the separator is `//`, on the other hand, then the pattern tests an ancestor-descendent relationship. For example, the pattern `Description//Character` matches `<Character>` elements that have a `<Description>` element as an ancestor at any level.

Location path patterns enable you to match elements according to the context in which they occur.

Identifying Elements in Different Contexts

In our XML document, `TVGuide3.xml`, there are some elements that have different meanings in different contexts. If you remember back that far, it was one of our design decisions when we first put together our XML structure that we would make use of the context an element was in, rather than use different names for elements in different contexts, to work out what an element meant and what we should do with it.

So now we have to deal with that decision by creating different templates with different match patterns for the different contexts in which an element can occur. The contexts within which different elements can occur are shown in the following table:

Element	Contexts
<Name>	child of <Channel> child of <Character> child of <Actor>
<Description>	child of <Program> child of <Character> child of <Actor>
<Character>	child of <CastMember> descendent of <Description>
<Actor>	child of <CastMember> descendent of <Description>
<Series>	child of <Program> descendent of <Description>
<Program>	child of <Channel> descendent of <Description>
<Channel>	child of <TVGuide> descendent of <Description>

A stylesheet that deals with documents that follow our markup language really needs to have templates that deal with elements occurring in each of these possible contexts, using patterns that include ancestry information.

Try It Out – Creating Templates for Context-Dependent Elements

At this stage, we'll create a new version of the stylesheet, `TVGuide9.xsl`, which contains separate templates for each of these elements in each of these contexts. You should be able to put together different templates for the elements in their different contexts as mapping rules. For example, the `<Name>` element can occur in three contexts – `<Channel>`, `<Character>`, and `<Actor>` – so there should be three corresponding templates:

```
<xsl:template match="Channel/Name">...</xsl:template>
<xsl:template match="Character/Name">...</xsl:template>
<xsl:template match="Actor/Name">...</xsl:template>
```

As you add these templates, you should consider whether some of the HTML that you're currently generating in higher-level templates can be generated in lower-level templates instead. For example, my feeling is that the `<Name>` element in the `<Channel>` element maps on to the `<h2>` heading element in the result, so the template should look like:

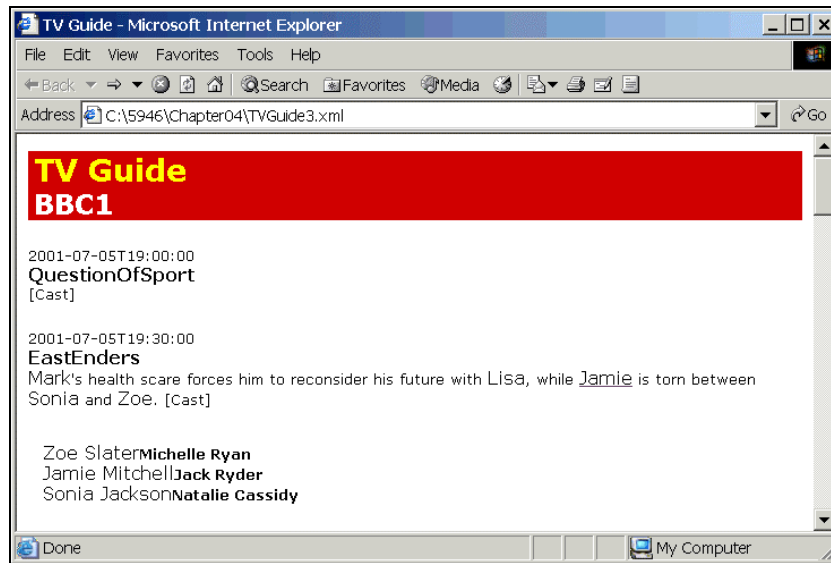
```
<xsl:template match="Channel/Name">
  <h2 class="channel"><xsl:value-of select="." /></h2>
</xsl:template>
```

But if you create the `<h2>` element in the above template, you don't need to create it in the template for the `<Channel>` element. So you need to change that template too, removing the `<h2>` element that you were creating within it:

```
<xsl:template match="TVGuide/Channel">
  <xsl:apply-templates select="Name" />
  <xsl:apply-templates select="Program" />
</xsl:template>
```

This template only applies to `<Channel>` elements that are children of the `<TVGuide>` element, not those that are descendants of `<Description>` elements.

If you go through this process religiously, you should end up with about 19 different templates, as in `TVGuide9.xsl`. Using `TVGuide9.xsl` with `TVGuide3.xml` results in a page in which both the `<Character>` elements within the cast list and those within the `<Description>` are treated properly, so the result looks like the following when you view it in Internet Explorer:



This process of adding templates on an element-by-element basis, taking account of the different contexts in which an element can occur, has left us with a lot of templates, but it has made the stylesheet more robust. We've handled the problem that we had with `<Character>` elements being treated differently in different contexts, and we've included templates to handle the possibility of things that aren't actually present in `TVGuide3.xml`, but which could happen in more complete documents that follow the markup language, such as `<Channel>` elements in `<Description>` elements.

