The o:XML Programming Language

Martin Klang

The o:XML Programming Language

Martin Klang

Table of Contents

List of Tables

List of Examples

Introduction

What Is o:XML

o:XML is an object-oriented, dynamically typed programming language based on XML [http://w3c.org] and XPath [http://www.w3.org/TR/xpath].

Every o:XML program consists of a valid XML document or fragment. The reserved o:XML elements provide the programmer with a vocabulary similar to that of many other procedural languages. This set of keywords can be extended with user-defined procedures, see Chapter 1, *Procedures*.

Keyword elements make up the atomic program instructions, other nodes (except white-space text nodes) are treated literally. The result of executing a program is an XML document or fragment where the program instructions have been substituted.

o:Path

Parameters are passed to procedures using o:Path expression syntax, a superset of XPath 1.0. The o:Path expression language can be extended with functions defined in o:XML in the same way as procedures extend the core language.

The basic types of XPath can also be extended, and user-defined types can have functions associated with them. Furthermore a type can inherit functionality from one or more parent types.

Objects are created using type constructors, functions can be invoked on an object with the **dot** operator.

Chapter 1. Procedures

A procedure is defined using the **procedure** key element. Once defined, it associates the program instructions it contains (the procedure body) with all elements with a matching name. A procedure definition may declare parameters that are accessible in the procedure body as variables.

Example,

```
<!-- procedure definition -->
<o:procedure name="ex:formatDate">
  <o:param name="day"/>
  <o:param name="month"/>
  <o:param name="year" select="2002"/>
  <o:do>
    <date>
      <day><o:eval select="$day"/></day>
      <month><o:eval select="$month"/></month>
      <year><o:eval select="$year"/></year>
    </date>
  </o:do>
</o:procedure>

<!-- procedure call -->
<!-- 'year' has a default value and so is optional -->
<ex:formatDate year="2002" month="'Aug'" day="31"/>
```

Procedure calls

In procedure calls the calling node is available and can be referenced from within the procedure body using the variable `this`.

Parameters are o:Path expressions passed in attributes of the calling element.

The return value of a procedure is the nodeset that results from executing the procedure body. The return type of all procedures is a nodeset.

Pre-order Descent

The child nodes of a node constituting a procedure call are executed before the procedure call is made.

Chapter 2. Functions

A user-defined function extends o:Path directly. The return value of a function is determined by its return statement (or statements), the functions type is that of its return value. If the return statement is missing, the function value is the nodeset that resulted from executing the function. A function definition may declare parameters that are accessible in the function body as variables.

Example,

```
<!-- function definition -->
<o:function name="ex:formatDate">
  <o:param name="day"/>
  <o:param name="month"/>
  <o:param name="year"/>
  <o:do>
    <date>
      <day><o:eval select="$day"/></day>
      <month><o:eval select="$month"/></month>
      <year><o:eval select="$year"/></year>
    </date>
  </o:do>
</o:function>

<!-- function call -->
<o:eval select="ex:formatDate(31, 'Aug', 2002)/>
```

A function call for which no matching function can be found will produce a `FunctionNotFound` exception.

Function Names

Function names follow the XML convention of partially or fully qualified names, which goes to say that they may be qualified by an optional namespace prefix. Just like with other qualified names, the namespace prefix must be defined within the context that it is referenced.

Function Overloading

Function overloading conflicts are resolved by finding the closest match on parameter types. A types primary parent type is considered closer than its secondary parent, its secondary parent closer than its tertiary parent etc, see the section called “Inheritance”.

Chapter 3. Programs

Program Statements and Sequences

Every node in an o:XML program document is a program statement. If there exists no Node Mapping for the statement, it is treated as a literal node and simply copied to the result document.

Control statements are program statements capable of changing the program location. They include procedures, function calls and the following o:XML key elements:

- **if**
- **choose**
- **while**
- **for-each**

A program sequence is a valid (well formed) XML fragment containing zero or more program statements.

Program Document

A program document is an XML document or document fragment (with any included documents or fragments) containing program statements. When a program executed, all include directives are completed and all program instructions followed in document order, or the order imposed by control statements.

Program Location

A program location is a reference to a node in the program document. A location cannot reference an object that is not contained in the program document, e.g. an object only referenced by a variable.

Execution Environment

A program runs in an environment that holds a reference to the current location in the document. It also keeps a stack of all variables currently in scope and their respective values.

A program environment exists for each concurrent thread.

