|  |
| --- |
|  |

MXML 2006 and 2009  
Language Specification

Author: Gordon Smith (gordonsmith@apache.org)  
Last Update: May 25, 2010

Contents

2 Introduction 6

2.1 Scope of this document 6

2.2 Purpose of this document 6

2.3 Overview of MXML 7

2.4 Advantages of MXML 7

2.4.1 Terseness 7

2.4.2 Naturalness 8

2.4.3 Features 10

2.4.4 Toolability 10

2.5 Key concepts of MXML 11

3 MXML files 12

3.1 File names 12

3.2 Byte order mark 12

3.3 Encodings 12

3.4 Illegal characters 12

3.5 Line endings 12

4 Lexical issues 13

4.1 MXML as XML 13

4.2 XML directive 13

4.3 Unicode 13

4.4 Numeric character references 13

4.5 Character entity references 14

4.6 Special characters in attribute values 14

4.7 Special characters in character data 15

4.8 Whitespace 15

4.9 Comments 15

4.10 Namespaces 16

4.11 Tag names 16

4.12 Tag order 16

4.13 Attribute names 16

4.14 Attribute order 16

4.15 Tag content 17

5 Namespaces in MXML 18

5.1 Language namespaces 18

5.2 Manifest namespaces 18

5.3 Package namespaces 20

5.4 Private namespaces 21

6 Tags in MXML 22

6.1 Instance tags 23

6.1.1 Instance tags for primitive builtin types 23

6.1.1.1 Boolean tag 23

6.1.1.2 int tag 24

6.1.1.3 uint tag 25

6.1.1.4 Number tag 27

6.1.1.5 String tag 28

6.1.1.6 Class tag 30

6.1.1.7 Function tag 30

6.1.1.8 Missing primitives 31

6.1.2 Instance tags for complex builtin types 31

6.1.2.1 Array tag 31

6.1.2.2 Vector tag 34

6.1.2.3 Object tag 36

6.1.2.4 XML tag 38

6.1.2.5 XMLList tag 41

6.1.2.6 RegExp tag 41

6.1.2.7 Date tag 42

6.1.3 Instance tags for non-builtin classes 44

6.1.4 Instance tags for dynamic classes 45

6.1.5 Instance tags with special compile-time treatment 46

6.1.5.1 Repeater tag 46

6.1.5.2 HTTPService tag 46

6.1.5.3 WebService tag 46

6.2 Property tags and property attributes 47

6.2.1 Boolean properties 48

6.2.2 int properties 48

6.2.3 uint properties 49

6.2.4 Number properties 50

6.2.4.1 PercentProxy metadata 51

6.2.5 String properties 51

6.2.5.1 CollapseWhiteSpace metadata 51

6.2.6 Class properties 52

6.2.6.1 IFactory properties 52

6.2.7 Function properties 52

6.2.8 Array properties 53

6.2.8.1 Implicit Array tags 54

6.2.9 Vector properties 54

6.2.9.1 Implicit Vector tags 54

6.2.10 Object properties 55

6.2.11 \* properties 57

6.2.12 XML properties 58

6.2.13 XMLList properties 58

6.2.14 Properties of other types 59

6.2.15 Implicit property tags 59

6.3 Style tags and style attributes 61

6.3.1 Named colors 61

6.4 Event tags and event attributes 63

6.5 id attribute 65

6.6 Class definition tags 67

6.6.1 Root tags 67

6.6.1.1 Properties, styles, and events 68

6.6.1.2 implements attribute 69

6.6.1.3 Other special attributes 69

6.6.1.4 Creating instances 69

6.6.2 Application tags 70

6.6.3 Ordinary attribute with special meaning 70

6.6.3.1 backgroundColor 70

6.6.3.2 styleName 70

6.6.3.3 width and height 70

6.6.4 Special attributes 70

6.6.4.1 frameRate 71

6.6.4.2 lib 71

6.6.4.3 pageTitle 71

6.6.4.4 preloader 71

6.6.4.5 rsl 71

6.6.4.6 scriptRecursionLimit 71

6.6.4.7 scriptTimeLimit 71

6.6.4.8 theme 71

6.6.4.9 usePreloader 71

6.6.5 Automatic imports 71

6.7 Language tags 73

6.7.1 Binding tag 73

6.7.2 Component tag 74

6.7.3 Declarations tag 76

6.7.4 Definition tag 77

6.7.5 DesignLayer tag 78

6.7.6 Library tag 79

6.7.7 Metadata tag 80

6.7.8 Model tag 81

6.7.9 Private tag 83

6.7.10 Reparent tag 84

6.7.11 Script tag 85

6.7.12 Style tag 86

7 States 88

7.1 Declaring each State 88

7.1.1 name attribute 88

7.1.2 stateGroups attribute 88

7.1.3 basedOn attribute 89

7.1.4 overrides attribute 89

7.1.5 Other attributes 89

7.2 Declaring the states of a component 89

7.3 Using state-dependent tags 90

7.4 Using state-dependent properties, styles, and events 90

8 File paths 92

9 Compiler directives 93

9.1 @Clear directive 93

9.2 @ContextRoot directive 94

9.3 @Embed directive 94

9.4 @Resource directive 95

10 Databinding expressions 96

# Introduction

## Scope of this document

This document is the most complete description of the MXML language to date. It is currently still a work-in-progress, but is intended to evolve into the official specification of the two versions of MXML supported by Flex 4: MXML 2006 and MXML 2009. It describes how the compiler interprets the tags, attributes, and character data in MXML files.

It is focused on the MXML language and does not cover how the MXML compiler (for example, as found in mxmlc and compc) compiles other files such as .as files, .css files, and .properties files. Hopefully other specifications will cover these topics at some point.

Nor does it cover how the MXML compiler deals with ActionScript metadata, unless that metadata affects the semantics of MXML. For example, this document discusses [DefaultProperty], [Style], and [Event] (because these affect how attributes and child tags in MXML are interpreted) but not [AccessibilityClass] (which provides conditional linkage of another ActionScript class) or [Inspectable] (which provides information for the property inspector in Flash Builder). Again, a complete specification of all meaningful metadata is much needed but wil be a separate document from this one.

This document does not cover prior versions of MXML in Flex 1, Flex 2, and Flex 3. Nor does it cover the version of MXML that applies when you specify the compilation option  
-compatibility-verson=3 in Flex 4, because this gives you the Flex 3 behavior.

Finally, we have omitted discussing effect attributes and effect tags, because effect triggers are a deprecated concept in Flex 4 and may be dropped from MXML in the future.

## Purpose of this document

The MXML language has developed “organically” over the last eight years, and it is time for a more systematic approach to its definition and evolution.

However, the impetus for this specification comes from the Next Generation Compiler project, which seeks to replace the current MXML compiler with one that is faster, capable of supporting large workspaces in limited memory, and more maintainable. We wanted to document the behavior of the current compiler so that we can re-implement -- as evolve as appropriate -- the behavior that Flex developers depend on.

Therefore an important purpose of this document is to lay the foundation for MXML 2012, the next version of the language, which will be supported by the new compiler. This document has numerous recommendations for MXML 2012.

## Overview of MXML

MXML can be thought of as a declarative language using XML syntax to provide syntactic sugar on top of ActionScript, which is a procedural language.

Each MXML file defines an ActionScript class (and sometimes additional classes as well). For example, when you write an MXML file – say HelloWorld.mxml – such as

<?xml version="1.0" encoding="utf-8"?>  
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"  
 xmlns:s="library://ns.adobe.com/flex/spark">  
 <s:Label text="Hello, World!"/>  
</s:Application>

you are defining a subclass of spark.components.Application named HelloWorld, as if you had written code similar to

import spark.components.Application;  
import spark.components.Label;  
  
public class HelloWorld extends Application  
{  
 public function HelloWorld  
 {  
 super();  
  
 var label1:Label = new Label();  
 label.text = "Hello, World!";  
 addChild(label1);  
 }  
}

The actual code generated by the MXML compiler is more complicated, but the code above gives the general idea. It is important to understand, however, that MXML can define any ActionScript class, not just application-level classes and not just visual classes that extend DisplayObject.