Unnecessary Templates

Adding templates to deal with every kind of element, in every context, within a stylesheet can leave you with a stylesheet that contains lots of templates that actually don't do very much. This makes the stylesheet harder for you to maintain (because it's longer) and it makes more work for the processor when it needs to identify which template to apply in a particular situation. You need to find a judicious balance between the two.

First, you don't need to create a template for every element in every context. Remember that if the XSLT processor can't find a template that matches a node, then it will use a built-in template. If an element (and all its content) gets processed by the built-in templates, then you'll just get the value of the node. So, if you have templates that look like either of the following:

```
<xsl:template match="...">
  <xsl:value-of select="." />
</xsl:template>

<xsl:template match="...">
  <xsl:apply-templates />
</xsl:template>
```

then you may as well get rid of them.

Second, you can get rid of templates that essentially process the children of the element that they match in the order that they appear. For example, the revised template for the `<Channel>` elements that are children of `<TVGuide>` elements is like that:

```
<xsl:template match="TVGuide/Channel">
  <xsl:apply-templates select="Name" />
  <xsl:apply-templates select="Program" />
</xsl:template>
```

When a `<Channel>` element appears as a child of a `<TVGuide>` element then it can only contain a `<Name>` element followed by any number of `<Program>` elements. So telling the processor to apply templates first to the `<Name>` element and then to the `<Program>` elements has the same effect as telling the processor to apply templates to all the `<Channel>` element's children in the order they occur, which would be the template:

```
<xsl:template match="TVGuide/Channel">
  <xsl:apply-templates />
</xsl:template>
```

This template has the same effect as the built-in template for elements, so you can delete it without affecting the result of the stylesheet.

Finally, you can get rid of templates that match elements that are never selected for processing. Remember that a template is never actually used if the processor never gets told to apply templates to a node that matches that template. A template that is never matched will never be used, and can therefore be safely removed.

Try It Out – Removing Unnecessary Templates

TVGuide9.xsl does contain a few templates that aren't really necessary, and to make it easier to understand we could prune it to create TVGuide10.xsl.

There are three templates that do the same thing as the built-in templates, and just contain an `<xsl:apply-templates>` or an `<xsl:value-of>` instruction that gives the value of the current node. They are the one matching `<Name>` element children of `<Character>` elements, the one matching `<Name>` element children of `<Actor>` elements, and the one matching `<Description>` element children of `<Program>` elements – you can safely delete them:

```
<xsl:template match="Character/Name">
  <xsl:value-of select="." />
</xsl:template>

<xsl:template match="Actor/Name">
  <xsl:value-of select="." />
</xsl:template>

<xsl:template match="Program/Description">
  <xsl:apply-templates />
</xsl:template>
```

We identified the template matching `<Channel>` element children of the `<TVGuide>` element as falling into the second category. It had two `<xsl:apply-templates>` in it, but these selected nodes in the same order as they occurred in the source document, essentially the same as applying templates to all the children, which again is just what the built-in templates do. So you can safely delete the following template:

```
<xsl:template match="TVGuide/Channel">
  <xsl:apply-templates select="Name" />
  <xsl:apply-templates select="Program" />
</xsl:template>
```

Finally, in `TVGuide9.xsl` there are two templates that can never be applied – the one matching `<Description>` elements within `<Character>` elements, and the one matching `<Description>` elements within `<Actor>` elements. These templates will never get activated because the templates that match `<Character>` and `<Actor>` elements (within `<CastMember>` elements) never apply templates to their child `<Description>` elements. So these two templates can also be deleted:

```
<xsl:template match="Character/Description" />

<xsl:template match="Actor/Description" />
```

Once you've done all that, you should have something like the following stylesheet, `TVGuide10.xsl`. The ordering of the templates within the stylesheet doesn't matter.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

  <xsl:template match="Channel/Name">
    <h2 class="channel"><xsl:value-of select="." /></h2>
  </xsl:template>

  <xsl:template match="Channel/Program">
    <div>
      ...
    </div>
  </xsl:template>

  <xsl:template match="Program/Series">
    <span class="title"><xsl:value-of select="." /></span>
  </xsl:template>

  <xsl:template match="CastMember">
    <li>
      <xsl:apply-templates select="Character" />
      <xsl:apply-templates select="Actor" />
    </li>
  </xsl:template>
```



```

    </li>
</xsl:template>

<xsl:template match="CastMember/Character">
    <span class="character">
        <xsl:apply-templates select="Name" />
    </span>
</xsl:template>

<xsl:template match="CastMember/Actor">
    <span class="actor">
        <xsl:apply-templates select="Name" />
    </span>
</xsl:template>

<xsl:template match="Description//Character">
    <span class="character">
        <xsl:apply-templates />
    </span>
</xsl:template>

<xsl:template match="Description//Actor">
    <span class="actor">
        <xsl:apply-templates />
    </span>
</xsl:template>