Chapter 4. Expressions

The expression language for o:XML is o:Path, which is a superset of XPath v1.0 [www.w3.org/TR/xpath].

Type Function Invocation

Type functions are invoked on an object instance with the binary operator **dot**:

[1]	TypeFunctionCall	::=	Expr '.' FunctionName '(' ArgList? ')'
[2]	ArgList	::=	Expr | ArgList ',' Expr

For example,

foo().bar()

$foo.bar($param)

Limitations and Conflicts

o:Path does not allow for direct indexing of elements whose name contains a dot (“.”), since this is interpreted to signify a type method invocation. Elements with dot names can still be indexed by use of predicates and the `name` or `local-name` XPath functions.

For example, given the following XML document:

```
<?xml version="1.0"?>
<abc>
  <foo.bar>
    <def/>
  </foo.bar>
</abc>
```

XPath expression:

```
/abc/foo.bar/def
```

o:Path expression:

```
/abc/*[name() = 'foo.bar']/def
```

Chapter 5. Objects

Objects are of two types: Nodes and Nodesets. All other types inherit from one of these two. Nodesets are container objects, Nodes generally represent data.

Instantiation

To create an object, a type constructor has to be invoked. This in turn invokes the constructors of any parents recursively until all inherited types have been instantiated. When the parent constructors have executed, any variables with default values or whose values are passed in as parameters to the constructor are set before the function body is executed. See the section called “Constructors” and the section called “Type Variables”.

Pass By Reference

When objects are passed as parameters to a function or procedure, it is always done on the same object instance. That is to say that no copy of the object is ever implicitly constructed.

Object Proliferation

The `copy` function returns a deep copy of an object: all objects contained within it in variables, content or parent types are also recursively copied. Calling the `copy` function is equivalent to calling an objects copy constructor.

Identity and Location

Every instance of a type, i.e. every object, is unique within a program. Furthermore an object can exist at most in one location in the program document. If an attempt is made to insert an object that already has a location (**parent() != false**) a `HierarchyViolation` exception is thrown.

Example:

```
<o:variable name="var1" select="Element('dyne')"/>
<o:variable name="var2" select="$var1"/> <!-- valid: an object can have any number of r
<foobar>
  <o:insert select="$var1"/>
  <o:eval select="$var1"/> <!-- valid: eval constructs a new object -->
  <o:insert select="$var2"/>  <!-- error: object already exists in document -->
</foobar>
```

Object Lifetime

An object is guaranteed to exist in the execution environment from the point where the constructor has completed for as long as there is still a reference to it.

Chapter 6. Variables

Variables are references to objects, they are created and assigned values with the **variable** key element. Dereferencing an undefined variable is an error, and will result in an `UnboundVariable` exception.

Variable Scoping and Assignment

A variable is bound in any child or following sibling node of the node where it was declared.

Subsequent declarations of the same variable within its scope has the effect of assigning to it a new value.

Within a procedure or function body only parameters and locally declared variables are in scope.

The scope of a variable is not propagated to any included documents or document fragments.

Chapter 7. Node Mappings

Node mappings give special meaning to program nodes. They come in three different flavours:

- Key Elements
- Procedures
- Language Extensions

Key Elements

Key elements map named elements in the o:XML namespace to core language functionality, such as program constructs and declarations. They make up the atomic o:XML program statements.

Procedures

Procedures extend the o:XML core language by associating named elements with parameterised program sequences.

Language Extensions

Language extensions is a means by which o:XML programs can be easily integrated in and customised for specific execution environments. Within an execution environment there may exist language extensions mapping from nodes of a particular type, name or value to native or external code.

A language extension may produce result nodes and/or change the program state. The null language extension is an extension that does neither.

Examples of possible language extensions include:

- Mapping Program Instruction nodes to application specific code
- Stripping comments and whitespace text nodes from a program by mapping them to the null extension

Chapter 8. Namespaces

The o:XML Namespace

The o:XML key elements are defined in a specific namespace:

http://www.o-xml.org/lang/

Note

In the examples in this book, the o:XML namespace declaration is omitted for brevity. The following can be assumed:
xmlns:o="http://www.o-xml.org/lang/"

Function and Procedure Namespaces

Functions and procedures may contain a namespace specifier, which declares that the entity will only be available within that namespace. If the namespace specifier is not declared, the function or procedure will be defined in the default namespace.

