Programming in o:XML

Martin Klang

Programming in o:XML

Martin Klang

Table of Contents

List of Examples

Introduction

This is a programming guide for the o:XML programming language. It assumes you have an understanding of XML and some programming experience, preferably with an object-oriented language. Some knowledge about the XPath Query Language and/or XSLT will also be useful. The guide starts with some basic concepts, then moves on to more advanced topics.

Chapter 1. Producing Nodes

The easiest way to produce elements, text and other XML nodes in o:XML is to include them as literal nodes in the program (remember that an o:XML program is also an XML document). However, sometimes you want to produce nodes programmatically, for which purpose there are specific commands that you can use.

Elements are created with the **element** element, which requires at least a *name* attribute and optionally *attributes*. You can also add attributes with repeated **attribute** elements.

Example 1.1. Creating Elements

```
<o:element name="bean">
  <o:attribute name="name" select="'Illy'"/>
  <o:attribute name="type">coffee</o:attribute>
  this is a coffee bean
</o:element>
```

will produce the nodes: `<bean name="Illy" type="coffee">this is a coffee bean</bean>`

To programmatically create a programming instruction, you need to specify the target and value to a **processing-instruction** element.

For example, to produce the programming instruction <?xml-stylesheet href="style.xsl" type="text/xsl"?> you could write (note the escaped quotes): <o:processing-instruction target="xml-stylesheet" select="'href="style.xsl" type="text/xsl"'"/>, or alternatively put the processing instruction value in the element value:

Example 1.2. Creating Processing Instructions

```
<o:processing-instruction target="xml-stylesheet">href="style.xsl" type="text/xsl"</o
```

Text and comments are even easier, you only need to specify a value as either the element value or with the *select* attribute:

Example 1.3. Creating Comments and Literals

```
<o:comment select="'i am commenting this'"/>
<o:text>  leading and trailing whitespace might otherwise have been removed  </o:text
```

Often you will want to output the value of an expression. This can simply be inserted at any point in the output with **eval** (similar to **copy-of** in XSLT).

Example 1.4. Dynamic Content

```
<example>here is some mixed static and <o:eval select="$dynamic"/> content.</example>
```

The expressions used can evaluate to text, elements, or any other node type which will simply be appended to the output.

> **Note**
> Nodes of all types can also be created using the type constructor functions - see Chapter 6, *Basic Types*.

Chapter 2. Log Output

ObjectBox, the o:XML engine, has built-in support for runtime logging. This valuable development tool can be used to trace a programs execution and state, and is very easy to use.

The **log** element takes either a *select* attribute that will log the value of an o:Path expression, or *msg* which will output text or attribute-escaped, mixed content - anything contained in curly braces will be evaluated as an o:Path expression. For example:

Example 2.1. Logging

```
<o:log msg="the string value of $i is: {$i}."/>
<o:log level="info">
  <info>the value of $i is: <o:eval select="$i"/></info>
</o:log>
```

The second example shows how you can specify which log level, or severity, that the message has. The default log level is “info”, other valid levels are “debug” (for more verbose output), “warning” and “error” (for less log output).

Log output will not affect the program result, however beware of using o:Path expressions, ie functions, that change the program state.

Chapter 3. Using Variables

A programming language would not be complete without variables in one form or another. In o:XML variables are declared and referenced in the same way as in XSLT but with the difference that they can also be reassigned.

To use a variable first you have to declare it with the **variable** or **set** element. Once declared it can be referenced and changed (reassigned) in any sibling or sibling child node - outside of this scope the variable is not defined!

The variable declaration can also assign a value at the same time, with either an expression in the *select* attribute or to whatever child nodes it has.

Anywhere the variable is in scope its value can be changed by using either **variable** or the **set** element. Reassignement is done in the same as declaring a variable, but with **set** you can assign more than one variable at a time.

Here's an example that declares a couple of variables and changes a variable value.

Example 3.1. Using Variables

```
<o:variable name="title" select="'Lissie'"/>
<o:variable name="channel">
  <channel>
    <title><o:eval select="$title"/></title>
  </channel>
</o:variable>
<o:set title="'Bessie'"/>
```

It is important to recognise that in o:XML the variables in themselves don't have types, even though variable values do - a variable is an untyped reference to an object. This means for instance that a variable with string value can be reassigned a nodeset value without changing or casting any objects or object types.