<xsl:template match="Link">
    <a href="{@href}">
        <xsl:apply-templates />
    </a>
</xsl:template>

<xsl:template match="Description//Program">
    <span class="program"><xsl:apply-templates /></span>
</xsl:template>

<xsl:template match="Description//Series">
    <span class="series"><xsl:apply-templates /></span>
</xsl:template>

<xsl:template match="Description//Channel">
    <span class="channel"><xsl:apply-templates /></span>
</xsl:template>

</xsl:stylesheet>

```

Using TVGuide10.xsl with TVGuide3.xml should give the same result as TVGuide9.xsl did.

Resolving Conflicts Between Templates

Whenever you have rules in a language, such as rules in CSS or templates in XSLT, you need some way to resolve conflicts when two of the rules apply to the same situation. What would happen, for example, if you had one template that matched all `<Character>` elements and another template that matched only those `<Character>` elements that had `<CastMember>` as their parent:

```
<xsl:template match="Character">
  <span class="character"><xsl:apply-templates /></span>
</xsl:template>

<xsl:template match="CastMember/Character">
  <span class="character"><xsl:apply-templates select="Name" /></span>
</xsl:template>
```

A `<Character>` element in a `<Description>` element only matches one of the templates, because it doesn't have a `<CastMember>` element as its parent, so obviously the XSLT processor uses that template with it. But what about a `<Character>` element in a `<CastMember>` element? It matches both of the template's patterns, so what should the XSLT processor do?

Template Priority

Well, the XSLT processor will only ever process one template when you apply templates to a node, so it has to choose between the two templates that it's presented with in some way. It does this by looking at the template's **priority**. A template with a high priority is chosen over a template with a lower priority.

You can specifically assign a template a priority using the `priority` attribute on the `<xsl:template>` element. The `priority` attribute can be set to any number, including decimal and negative numbers. For example, you can give the two templates different specific priorities, as follows:

```
<xsl:template match="Character" priority="-1">
  <span class="character"><xsl:apply-templates /></span>
</xsl:template>

<xsl:template match="CastMember/Character" priority="2">
  <span class="character"><xsl:apply-templates select="Name" /></span>
</xsl:template>
```

Default Priorities

However, it would be very difficult to assign and keep track of priorities if the `priority` attribute was your only option. If you don't specify a `priority` attribute on a template, the XSLT processor assigns it a priority based on how specific its match pattern is. XSLT processors recognize three levels of priority in patterns:

- ❑ Patterns that match a class of nodes, such as `*`, which matches all elements, are assigned an implicit priority of `-0.5`
- ❑ Patterns that match nodes according to their name, such as `Character`, which matches `<Character>` elements, are assigned an implicit priority of `0`

- ❑ Patterns that match nodes according to their context, such as `CastMember/Character`, which matches `<Character>` elements whose parent is a `<CastMember>` element, are assigned an implicit priority of 0.5

When assigning priorities based on patterns, it doesn't matter how specific the context information is: if you specify any context for a node then the template has a priority of 0.5. For example, `Description/Link/Character` has exactly the same priority as `Description//Character`.

Technically, it's an error if you have two templates that match the same node and the match patterns for the two templates have the same specificity. However, most processors recover from the error and use the template that you've defined last in the stylesheet. You should try to avoid having templates that have the same priority and can feasibly match the same node; use the `priority` attribute to assign them specific, different priorities.

If two templates match the same node, the processor uses the one with the highest priority. Priority can be assigned explicitly with the `priority` attribute or determined implicitly from the template's match pattern. As a last resort, the processor can select the last matching template in the stylesheet.

Try It Out – Using Priorities

Currently, the templates that our stylesheet contains don't have any conflicts with each other because each of them only matches elements in a fairly specific context. To try out priorities, let's try making some of the templates conflict by removing some of that context information from one of them. For example, in `TVGuide11.xsl`, let's change the template that matches `<Program>` elements within `<Description>` elements to match any `<Program>` element:

```
<xsl:template match="Program">
  <span class="program"><xsl:apply-templates /></span>
</xsl:template>
```

Now a `<Program>` element will always be matched by this template, and if it's a child of a `<Channel>` element then it will also match the following template:

```
<xsl:template match="Channel/Program">
  <div>
    <p>
      <span class="date"><xsl:apply-templates select="Start" /></span>
      <br />
      <xsl:apply-templates select="Series" />
      <br />
      <xsl:apply-templates select="Description" />
      <span onclick="toggle({Series}Cast);">[Cast]</span>
    </p>
    <div id="{Series}Cast" style="display: none;">
      <ul class="castlist">
        <xsl:apply-templates select="CastList/CastMember" />
      </ul>
    </div>
  </div>
```

