Schema-Aware Queries and Stylesheets

By: Dr. Michael Kay

In the previous articles in this series I gave a basic introduction to XQuery and a more detailed overview of what you can achieve using XQuery FLWOR expressions.

One of the most significant new features in the Stylus Studio 2006 edition is the introduction of support for handling schema-aware XSLT and XQuery. So in this article, that's going to be my focus. The examples in this article requires downloading and installing Stylus Studio 2006 edition, build 501f or later: this build includes Saxon 8.6.1, an XML schema-aware XSLT and XQuery processor.

XML Schema-awareness is an optional feature. Not every product will support it, and not every query or stylesheet will use it. You might consider it to be an advanced feature, which will only be needed by the most demanding applications. However, I'd like to persuade you that schema-awareness in XSLT and XQuery is something that you should consider using all the time. It's similar to the situation with XML itself: you can create and manipulate XML documents without ever validating them against a DTD or schema, but many experts will tell you that that's not good practice in anything other than a throwaway use-once application. Similarly, you can write queries and stylesheets that don't use schema-aware features, but I think that you're losing something by doing that.

"Queries and stylesheets" is a bit of a mouthful, and it's just one example of where XQuery and XSLT have different names for what's essentially the same concept. But I think it's worth covering both languages in the same article, because the way they handle schema-awareness is so similar. Also, I think all serious XML practitioners should have both tools in their kitbag. There are plenty of occasions when one tool is better than the other for a particular job, so sticking to one language alone is like going on the golf course with a single club.

What's the connection?

Where do queries/stylesheets and schemas fit together? Why should a transformation need to be schema-aware? I'll try and answer that question first on a theoretical level (but don't worry, there's no math). Later in the article, we'll see what this means in practice.

The role of type systems in programming language theory has a long and distinguished history. Ever since Fortran decided that variables beginning with I-N were integers while everything else was floating point, it's been recognized that different variables hold different kinds of value, and that it's a good thing if the programmer and the compiler both know clearly what kind of value is held in each variable. This gives several benefits. It means you can use the same symbol "+" to refer to integer addition or floating-point addition, while still allowing the compiler to know in advance which is intended (a property called *polymorphism*). It means you can get better error messages if you try to perform operations that don't make sense, like multiplying a date by an integer. And it means, in general, that a compiler can generate more efficient code, simply because it has more information to go on.

The key benefit, however, is error checking. If you've written stylesheets in XSLT 1.0, then you're probably painfully familiar with the fact that when you get your code wrong, the effects can be very baffling. Usually you get no error message telling you what you've done wrong, just "wrong" output, or no output at all. You then have to go through lengthy debugging procedures (greatly assisted by the XSLT debugger and XQuery debugger in Stylus Studio, of course, but still laborious) to trace the incorrect output to the actual source of the problem.

An Experiment: Spot the Error

In case you're not convinced, let's try a little experiment.

We'll use the same source data file as in my previous articles: the `videos.xml` file included in the Stylus Studio `examples/VideoCenter` folder. You'll see it listed if you open the Examples project.

Now here's one of the queries from my previous article, but with a deliberate error:

```
declare variable $input := doc('videos.xml')/*;

for $v in $input/videos/video
where $v/@year = 1999
return $v/title
```

Try running this under Stylus Studio. Open the Examples project. Select File/New/XQuery File, and paste in the above code. Configure Stylus Studio to use Saxon (version 8.6.1 or later) (select Tools/Options/XQuery/Processor Settings, then select Saxon 8.6.1 from the drop-down, and tick the box marked "Use as default processor"). Repeat for XSLT if you want to try the XSLT examples. Click on "Create Scenario", and in the box labelled "Main input (optional)" navigate your way to the `videos.xml` file in `examples/VideoCenter`, and select it. Then click the triangle icon to execute the query. It will prompt you to save the query as a file, and I suggest you call it "query1.xquery" and place it in the same directory as the `videos.xml` file.

Result? A blank screen.

If you prefer XSLT, it's just the same. Try this stylesheet. Select File/New/XSLT Text Editor, point the Source XML URL to the `videos.xml` file and paste in the following XSLT stylesheet code:

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xpath-default-namespace="http://www.stylusstudio.com/videos/ns">

<xsl:template match="/*">
   <xsl:for-each select="videos/video[@year = 1999]">
      <title><xsl:value-of select="title"/></title>
   </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Result? Another blank screen.