Example:

```
<o:function name="mm:string" xmlns:mm="http://mydomain.com/mynamespace">
  <o:param name="arg"/>
  <o:do>
    <string><o:eval select="$arg"/></string>
  </o:do>
</o:function>

<program xmlns:m="http://mydomain.com/mynamespace">
  <o:eval select="string('this is my string')"/> <!-- calls o:Path string function -->
  <o:eval select="m:string('this is my string')"/> <!-- calls user string function -->
</program>
```

Chapter 9. Types

Every object in an o:XML program is an instance of a type. Furthermore, all nodes in a program document are objects.

A type is defined by its name, parents, variables, constructors and functions.

Basic Types

The o:XML basic types consist of an extended XPath document model:

- Node
- Nodeset
- Element
- String
- Number
- Boolean

Node is an abstract class, which means it doesn't define any public constructors. It is the parent type of all other basic types, except Nodeset which has no parent type.

Content

Every object has a content, which is defined by the object type. String objects for example have a simple string as content, the content of a `Nodeset` is 0 or more nodes, and the content of an `Element` is an XML element node with its attributes and child nodes.

The content of a type is equal to the return value of its constructor. A type can generate objects with structurally and semantically different content.

The content can be accessed from within the body of a type function through the `this` variable.

Type Functions

A type function is a function defined as part of a type definition. Type variables can be reassigned values and referenced, and the object content is available through the special `this` variable. If a function parameter name is identical to that of a type variable, the type variable is assigned the parameter value before the body of the function executes. Parent instances are accessible by calling `parents`, a type function defined by `Node`.

Type functions (not including constructors) can only be invoked through a type instance with the **dot** operator.

Constructors

A type can define one or more functions bearing the name of the type itself. These functions are the constructors of the type, and are invoked to construct a type instance.

Unlike other type functions, constructors are global o:Path functions that must be called directly.

The return value of a constructor constitute the objects content, and forms part of its state and public interface - see the section called “Content”.

Parent types are also initialised by the constructor. This can be done explicitly with the **parent** key element.

If a default constructor exists for a parent that is not explicitly initialised, it will be called after any other parent initialisers, before the function body executes. It is an error to define a constructor that does not initialise all parent types.

The content of parent instances is accessible to an object through the protected `parents` function defined by `Node`.

For example,

```
<o:type name="foo">
  <o:parent name="bar"/>
  <o:function name="foo">
    <!-- default constructor of 'foo' -->
    <o:parent select="bar()"/>
    <!-- calls default constructor of 'bar' -->
    <o:do>
      <foo><o:eval select="parents()/bar"/></foo>
      <!-- inserts contents of parent 'bar' -->
    </o:do>
  </o:function>
</o:type>
```

Parent types are instantiated exactly once even if shared by multiple declared parent types, which means that only the first constructor for a certain type that is reached is executed.

Generated Default Constructors

A default constructor is a constructor that takes no parameters.

If (and only if) no constructors are defined, a default constructor is generated by the execution environment.

A generated default constructor calls only the default constructors of the parent types. If the type has one or more parent types with no default constructor it is an error not to define a valid constructor.

Copy Constructor

A copy constructor is a constructor that takes a single parameter of the same type as the type being declared. It is called by the `copy` function and its purpose is to create an identical copy of a given object. Every valid type has exactly one valid copy constructor.

If a type does not declare its own copy constructor, then one will be automatically generated.

Example 9.1. Generated Copy Constructor

```
<o:function name="type-name">
  <o:param name="other" type="type-name"/>
  <!-- for each parent in the order they were declared -->
  <o:parent select="parent-name($other.parents()/parent-name)"/>
  <o:do>
    <!-- for each variable in the order they were declared -->
    <o:variable name="variable-name" select="copy($other.variables()/variable-name)"/>
    <o:eval select="$other/*"/>
  </o:do>
</o:function>
```

Type Variables

A type variable is a variable defined within a type definition. Type variables with names matching that of a type function parameter will be initialised to the respective parameter value before the function body executes. Others will be assigned their default value if defined, otherwise given the value of an empty nodeset.

For example,

```
<o:type name="foo">
  <o:variable name="a"/>
  <o:variable name="b"/>
  <o:variable name="c" select="'c'"/>
  <o:variable name="d"/>
  <o:function name="foo">
    <o:param name="a"/>
    <o:do>
      <o:variable name="b" select="'b'"/>
      <foo/>
    </o:do>
  </o:function>
</o:type>

<o:variable name="bar" select="foo('a')"/>
```

would result in a `bar` variable of type foo with `$a = 'a', $b = 'b', $c = 'c'` and `$d` an empty nodeset.