Chapter 4. Conditionals

o:XML supports not only **if** and **choose** but also conditional looping with **while**. This chapter looks closer at how conditionals can be used to express logic and control program flow in o:XML.

The simplest conditional is **if**, which simply states that if its *test* expression cannot be evaluated to boolean `true` then its child nodes will be ignored.

Example 4.1. If-statement

```
<o:if test="1 = 0">never!</o:if>
<o:if test="1 > 0">always!</o:if>
```

will produce the text “always!” on computers that think that 1 is larger than 0

For multiple conditionals, such as if/then or if/else statements, you will need to use **choose**. The child element **when** can be repeated any number of times, however only the first one that evaluates to `true` will be considered. As a fallback or catch-all, you may specify in the optional **otherwise** child element what should happen if none of the **when** tests `true`. Example:

Example 4.2. Choose-statement

```
<o:choose>
  <o:when test="1 = 0">
    <o:log msg="we know this will never happen"/>
  </o:when>
  <o:when test="$i = 0">
    <o:log msg="this might happen"/>
  </o:when>
  <o:otherwise>
    <o:log msg="otherwise this will happen"/>
  </o:otherwise>
</o:choose>
```

So what does this mean? Only one of these log messages will ever be printed, which one depends on the value of the variable `$i`. If you provide an **otherwise** element, then exactly one block will execute. Otherwise at most one - the one designated by the first **when** element to hold true - will run.

If you want something to be not only conditionally executed but also repeated for as long as the condition holds true, use **while** just like you would use **if**, though beware of infinite loops. For example, the following fragment will produce `yes` elements until the end of time (or until you get bored and stop the program, whichever comes first):

Example 4.3. While-statement

```
<o:while test="true()">
  <yes/>
</o:while>
```

Chapter 5. Looping the Loop

Iteration is achieved in o:XML, as in XSLT, with a **for-each** statement. In contrast with XSLT however, there are three ways to specify what objects to iterate over. If you want to iterate over a nodeset, use the attribute *select*. If you want to iterate over say all even numbers from 2 to 10, use *from*, *to* and *step*. If you want to do process each of the words in the phrase "from start to finish" use *in* (and optionally *delim*), which works similarly to a shell script loop in that it takes a string and breaks it into tokens at certain delimeter characters.

However you declare the iterated nodes, you may also specify which variable they should be assigned by using the *name* attribute. If no variable name is given, the context node will be assigned instead (you can always get the current context node with the `current` function). See these examples:

Example 5.1. Iteration over nodeset

```
<o:variable name="planets">
  <pluto/>
  <jupiter/>
  <mercurius/>
</o:variable>
<o:for-each name="planet" select="$planets">
  Kelly sees <o:eval select="name($planet)"/>.
</o:for-each>
```

Example 5.2. Iteration over numbers

```
<o:variable name="sum" select="0"/>
<o:for-each to="10">
  <o:set sum="$sum + current()"/>
</o:for-each>
<o:eval select="$sum"/>
```

produces the number 45 (0+1+2+3+4+5+6+7+8+9), because the default value of *from* is 0, and the default of *step* is 1. Note that only the lower limit of the range, denoted by *from*, is included.

Example 5.3. Iteration over string tokens

```
<o:for-each name="word" in="comma,separated,values" delim=",">
  <!-- the default value of 'delim' is space, ' ' -->
  <o:element name="{$word}"/>
</o:for-each>
```

produces the elements <comma/><separated/><values/>

Chapter 6. Basic Types

In o:XML, things like elements, attributes and text are not just markup but also objects that you can invoke functions on. The basic types of o:XML have their own constructor functions, so you can create nodes in o:Path expressions. Other useful type functions include conversion to string, number and boolean values, and node manipulation functions (see Chapter 7, *Node Manipulation*).