## Advantages of MXML

Although there are some ActionScript *afficianados* in the Flex community who don’t care for MXML, most Flex developers find MXML to be the more productive language for application development. MXML provides four main advantages over ActionScript:

### Terseness

Since most MXML tags represent ActionScript instances, XML attributes can be used to tersely configure the details of these objects, such as their properties, styles, and events. For example, writing an “instance tag” such as

<s:Button id="okButton" label="OK"  
 fontSize="16" click="trace('click')"/>

is convenient shorthand for writing code similar to

public var okButton:Button;  
...  
okButton = new Button();  
okButton.label = "OK";  
okButton.setStyle("fontSize", 16);  
okButton.addEventListener("click", okButton\_clickHandler);  
...

private function okButton\_clickHandler(event:MouseEvent):void  
{  
 trace('click');  
}

The MXML version is simpler, more elegant, easier to learn, and involves far fewer keystrokes.

### Naturalness

The nestability of of character data inside XML tags, and of tags inside other tags, is a natural way of representing common relationships such as type, name/value, “has”, containment, and parent/child. Here are examples of each relationship:

a) Type

In

<fx:String>true</fx:String>

the nesting of the characters ‘true’(which normally mean the Boolean value true) inside an <fx:String> tag indicates that it has type String and therefore represents the value "true".

b) Name/value

In

<s:label><fx:String>OK</fx:String></s:label>

or, more simply,

<s:label>OK</s:label>

the nesting of the value ‘OK’ inside an <s:label> tag indicates that the value of the label property of some object is "OK".

c) “Has”:

In

<s:Button>  
 <s:label>OK</s:label>  
</s:Button>

the nesting of the label tag inside the Button tag indicates “This instance of Button has a label property whose value is "OK".

Note: Simple property values are typically written using attributes instead of child tags. But child tags are sometimes required for more complex property values.

d) Containment:

In

<fx:Array>  
 <fx:String>California</fx:String>  
 <fx:String>Massachusets</fx:String>  
</fx:Array>

the nesting of the String tags inside the Array tag indicates an Array value containing two elements, each of type String .

e) Parent/child:

In

<mx:Form>  
 <mx:FormItem label="Name:">  
 <mx:TextInput id="nameTextInput"/>  
 </mx:FormItem>  
 <mx:FormItem label="Password:">  
 <mx:TextInput id="passwordTextInput"  
 displayAsPassword="true"/>  
 </mx:FormItem>  
</mx:Form>

the nesting of the FormItem tags inside the Form tag, and the TextInput tags inside the FormItem tags, indicate parent/child relationships between DisplayObjects in the Flash Player: The grandparent Form has two FormItems as children, and each of them has a child which is a TextInput. Code such as

public var nameTextInput:TextInput;  
public var passwordTextInput:TextInput;  
...  
var form:Form = new Form();  
var formItem1:FormItem = new FormItem();  
formItem1.label = "Name:";  
form.addChild(formItem1);  
nameTextInput = new TextInput();  
formItem1.addChild(nameTextInput);  
var formItem2:FormItem = new FormItem();  
formItem2.label = "Password:";  
form.addChild(formItem2);  
passwordTextInput = new TextInput();  
passwordTextInput.displayAsPassword = true;  
formItem1.addChild(passwordTextInput);

which constructs such a hierarchy using calls to addChild() is considerably harder to understand at first glance.

Each example of nesting in MXML given above seems quite natural. But, to be honest, nesting in MXML has so many different meanings that the semantics can become challenging when it comes to the more obscure combinations of tags, such as

<mx:DataGrid>  
 <mx:itemRenderer>  
 <fx:Component>  
 <mx:HBox>  
 …  
 </mx:Hbox>  
 </fx:Component>  
 </mx:itemRenderer>  
</mx:DataGrid>

(Here the itemRenderer property of the DataGrid is being set to a factory the produces instances of an “inner component” which is a subclass of HBox!)

Incidentally, in the remainder of this document, we will frequently use the terminology “child tag” to describe any nested tag, even if there is not a strict parent/child relationship between the two objects.

### Features

MXML supports higher-level concepts such as states and databinding that are not present in ActionScript. Most Flex developers rely heavily on these features and consistently rate them as extremely important.

### Toolability

The final advantage of MXML is that it is more “toolable” than ActionScript due to its simpler lexical structure: In XML everything is simply a tag, an attribute, or character data.

Of course, understanding what each of these actually means – the semantics – constitutes the difficulty, for either a human or an MXML parser. But at least the XML-based syntax is easy and fast to scan. By contrast, the token structure of ActionScript is two orders of magnitude more complicated.

Although developers don’t really care whether a language is easy or hard to support with tooling, we do because we write the tools. It is easy to provide editing support in Flash Builder for MXML than for ActionScript.

## Key concepts of MXML

Here are the most important things to understand about MXML:

* It's just XML, so everything is either a tag, an attribute, or character data.
* The root tag defines an ActionScript class, and its attributes provide default settings for all instances of that class.
* The <fx:Declarations> tag lets you use other MXML tags to declare public ActionScript variables in the class and set their initial values.
* The <fx:Script> tag lets you put arbitrary ActionScript code, such as methods, into the body of the class you are defining.
* The <fx:Metadata> tag lets you specify ActionScript metadata, such as that declaring styles and events, for the class you are defining.
* There are a number of other special purpose tags.
* Other child tags that aren't special-purpose are used, together with the attributes on the root tag, to set the properties, styles, and event handlers of the class.
* If the root tag represents a DisplayObject class, child tags are used to specify its children and their properties/styles/events; and their children…, etc.

# MXML files

## File names

An MXML file must have the extension .mxml. The extension is case-insensitive.

The file name (without the extension) must be a valid ActionScript identifier, because an MXML file defines a class with that name.

The names of the enclosing directories must also be valid ActionScript identifiers, because these become the package name for that class. (For example, the file foo/bar/MyClass.mxml defines the class foo.bar.MyClass.) TODO: Explain how the compiler knows where the “top level” directory is.

## Byte order mark

TODO: Explain this.

## Encodings

TODO: What encodings for MXML files does Flex 4’s MXML compiler support?

Recommendation for MXML 2012

Consider supporting only the various Unicode encodings and dropping support for Windows-1252, ISO-8859-1, etc. if these are currently supported.

## Illegal characters

TODO: What happens if a byte sequence in the file does not represent a Unicode character?

## Line endings

Platform-appropriate line endings are not required; an MXML file can mix \r, \n, and \r\n.

TODO: But are the line numbers correct in this case?

TODO: Are there other line ending sequences in Unicode that are supported?

# Lexical issues

## MXML as XML

Lexically, MXML is simply XML 1.0 and therefore MXML files can be read and written by a wide variety of XML tools. Understanding MXML means understanding the semantics of the various tags, attributes, and character data in an MXML file.

Structurally, the XML-ness of MXML implies that

* each MXML file can have only one top-level tag;
* namespace prefixes for qualified tag names and qualified attribute names must be declared on, or above, the tag where they are used;
* a tag cannot have multiple attributes with the same qualified name.

## XML directive

An XML directive is optional in MXML files. If present, it must immediately follow the byte order mark, or, if there is no byte order mark, be at the beginning of the file.

If the XML directive is present, it must have the attribute version="1.0".

The encoding attribute is optional; if missing, UTF-8 encoding is assumed.

Tools writing MXML files should output a complete XML directive of the form

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

## Unicode

TODO: What version of Unicode is supported? This should affect which byte sequences are considered to represent characters and which are illegal. In particular, do we support only Unicode’s Basic Multilingual Plane (BMP), which extends between U+0000 and U+FFFF, or do we support characters in the higher planes?

## Numeric character references

In XML, any Unicode character can be written using the notation &#nnnn; or &#xhhhh; where nnnn is the character's code point in decimal and hhhh is the code point in hexadecimal. (In the prevous sentence, the semicolons are part of the numeric character references.)

The x must be lowercase. The nnnn or hhhh may be any number of digits and may include leading zeros. The hhhh may mix uppercase and lowercase, though uppercase is the usual style.

The following example has a registered trademark symbol (a circled latin capital letter r, U+00AE, 'REGISTERED SIGN') after “Adobe”and “Flash”, and an unregistered trademark symbol (raised latin capital letters TM, U+2122, 'TRADE MARK SIGN') after “Builder”:

<s:Label text="Adobe&#x00AE; Flash&#x00AE; Builder&#x2122; 4"/>

Note that numeric character references make it possible to write MXML files using all of Unicode using 7-bit ASCII, if this is necessary for some reason.

Numeric character references are processed before ActionScript code is generated from MXML; the ActionScript compiler sees only the Unicode character that they specify.

## Character entity references

The following predefined entities, required by XML 1.0 for representing characters that would otherwise have special meaning in XML, are also supported:

|  |  |
| --- | --- |
| **Character entity reference** | **Character** |
| &amp; | & |
| &apos; | ' |
| &gt; | > |
| &lt; | < |
| &quot; | " |

TODO: What other character entities are predefined? &nbsp; seems to be, but not &tm; .