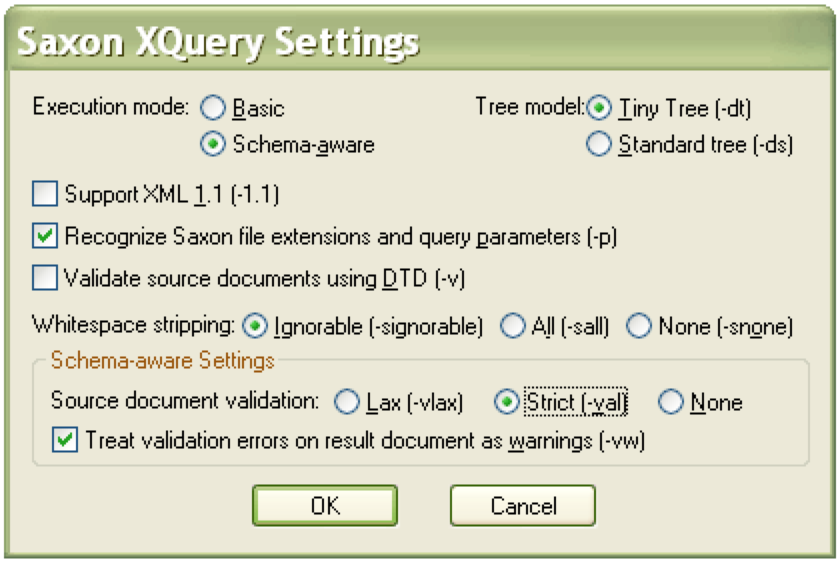
Now let's see what happens if we add a schema. I've written one for the purpose of this tutorial, and you'll find it in the Stylus Studio default project if you downloaded Stylus Studio after December 6, 2005, otherwise you can also download the example schema here. Actually, I didn't write it: Stylus Studio did. I simply used the "XML to XML Schema" wizard to generate the XML Schema. I made one or two small tweaks to add information that the wizard couldn't have guessed, for example that the year element is of type xs:gYear. I suggest you save the schema as videos.xsd into the same folder as the videos.xml file. Just for confidence, associate the data file with the schema file (XML/Associate XML With Schema) and validate it (XML/Validate Document) using your favorite schema validator just to make sure.

Now let's edit the query and stylesheet to make them schema aware. The query now looks like this:

```
import schema default element namespace "" at "videos.xsd";
declare variable $input as schema-element(result)
   := doc('videos.xml')/*;

for $v in $input/videos/video
where $v/@year = 1999
return $v/title
```

Open the scenario, select Processor, click the Settings button, and under "Schema Aware Settings / Source document validation" click the radio button labelled "Strict (-val)" as shown here:


Saxon XQuery Settings

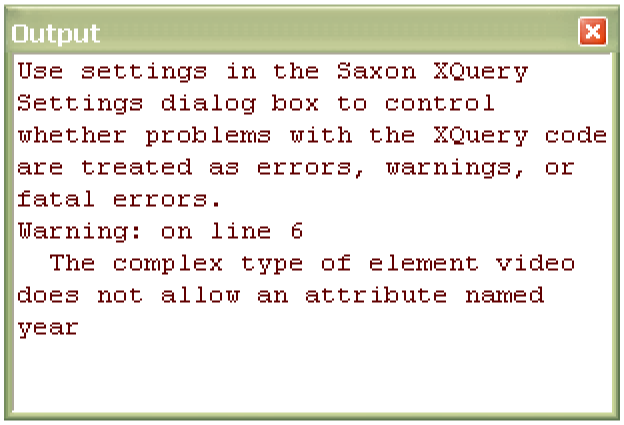
Execution mode: Basic [✓ Schema-aware]
Tree model: [✓ Tiny Tree (-dt)] Standard tree (-ds)

☐ Support XML 1.1 (-1.1)
☑ Recognize Saxon file extensions and query parameters (-p)
☐ Validate source documents using DTD (-v)

Whitespace stripping: [✓ Ignorable (-signorable)] All (-sall) None (-snone)

Schema-aware Settings
Source document validation: Lax (-vlax) [✓ Strict (-val)] None
☑ Treat validation errors on result document as warnings (-vw)

When you run this, you should get a pop-up with the message:



```
Use settings in the Saxon XQuery
Settings dialog box to control
whether problems with the XQuery code
are treated as errors, warnings, or
fatal errors.
Warning: on line 6
  The complex type of element video
does not allow an attribute named
year
```

It's pinpointed the deliberate error in the query: year is an element, not an attribute.