Example 6.1. Node Functions

```
<o:variable name="stock">
  <product name="stereo">
    <price>250.00</price>
    <condition>flawless</condition>
  </product>
</o:variable>

<!-- output the price plus VAT -->
<o:eval select="$stock/product/price.number() * 1.175"/>

<!-- create an element that has the name of the product -->
<o:variable name="stereo" select="Element($stock/product/@name)"/>
```

Chapter 7. Node Manipulation

Generally in a program you will want to not only create nodes dynamically, but also change the nodesets you're working with. The basic o:XML types provide an interface for doing this directly on the nodes themselves. The functions available include:

Node Manipulation Type Interface

Node.remove()	Remove this node from its parent
Node.replace(Node)	Replace this node with another
Node.append(Node)	Add a node or nodeset to the end of this nodes child nodes (defined only in Element and Nodeset)
Node.insertBefore(Node)	Add a node before this node
Node.insertAfter(Node)	Add a node after this node

The functions are similar to those in the W3C DOM (Document Object Model) specification, but instead of operating on the child nodes of the current node, these functions generally act directly on the current node and its parent node. This means that you don't have to specify *which* node to remove, or replace, because it is simply the node you've selected with an o:Path expression to invoke the function on.

Example 7.1. Adding, Replacing and Removing Nodes

```
<!-- create an RSS channel -->
<o:variable name="channel">
  <title>XML.com</title>
  <link>http://xml.com/pub</link>
  <description>
    XML.com features a rich mix of information and services
    for the XML community.
  </description>

  <items>
    <rdf:Seq>
      <rdf:li resource="http://xml.com/pub/2000/08/09/xslt/xslt.html" />
      <rdf:li resource="http://xml.com/pub/2000/08/09/rdfdb/index.html" />
    </rdf:Seq>
  </items>
</o:variable>

<!-- replace the description -->
<o:do select="$channel/description/text().replace('XML.com is great!')"/>

<!-- add some items to the sequence -->
<o:do select="$channel/items/rdf:Seq.append($newItems)"/>

<!-- remove all items with resources outside xml.com -->
<o:do select="$channel//rdf:li[not(starts-with(@resource, 'http://xml.com/'))].remove
```

Chapter 8. Functions and Procedures

Typically, if a logical piece of code can be reused, it makes sense to encapsulate it one way or another, to separate it out from the rest of your program. In this chapter we shall look into how functions and procedures can be used to get more out of o:XML.

Procedures

With the use of procedures, you can quickly build up and generate complex structures of dynamic XML without exposing any more than necessary of the underlying logic. They also provide very powerful ways of creating streams of output, and for processing input streams. In the next section we will look closer at how procedures in o:XML can be put to use implement reusable logic and ensure data consistency.

A Simple Example

Say that you need to generate an XML structure for each user on the system, and that each user has a certain set of attributes associated with it. The user data may come from a database, from user input or from a flat file, but the XML should always be generated the same. First define the XML that represents a user - let's say that the only thing we know about them is their username, password and their role and associated privileges on the system:

Example 8.1. User Example, the XML

Here's the XML as we want it to look for Freddie, who is an administrator and should have read, write and remove privileges.

```
<user name="Freddie" role="administrator">
  <password>banana</password>
  <privileges>
    <read/>
    <write/>
    <remove/>
  </privileges>
</user>
```

Now we want to write a producedure that generates this structure, with the right set of privileges which is determined by the role the user has, based on the three parameters given: name, password and role.

Example 8.2. User Example, the Procedure

```
<o:procedure name="ex:user">
  <o:param name="name"/>
  <o:param name="password"/>
  <o:param name="role" select="'guest'"/><!-- 'guest' is the default value -->
  <o:do>
    <user>
      <!-- add the 'name' attribute to the 'user' element -->
      <o:attribute name="name" select="$name"/>
      <!-- add the 'role' attribute -->
      <o:attribute name="role" select="$role"/>
      <password><o:eval select="$password"/></password>
```

```
      <privileges>
        <o:if test="$role = 'administrator'">
          <!-- only administrators are allowed to remove -->
          <remove/>
        </o:if>
        <o:if test="$role != 'guest'">
          <!-- non-guests are allowed to write -->
          <write/>
        </o:if>
        <!-- others (guests) may only read -->
        <read/>
    </privileges>
  </user>
</o:do>
```

Declaring the procedure doesn't produce any nodes in itself, but it binds the logic and information contained in the declaration to all **ex:user** elements (that's the name of the procedure). The result of a procedure call is always a nodeset, the output from executing the contents of its **do** element.