TODO: Can MXML files have a DTD with entity declarations?

Character entity references are processed before ActionScript code is generated from MXML; the ActionScript ompiler sees only the Unicode character that they specify.

## Special characters in attribute values

As in XML, if you need to write the less-than character (<) or the ampersand character (&) in an attribute value, you must write them using a character entity reference or a numeric character reference.

Similarly, the double-quote character (") cannot be written as itself inside an attribute value that uses this character as its delimiter, and the same applies to the apostrophe character (') . They must be written using a character entity reference or a numeric character reference.

## Special characters in character data

As in XML, if you need to write the less-than character (<) or the ampersand character (&) in character data, you must either use a CDATA block or write them using a numeric character reference or a character entity reference.

## Whitespace

TODO: What set of Unicode whitespace characters are recognized as whitespace? Does it depend on context?

The normal rules for XML whitespace inside directives and tags apply: Whitespace is required after the tag (or directive) name and between attributes. Whitespace is optional around the equals sign in attributes, and before the close of the directive/tag. Whitespace is not allowed between the less-than character and the possibly-qualified tag name, or around the semicolon that separates a namespace prefix from a tag name or attribute name.

The following examples illustrate these rules. Whitespace is required in the places indicated by a red plus sign. Whitespace is optional in the places indicated by a red asterisk.

<?xml+version\*=\*"1.0"+encoding\*=\*"utf-8"\*?>  
  
<ns:tagname+ns:attr1\*=\*"value1"+ns:attr2\*=\*"value2"\*>  
  
</ns:tagname\*>  
  
<ns:tagname+ns:attr1\*=\*"value1"+ns:attr2\*=\*"value2"\*/>

None of this whitespace has any semantic meaning.

No whitespace is allowed before the XML directive. Whitespace is allowed between the XML directive and the root tag, and after the root tag, but it has no semantic meaning.

Whitespace inside of attribute values, and inside of character data, can have semantic meaning, depending on context. This is described in detail below.

## Comments

MXML uses standard XML comments of the form

<!-- This is a comment -->

They can be used to “comment out” tags in an MXML file:

<s:TextInput .../>  
<!-- <s:TextInput .../> -->  
<s:Button .../>  
<!--  
<s:Button .../>  
<s:Button .../>  
-->

Comments are not nestable. There is no way to comment out attributes within a tag.

MXML comments are ignored when compiling a SWF, but processed when producing ASDoc from source code.

## Namespaces

The way that MXML uses XML namespaces is explained later. XML namespace prefixes may be declared on any tag, so when we discuss below what attributes are allowed on each tag, we don't bother to mention that you can *always* write namespace prefix attributes on *any* tag.

As in XML, these namespace prefixes can then be used within the tag where it is declared, or inside child tags.

MXML places no restrictions beyond those of XML on namespace prefixes or values, except that a special language namespace must be declared on the root tag. This is explained later.

## Tag names

TODO: Explain any restrictions on unqualified tag names.

## Tag order

The order in which child tags appear within a parent tag often, but not always, has meaning in MXML. For example, inside an <s:Group> tag, the order of the child tags determines which is the first child, the second child, etc. But inside an <s:Application> tag, it is meaningless whether <fx:Script> appears before or after <fx:Declarations>. As we discuss various kinds of tags, we will explain whether the order of child tags matters.

## Attribute names

TODO: Explain any restrictions on unqualified attribute names.

## Attribute order

The order in which attributes appear within a tag does not have meaning in MXML. For example, when you write

<s:Button label="OK" width="100" height="100"/>

the compiler does not guarantee that the Button's label property will be set first, followed by its width property, and then its height property; they might get set in any order.

Recommendation for MXML 2012

This is unintuitive, and in the unfortunate situations where there are interdependent proproperties, it makes it difficult to set them in the desired order. We should ensure that the attributes are applied in the order they are written.

## Tag content

By “content” we mean everything between a start tag and an end tag. There are three possibilities:

1. There is no content, as in

<s:Button label="OK"></s:Button>

or its equivalent form

<s:Button label="OK"/>

2. The content is all character data, as in

<s:Button>OK</s:Button>

or perhaps

<s:Button label="OK">  
</s:Button>

(Note that the character format in the second case is all whitespace.)

3. The content is one or more child tags with meaningless whitespace before, after, and between them, as in

<s:HGroup>  
 <s:Button label="OK"/>  
 <s:Button label="Cancel"/>  
</s:HGroup>

4. The content is "mixed", meaning that there are one or more child tags with meaningful character data before, after, and between them, as in

<s:RichText>Hello, <s:span  
 fontWeight="bold">World!</s:span><s:RichText>

When we discuss each tag, we will explain which kind of content is allowed and whether the whitespace in the character data is meaningful or not.

# Namespaces in MXML

MXML uses XML namespaces to

* declare the version of MXML being used;
* distinguish tags such as <s:Button> and <mx:Button> by mapping them to different ActionScript classes;
* mark attributes as being “private”.

Language namespaces serve the first purpose; manifest namespaces and package namespaces serve the second; private namespaces serve the third.

The language namespace must be declared on the root tag. Other namespaces may be declared or redeclared on any tag.

## Language namespaces

The root tag of an MXML file must declare a special namespace, which declares the version of MXML used within that file. Currently there are two supported language namespaces:

|  |  |  |
| --- | --- | --- |
| Version of MXML | Language namespace | Conventional prefix |
| MXML 2006 | http://www.adobe.com/2006/mxml | mx: |
| MXML 2009 | http://ns.adobe.com/mxml/2009 | fx: |

Future versions of MXML are likely to follow the more recent convention of using the form http://ns.adobe.com/mxml/<year>.

The namespace prefix assigned to a language namespace has no semantics. The current convention is to use fx: for MXML 2009 and mx: for MXML 2006. It is possible to use the empty prefix for the language namespace (or any other namespace) if you prefer to write, for example, <Script> instead of <fx:Script>.

All “language tags” (which are explained later) must be in the language namespace. The two language namespace support different language tags, as discussed later.

TODO: What other features of MXML depend on the language namespace?

## Manifest namespaces

A manifest namespace – which is a namespace associated with an manifest file – is one way of mapping an MXML tag to an ActionScript class.

A manifest file is an XML file that maps unqualified MXML tag names to fully qualified ActionScript class names. It has the format

<?xml version="1.0"?>  
<componentPackage>  
 <component id="Button" class="mx.controls.Button"/>  
 ...  
</componentPackage>

When compiling a SWF, you can use the -namespace option to specify mappings between manifest namespaces and manifest files.

The default configuration files frameworks/flex-config.xml and frameworks/air-config.xml set up the following manifest-namespace-to-manifest-file mappings

<namespaces>  
 <namespace>  
 <uri>http://ns.adobe.com/mxml/2009</uri>  
 <manifest>mxml-2009-manifest.xml</manifest>  
 </namespace>  
 <namespace>  
 <uri>library://ns.adobe.com/flex/spark</uri>  
 <manifest>spark-manifest.xml</manifest>  
 </namespace>  
 <namespace>  
 <uri>library://ns.adobe.com/flex/mx</uri>  
 <manifest>mx-manifest.xml</manifest>  
 </namespace>  
 <namespace>  
 <uri>http://www.adobe.com/2006/mxml</uri>  
 <manifest>mxml-manifest.xml</manifest>  
 </namespace>  
</namespaces>

and a typical MXML 2009 file declares

<s:Application xmlns:fx=http://ns.adobe.com/mxml/2009  
 xmlns:s="library://ns.adobe.com/flex/spark"  
 xmlns:mx="library://ns.adobe.com/flex/halo">

Therefore this file can use the s: prefix to refer to Spark classes listed in frameworks/spark-manifest.xml and the mx: prefix to refer to MX classes listed in frameworks/mx-manifest.xml. Since spark-manifest.xml has the mapping

<component id="Button" class="spark.components.Button"/>

and mx-manifest.xml has the mapping

<component id="Button" class="mx.controls.Button"/>,

the MXML compiler resolves the MXML tag <s:Button> to the ActionScript class spark.components.Button and the tag <mx:Button> to the class mx.controls.Button.

TODO: What happens if you map the same namespace to two manifests (-namespace uri manifest1 uri manifest2) and the two manifests don't agree on what a Button tag means?

TODO: Explain namespaces in SWCs.