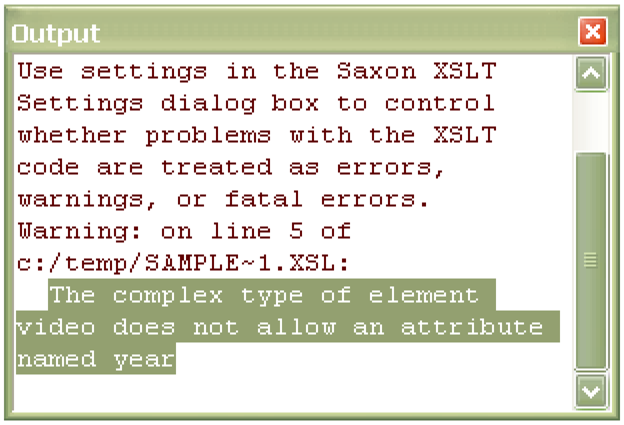
In XSLT we can do the same thing. The schema-aware version of the stylesheet looks like this:

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:import-schema schema-location="videos.xsd"/>
<xsl:template match="schema-element(result)">
   <xsl:for-each select="videos/video[@year = 1999]">
      <title><xsl:value-of select="title"/></title>
   </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

and we get essentially the same error message:

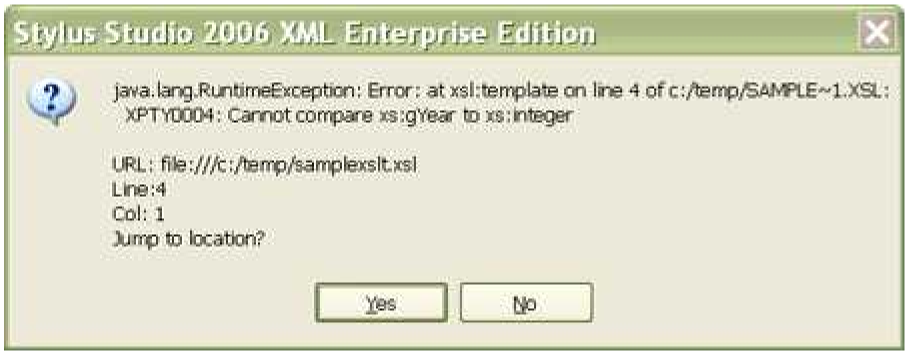


```
Use settings in the Saxon XSLT
Settings dialog box to control
whether problems with the XSLT
code are treated as errors,
warnings, or fatal errors.
Warning: on line 5 of
c:/temp/SAMPLE~1.XSL:
  The complex type of element
video does not allow an attribute
named year
```

You might be wondering why this is a warning rather than an error. The answer is that it's not actually an error to ask for an attribute that isn't there. The query is well defined, it just doesn't produce the output that we wanted. There might be good reasons for attempting to read an attribute that the schema doesn't allow, for example you might be anticipating a change in the next version of the schema. In XQuery there's an option (strict static typing) that does more draconian checks of your query and makes almost everything that might possibly be wrong a fatal error, but Saxon-SA doesn't currently implement this option.

The experiment doesn't end here. Let's see what happens when we correct the error, changing "@year" to "year". You can do this with either the XQuery or the XSLT version.

Whether we use the XQuery version or the XSLT version, we now get the error: Cannot compare `xs:gYear` to `xs:integer`



```
java.lang.RuntimeException: Error: at xsl:template on line 4 of c:/temp/SAMPLE~1.XSL:
  XPTY0004: Cannot compare xs:gYear to xs:integer

URL: file:///c:/temp/samplexslt.xsl
Line:4
Col: 1
Jump to location?
```

To correct this we have to change the where condition to `($v/year = xs:gYear('1999'))`.

You have to get used to this kind of error message when doing schema-aware processing. Sometimes the message identifies a real error, for example comparing an integer to a date or adding two `xs:gYear` values. Sometimes it's more frustrating, because you feel the system should be capable of comparing a year to an integer. This is the price you pay for a system that tries its hardest to catch all your errors: occasionally it tries just a little bit too hard. But since the comparison can only be done by converting the integer to a year (or the year to an integer) it's no hardship to do it explicitly.

The Three Aspects of Schema-Awareness

We've seen an example of schema-awareness in action: now let's stand back and see what's going on. There are three ways in which you can exploit schema-awareness:

- You can refer to elements, attributes, and types defined in a schema, provided that you import the schema into your stylesheet or query
- You can validate input documents, and take advantage of the fact that this attaches type information to the nodes in an input document
- You can validate output documents, getting explicit error messages if you try to produce a document that isn't valid according to the target schema.

In the rest of the article we'll look at each of these in turn.

Importing Schemas and declaring Types

Importing a schema simply makes the XQuery or XSLT processor aware at compile-time of the definitions contained in your schema: specifically, the top-level element and attribute declarations, and the named simple and complex type definitions. We've already seen an example of the syntax for importing a schema in both XSLT and XQuery. There are a couple of differences to be aware of:

- In XQuery you need to import a schema separately into each module where it is used. In XSLT, importing a schema in one module makes the definitions available throughout all modules of the stylesheet.
- XSLT also allows you to define a schema inline, within the stylesheet. This is useful if you want to declare types for local use within the stylesheet, for example a schema that describes the structure of intermediate results. This is possible because XSLT and XML Schema both use an XML-based syntax.