Type Namespaces

Just like functions and procedures, type definitions may contain a namespace specifier. If it does, a type instance can only be created with a fully qualified name.

Example:

```
<o:type name="mm:String" xmlns:mm="http://mydomain.com/mynamespace">
  <o:function name="mm:String">
    <o:param name="arg"/>
    <o:do>
      <string><o:eval select="$arg"/></string>
    </o:do>
  </o:function>
</o:type>

<program xmlns:m="http://mydomain.com/mynamespace">
  <o:variable name="str1" select="String('this is my string')"/> <!-- calls o:Path Stri
  <o:variable name="str2" select="m:String('this is my string')"/> <!-- calls user Stri
</program>
```

Inheritance

An o:XML type inherits functionality, meaning type functions, from one or more parent types. The first declared inheritance constitutes a types primary parent, second declared its secondary parent etc.

A type that inherits from another type is said to be the child type of its parent. Public and protected functions declared in a parent type are available in all types that inherit from it.

If a user defined type does not declare any parents, it is given a default parent of Node.

Polymorphism

Object types are determined dynamically at runtime.

If a type declares a function that is already declared in one or more of its parent types, the most specialised function takes precedence. That is to say the function with the closest matching signature is invoked regardless of whether it is declared in a parent or child type.

Type functions are first searched for in the type itself, then in all its parent types in the order they were declared, then in the parents of the parent types until all parent types have been exhausted.

If a function with identical signature is declared in both parent and child types, the child types function takes precedence.

Access Specifiers

Type methods can contain an access specifier as an attribute to the declaring element.

Possible access declarations are:

Access Declarations

public	Accessible to anyone.
protected	Accessible to instances of this type and any of its child types.
private	Accessible to instances of this type only.

A type function can be declared *public*, *protected* or *private*. If no access specifier is present the default is *public*.

All type variables are *private*, protected and public variables are not allowed.

Constructors can also declare an access specifier. A protected type is a type with no public constructors - the same as an abstract type. A type with only private constructors is formally correct but would be non-sensical.

For example,

```
<o:type name="Foo">
  <o:function name="Foo" access="private">
    <o:param name="arg"/>
    <o:do>
      <o:eval select="$arg"/>
    </o:do>
  </o:function>
  <o:function name="Foo" access="protected">
    <o:do><tb:eval select="Foo('foo')"/></o:do>
  </o:function>
</o:type>

<o:type name="Bar">
  <o:parent name="Foo"/>
  <o:function name="Bar" access="protected">
    <o:param name="arg"/>
    <o:parent select="Foo($arg)"/> <!-- illegal: private constructor of Foo -->
    <o:do/>
  </o:function>
  <o:function name="Bar">
    <o:parent select="Foo()"/>
    <o:do/>
  </o:function>
</o:type>
```

```
<o:variable name="var1" select="Foo()"/> <!-- illegal: protected class -->
<o:variable name="var2" select="Bar('foo')"/> <!-- illegal: protected constructor -->
<o:variable name="var3" select="Bar()"/> <!-- correct -->
```

Chapter 10. Builtin Type Interfaces

Node

Table 10.1. Node Interface

Function Name	Return Value	Description
`id()`	Number	returns a unique identifier for this object
`type()`	String	get the name of this objects runtime type
`string()`	String	get the string value of this object, synonymous to calling `string(Object)`
`number()`	Number	get the number value of this object
`boolean()`	Boolean	get the boolean value of this object
`replace(Node other)`	Nodeset	returns a nodeset containing this object after it has been replaced with *other*
`parent()`	Nodeset	returns a nodeset containing the parent location
`protected parents()`	Nodeset	returns a nodeset with all parent instances

Nodeset

Table 10.2. Nodeset Interface

Function Name	Return Value	Description
`append(Node other)`	Node	appends *`other`* to the list of nodes in this nodeset

Element

Table 10.3. Element Interface

Function Name	Return Value	Description
name	type	returns
`name()`	String	name of this element, including any namespace prefix

String

Table 10.4. String Interface

Function Name	Return Value	Description
`length`	Number	number of characters in this string

Chapter 11. o:XML Language Reference

Top Level o:XML Elements

- **variable**
- **function**
- **procedure**
- **type**
- **if**
- **choose**
- **while**
- **for-each**
- **try**
- **throw**
- **include**
- **insert**
- **eval**

o:Path Core Functions

Table 11.1. o:Path Core Functions

Function Name	Return Value	Description
`type(Object)`	String	get the runtime type name of an object
`copy(Object)`	Object	create a deep copy of an object