The following table shows the manifest namespace associated with various SWCs in the Flex SDK:

|  |  |
| --- | --- |
| SWC | Namespace |
| airframework.swc | library://ns.adobe.com/flex/mx |
| airspark.swc | library://ns.adobe.com/flex/spark |
| datavisualization.swc | http://www.adobe.com/2006/mxml |
| framework.swc | library://ns.adobe.com/flex/mx |
| osmf.swc | none |
| spark.swc | library://ns.adobe.com/flex/spark |
| textLayout.swc | library://ns.adobe.com/flashx/textLayout |

## Package namespaces

A package namespace – which is a namespace associated with one particular package – is the second way of mapping an MXML tag to an ActionScript class.

The value of a package namespace has the form "\*", "foo.\*", "foo.bar.\*", etc. The part of the package namespace before the final \* is prepended to the unqualified tag name to form the fully qualified classname of the class represented by the tag, which must then be found somewhere on the compiler’s search path (i.e., its -source-path, -library-path,  
or –external-library-path).

For example, you can declare xmlns:fd="flash.display.\*" and then write <fd:Sprite> to represent the flash.display.Sprite class as an MXML tag. Note that this a way of getting around the fact that none of the Adobe-defined manifests include the Sprite class and therefore the compiler doesn't recognize the use of a manifest namespace, as in <s:Sprite> or <mx:Sprite>.

However, a more common use of package namespaces is to locate developer-written classes. For example, suppose the application MyApp.mxml lives in a myapp directory and suppose it uses an MXML component MyComp.mxml. If you put MyComp.mxml into the same myapp directory, then you can declare xmlns:my="\*" and reference the MyComp class as <my:MyComp>. If you put MyComp.mxml into a myapp/components directory, then you would declare xmlns:my="components.\*" but still reference it as <my:MyComp>.

Package namespaces can be convenient for quick-and-dirty development, but they are not considered best practice because using them in MXML code requires more changes when packages are reorganized. Also, if you organize your components into multiple packages, you need a different namespace prefix for every package, which gets tedious. Instead, it is recommended that, for non-trivial applications, developers maintain a manifest file and use a manifest namespace for their components.

## Private namespaces

Any namespace that is not a language namespace, a manifest namespace, or a package namespace is considered a private namespace. Private namespaces can be used to put private attributes on any tag. Private attributes are simply ignored by the MXML compiler, but may have meaning to tools such as Flash Builder or Flash Catalyst.

Here is an example of using private attributes on a root tag, a language tag, and an instance tag:

<s:Application xmlns:fx=http://ns.adobe.com/mxml/2009  
 xmlns:s="library://ns.adobe.com/flex/spark"  
 xmlns:mx="library://ns.adobe.com/flex/halo"  
 xmlns:private="http://private"  
 private:foo="1">  
  
 <fx:Script script="myapp.as" private:bar="2"/>  
  
 <s:Button label="OK" private:baz="3"/>  
  
</s:Application>

# Tags in MXML

Given the minimalist tag-based syntax of XML, it isn't surprising that tags are used for a variety of different purposes in MXML:

* Defining a class.
* Declaring ActionScript metadata for that class.
* Declaring variables in that class.
* Declaring ActionScript code inside that class.
* Representing instances of ActionScript types.
* Declaring CSS selectors.
* Declaring databinding relationships.
* etc.

Understanding what various tags mean is critical to understanding the language. The core of this document is the following discussion of the various types of tags that can be used in MXML files.

## Instance tags

The best way to understand tags in MXML is to start “at the bottom” with the simplest tags which represent instances of ActionScript types. In this way, three fundamentals of MXML syntax will become apparent:

* how primitive typed values are represented;
* how nested data structures are represented;
* how attributes can sometimes be used instead of child tags.

After discussing various kinds of instance tags (which will involve also introducing the concept of a property tag) we will discuss the other types of tags: style tags, event tags, definition tags, and language tags.

### Instance tags for primitive builtin types

The primitive builtin types are Boolean, int/uint/Number, String, Class, and Function.

The MXML tags for these types simply “wrap” a value, expressed as character data, to indicate its type, since this can otherwise be ambiguous. For example, <fx:Boolean>true</fx:Boolean> means the Boolean value true while <fx:String>true</fx:String> means the String value "true".

The tags for these primitive types live in the language namespace, not in a manifest or package namespace.

Primitive tags cannot have child tags; their content can only be character data, which is used to determine the value of the primitive instance.

#### Boolean tag

A <Boolean> tag in the language namesapce represents a value of the builtin type Boolean; it is the equivalent of the ActionScript literal true or false.

Examples:

<fx:Boolean>true</fx:Boolean>  
  
<fx:Boolean>false</fx:Boolean>

An empty tag, or one whose character data is all whitespace, represents the default value for type Boolean, which is false.

Recommendation for MXML 2012

An empty Boolean tag, or one whose character data is all whitespace, should not be allowed.

The character data ‘true’and ‘false’ can be written in any case; for example, the tag <fx:Boolean>fAlSe</fx:Boolean> is recognized as the ActionScript value false.

Recommendation for MXML 2012

Lowercase ‘true’ and ‘false’ in the character data should be required because MXML is generally case-sensitive and in ActionScript Boolean values must be written in lowercase.

Whitespace is allowed before and after ‘true’ but not before or after ‘false’. This inconsistency is presumably unintentional.

Recommendation for MXML 2012

Leading and trailing whitespace should be allowed so that you can format code like this:  
  
<fx:Boolean>  
 false  
</fx:Boolean>

Recommendation for MXML 2012

Consider adding <fx:true/> and <fx:false/> as shorthand for <fx:Boolean>true</fx:Boolean> and <fx:Boolean>false</fx:Boolean>.

#### int tag

An <int> tag in the language namespace represents a value of the builtin type int; it is the equivalent of an ActionScript numeric literal such as -3 or 0x01A3.

Examples:

<fx:int>-3</fx:int>  
  
<fx:int>0x01A3</fx:int>  
  
<fx:int>#01A3</fx:int>

An empty tag, or one whose character data is all whitespace, represents the default value for type int, which is 0.

Recommendation for MXML 2012

An empty int tag, or one whose character data is all whitespace, should not be allowed.

A leading plus sign is not allowed.

Recommendation for MXML 2012

A leading plus sign should be allowed, with no whitespace between it and the rest of the integer.

A negative sign is allowed, even in front of a value representing 0. Whitespace is not allowed between it and the rest of the integer.

The value may be written in either decimal or hexadecimal.

If written in decimal, it must not have leading zeros, or commas as a thousands separator, or a period as a decimal point.

Recommendation for MXML 2012

Leading zeros for ints should be allowed in MXML since they are allowed in ActionScript.

If written in hexadecimal, it must start with 0x, 0X or # and may have an unlimited number of leading zeros. It may use either A-F or a-f for hexadecimal digits.

Recommendation for MXML 2012

Using # as a indicator of hexadecimal should not be allowed for an int tag; this syntax should be allowed only for RGB style values.

The value must be within the range of possible values for int, which extend from -2147483648 (-0x80000000) to 2147483647 (0x7FFFFFFF).

If you want to write -1 in hexadecimal, you cannot write it as the bit pattern 0xFFFFFFFF; you must write it as -0x1, -0x01, etc.

Leading and trailing whitespace is not allowed.

Recommendation for MXML 2012

Leading and trailing whitespace should be allowed so that you can format code like this:  
  
<fx:int>  
 -3  
</fx:int>

#### uint tag

A <uint> tag in the language namesapce represents a value of the builtin type uint; it is the equivalent of an ActionScript numeric literal such as 3 or 0x01A3.

Examples:

<fx:uint>3</fx:uint>  
  
<fx:uint>0x01A3</fx:uint>  
  
<fx:uint>#01A3</fx:uint>

An empty tag, or one whose character data is all whitespace, represents the default value for type uint, which is 0.

Recommendation for MXML 2012

An empty uint tag, or one whose character data is all whitespace, should not be allowed.

A leading plus sign is not allowed.

Recommendation for MXML 2012

A leading plus sign should be allowed, with no whitespace between it and the rest of the integer.

The value may be written in either decimal or hexadecimal.

If written in decimal, it must not have leading zeros, or commas as a thousands separator, or a period as a decimal point.

Recommendation for MXML 2012

Leading zeros for uints should be allowed in MXML since they are allowed in ActionScript.

If written in hexadecimal, it must start with 0x, 0X or # and may have an unlimited number of leading zeros. It may use either A-F or a-f for hexadecimal digits.

Recommendation for MXML 2012

Using # as a indicator of hexadecimal should not be allowed for a uint tag; this syntax should be allowed only for RGB style values.