Example 8.3. User Example, Using the Procedure

This is what the procedure call will look like for Eddie (with the default value for *role*, 'guest'):

```
<ex:user name="'Eddie'" password="'salad'"/>
```

And this is the result it yields - an XML structure just like we specified it:

```
<user name="Eddie" role="guest">
  <password>salad</password>
  <privileges>
    <read/>
  </privileges>
</user>
```

Note that arguments passed to procedures have to be o:Path expressions, that's why “Eddie” and “salad” were quoted in the above example.

Wrapping it Up

Any child nodes of a mapped element will be passed to the procedure as the context nodeset (remember that the context node is represented in o:Path by dot, “.”, or accessed with the `current()` function). This is best exemplified by another example:

Example 8.4. Element Uppercasing Procedure

This procedure uses the context node to change the names of all child elements of the mapped element to upper case.

```
<o:procedure name="ex:uppercase">
  <o:do>
    <o:for-each name="node" select=".//*">
      <o:set upper="name($node).upper()"/>
      <o:do select="$node.name(Name($upper))"/>
    </o:for-each>
    <o:eval select="."/>
  </o:do>
</o:procedure>
```

When the procedure is called it will transmogrify its content:

```
<ex:uppercase>
  <table>
    <tr><td>Text nodes will not be changed.</td></tr>
    <tr><td>Only element names.</td></tr>
  </table>
</ex:uppercase>
```

Functions

While procedures are ideal for creating XML datasets and producing streams, they don't give you the full process flow control and flexibility that functions do. By defining your own o:XML functions you can easily write simple or recursive behaviour and even take full advantage of the parameter overloading capabilities of the language.

A function is declared in much the same way as a procedure by specifying parameters and the function body. A user-defined function is called exactly as other o:Path (or XPath) functions and can be part of any expression.

Example 8.5. String Replace Function

A function that replaces all occurences of one string with another.

```
<o:function name="ex:replace">
  <o:param name="input" type="String"/>
  <o:param name="from" type="String"/>
  <o:param name="to" type="String"/>
  <o:do>
    <o:variable name="result"/>
    <o:while test="contains($input, $from)">
      <o:set result="concat($result, substring-before($input, $from), $to)"/>
      <o:set input="substring-after($input, $from)"/>
    </o:while>
    <o:return select="concat($result, $input)"/>
  </o:do>
</o:function>
```

Here's an example of how it could be called:

```
<o:eval select="ex:replace('there is no place like home', 'home', 'space')"/>
```

Which would produce the output “there is no place like space”.

Chapter 9. Programs

As mentioned earlier, all XML content that does not lie in the o:XML namespace, or match an existing mapping rule, is simply copied to the result. This means that most every valid XML document is a valid o:XML program. If it contains o:XML instructions they will be executed when the file is interpreted by an o:XML engine. This makes it very easy to embed dynamic content in XML files, and makes for a highly flexible processing model.

If we want to pass parameters we need to include a **program** statement within our source XML file. Parameters are declared on programs in much the same way as with procedures - they have a name and an optional default value. There are also a couple of optional program attributes that control how whitespace and comments should be treated.

Example 9.1. Program Declaration

```
<?xml version="1.0"?>
<o:program comments="preserve" space="ignore"
           xmlns:o="http://www.o-xml.org/lang/">
  <o:param name="title" select="'DefaultTitle'"/>

...

</o:program>
```

Default values for *comments* and *space* is “ignore”. If a **program** declaration is not the first element of the program file, whitespace and comments will be preserved, at least up to the point where a **program** tag says otherwise.

The **program** declaration is typically the root element of the source file, however it is not stipulated anywhere that it has to be. The only restriction is that a program, ie an o:XML document, may contain at most one **program** declaration. It is also worth noting that you can't execute a program if it has parameters that have not been set, unless they have a default value.

The o:XML engine that runs the program is allowed to decide how to set parameters. For example when running the ObjectBox, the command line utility allows the user to pass parameters as command line arguments. When invoked as an Ant task it takes parameter values from the build file, while the servlet instance picks parameters from the HTTP request that called it. This makes it easy to write reusable, parameterized programs that can be used and tested independent of the execution environment.

Chapter 10. Regular Expressions

Since XML is a text markup language, it seems appropriate to use that most powerful of text processing tools - regular expressions. o:Path comes with native support in the functions `split`, `match` and `substitute`, furthermore these functions are integrated in the String type interface.