There are various places where you can refer to these definitions, for example you can use them to define the types of variables or of function parameters, and you can test whether a particular value conforms to a given type using an `instance of` test. Most of these constructs make use of a piece of syntax called a `SequenceType`. The table below shows the more common forms of `SequenceType` you are likely to use, and explains their meaning:

Syntax	Matches
`element(xyz)`	Any element named `xyz`. This form doesn't require `xyz` to be defined in an imported schema: in fact it doesn't require schema-awareness at all, but I've included it for completeness
`schema-element(xyz)`	Any element whose name is `xyz`, or whose name is the same as an element in the schema-defined substitution group of `xyz`. In this case `xyz` must be the name of a global element definition in an imported schema, and the element being tested must have been validated against the type defined for `xyz` in the schema.
`element(*, mytype)`	Any element that has been validated against the schema-defined type `mytype`. This may be a simple type or a global type, and unless it is one of the built-in types such as `xs:integer`, it must be defined in an imported schema.
`element(xyz, mytype)`	This SequenceType matches any element that satisfies both `element(xyz)` and `element(*, mytype)`

`attribute(abc)`
`schema-attribute(abc)`
`attribute(*, mytype)`
`attribute(abc, mytype)`
	These SequenceTypes match attributes in much the same way as the corresponding `element()` variants. However, it's unusual for schemas to define many top-level attribute definitions so some of these forms are rarely used. Also, of course, there's no equivalent of substitution groups in the case of attributes.

Atomic type name, for example: `xs:integer`	An atomic type name can be used as a SequenceType on its own, and matches any atomic value of that type: for example `xs:integer` matches an integer. The type name can either be a

`my:part-number`	built-in name such as `xs:integer` or `xs:date`, or a user-defined atomic type in an imported schema

Any of these constructs can be followed by an occurrence indicator, which indicates how many instances of the relevant item may occur within a value. The occurrence indicators are shown in the next table:

Occurrence indicator	Meaning
?	zero or one
+	one or more
*	zero or more
(absent)	exactly one

So, for example, `element(foo)*` matches a sequence of zero or more foo elements, while `xs:date?` matches an optional date.

We've seen a couple of examples of SequenceType constructs in the experiment we did at the start. In the XQuery example, we declared the type of a variable:

```
declare variable $input as schema-element(result)
   := doc('videos.xml')/*;
```

while in XSLT, we used a SequenceType in a match pattern to indicate which nodes a template rule should match:

```
<xsl:template match="schema-element(result)">
```

Both these constructs had the effect of telling the compiler what kind of value to expect. In the XQuery case the type declaration also causes an error message if the input doesn't conform to the required type. In the XSLT case, input that doesn't match this type will simply cause this template rule not to be selected.

Both languages allow you to use SequenceTypes to declare the types of function parameters and results. This is one of the most important uses of type declarations, so we'll examine it in some detail in the next section.

Writing Schema-Aware Functions

Both XQuery and XSLT 2.0 allow you to write libraries of functions, which you can invoke from within XPath expressions. Such functions can be extremely useful to hide some of the complexity of real-life schemas — the kind of schemas like XBRL and FpML which try to cater for all possible varieties of transaction across a whole industry, and which therefore have an immense amount of generality built in. The difficulty with such schemas is that it's very hard to remember the names of all the elements, let alone their detailed nesting.

(Don't panic if you don't know what XBRL and FpML are: I won't even expand the acronyms. They are two examples of standardized XML vocabularies designed to move data within and between organizations, and like many such vocabularies, they contain many hundreds of different element definitions.)

A well-designed library of functions can hide this complexity: for example, it allows the person who wants to select all trades worth more than $1m to write `[f:dollar-value($trade) gt 1000000]` rather than something like `[$trade/trade/tradeHeader/partyTradeInformation/revenue/event/payment/paymentAmount[currency='USD' and amount gt 1000000]]`. The people who write the function library need to understand the underlying complexity; those who write the queries don't.

(If you know FpML, you will probably be bristling to tell me that this is all wrong. OK, I admit it, I'm simplifying grossly. Real life is even worse.)