The value must be within the range of possible values for int, which extend from 0 to 4294967295 (0xFFFFFFFF).

Leading and trailing whitespace is not allowed.

Recommendation for MXML 2012

Leading and trailing whitespace should be allowed so that you can format code like this:  
  
<fx:uint>  
 3  
</fx:uint>

#### Number tag

A <Number> tag in the language namespace represents a value of the builtin type Number; it is the equivalent of an [ActionScript](https://zerowing.corp.adobe.com/display/FlashPlayer/ActionScript) numeric literal such as 3.14159 or 6.02e-23.

Examples:

<fx:Number>3.14159</fx:Number>  
  
<fx:Number>6.02e-23</fx:Number>  
  
<fx:Number>NaN</fx:Number>  
  
<fx:Number>+Infinity</fx:Number>

An empty tag, or one whose character data is all whitespace, represents the default value for type Number, which is NaN.

Recommendation for MXML 2012

An empty Number tag, or one whose character data is all whitespace, should not be allowed.

Unlike for int and uint, a leading plus sign is allowed for Number.

The value may be written in either decimal or hexadecimal.

If written in decimal, it can have leading/trailing whitespace. After the whitespace is trimmed, it must parse as an integer, or have a decimal point, or be written in exponential notation using e, E, e+, E+, e-, or e+ to separate the mantissa and the exponent.

If written in decimal, the value may not have leading zeros, or commas as a thousands separator, or a period as a decimal point.

If writen in decimal, there is no range checking; for example, 1e-10000 is compiled as 0 and 1e+10000 is compiled as Infinity because those are the closest Numbers.

If written in hexadecimal, the value cannot have leading/trailing whitespace and must start with 0x, 0X, or #. It may use either A-F or a-f for hex digits.

Recommendation for MXML 2012

Using # as a indicator of hexadecimal should not be allowed for Number; it should be allowed only for uints representing RGB colors.

TODO: Range checking for hex?

The special values NaN, +NaN, -NaN, Infinity, +Infinity, and -Infinity are allowed.

Recommendation for MXML 2012

Leading and trailing whitespace should always be allowed.

Recommendation for MXML 2012

Consider adding <fx:NaN/>, <fx:Infinity/>, and <fx:NegativeInfinity/> as shorthand for <fx:Number>NaN</fx:Number>, <fx:Number>Infinity</fx:Number>, and <fx:Number>-Infinity</fx:Number>.

#### String tag

A <String> tag in the language namespace represents a value of the builtin type String; it is the equivalent of an ActionScript String literal such as "Hello".

Examples:

<fx:String></fx:String>  
  
<fx:String>Hello</fx:String>  
  
<fx:String source="stuff/mystring.txt"/>  
  
<fx:String>  
 Whitespace all around me  
</fx:String>

An empty tag, or one whose character data is all whitespace, represents the default value for type String, which is null. This means that there is no way to represent the empty String "" with a String tag.

Recommendation for MXML 2012

An empty String tag should represent the empty String "".

If the character data contains non-whitespace, then the entire character data becomes the String value. For example, in the last example above, the value of the String would be "\n    Whitespace all around me\n".

Recommendation for MXML 2012

All whitespace in a String tag should always be preserved. For example, you should be able to represent a space as <fx:String> </fx:String>.

TODO: Do multiple lines of MXML cause \r, \n, or \r\n in the ActionScript value? Does it depend on what line endings the MXML file uses?

The String tag supports a compile-time source attribute. (Strings do not have such a property at runtime.) This attribute instructs the compiler to read the String’s value from a file. The syntax for the value of the source attribute is specified in the section “File paths”. The specified file must exist. (TODO: What if it is a binary file?) The extension of the file is irrelevant. This source attribute introduces a dependency between the MXML file and the source file, so that if the source file is updated, the MXML file must be recompiled.

TODO: How is whitespace in the file handled?

TODO: What encoding is the file assumed to have?

Recommendation for MXML 2012

Support an encoding attribute on the String tag, with the same supported encodings as for MXML files.

TODO: What happens to \r, \n, and \r\n in the file?

If source is specified and there is also non-whitespace character data in the String tag, the compiler warns that the character data will be ignored.

Recommendation for MXML 2012

Having both a source attribute and non-whitespace character data should be an error.

Unfortunately, curly braces inside the source file are interpreted as binding expressions, making it impossible to set the String to the contents of an arbitrary text file.

Recommendation for MXML 2012

Curly braces inside the source file should not be interpreted as binding expressions, or there should be a way to disable this interpretation so that the characters in the source file simply become the character in the string.

There seems to be where a text file containing a single lne of text without a line ending is not read properly and the resulting String is empty.

Recommendation for MXML 2012

Fix this bug.

You must use an attribute, not a child tag, to specify a source file. You cannot qualify the source attribute with the language namespace, as you might expect.

Recommendation for MXML 2012

Support both <fx:String source="…"/> and <fx:String fx:source="…"/>

If you qualify the source attribute with a manifest or package namespace, it will be ignored, as if it were a private attribute.

Recommendation for MXML 2012

Make this a compilation error, reporting an unknown attribute.

#### Class tag

A <Class> tag in the language namespace represents a value of the builtin type Class; it is the equivalent of an ActionScript Class literal such as Button.

Example:

<fx:Class>spark.components.Button</fx:Class>

An empty tag, or one whose character data is all whitespace, represents the default value for type Class, which is null.

The classname must be fully qualified and must use dot notation, as in spark.components.Button; other notations such as spark.components:Button, spark.components::Button and spark.components/Button are not allowed.

The specified class must exist in the compiler’s search path.

Leading/trailing whitespace is allowed.

#### Function tag

A <Function> tag in the language namespace represents a value of the builtin type Function, i.e., a function reference.

Examples:

<fx:Function>doSomething</fx:Function>  
  
<fx:Function>someComponent.doSomethingElse</fx:Function>

An empty tag, or one whose character data is all whitespace, represents the default value for type Function, which is null.

TODO: Explain what kind of symbolic references are allowed.

Anonymous functions such as

<fx:Function>  
 function():int { return 3; }  
</fx:Function>

are not supported; for some reason – probably a bug – this code compiles to a null Function value.

Leading/trailing whitespace is allowed.

Recommendation for MXML 2012

The Function tag should be removed from the language because it is too order-dependent and insufficiently useful.

#### Missing primitives

In ActionScript, undefined and null are special primitive values, but MXML has no tags for representing them. This is seldom a problem in practice, but the language seems incomplete without them.

Recommendation for MXML 2012

Add <fx:undefined/> and <fx:null/> tags to the language namespace so that any ActionScript value can be represented as an MXML tag.

### Instance tags for complex builtin types

The complex builtin types are Array/Vector, Object, XML/XMLList, Date, and RegExp.

The tags for these types live in the language namespace, not in a library or package namespace.

#### Array tag

An <Array> tag in the language namespace represents a value of the builtin type Array; it is the MXML equivalent of an ActionScript Array literal such as [ "abc", 3.14159 ].

Examples:

<fx:Array>  
 <fx:String>abc</fx:String>  
 <fx:Number>3.14159</fx:Number>  
</fx:Array>  
  
<fx:Array>  
 [ "abc", 3.14159 ]  
</fx:Array>

The content of the tag specifies the elements in the Array, if any. There are four possibilities:

1. An empty tag, or one whose content is all whitespace, does *not* represent the default value for type Array, which is null; instead, it represents the empty Array []. This is more useful, but is inconsistent with the primitive types.

2. Each element of the Array can be specified by a child tag, with ignored whitespace before, between, and after the child tags, as in the first example above; this makes the type of each element explicit and makes it possible to have

<fx:Array>  
 <fx:int>123</fx:int>  
 <fx:String>123</fx:String>  
</fx:Array>

represent

[ 123, "123" ]

and

<fx:Array>  
 <fx:Boolean>false</fx:Boolean>  
 <fx:String>false</fx:String>  
</fx:Array>

represent

[ false, "false" ]

3. Character data alone can specify the entire Array using an ActionScript-like square-bracket syntax. Unfortunately, the details are sufficiently different from ActionScript to be confusing:

Whitespace before and after the left and right square brackets is ignored. The character data between thebrackets is split on the comma character, and whitespace before and after is removed, to produce character data that is then intepreted for each element. This means, for example, that

<fx:Array>  
 [,,]  
</fx:Array>

represents

[ "", "", "" ]

rather than

[ undefined, undefined ]

as in ActionScript.