Split

The `split` function is used to divide a string into several parts. This is also known as tokenising and is extremely useful for breaking up a character sequence into appropriate chunks. The function's first argument is the node that you want to use as input (the string value of the node will be used). The second argument is the pattern, the regular expression which says where to split the input. There's an optional third argument which if used gives a limit to the number of chunks that will be created.

Example 10.1. Using Regular Expressions: split()

```
<o:for-each select="split('Here is a text with words, spaces and punctuation marks',
  <o:log msg="word: {.}"/>
</o:for-each>
```

Here the pattern says to split at any of the characters space, comma or dot. The result of running this fragment will be a log output for each word in the phrase.

Note that dot, ".", must be escaped with a backslash because it otherwise has special meaning to the regular expression. Enclosing characters in angle brackets denotes a character class. The plus sign means that one or more repetitions of any of the three characters constitutes a split.

Match

The `match` function gives you all parts of the input that matches the pattern, which is pretty much the opposite of `split` which gives you the parts in between the matches. It is used in the same way, and also has an optional limit argument. The next example extracts the first three upper-case words of the input string:

Example 10.2. Using Regular Expressions: match()

```
<o:eval select="match('I AM an EVIL hAx0r', '[A-Z]+ ', 3)"/>
```

Similar to the `split` function, the result of the `match` function call is a nodeset with a String node for each match. Even though in the above example we use the value of the whole set, each String can be accessed individually with a simple XPath expression.

It is also worth noting that `match` produces a separate match for each subgroup in the expression (subgroups are marked by parenthesis). This is useful when splitting text into several groups at once.

Substitute

Finally there's the `substitute` function, which does a pattern substitution on the input string of characters. It works much like the substitute expression of the Unix command **sed**, or Perl 5 substitutions. It takes one more argument than the two other functions explained in this chapter, and that is the substitution expression. A substitution expression can be a normal string, or it can reference some part, or parts, of the matched string. The optional limit argument specifies the maximum number of substitutions to be performed on the input.

Example 10.3. Using Regular Expressions: substitute()

The following example shows how to generate a list of image hrefs (from for example an input XHTML file), where the absolute links have been substituted for relative ones.

```
<o:for-each select="$input//img">
  <img>
    <o:attribute name="src"
      select="substitute(./@src, 'http://([^/]+)/(.*)', '$2')"/>
  </img>
</o:for-each>
```

String Type Interface

In o:XML the regular expression functions are also integrated in the String type interface. This has at least two advantages. Firstly it saves you a bit of typing, since the String itself is the input to the function. Secondly, and more importantly, it means that you can apply the functions to a whole set of String nodes at once. Let's look at an example.

Example 10.4. Regular Expressions: String.match() and String.split()

This example turns dates that are of the form (loosely) “dd/mm/yyyy” into formatted XML.

```
<o:for-each name="date" select="$input//text().match('\d{1,2}/\d{1,2}/\d{2,4}')">
  <date>
    <day><o:eval select="$date.split('/')[1]"/></day>
    <month><o:eval select="$date.split('/')[2]"/></month>
    <year><o:eval select="$date.split('/')[3]"/></year>
  </date>
</o:for-each>
```

What we have here is a variation of the Composite design pattern - sets of nodes, subtrees and individual objects are manipulated uniformly. The `match` function will be invoked on every text node in the input nodeset, creating a single result nodeset of all matching strings.

Summary

Regex functions opens up the text content of XML to processing in much the same way that XPath lets us navigate the node structure. Since much of real-life XML contains mountains of formatted text, and because many applications combine XML with non-XML text data, these tools prove almost indispensable.

For more information about regular expressions, see for example this useful web page: http://etext.lib.virginia.edu/helpsheets/regex.html.

Chapter 11. Types

Since o:XML is an object-oriented language, what we're really interested in is describing types of objects and how they relate to each other. In o:XML, classes of objects are called types, and objects are fundamentally nodes. Hence, the abstract parent type of all other types is Node (an abstract type in o:XML is a type that has no public constructor, but more about that later).

Declaring Types

A type declaration needs to say at least two things - the name of the type, and which other type or types it is derived from. For the type to be really useful, you probably want to add some function declarations too. You can also specify any number of type variables - they're great for storing object state. Apart from the type's name, there are in total three parts to the type declaration - parent types, type variables and type functions. Lets have a look at the parent types first.