Both XSLT and XQuery allow you to write functions without declaring the types of the parameters and result, but when you're handling such complex data this really isn't a good idea, because people are very likely to pass the wrong input when calling the function (for example, the wrong start node for the selection) and the result will be garbage output and general incomprehension. Here's how you might write this function in a schema-aware query:

```
declare function f:dollar-value(
   $in as element(*, fpml:tradeRequest))
as xs:decimal {
   $in/trade/tradeHeader/
      partyTradeInformation/revenue/event/
         payment/paymentAmount[currency='USD']/amount
}
```

In XSLT 2.0 the function is exactly the same, except for the surface syntax:

```
<xsl:function name="f:dollar-value" as="xs:decimal">
   <xsl:param name="in" as="element(*, fpml:tradeRequest)"/>
   <xsl:sequence select="
   $in/trade/tradeHeader/
      partyTradeInformation/revenue/event/
         payment/paymentAmount[currency='USD']/amount"/>
</xsl:function>
```

Let's go back to a simpler schema, the one for the videos. There are still many useful functions we can write to make it easier to get the data. Here's one that gets the first name of an actor, which I'll write in XQuery:

```
declare function f:first-name(
   $in as schema-element(actor))
as xs:string {
   normalize-space(substring-after($in, ','))
}
```

And here's one that finds all the videos featuring a particular actor. To keep you on your toes, this one uses XSLT syntax:

```
<xsl:function name="f:videos-for-actor" as="schema-element(video)*">
   <xsl:param name="actor" as="schema-element(actor)"/>
   <xsl:sequence select="$actor/(//video[actorRef=$actor/@id])"/>
</xsl:function>
```

Notice here that the result is a sequence whose items must all be video elements: the "as" attribute defining the return type uses the occurrence indicator "*".

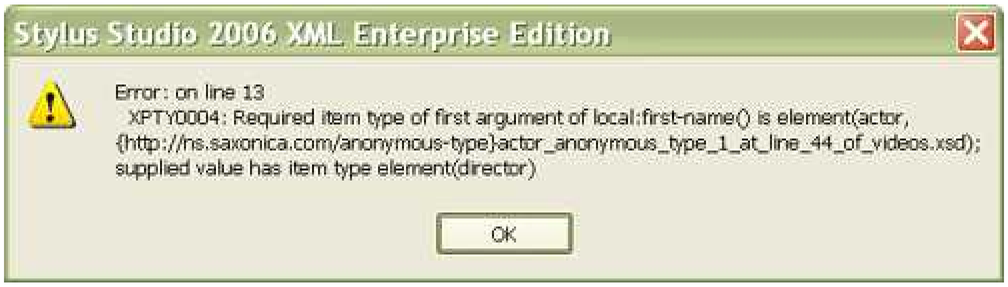
Because the types of the arguments are known, you'll get an error if you call the function incorrectly. Here's a complete query that declares the first function above and then tries to call it to get the first name of a director:

```
import schema default element namespace "" at "videos.xsd";
declare variable $input as schema-element(result) := doc('videos.xml')/*;

declare function local:first-name(
   $in as schema-element(actor))
as xs:string {
   normalize-space(substring-after($in, ','))
};

for $v in $input/videos/video
where $v/title="The Fugitive"
return local:first-name($v/director)
```

Saxon throws this out at compile time, with the rather convoluted message:



```
Error: on line 13
  XPTY0004: Required item type of first argument of local:first-name() is element(actor,
{http://ns.saxonica.com/anonymous-type}actor_anonymous_type_1_at_line_44_of_videos.xsd);
supplied value has item type element(director)
```

Error on line 13 of file:/c:/temp/videos/query3.xq:
XPTY0004: Required item type of first argument of local:first-name() is element(actor, {http://ns.saxonica.com/anonymous-type}
actor_anonymous_type_1_at_line_44_of_videos.xsd); supplied value has item type element(director)

Believe it or not, it's worth trying to understand this error message:

- The first line tells you where the error occurred: always useful
- The second line gives you an error code XPTY0004 which could possibly be useful if you want to test for different error codes in your calling application, or to find how the error is described in the XQuery language specification
- The message that starts on the second line has a general form which you will see quite often: Required type of XXX is RRR; supplied value has type TTT
- XXX gives you the context: in this case, it's the first argument of the call to the user-written function local:first-name() that's at fault
- RRR tells you what type was required, in a rather unwieldy internal form. Saxon has expanded the SequenceType schema-element(actor) into an element() test using an internally-generated name for the anonymous type of the actor element. The namespaces are spelt out in full because getting the namespace wrong is a common cause of type errors
- TTT tells you the type of the value that was supplied, in this case an element named "director". The error occurs because this type doesn't match the required type RRR.

What would happen if you didn't declare the required type of the function argument? Try it and see (just remove the "as" clause). Blank output again! Declaring the required types is optional, but I really wouldn't recommend it: it's a short-cut that you will later regret.