```

    </ul>
  </div>
</div>
</xsl:template>

```

However, if you run TVGuide11.xsl with TVGuide3.xml, it won't make any difference to the result because the latter template, matching `<Program>` element children of `<Channel>` elements, has a higher priority. When the `<Program>` element that the XSLT processor is trying to process is a child of a `<Channel>` element, it will use the latter template, with the match pattern of `Channel/Program`; when it's not (such as when it's a child of a `<Description>` element), the processor will use the former template, with the match pattern of `Program`.

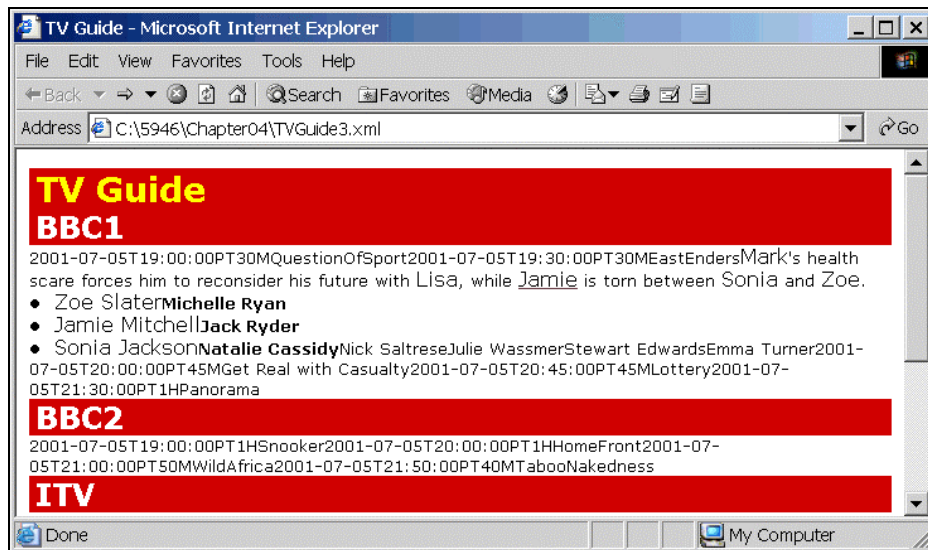
But we can change that in two ways. First, we can assign different priorities to the two templates using the `priority` attribute. Assign the general template a priority of 1, to create TVGuide12.xsl:

```

<xsl:template match="Program" priority="1">
  <span class="program"><xsl:apply-templates /></span>
</xsl:template>

```

This explicit priority is higher than the implicit priority of the second template. If you run TVGuide12.xsl with TVGuide3.xml, you get the following mess:

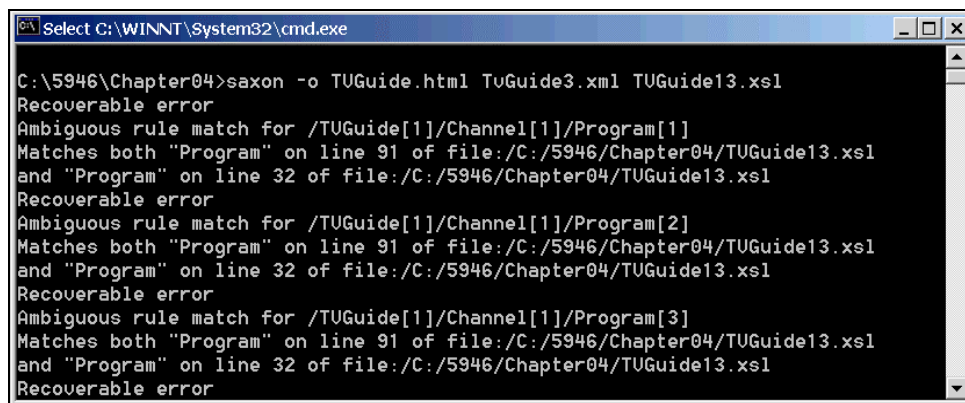


Some of the formatting is still present because the elements inside the `<Program>` element still get processed by this template. However, the parts of the result that were generated by the second template, such as the `<div>` and `<p>` elements, are no longer present. You can get the same effect by removing the priority on the template matching all `<Program>` elements and giving the template matching `<Program>` elements within `<Channel>` elements a priority lower than `-0.5`.

A second way of manipulating the priority of the templates is to remove the context from the path for the second template (and remove the priority attributes that you just added), so the template that's supposed to be used to process <Program> elements within <Channel> elements looks like:

```
<xsl:template match="Program">
  <div>
    <p>
      <span class="date"><xsl:apply-templates select="Start" /></span>
      <br />
      <xsl:apply-templates select="Series" />
      <br />
      <xsl:apply-templates select="Description" />
      <span onclick="toggle({Series}Cast);">[Cast]</span>
    </p>
    <div id="{Series}Cast" style="display: none;">
      <ul class="castlist">
        <xsl:apply-templates select="CastList/CastMember" />
      </ul>
    </div>
  </div>
</xsl:template>
```

Now the templates both apply to the same <Program> elements and have the same priority, which is an error. Move the template that's supposed to be for <Program> elements within <Description> elements below the above in the stylesheet (if it's not there already), to give TVGuide13.xsl. Most likely your processor will use the lower template in the stylesheet to process the <Program> elements, and you'll get the same mess as when you changed the priority explicitly. Processors are within their rights to terminate the stylesheet and give you an error, though, and some processors might warn you that there are two templates that match the same node with the same priority. Saxon, for example, gives reams of recoverable error messages, though it produces the result perfectly well:



```
Select C:\WINNT\System32\cmd.exe

C:\5946\Chapter04>saxon -o TUGuide.html TvGuide3.xml TUGuide13.xsl
Recoverable error
Ambiguous rule match for /TUGuide[1]/Channel[1]/Program[1]
Matches both "Program" on line 91 of file:/C:/5946/Chapter04/TUGuide13.xsl
and "Program" on line 32 of file:/C:/5946/Chapter04/TUGuide13.xsl
Recoverable error
Ambiguous rule match for /TUGuide[1]/Channel[1]/Program[2]
Matches both "Program" on line 91 of file:/C:/5946/Chapter04/TUGuide13.xsl
and "Program" on line 32 of file:/C:/5946/Chapter04/TUGuide13.xsl
Recoverable error
Ambiguous rule match for /TUGuide[1]/Channel[1]/Program[3]
Matches both "Program" on line 91 of file:/C:/5946/Chapter04/TUGuide13.xsl
and "Program" on line 32 of file:/C:/5946/Chapter04/TUGuide13.xsl
Recoverable error
```

It's often simpler to make templates whose match patterns don't include any information about the ancestry of the element, and it's easier for the processor too, because it doesn't have to check. For our TV guide stylesheet, we could adopt one of two styles to deal with elements that need to be treated differently in different places – add ancestry information for those templates that deal with elements in the descriptions, or add ancestry information to the other templates, those that deal with the bulk of the result. I think it makes more sense to keep ancestry information on the templates that deal with descriptions, because that way it's easy to tell which templates are those that deal with descriptions and which aren't. `TVGuide14.xsl` shows the result of doing that.

Processing with Push and Pull

Using templates as mapping rules really makes explicit the correspondence between a bit of the source XML and the result that you desire from it. As we've seen in `TVGuide14.xsl`, we can adopt this approach in data-oriented XML as well as document-oriented XML – you can have different templates matching different elements, even if those elements follow the traditionally 'data-oriented' pattern of just having element or text children. The approach that we were working with at the start of this chapter was quite different. There, we had very few templates (we started with just one!), and where we did use them it was really to make the stylesheet more manageable and to get reuse.

These two approaches to transformations with XSLT are termed **push** and **pull**.

Processing with Push

In the push approach, templates specify fairly low-level rules and the source XML document gets pushed through the stylesheet to be transformed on the way. Stylesheets that use the push approach tend to have a lot of templates, each containing a snippet of XML with an `<xsl:apply-templates>` instruction that moves the processing down to all an element's children. The final structure of the result is highly determined by the structure of the source.

Here's an example of a stylesheet, `TVGuide15.xsl`, which demonstrates a push approach. You can see how multiple templates are used to build up the result, but without knowing the structure of the source XML document it's hard to tell exactly what result you'll get (I've highlighted the main changes from `TVGuide14.xsl`, though it was actually quite push-like already):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <head>
        ...
      </head>
      <body>
        <h1>TV Guide</h1>
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>

  <xsl:template match="Channel/Name">
```

```

    <h2 class="channel"><xsl:apply-templates /></h2>
  </xsl:template>

  <xsl:template match="Program">
    <div>
      <p>
        <xsl:apply-templates select="Start" /><br />
        <xsl:apply-templates select="Series" /><br />
        <xsl:apply-templates select="Description" />
        <span onclick="toggle({Series}Cast);">[Cast]</span>
      </p>
      <div id="{Series}Cast" style="display: none;">
        <ul class="castlist">
          <xsl:apply-templates select="CastList" />
        </ul>
      </div>
    </div>
  </xsl:template>

  <xsl:template match="Start">
    <span class="date"><xsl:apply-templates /></span>
  </xsl:template>

  <xsl:template match="Series">
    <span class="title"><xsl:apply-templates /></span>
  </xsl:template>

  <xsl:template match="CastMember">
    <li><xsl:apply-templates /></li>
  </xsl:template>

  <xsl:template match="Character">
    <span class="character"><xsl:apply-templates /></span>
  </xsl:template>

  <xsl:template match="Actor">
    <span class="actor"><xsl:apply-templates /></span>
  </xsl:template>

  <xsl:template match="Character/Description" />

  <xsl:template match="Actor/Description" />

  <xsl:template match="Description//Character">
    <span class="character"><xsl:apply-templates /></span>
  </xsl:template>

  <xsl:template match="Description//Actor">
    <span class="actor"><xsl:apply-templates /></span>
  </xsl:template>

  <xsl:template match="Description//Link">
    <a href="{@href}"><xsl:apply-templates /></a>
  </xsl:template>

```

```
</xsl:template>

<xsl:template match="Description//Program">
  <span class="program"><xsl:apply-templates /></span>
</xsl:template>

<xsl:template match="Description//Series">
  <span class="series"><xsl:apply-templates /></span>
</xsl:template>

<xsl:template match="Description//Channel">
  <span class="channel"><xsl:apply-templates /></span>
</xsl:template>

</xsl:stylesheet>
```

Note that in this stylesheet I've used empty templates to do nothing with the <Description> elements within <Character> and <Actor> elements within the cast list. I apply templates to both the <Name> and <Description> children of these elements, but then ignore the <Description> element. This contrasts with a pull approach, which would only apply templates to the <Name> element in the first place.