A type that derives from another type (said to be the parent type), inherits all functions that it defines. This means that all types inherit the functions defined by Node, because all types derive from Node either directly or through another parent type.

If you define a function that has the same signature (i.e. name and parameter types) as one already defined in a parent type, you're in fact overloading that function. This means that all objects of your type will use your function instead of the one defined by the parent type. If the parameter types differ between the two functions, the one that has the closest match to the actual argument types will be picked (using runtime type resolution).

Now lets start defining some types.

Example 11.1. Declaring Types: Translator

```
<o:type name="Translator">
  <o:parent name="Node"/>
  <o:variable name="map"/>

  <o:function name="translate">
    <o:param name="text" type="String"/>
    <o:do>
      <o:return select="$map/word[from = $text]/to/text()"/>
    </o:do>
  </o:function>

  <o:function name="setMap">
    <o:param name="map"/>
    <o:do/>
  </o:function>

</o:type>

<o:set ling="Translator()"/>
<o:do select="$ling.setMap($espanol)"/>

<o:eval select="$ling.translate('hello')"/>
```

The type definition shows a function called `translate` that looks up a word in an XML vocabulary (`$map`) and returns its translation. It also has a function to set the vocabulary. What this definition does not show is how to create nodes of this type - it doesn't have any constructor functions. Instead we're relying on the generated default constructor.

Constructor Functions

Constructors are defined just like any other function, but they have to have the same name as the type. (Reversely, any function that has the same name as the type that defines it is a constructor function). If one is not supplied, a default constructor will be generated. The generated default constructor takes no parameters and does nothing apart from initialise any type variables to empty nodesets, and call the default constructors of the parent types.

It is the responsibility of the constructor to correctly instantiate all parent types. In the case of the Translator this is not a problem, because the only parent is *Node* which is instantiated by a default constructor.

Note that type variables are directly assigned to any parameter value of the same name. This is particularly useful with constructors, as it gives a fast and easy way to set variable values. Now we'll extend our type definition with a constructor that does exactly that.

Example 11.2. Declaring Types: Translator take Two

```
<o:type name="Translator">
  <o:variable name="map"/>

  <o:function name="Translator">
    <o:param name="map"/>
    <o:do/>
  </o:function>

  ...

</o:type>

<o:set ling="Translator($espanol)"/>

<o:eval select="$ling.translate('hello')"/>
```

Calling a constructor function returns a new node of that type. Here we create a translator that, given the right dictionary file, knows how to translate spanish. In the revised type definition above we've not included the parent directive for *Node*, as this is the default parent type anyhow.

Inheritance

Our translator only knows how to translate single words. It would be more useful with a translator that can process an entire document or fragment in one go. Let's create another type that derives from Translator.

Example 11.3. Declaring Types: DocumentTranslator

```
<o:type name="DocumentTranslator">
  <o:parent name="Translator"/>

  <o:function name="DocumentTranslator">
    <o:param name="map"/>
    <o:parent name="Translator" select="Translator($map)"/>
    <o:do/>
  </o:function>

  <o:function name="translate">
```

```
      <o:param name="doc"/>
      <o:do>
        <o:for-each name="phrase" select="$doc//text()">
          <o:variable name="translated">
            <o:for-each name="word" select="$phrase.match('(\w+)(\W+)')">
              <o:set match="$this.translate($word)"/>
              <o:choose>
                <o:when test="$match">
                  <o:eval select="$match"/>
                </o:when>
                <o:otherwise>
                  <o:eval select="$word"/>
                </o:otherwise>
              </o:choose>
            </o:for-each>
          </o:variable>
          <o:do select="$phrase.replace($translated)"/>
        </o:for-each>
        <o:return select="$doc"/>
      </o:do>
    </o:function>

  </o:type>
```

There are a few things worth noting about the DocumentTranslator. Firstly the constructor explicitly calls the parent type constructor. It doesn't have it's own `map` variable, in fact it wouldn't know what to do with it. Instead it uses the parent type's `translate` function.

As you can see, the DocumentTranslator overloads the `translate` function, but accepts a Node argument instead of the more specific String of the parent class. So when the DocumentTranslator (or a user) calls `translate` with a String, it is in fact the parent type function that is invoked. Just as with normal overloading, it is the function with the closest matching signature that is picked.