The character data for each element is automatically typed as follows:

1. Double-quotes or single-quotes are stripped off and the character data becomes a String.
2. Any casing of ‘true’ and ‘false’ becomes a Boolean.
3. Anything that would be legal inside a Number tag becomes a Number.
4. Anything that would be legal inside an int/uint tag becomes an int/uint.
5. RegExp, XML, and XMLList literal syntax is not recognized.
6. Otherwise, the character data is “auto-quoted” and assumed to be a String, so that

<fx:Array>  
 [ "abc", 'abc', abc ]  
</fx:Array>

represents

[ "abc", "abc", "abc" ]

Various element character data confuses the MXML compiler so much that it produces inexplicable Array values. For example,

<fx:Array>  
 [ 1, [] ]  
</fx:Array>

represents

[ 1, "", "" ]

Recommendation for MXML 2012

When specifying the Array elements in this way, the square-bracket expression should simply be used as the Array initializer in the generated ActionScript code, and therefore should obey all the normal ActionScript Array literal rules.

4. Mixed content is allowed. For example,

<fx:Array>  
 1  
 <fx:int>2</fx:int>  
 abc  
 <fx:String>def</fx:String>  
 true  
</fx:Array>

represents the 5-element array

[ 1, 2, "\n abc\n ", "def", true ]

Notice that the whitespace around ‘abc’ is significant because it was interpreted as a String, but the whitespace around ‘1’ and ‘true’ is not because they were interpreted as int and Boolean.

Except for whitespace being significant for String elements, the character data before, between, and after child tags is interpreted according to the same rules as for each comma-separated element in the square-bracket notation. Again, certain character data produce inexplicable results. For example,

<fx:Array>  
 <fx:int>1</fx:int>  
 []  
</fx:Array>

represents

[ 1, null ]

Recommendation for MXML 2012

The Array tag should not allow mixed content. There seems to be no reasonable use case for it.

Even though the Array class has length as a settable property, this property cannot be set either with a length attribute or a length child tag.

Recommendation for MXML 2012

The Array tag should support a length attribute or child tag. If either is present, other content representing elements should not be allowed. The child tag should be required to be qualified with the language namespace. The attribute should be allowed to be qualified with the language namespace.

#### Vector tag

NOTE: The Vector tag was added in MXML 2009. It does not exist in MXML 2006.

A <Vector> tag in the language namespace represents a value of the builtin type Vector.<T>; it is the MXML equivalent of an ActionScript Vector literal such as

new <String>.[ "abc", "def" ].

Example:

<fx:Vector type="String">  
 <fx:String>abc</fx:String>  
 <fx:String>def</fx:String>  
</fx:Vector>

An empty tag, or one whose content is simply whitespace character data, does *not* represent the default value for type Vector.<T>, which is null; instead, it represents an empty Vector.<T>. This is more convenient, but is inconsistent with the primitive types.

You specify the element type T using a type attribute. The value of the attribute must be a fully-qualified classname using dot notation, as in type="String" or type="spark.components.Button". Other notations such as spark.components:Button, spark.components::Button and spark.components/Button are not allowed.

NOTE: Whitespace before or after the type value causes an internal error in Flash Builder.

Recommendation for MXML 2012

Whitespace should be allowed around the type value.

NOTE: type="\*" causes a strange compilation error; this is presumably a bug.

Recommendation for MXML 2012

type="\*" should be allowed.

You specify the elements of the Vector using child tags, which must be type-assignable to the type specified by the type attribute. For example, you can make an inhomogeneous Vector like this:

<fx:Vector type="Object">  
 <fx:String>abc</fx:String>  
 <fx:Number>3.14159</fx:Number>  
</fx:Vector>

Mixed content is not allowed inside a Vector tag. The only character data allowed before, between, or after child tags is whitespace.

Even though Vector.<T> classes have length as a settable property, this property cannot be set either with a length attribute or a length child tag. The child tag should be required to be qualified with the language namespace. The attribute should be allowed to be qualified with the language namespace.

Recommendation for MXML 2012

The Vector tag should support a length attribute or child tag. If either is present, other content representing elements should not be allowed.

TODO: Explain the fixed attribute.

You must use an attribute, not a child tag, to specify type and fixed. You cannot qualify them with the language namespace, as you might expect.

Recommendation for MXML 2012

Support both <fx:Vector type="…" fixed="…"/>  
and <fx:Vector fx:type="…" fx:fixed="…"/>.

If you qualify either attribute with a manifest or package namespace, it will be ignored, as if it were a private attribute. With type this will cause an error since type is required.

Recommendation for MXML 2012

Make both an error, and the error should say that the attribute is not recognized.

#### Object tag

An <Object> tag in the language namespace represents a value of the builtin type Object; it is the MXML equivalent of an ActionScript Object literal such as

{ first: "Gordon", last: "Smith" }

An empty tag, or one whose content is simply whitespace character data, does *not* represent the default value for type Object, which is null; instead, it represents the empty Object {}. This is more convenient, but it is inconsistent with the primitive types.

An Object is essentially just a collection of name/value pairs for its properties, where the value can be of any type. Name/value pairs for properties can be written in MXML using a child tag for the name and a grandchild tag for the typed value:

Example:

<fx:Object>  
 <fx:first>  
 <fx:String>Gordon</fx:String>  
 </fx:first>  
 <fx:last>  
 <fx:String>Smith</fx:String>  
 </fx:last>  
</fx:Object>

The child tag for the name is known as a *property tag*; unlike the instance tags that we’ve been discussing, it does not represent an ActionScript instance; instead it represents a property of an ActionScript instance.

A property tag must be in the same namespace as the instance of which it is a property. In this case, since first is a property of the Object, whose tag is in the language namespace, the property tags <fx:first> and <fx:last>are also in the language namespace.

The instance tags for Boolean, int/uint/Number, and String property values can be omitted for convenience, as in

<fx:Object>  
 <fx:first>Gordon</fx:first>  
 <fx:last>Smith</fx:last>  
</fx:Object>

In this case, the character data for property values is auto-typed, so that

<fx:Object>  
 <fx:a> TRUE </fx:a>  
 <fx:b> 1 </fx:b>  
 <fx:c> abc </fx:c>  
</fx:Object>

represents

{ a: true, b: 1, c: " abc " }

Using two child tags to set the same property (even to the same value) is an error:

<fx:Object>  
 <fx:a>1</fx:a>  
 <fx:a>2</fx:a> <!-- error -->  
</fx:Object>

As an even terser syntax, you can write the property name/value pairs as attributes, as in

<fx:Object first="Gordon" last="Smith"/>

or

<fx:Object a=" TRUE " b=" 1 " c=" abc "/>

Note that whitespace around Strings is relevant but whitespace around Booleans and numeric values is not.

Using both a child tag and an attribute to set the same property (even to the same value) is an error:

<fx:Object a="1">  
 <fx:a>2</fx:a>  
</fx:Object>

Namespaces on attributes of the Object tag are peculiar in two ways.

First, giving an attribute the same namespace as the tag (which in this case is the language namespace) is not allowed, apparently because of the specialness of the language namespace.

Recommendation for MXML 2012

Since you can write <s:Button s:label="OK"/>, you should be able to write  
<fx:Object fx:a="1"/>.

Second, using a manifest or package namespace on an attribute causes the attribute to simply be ignored, as if it were a private attribute.

Recommendation for MXML 2012

Using an improper namespace on an attribute, as in <fx:Object s:a="1"/>, should cause an error.

Mixed content is not allowed inside an Object tag; the character data before, between, and after child tags must be whitespace.

#### XML tag

An <XML> tag in the language namespace represents a value of the builtin type XML; it is the MXML equivalent of an ActionScript XML literal such s

<a c="1">  
 <b d="2"/>  
</a>

An empty XML tag, or one whose content is simply whitespace character data, represents the default value for type XML, which is null.

The contents of the XML tag specify the XML value. There must be a single root tag. Whitespace before and after it is ignored. Inside the root tag, if a piece of character data is all-whitespace, then it is ignored, but otherwise it is preserved.