Let's do the same exercise with our XSLT function. Here's a complete stylesheet that calls the function incorrectly:

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:f="http://localhost/functions/">
```

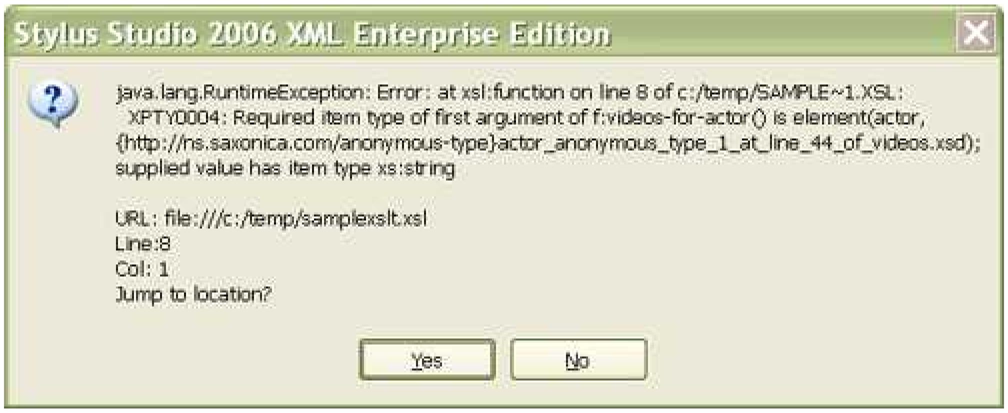
```
<xsl:import-schema schema-location="videos.xsd"/>

<xsl:function name="f:videos-for-actor" as="schema-element(video)*">
   <xsl:param name="actor" as="schema-element(actor)"/>
   <xsl:sequence select="$actor/(//video[actorRef=$actor/@id])"/>
</xsl:function>

<xsl:template match="/">
   <videos for="David Schwimmer">
      <xsl:for-each select="f:videos-for-actor('Schwimmer, David')">
         <title><xsl:value-of select="title"/></title>
      </xsl:for-each>
   </videos>
</xsl:template>

</xsl:stylesheet>
```

Can you spot the error? Perhaps the error message will help:



```
java.lang.RuntimeException: Error: at xsl:function on line 8 of c:/temp/SAMPLE~1.XSL:
  XPTY0004: Required item type of first argument of f:videos-for-actor() is element(actor,
{http://ns.saxonica.com/anonymous-type}actor_anonymous_type_1_at_line_44_of_videos.xsd);
supplied value has item type xs:string