The `this` variable is always defined within a type function definition, it's value is the node that the function was invoked on. We use it to access the content of the node (the nodeset created by the constructor function), and to call other type functions.

Chapter 12. Exception Handling

Sometimes things go wrong when running a program - files might have moved or gone corrupt, network connections can disconnect, filesystems run out of space and evil bugs sometimes cause application errors. These exceptional events generally cause an exception to be thrown, which means that program excecution stops until the exception is caught. If there's no code to catch it, the exception will cause the program to terminate.

Throwing and catching exceptions is generally the preferred way to do error handling. If something happens that shouldn't happen, say you get the wrong type of data or critical resources fail, throwing the right type of exception will not only allow a user, or your own code higher up in the hierarchy, to recover from the problem but it can also give valuable information about what went wrong.

What is an Exception

An exception can be any node of any type, you can throw a String or a Number or anything you want. Two specific types of nodes are thrown by the system under certain circumstances. If there's a system error, a SystemError exception is thrown. If you catch this you can get information about what happened that shouldn't have. Secondly there's AssertionError, a type of node which is thrown if an assertion fails (we'll look more into that later). Both of these types inherit from Exception, a useful parent type that you can also use for your own exceptions.

Throwing and Catching Exceptions

Let's look at a simple example.

```
<o:function name="open">
  <o:param name="url"/>
  <o:do>
    <o:if test="not($url.startsWith('http://'))">
      <o:throw select="'Invalid URL, unsupported protocol!'"/>
    </o:if>
    ... do something with the URL
  </o:do>
</o:function>
```

Here we **throw** an exception, because we've been asked to do something with a URL that we don't know how to handle. The exception itself is a simple String that holds the error message. To catch the exception, we need to somehow enclose the code in a **catch** block:

```
<o:catch>

  <o:do select="open($url)"/>
  <o:log msg="opened: {$url}"/>

</o:catch>
```

In the above **catch** block we're relying on the default values of two optional attributes - *exceptions* and *handler*. The first one says which types of exceptions we want to catch. The default is to catch all exceptions. If we wanted to we could instead specify a (comma separated) list of type names. Any exceptions not of a type in that list would simply not be caught, but instead be propagated to the next (if any) catch block. The second optional attribute, *handler*, specifies which exception handler to use. An exception handler can be any object that implements the function `handle(Node exception)`. The default here is to create a new instance of *ExceptionHandler*, which simply logs the exception to the error log. When an exception has been caught, the program

continues to execute right where the **catch** block ends. In the example above that means that the log message “opened:” will not be printed if the open function throws an exception.

Defining Exceptions and Exception Handlers

For more fine grained control over the exception handling in your applications you will generally want to define your own type hierarchies of exceptions and exception handlers.

Example 12.1. Defining Exceptions and Exception Handlers

```
<o:type name="URLException">
  <o:parent name="Exception"/>
  <o:variable name="url"/>

  <o:function name="URLException">
    <o:param name="error"/>
    <o:param name="url"/>
    <o:parent name="Exception" select="Exception($error)"/>
    <o:do/>
  </o:function>

  <o:function name="url">
    <o:do>
      <o:return select="$url"/>
    </o:do>
  </o:function>
</o:type>

<o:type name="NetworkExceptionHandler">
  <o:parent name="ExceptionHandler"/>

  <o:function name="handle">
    <o:param name="exception" type="URLException"/>
    <o:do>
      <o:log level="error" msg="{$exception.message()}"/>
      <o:log level="error" msg="in URL: {$exception.url()}"/>
    </o:do>
  </o:function>
</o:type>

<o:function name="open">
  <o:param name="url"/>
  <o:do>
    <o:if test="not(starts-with($url, 'http://'))">
      <o:throw select="URLException('unsupported protocol!', $url)"/>
    </o:if>
    <o:log msg="opening URL: {$url}"/>
  </o:do>
</o:function>

<o:catch exceptions="URLException" handler="NetworkExceptionHandler()">

  <o:do select="open('ftp://somewhere.com')"/>

</o:catch>
```

By using *ExceptionHandler* as the parent type of our error handler, and further qualifying the `handle` function with a typed argument, we're ensuring that unknown exceptions are handled by the parent type. Of course, if we only specify *URLException* in our catch blocks then we won't be handling any other exceptions anyhow.