Example:

<fx:XML>  
 <a c="1">  
 <b d="2"/>  
 </a>  
</fx:XML>

represents the ActionScript value

<a c="1"><b d="2"></b></a>

but

<fx:XML>  
 <a c="1">  
 <b d="2"/>  
 e  
 </a>  
</fx:XML>

represents

<a c="1"><b d="2"/>  
 e  
</a>

Any namespace attributes on the XML tag are pushed down onto each tag that uses – either for itself or for any of its attributes – the specified namespace prefix. (TODO: What happens if a tag inside an XML tag also defines that namespace prefix?) For example,

<fx:XML xmlns:ns1="foo" xmlns:ns2="bar">  
 <ns1:a ns2:c="1">  
 <ns2:b d="2"/>  
 </ns1:a>  
</fx:XML>

represents

<ns1:a ns2:c="1" xmlns:ns1="foo" xmlns:ns2="bar"><ns2:b d="2" xmlns:ns2="bar"></ns2:b></ns1:a>

TODO: What happens if a tag inside an XML tag uses a namespace which is defined above the XML tag?

The XML tag supports a compile-time source attribute. (XML instances do not have such a property at runtime.) This attribute instructs the compiler to read the XML value from a file. The syntax for the value of the source attribute is specified in the section “File paths”. The specified file must exist. (TODO: What if it is a binary file?) The extension of the file is irrelevant. This source attribute introduces a dependency between the MXML file and the source file, so that if the source file is updated, the MXML file must be recompiled.

TODO: How is whitespace in the file handled?

TODO: What encoding is the file assumed to have?

Recommendation for MXML 2012

If the source file has an <?xml?> directive, use its encoding attribute, with the same supported encodings as for MXML files.

Recommendation for MXML 2012

Support an encoding attribute on the XML tag, with the same supported encodings as for MXML files. It should apply if the file does not have an <?xml?> directive. If the directive is present but doesn’t support the encoding, the encoding should be assumed to be UTF-8, because this is what XML requires.

TODO: What happens to \r, \n, and \r\n in the file?

If source is specified and there is also non-whitespace character data in the XML tag, the compiler warns that the character data will be ignored.

Recommendation for MXML 2012

Having both a source attribute and non-whitespace character data should be an error.

There seems to be where a text file containing a single lne of text without a line ending is not read properly and the resulting String is empty.

Recommendation for MXML 2012

Fix this bug.

You must use an attribute, not a child tag, to specify a source file. You cannot qualify the source attribute with the language namespace, as you might expect.

Recommendation for MXML 2012

Support both <fx:XML source="…"/> and <fx:XML fx:source="…"/>

If you qualify the source attribute with a manifest or package namespace, it will be ignored, as if it were a private attribute.

Recommendation for MXML 2012

Make this a compilation error, reporting an unknown attribute.

TODO: Explain the format attribute.

TODO: What about curly brackets insideXML tags? Do they indicate Flex databinding or E4X variable interpolation?

#### XMLList tag

An <XMLList> tag in the language namespace represents a value of the builtin type XMLList; it is the MXML equivalent of an ActionScript XMLList literal such as

<>  
 <a c="1"/>  
 <b d="2"/>  
</>

Example:

<fx:XMLList>  
 <a c="1">  
 <b d="2">  
</fx:XMLList>

represents the XMLList value above.

An empty tag, or one whose content is simply whitespace character data, does *not* represent the default value for type XMLList, which is null; instead it represents the empty XMLList <></>. This is more convenient, but is inconsistent with the primitive types.

TODO: Explain whitespace rules.

TODO: Explain namespace rules.

#### RegExp tag

A <RegExp> tag in the language namespace represents a value of the builtin type RegExp. However, it is not currently useful because all you can write is an empty tag or a tag whose character data is all whitespace.

NOTE: You can set the lastIndex property in an attribute, but this seems pointless.

An empty tag, or one whose content is simply whitespace character data, does *not* represent the default value for type RegExp, which is null; instead it represents the empty RegExp // whose source is the empty String "" and whose flags are all false. This is more convenient, but is inconsistent with the primitive types and with other instances types like Button.

Recommendation for MXML 2012

Should an empty RegExp tag represent // or null?

Recommendation for MXML 2012

Support ActionScript's RegExp literal syntax in character data. For example,

<fx:RegExp>/abc/g</fx:RegExp>

or   
  
<fx:RegExp>  
 /abc/g  
</fx:RegExp>

should be equivalent to the ActionScript RegExp literal

/abc/g;

Recommendation for MXML 2012

Support constructor parameters as attributes or child tags:

<fx:RegExp source="abc" flags="g"/>

and

<fx:RegExp>  
 <fx:source>abc<fx:source>  
 <fx:flags>g</fx:flags>  
</fx:RegExp>  
  
should have the meaning

new RegExp("abc", "g");

#### Date tag

A <Date> tag in the language namespace represents a value of the builtin type Date; it is the MXML equivalent of an ActionScript expression such as

new Date(2010, 1, 19)

The character data of the Date tag cannot be used to set its value. The character data must be empty or only whitespace.

An empty tag, or one whose content is simply whitespace character data, does *not* represent the default value for type Date, which is null; instead it represents the expression

new Date()

for the current date and time. This is more convenient, but is inconsistent with the primitive types and with other instance types like Button.

Although you cannot set the desired date using character data, you can do so with property tags or attributes. The Date class has a number of settable properties – such as fullYear, month, and date – which can be set as in the following examples:

Examples:

<fx:Date/>  
  
<fx:Date>  
 <fx:fullYear><fx:Number>2010</fx:Number></fx:fullYear>  
 <fx:month><fx:Number>1</fx:Number></fx:month>  
 <fx:date><fx:Number>19</fx:Number></fx:date>  
</fx:Date>

<fx:Date>  
 <fx:fullYear>2010</fx:fullYear>  
 <fx:month>1</fx:month>  
 <fx:date>19</fx:date>  
</fx:Date>  
  
<fx:Date fullYear="2010" month="1" date="19"/>

Note that the way you set the non-dynamic properties of a Date is similar to how you set dynamic properties of a plain Object.

As with the Object tag, the Date tag does not allow its attributes to have the same namespace of the tag, apparently because this is the language namespace. And manifest or package namespaces cause the attribute to be ignored.

For maximum type explicitness, you can specify each property using a child property tag and an grandchild instance (i.e., value) tag, as in the second example. In this case the instance tag must be type-compatible with the type of the property that you are setting. For example, writing

<fx:Date>  
 …  
 <fx:month><fx:String>1</fx:fx:String></fx:month>  
 …  
</fx:Date>

produces a compilation error because the month property is typed as Number and you cannot set it to a String.

However, there is no reason to write the value tags because the compiler knows the type of each property. In the third example, the compiler does not treat 1 as an int because it looks like an integer; it treats it as a Number because month is typed as Number.

Finally, since the compiler knows the type of each property, it is simplest to just write attributes to specify the name/value pairs for whatever properties you want to set on the instance, as in the fourth example.

### Instance tags for non-builtin classes

Non-builtin classes are those that are not part of the core ActionScript language itself. Some, like flash.display.DisplayObject, are defined inside the Flash Player. Others, like spark.components.Button, are defined in SWF files. Such types can be either dynamic or non-dynamic. They can represent visual objects or non-visual (“faceless”) ones.

MXML tags representing instances of these classes follow the similar rules as for builtin types like Date and Object.

Examples:

<mx:DateFormatter>  
 <mx:formatString>  
 <fx:String>YYYY-MM-DD</fx:String>  
 </mx:formatString>  
</mx:DateFormatter>  
  
<mx:DateFormatter>  
 <mx:formatString>YYYY-MM-DD</mx:formatString>  
</mx:DateFormatter>  
  
<mx:DateFormatter formatString="YYYY-MM-DD"/>

The general rule is that the tag name specifies the class of the instance, and its attributes and/or child tags specify properties to be set on that instance. (Attributes and child tags on instance tags can also have additional meanings that will be explained later.)

The differences between tags for builtin classes and tags for non-builtin classes are:

* Tags for non-builtin classes use a manifest or package namespace rather than the language namespace.
* Property attributes may use the same namespace as the tag, as in  
  <mx:DateFormatter mx:formatString="YYYY-MM-DD"/>.

Property attribute using another manifest or package namespace are ignored, as if they were private attributes.

Recommendation for MXML 2012

Make this a compilation error. To be a private attribute, you should use a namespace which is neither a language namesapce nor a manifest namesapce nor a package namespace.

With an understanding of instance and property tags, we are now at the point where we understand all the various ways that MXML allows a property to be set. Consider setting the text of a Spark Label, for example. We can write

<s:Label>  
 <s:text><fx:String>Hello World</fx:String></s:text>  
</s:Label>

or

<s:Label>  
 <s:text>Hello World</s:text>  
<s:Label>

or

<s:Label label="Hello World">

or

<s:Button s:label="Hello World">

or

<s:Label>Hello World<s:Label>

(The last example works because text is the default property of Label.)