URL: file:///c:/temp/samplexslt.xsl
Line:8
Col: 1
Jump to location?
```

In other words, the function was written to expect an actor *element*, but we called it supplying an actor's *name* as a string. In this case, as it happens, we would have got an error even with the non-schema-aware product, because you can't use a string on the left-hand-side of the "/" operator. But the error with an explicit declaration of the parameter type is much more precise.

We said we'd look at three aspects of schema-awareness, and we've finished the first: importing schemas and declaring types. We'll now look at input validation and output validation.

Validating Input

In all the examples in the previous section, the input to the stylesheet or query was an XML document (`videos.xml`) that had been validated against a schema. If you forgot to tick the checkbox asking for strict input validation, you'll have seen what happened: the input doesn't match the types declared in the query/stylesheet.

It's important here to distinguish input that is valid from input that has been validated. If you write a function that expects an argument declared as `schema-element(actor)`, then it's not enough that the element you pass in should be valid according to the schema definition: it must be certified as valid, and to achieve that, it must have already gone through the validation process.

The same is true for atomic values. It might come as a surprise that when you declare a function `f:sqrt()` as expecting an argument of type `xs:positiveInteger`, the call `f:sqrt(5)` should fail on the grounds that 5 is not a positive integer! The point is that using a value in the function call doesn't cause it to be validated against the type definition; the value must already be labelled with the right type, and that means the caller has to ensure that the validation has been done in advance. In the case of atomic values you can do this simply with a constructor function: `f:sqrt(xs:positiveInteger(5))`.

Validating the input to the query or stylesheet is normally controlled using options set in the calling environment. This applies whether the input is supplied as the context document, or the value of a parameter, or whether it's read using the `doc()` or `document()` function. In Stylus Studio, as you've already discovered, you have to set the right options in the *Scenario*. If you call Saxon directly from the command line, there's a switch (-val) that has the same effect. There are equivalent options you can set if you invoke a transformation or query from a Java application using the Saxon API.

Because validation of the input happens outside the control of the language itself, the specs are a little vague as to how the system goes about finding a schema to do the validation. In Saxon, and probably in most other products, there's an internal cache of schema definitions that are used, and you can load schemas into this cache in various ways: by using import schema, by using `xsi:schemaLocation` in an instance document, or by using the Java API directly. An important point to note, however, is that the cache can only hold one schema for each namespace. This means that if you want to write code to transform data from version N to version N+1 of the same schema, you can't have both the input and output validated.

If you don't validate the input, then the document is said to be *untyped*. All the elements are marked as having the type `xdt:untyped`, while the attributes are marked as xdt:untypedAtomic. (The namespace prefix xdt here refers to a namespace defined in the XSLT/XQuery specifications.) In many ways these are just types like any other, for example you could write a function that actually requires the input to be untyped. There's one important difference, however: unlike any other type, untypedAtomic data is automatically converted to the required type when passed to a function or operator that expects some specific type. For example, if the function f:sqrt() declares that it expects an xs:positiveInteger and you supply the unvalidated element <a>1920</a>, then the system will happily calculate the square root of 1920. Contrast this with what happens when <a> is validated against a schema that declares its type as xs:gYear: in this case you will be told that an xs:gYear cannot be used where an xs:positiveInteger is expected. (Perhaps you're starting to see the point of that irritating error we hit earlier.) However, although untyped atomic values will automatically be validated and converted to the required type, the same isn't true of untyped elements: you have to validate these explicitly.

Remember: if you want the system to spot errors in path expressions, as we saw at the beginning of this article, then you need to declare the types of your variables and functions. And if you declare the types, then the input has to satisfy those types, and it will only do so if it has been validated on the way in.

Validating Output

Let's now look at how you can write stylesheets and queries that validate their output. This is one of the most effective ways of exploiting schema-awareness, and it can make a dramatic difference to the way in which you tackle development. Let's see it in action.

It's quite common for stylesheets and queries to produce XHTML as their output, so we'll use that as our example. Let's try to produce an alphabetical list of actors and the videos they appear in. Here's our first attempt in XSLT:

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html xmlns="http://www.w3.org/1999/xhtml">
      <xsl:call-template name="header"/>
      <body>
         <h1>Index of Actors</h1>
         <table>
            <xsl:for-each select="//actor">
               <xsl:sort select="."/>
               <tr>
                  <td><xsl:value-of select="."/></td>
                  <td>
                     <xsl:for-each select="//video[actorRef=current()/@id]">
                        <xsl:value-of select="title"/>
                        <xsl:if test="position() != last()"><br/></xsl:if>
                     </xsl:for-each>
                  </td>
               </tr>
            </xsl:for-each>
         </table>
      </body>
   </html>
</xsl:template>

<xsl:template name="header">
   <head><title>Index of Actors</title></head>
</xsl:template>

</xsl:stylesheet>
```

Run this, and it works. You can view the output in Firefox or Internet Explorer. (It's not pretty, but a bit of CSS code would quickly fix that.) But is it XHTML? Try submitting the XSLT output to the official W3C XHTML validation service, and you'll find that it isn't.

Now stop and think about the way you develop stylesheets. How do you check that the output is correct? Do you simply view it in a browser or two, or do you actually validate it? And if you do validate it, what happens when you see a message like this:

```
Validation error on line 2 column 19 of file:/c:/temp/videos/out.html:
In content of element <html>: The content model does not allow element <head> to appear
here. Expected: {http://www.w3.org/1999/xhtml}head
```

You have to open up the (alleged) XHTML in a text editor, find the relevant line and column, work out why it's invalid (in this case, the <head> element is in the wrong namespace), and then find the code in the stylesheet that generated the <head> element and correct it. If your HTML contains thousands of lines of unindented code, and your stylesheet is equally complex, that's not an enviable task.

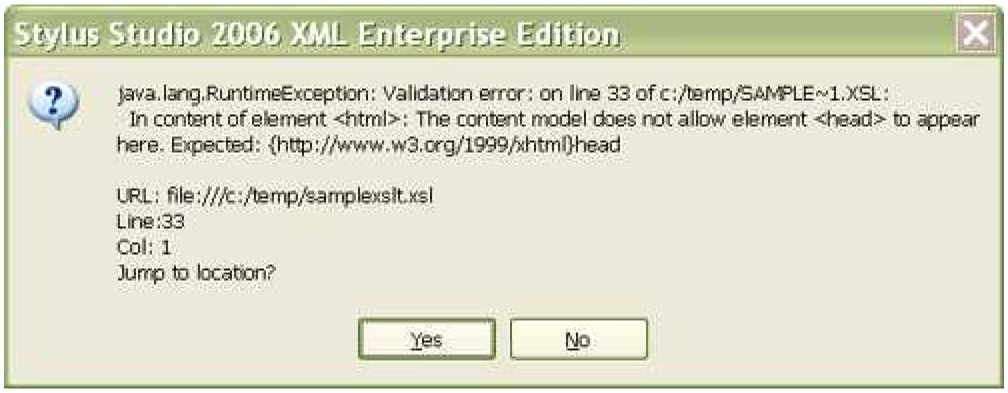
The good news is that there is an easier way. Change the <html> element in the stylesheet to say

```
<html xmlns="http://www.w3.org/1999/xhtml" xsl:validation="strict">
```

and add after the `xsl:stylesheet` element:

```
<xsl:import-schema namespace="http://www.w3.org/1999/xhtml"
   schema-location="http://www.w3.org/2002/08/xhtml/xhtml1-transitional.xsd"/>
```