Note that the *handler* attribute in the **catch** statement accepts any expression, as long as it evaluates to an object

with a `handle` function. This means that we can, for example, create a single handler and reuse it.

Example 12.2. Reusing an Exception Handler

```
<o:set errorhandler="SpecialExceptionHandler()"/>
<o:catch exceptions="URLException" handler="$errorhandler">
  ...
</o:catch>
```

This way we can easily perform tasks like count the total number of exceptions (allow `n` number of retries), or queue the exceptions to a remote handler or suchlike. Furthermore, when developing types, a type implementing handler functions can use itself, i.e. the `$this` variable, as the exception handler instance.

Chapter 13. Threads

Computer programs are fundamentally based on instructions, generally written as a sequence of commands. This reflects how the machine executes the instructions, one after the other. However it is often necessary for an application to perform several tasks at the same time. To represent concurrent execution paths we use threads. Each application thread can run in parallel with every other.

To make things easy, o:XML provides a *Thread* type. Threads can be started and put to sleep, or made to wait for other threads. They can be given higher or lower priority, depending on how important or time-consuming they are.

To make things easier still, you can create and use threads without even using types. The **thread** element tells the machine to run all instructions inside the element in parallel with the current thread of execution.

Example 13.1. Threads Made Easy

```
<o:thread>
  ... concurrent code ...
</o:thread>
```

What the **thread** element actually does is to create a new Thread object, start it, and then let the machine carry on with what it was doing. The output of executing **thread** is that object, which can be captured and further manipulated.

Example 13.2. Asynchronous Functions

```
<o:function name="fire-and-forget">
  <o:do>
    <o:thread>
      ... do something ...
    </o:thread>
  </o:do>
</o:function>

<o:set thread="fire-and-forget()"/>

... do something else ...
```

In the above example, calling `fire-and-forget` will immediately return a *Thread* object. While not very useful in itself, the function body is executed while the current thread carries on.

As you know by now, each thread represents a sequence of instructions that are executed when the thread runs. With o:XML *Thread* objects, these instructions reside in the `Thread.run()` function. The Thread type itself has a run function that doesn't do anything. When you use the element **thread**, a *Thread* object is created that knows to execute the instructions inside the element when its `run()` function is called. Another way to implement concurrent behaviour is to create a type with Thread as parent type, and declare a `run()` function that takes care of business. Objects of this type can then easily be created and controlled using the usual *Thread* functions.

Example 13.3. Thread Types

```
<o:type name="MyThread">
  <o:parent name="Thread"/>

  <o:function name="run">
    <o:do>
      <o:while test="true()">
        <o:do select="$this.sleep(1000)"/>
        <o:log msg="i sleep all day!"/>
      </o:while>
    </o:do>
  </o:function>
</o:type>
```

Now for another example of asynchronous functions: Consider that we want to collect some information from our servers, but we don't want to wait with what we're currently doing to get the results. Instead we create a thread that collects the data for us. Subsequently calling the function `return()` causes the program to wait for the thread to finish, and then return its output, or return value.

Example 13.4. Thread Results

```
<o:function name="collect">
  <o:do>
    <o:thread>
      <o:eval select="collect('server-1')"/>
      <o:eval select="collect('server-2')"/>
      <o:eval select="collect('server-3')"/>
    </o:thread>
  </o:do>
</o:function>

<o:set thread="collect()"/>

... do something else ...

<o:set result="$thread.return()"/>
```

At the beginning of this chapter we made mention of thread priorities. You can set or get the priority of a thread by calling its `priority(Number)` or `priority()` function respectively. Meaningful priorities are in the range from 1 to 10, with 1 representing the lowest precedence and 10 the highest.

The purpose of some threads is to work in the background, performing maintenance tasks or provide services to the application. These threads are called daemon threads, and they operate more or less independently of the main program. Daemons are automatically terminated when all other threads have finished, to avoid the program running indefinitely. To mark a thread as daemon, or determine its daemon status, there are the functions `daemon(Boolean)` and `daemon()`.

When creating threads using the **thread** element, both the priority and daemon status can be set with attributes.

Example 13.5. Thread Control Made Easy

```
<o:thread priority="1" daemon="true()">
  ... slow code ...
</o:thread>
```