Processing with Pull

In the pull approach, the stylesheet pulls in information from the source XML document to populate a template structure. Stylesheets that use the pull approach tend to have only a few templates and to use <xsl:for-each> and <xsl:value-of> to generate the result. The final structure of the result is mainly determined by the structure of the stylesheet and how the templates fit together.

Here's an example of a stylesheet that demonstrates a pull approach (actually, it's TVGuide2.xsl – the first stylesheet we used in this chapter, more or less). As you can see, there's only one template and its content follows the structure of the result that it generates very closely:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <head>
        ...
      </head>
      <body>
        <h1>TV Guide</h1>
        <xsl:for-each select="/TVGuide/Channel">
          <h2 class="channel"><xsl:value-of select="Name" /></h2>
          <xsl:for-each select="Program">
            <div>
              <p>
                <span class="date"><xsl:value-of select="Start" /></span>
                <br />
                <span class="title"><xsl:value-of select="Series" /></span>
```



```

        <br />
        <xsl:value-of select="Description" />
        <span onclick="toggle({Series}Cast);">[Cast]</span>
    </p>
    <div id="{Series}Cast" style="display: none;">
        <ul class="castlist">
            <xsl:for-each select="CastList/CastMember">
                <li>
                    <span class="character">
                        <xsl:value-of select="Character/Name" />
                    </span>
                    <span class="actor">
                        <xsl:value-of select="Actor/Name" />
                    </span>
                </li>
            </xsl:for-each>
        </ul>
    </div>
</div>
</xsl:for-each>
</xsl:for-each>
</body>
</html>
</xsl:template>

</xsl:stylesheet>

```

If you look carefully, you'll notice that I haven't even tried to style the content of the `<Description>` element in this stylesheet – it's very hard to process document-oriented XML with a pull style.

When to Use Push and Pull

Push and pull approaches both have their own advantages and disadvantages, and the best stylesheets use both approaches in tandem to process different parts of a particular XML document. You might use pull for the data-oriented parts of the XML and push for the document-oriented parts, for example. Here are some general guidelines about where to use push and where to use pull:

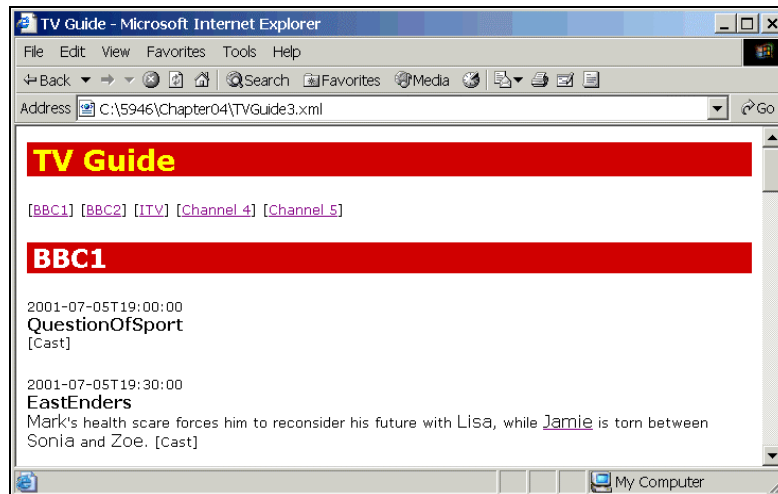
- ❑ Use multiple templates, each matching different types of nodes (a push approach) to process document-oriented XML.
- ❑ Select the nodes that you specifically want to process (a pull approach) to change the order in which the source is processed or to only process certain portions of the source. This is very common in data-oriented XML where the order in which information is contained in the source is not the same as the order in which it is required in the result.
- ❑ Apply templates to nodes, rather than getting their value directly (a push approach) to allow for extensibility in areas where the structure of the source XML, or the result that you want to generate from it, might change in the future.
- ❑ Access the values of nodes directly (a pull approach) when the structure of the source XML is fixed and you know exactly what result you want from it.

Taking these guidelines into account, `TVGuide16.xsl` gives a stylesheet that balances the push and pull approaches. It mainly uses a push style, but selects nodes directly for processing in certain places, notably to get only the `<Name>` child of `<Character>` and `<Actor>` elements and ignore the `<Description>`.

Using templates to match nodes is a push approach, which is good for document-oriented XML and for extensibility. Selecting nodes to process is a pull approach, which is good for changing the structure of a document.

Using Moded Templates

We've seen how to use separate templates to process different nodes in different ways. However, one thing that we might lose when we use templates like this is the ability to process the *same* node in different ways in different situations. For example, say we want to create a table of contents for the HTML page that we're generating, giving a list of the channels that the TV guide offers, as shown in the following screenshot:



The HTML underlying the above page is as follows:

```
...
<h1>TV Guide</h1>
<p>
  [<a href="#BBC1">BBC1</a>]
  [<a href="#BBC2">BBC2</a>]
  [<a href="#ITV">ITV</a>]
  [<a href="#Channel 4">Channel 4</a>]
  [<a href="#Channel 5">Channel 5</a>]
</p>
<h2 class="channel"><a name="BBC1" id="BBC1">BBC1</a></h2>
...
```

We can get the table of contents by processing the `<Channel>` elements, but we're already processing the `<Channel>` elements to get the lists of the programs available on each channel. We want to process the same `<Channel>` elements twice: once for their entry in the channel list and once to get their details.

This is fairly easy with `<xsl:for-each>` because the content of the particular `<xsl:for-each>` determines the result that you get. So we could do:

```
<h1>TV Guide</h1>
<p>
  <xsl:for-each select="TVGuide/Channel">
    [<a href="#{Name}"><xsl:value-of select="Name" /></a>]
  </xsl:for-each>
</p>
<xsl:apply-templates />
```

to get the entries in the channel list, and then either use another `<xsl:for-each>` or apply templates (as above) to the `<Channel>` elements to get their content later on.

However, you can also achieve this by using template **modes**. Modes allow you to process the same node with different templates in different situations. You can apply templates in a particular mode using the mode attribute on `<xsl:apply-templates>`, and you can define the mode for a template with the mode attribute on `<xsl:template>`. When you apply templates in a particular mode then the XSLT processor will only look at those templates with that mode.

With modes, then, you can apply templates to the same node in different modes to get different results. So in this case, we can use a template in `ChannelList` mode to generate the result for each channel in the channel list, as follows:

```
<xsl:template match="Channel" mode="ChannelList">
  [<a href="#{Name}"><xsl:value-of select="Name" /></a>]
</xsl:template>
```

This template will only match `<Channel>` elements if you apply templates in `ChannelList` mode. So we need to have an `<xsl:apply-templates>` element in the template for the root node that applies templates in `ChannelList` mode:

```
<h1>TV Guide</h1>
<p>
  <xsl:apply-templates select="TVGuide/Channel" mode="ChannelList" />
</p>
<xsl:apply-templates />
```

When you apply templates without setting the mode, then the processor uses templates that don't have a mode attribute.

You can use moded templates to get different processing for the same node in different situations.

Built-in Moded Templates

As you'll recall, when an XSLT processor can't find a template that matches a particular node then it will use a built-in template instead. There are similar built-in templates for moded templates, one for elements, which simply applies templates to their content in the same mode:

```
<xsl:template match="*" mode="any-mode">
  <xsl:apply-templates mode="any-mode" />
</xsl:template>
```

and another that matches text nodes in that mode and gives their value:

```
<xsl:template match="text()" mode="any-mode">
  <xsl:value-of select="." />
</xsl:template>
```

These built-in templates mean that you can get rid of superfluous templates and apply templates without explicitly specifying the nodes to which you're applying them in exactly the same way as you can with the default mode.

When applying templates in `ChannelList` mode, then, we don't have to specify the nodes to which we're applying templates and can just use:

```
<h1>TV Guide</h1>
<p>
  <xsl:apply-templates mode="ChannelList" />
</p>
<xsl:apply-templates />
```

The current node in the template (which matches the root node) is the root node, so the `<xsl:apply-templates>` in `ChannelList` mode with no `select` attribute will apply templates to the document element, the `<TVGuide>` element in `ChannelList` mode. There isn't a template for the `<TVGuide>` element in `ChannelList` mode, so the processor will apply the built-in moded template. This template selects the children of the `<TVGuide>` element, the `<Channel>` elements, and applies templates to them in `ChannelList` mode.

There are built-in moded templates in the same way as there are built-in normal templates.

Try It Out – Creating a Channel List

There are three things that we need to do to `TVGuide16.xsl` to create a linked list of channels in our page:

- 1.** Add anchors to the headings for the channels in the main body of the page
- 2.** Create a template in `ChannelList` mode to give the link to each channel
- 3.** Apply templates in `ChannelList` mode at the point at which the channel list should be given on the page

We'll make these three changes to create a new version of our stylesheet, `TVGuide17.xsl`.

You can add the anchors to the headings by changing the template for the `<Name>` element child of the `<Channel>` element to include an `<a>` element whose `name` and `id` attributes give the name of the channel:

```
<xsl:template match="Channel/Name">
  <h2 class="channel">
    <a name="{.}" id="{.}"><xsl:value-of select="." /></a>
  </h2>
</xsl:template>
```

The XHTML Recommendation advises that you use both the `name` and `id` attributes when creating anchors for backward and forward compatibility. We're not yet generating proper XHTML, but it's a good guideline to follow, as that's our final goal.

You can create the template for `<Channel>` elements in `ChannelList` mode to reference these links, as follows:

```
<xsl:template match="Channel" mode="ChannelList">
  [<a href="#{Name}"><xsl:value-of select="Name" /></a>]
</xsl:template>
```

Finally, you can apply templates in `ChannelList` mode in the template matching the root node to insert the channel list just under the title. The main content of the page comes after this channel list and is generated by applying templates in the normal mode:

```
<xsl:template match="/">
  <html>
    <head>
      ...
    </head>
    <body>
      <h1>TV Guide</h1>
      <p>
        <xsl:apply-templates mode="ChannelList" />
      </p>
      <xsl:apply-templates />
    </body>
  </html>
</xsl:template>
```

You could repeat the same list again at the bottom of the page very easily, by repeating the same instruction after the `<xsl:apply-templates>` that generates the main body of the page:

```
<xsl:template match="/">
  <html>
    <head>
      ...
```

```
</head>
<body>
  <h1>TV Guide</h1>
  <p>
    <xsl:apply-templates mode="ChannelList" />
  </p>
  <xsl:apply-templates />
  <p>
    <xsl:apply-templates mode="ChannelList" />
  </p>
</body>
</html>
</xsl:template>
```

This demonstrates one of the advantages of using templates over using `<xsl:for-each>` – you can reuse the same code by applying the same template. Transforming `TVGuide3.xml` with `TVGuide17.xsl` gives the display that we were aiming for, with a list of channel names at the top and the bottom of the page. Clicking on the channel name takes you to the program listing for that channel.

Summary

We've covered a lot of ground in this chapter. We've looked at full stylesheets for the first time – XML documents whose primary markup language is XSLT. You've learned how to convert from the starting point of a simplified stylesheet into a full stylesheet, by adding an `<xsl:stylesheet>` document element and an `<xsl:template>` element to give a template for the root node.

You've seen how an XSLT processor sees an XML document, as a node tree, and learned how to make an XSLT processor give you the output you want by telling it to apply templates to a bunch of nodes and providing the templates that it should use with them. You've discovered how to create templates that match various types of nodes:

- ☐ The root node
- ☐ Text nodes
- ☐ All element nodes
- ☐ Element nodes with particular names
- ☐ Element nodes with particular parents or ancestors

XSLT processors can only use one template to process a node when you apply templates to it. We've talked about how the processor deals with finding more than one template that matches a node by looking at the templates' priorities, and how it handles not finding one at all by using the built-in templates. You've also learned how to use modes to get the XSLT processor to use different templates in different situations.

Templates are the main constituent of a stylesheet, and the templates that you produce and the way you fit them together has a big effect on the stylesheet. You've now experienced the two main approaches used in stylesheets – push and pull – and we discussed the advantages and disadvantages of using each of them.

The XSLT that you've learned in this chapter hasn't much changed what you can generate from a stylesheet, just the way in which you get it. In the next chapter, we'll start looking at how to get a stylesheet to do more complicated processing dependent on the values of elements and attributes.

Review Questions

1. Turn the following simplified stylesheet into a full stylesheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="1.0">
  <head><title>Films</title></head>
  <body>
    <ul>
      <xsl:for-each select="/Films/Film">
        <li><xsl:value-of select="Name" /></li>
      </xsl:for-each>
    </ul>
  </body>
</html>
```

2. What is the term for the node at the top of the node tree? What relationship does it have to the document element?
3. Draw a node tree for the following XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="Films.xsl"?>
<Films>
  <Film rating="12">
    <Name>Crouching Tiger Hidden Dragon</Name>
    <Notes>
      Directed by <Director>Ang Lee</Director>.
    </Notes>
  </Film>
</Films>
```

4. Which template is the first template to be processed when you use a stylesheet with an entire document?
5. What kind of nodes does the following template process, and what does it do with them?

```
<xsl:template match="Description//Film">
  <a href="http://www.imdb.com/Find?for={.}">
    <xsl:value-of select="." />
  </a>
</xsl:template>
```

- 6.** What will be the result of applying the following stylesheet to a document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:apply-templates />
  </xsl:template>

</xsl:stylesheet>
```

- 7.** Which of the following templates will be applied to a <Film> element that's a child of a <Link> element that's a child of a <Description> element:

```
<xsl:template match="Description/Link/Film">...</xsl:template>
<xsl:template match="Film" priority="1">...</xsl:template>
<xsl:template match="Description//Film" priority="-1">...</xsl:template>
<xsl:template match="*">...</xsl:template>
<xsl:template match="Link/Film" mode="Description">...</xsl:template>
```

- 8.** Which of the templates given in the previous question will be applied to the <Film> element if it's selected with the instruction:

```
<xsl:apply-templates mode="Description" />
```