Now try running it again (you'll need to be online, because Saxon is going to fetch that schema from the web). You'll need to associate the videos.xml file with the videos.xsd file if you haven't done so already. This is the output:



```
java.lang.RuntimeException: Validation error: on line 33 of c:/temp/SAMPLE~1.XSL:
  In content of element <html>: The content model does not allow element <head> to appear
here. Expected: {http://www.w3.org/1999/xhtml}head

URL: file:///c:/temp/samplexslt.xsl
Line:33
Col: 1
Jump to location?
```

OK, you already knew it was invalid — but only because I persuaded you to check the XHTML. Would you have done it if I hadn't asked you to? And note the subtle difference from the earlier error message: this time, it's not telling you that line 2 in the XHTML is wrong, it's telling you that there's an error at line 33 in the stylesheet. So you can go straight there, and replace the incorrect code by:

```
<xsl:template name="header">
   <head xmlns="http://www.w3.org/1999/xhtml"><title>Index of
Actors</title></head>
</xsl:template>
```

(There's a checkbox in the processor options for Saxon labelled "Treat validation errors on result document as warnings". The output shown above is what happens when you leave this unchecked. If you check the option, Saxon will produce the output as requested, interspersed with comments telling you where it is invalid. This can be useful if you want to get the broad outline of your code right first, and sort out the detail later.)

This was all done in XSLT, but it works just as well for XQuery. In case you need convincing, here's a query that (incorrectly) attempts to copy the record for a selected video, while adding one to the value of vhs_stock:

```
let $v := //video[@id="id1256990"]
return
   <video>
      {$v/@*,
         for $e in $v/*
         return
            if ($e instance of element(vhs_stock))
            then $e + 1
            else $e
      }
   </video>
```

As with the XSLT example, it runs without errors, so you might be misled into thinking that all is well. Now change it to validate the output:

```
import schema default element namespace "" at "videos.xsd";
let $v := //video[@id="id1256990"]
return
validate {
      <video>
         {$v/@*,
            for $e in $v/*
            return
               if ($e instance of element(vhs_stock))
               then $e + 1
               else $e
         }
   </video>
}
```

and the error becomes apparent (again, note how the message pinpoints its location):

```
Validation error on line 10 of file:/c:/temp/videos/query6.xq:
The content model for element <video> does not allow character content
```

What has happened here is that the "else" branch $e copies the element $e to the output, while the "then" branch atomizes the element, adds one, and then writes a number to the output, with no containing element tags. To correct the query you need to write then `<vhs_stock>{$e + 1}</vhs_stock>`.

In my own work I have found that validating the output of a query or stylesheet gives a dramatic improvement in the ability to quickly detect and diagnose a great number of coding errors. The more complex the XML you are working with, the more useful it becomes.

I've talked here about validating the input and output of a stylesheet or query. But of course you can also validate intermediate results. When you write a multi-phase transformation, the output of one phase is the input to the next, so validating the intermediate data kills both birds with one stone. You might feel it's rather an effort to write a schema that describes such transient data: but if the query or stylesheet is forms part of an operational system that someone else will have to maintain, then the effort will probably pay off.

Schema Aware XSLT and XQuery Processing Summary

I hope you've got this far, and I hope you've tried running the examples in Stylus Studio. If so, I hope I will have convinced you of the advantages you get from making your queries and stylesheets schema-aware.

I outlined the three aspects of schema-awareness in XQuery and XSLT: the ability to import a schema and thus to declare the types of your variables, parameters, and results; the ability to validate input documents; and the ability to validate the output on the fly. This all adds up to a considerable improvement in your ability to diagnose and correct errors, and in the long run to an increase in the robustness of your finished code. It might feel tedious at first having to define all your data types, but you'll quickly find that it becomes second-nature, and when you come to write the coding standards for your organization you'll probably want to make it mandatory good practice.

If you want to know more about schema awareness, Chapter 4 of my book XSLT Programmer's Reference goes into more detail, and it's probably worth reading even if you intend to use XQuery most of the time. Watch out for Priscilla Walmsley's forthcoming book on XQuery, too: I haven't seen it yet, but I'm sure she'll do the topic justice.