TODO: Explain exactly how Label inside of Application makes a parent/child relationship.

### Instance tags for dynamic classes

One would expect that, for a dynamic class, you could use property tags and attributes to set name/pairs for any property. For some reason, this is not the case.

The following results are from trying tests such a

<e:FlexEvent e="mx.events.\*" foo="1">  
 <e:bar>2</e:bar>  
</e:FlexEvent>

Trying to use a child tag to do this, using any declared namespace, results in a compilation error that the tag cannot be resolved. Trying to use an attribute to do this results in one of two compilation errors. If you use the default namespace, or the language namespace, the message says that the attribute cannot be resolved. If you use a manifest namespace such as s: or mx:, the message says “Incorrect number of arguments”, which is even more unexpected.

Recommendation for MXML 2012

Allow name/value pairs for properties to be set on dynamic classes, using either child tags or attributes.

### Instance tags with special compile-time treatment

A few ActionScript classes receive special treatment by the MXML compiler.

#### Repeater tag

TODO

#### HTTPService tag

TODO

#### WebService tag

TODO

## Property tags and property attributes

We have already seen that , in addition to *instance tags*, MXML has *property tags* and *property attributes* for setting properties on instances. In the example above,

<mx:formatString>…</mx:formatString>

Is a property tag, and

formatString="YYYY-MM-DD"

is a property attribute. The attribute form is more convenient when the property value is a simple scalar.

Property tags must have the same namespace – which doesn’t necessarily mean the same namespace prefix – as their instance tag.

Property tags do not have tag-specific attributes.

Property attributes do not require any namespace prefix, but may be written with one as long as it is for the namespace of the instance tag. Therefore, it is possible to write either

<mx:DateFormatter formatString="YYYY-MM-DD"/>

or

<mx:DateFormatter mx:formatString="YYYY-MM-DD"/>

to set the formatString property of the DateFormatter.

If the instance is of a non-dynamic class, then its properties are defined by its public variables and setter functions, all of which have a type. (Here we are considering \* to be a type, even though it technically isn’t. The absence of any type annotation is equivalent to the :\* annotation.)

The value specified in MXML for the property must be compatible with the type of the property. For example, writing

<mx:DateFormatter>  
 <mx:formatString>  
 <fx:int>123</fx:int>  
 </mx:formatString>  
</mx:DateFormatter>

is an error; the formatString property of DateFormatter is a String, and you cannot assign an int to it.

### Boolean properties

A property tag for a Boolean property must have content specifying a Boolean value. The content can be either a Boolean tag (with optional leading or trailing whitespace) or character data which is intepretable as a Boolean value.

An attribute for a Boolean property must have a value which is interpretable as a Boolean value.

The interpretation in both cases is the same as that which is applied to the character data of a Boolean tag, with the exception that emptiness, or all whitespace, is not allowed.

Examples:

Consider a component with a booleanProperty:Boolean property. All of the following

<my:Component>  
 <my:booleanProperty>  
 <fx:Boolean>true</fx:Boolean>  
 </my:booleanProperty>  
</my:Component>  
  
<my:Component>  
 <my:booleanProperty>true</my:booleanProperty>  
</my:Component>  
  
<my:Component booleanProperty="true"/>

set this property to true.

NOTE: In ActionScript, you can assign any value to a Boolean variable and implicit coercion occurs, but this is not the case in MXML.

Recommendation for MXML 2012

Should any value coerce to Boolean, as in ActionScript?

### int properties

A property tag for an int property must have content specifying an int value. The content can be either an int tag (with optional leading or trailing whitespace) or character data which is interpretable as an int value.

An attribute for an int property must have a value which is interpretable as an int value.

The interpretation in both cases is the same as that which is applied to the character data of an int tag, with the exception that emptiness, or all whitespace, is not allowed.

Examples:

Consider a component with an intProperty:int property. All of the following

<my:Component>  
 <my:intProperty>  
 <fx:int>-1</fx:int>  
 </my:intProperty>  
</my:Component>  
  
<my:Component>  
 <my:intProperty>-1</my:intProperty>  
</my:Component>  
  
<my:Component intProperty="-1"/>

set this property to -1.

NOTE: In ActionScript, you can assign a value of type uint or Number to a variable of type int, and implicit coercion occurs. This is not the case in MXML.

Recommendation for MXML 2012

uint and Number tags should be allowed as values of int property tags, because ActionScript allows a uint or a Number to be assigned to an int.

### uint properties

A property tag for a uint property must have content specifying a uint value. The content can be either a uint tag (with optional leading or trailing whitespace) or character data which is interpretable as a uint value.

An attribute for a uint property must have a value which is interpretable as a uint value.

The interpretation in both cases is the same as that which is applied to the character data of a uint tag, with the exception that emptiness, or all whitespace, is not allowed.

Examples:

Consider a component with a uintProperty:uint property. All of the following

<my:Component>  
 <my:uintProperty>  
 <fx:uint>0x8A</fx:uint>  
 </my:uintProperty>  
</my:Component>  
  
<my:Component>  
 <my:uintProperty>0x8A</my:uintProperty>  
</my:Component>  
  
<my:Component uintProperty="0x8A"/>

set this property to 0x8A.

NOTE: In ActionScript, you can assign a value of type int or Number to a variable of type uint, and implicit coercion occurs. This is not the case in MXML.

Recommendation for MXML 2012

int and Number tags should be allowed as values of uint property tags, because ActionScript allows an int or a Number to be assigned to a uint.

### Number properties

A property tag for a Number property must have content specifying a Number value. The content can be either a Number tag (with optional leading or trailing whitespace) or character data which is interpretable as a Number value.

An attribute for a Number property must have a value which is interpretable as a Number value.

The interpretation in both cases is the same as that which is applied to the character data of a Number tag, with the exception that emptiness, or all whitespace, is not allowed.

Examples:

Consider a component with a numberProperty:Number property. All of the following

<my:Component>  
 <my:numberProperty>  
 <fx:Number>1.2</fx:Number>  
 </my:numberProperty>  
</my:Component>  
  
<my:Component>  
 <my:numberProperty>1.2</my:numberProperty>  
</my:Component>  
  
<my:Component numberProperty="1.2"/>

set this property to 1.2.

NOTE: In ActionScript, you can assign a value of type int or uint to a variable of type Number, and implicit coercion occurs. This is not the case in MXML.

Recommendation for MXML 2012

int and uint tags should be allowed as values of Number property tags, because ActionScript allows int and uint to be assigned to a Number.

#### PercentProxy metadata

TODO

### String properties

A property tag for a String property must have content specifying a String value. The content can be either a String tag (with optional leading or trailing whitespace) or arbitrary character data, which will be treated as the String value.

TODO: Explain whitespace rules.

Examples:

Consider a component with a stringProperty:String property. All of the following

<my:Component>  
 <my:stringProperty>  
 <fx:String>Hello</fx:String>  
 </my:stringProperty>  
</my:Component>  
  
<my:Component>  
 <my:stringProperty>Hello</my:stringProperty>  
</my:Component>  
  
<my:Component stringProperty="Hello"/>

set this property to "Hello".

Since the property is known to have type String, no type inference occurs on property values that are parseable as other types. For example,

<my:Component>  
 <my:stringProperty>123</my:stringProperty>  
</my:Component>

and

<my:Component stringProperty="123"/>

set stringProperty to "123", not 123.

#### CollapseWhiteSpace metadata

TODO

### Class properties

A property tag for a Class property must have content specifying a Class value. The content can be either a Class tag (with optional leading or trailing whitespace) or character data which is interpretable as a Class value.

An attribute value for a Class property will be interpreted as a Class value.

The interpretation in both cases is the same as that which is applied to the character data of a Class tag, with the exception that emptiness, or all whitespace, is not allowed.

Examples:

Consider a component with a classProperty:Class property. All of the following

<my:Component>  
 <my:classProperty>  
 <fx:Class>spark.components.Button</fx:Class>  
 </my:classProperty>  
</my:Component>  
  
<my:Component>  
 <my:classProperty>  
 spark.components.Button  
 </my:classProperty>  
</my:Component>  
  
<my:Component classProperty="spark.components.Button"/>

set this property to spark.components.Button.

#### IFactory properties

TODO

### Function properties

A property tag for a Function property must have content specifying a Function value. The content can be either a Function tag (with optional leading or trailing whitespace) or character data which is interpretable as a Function value.

An attribute value for a Function property will be interpreted as a Function value.

The interpretation in both cases is the same as that which is applied to the character data of a Function tag, with the exception that emptiness, or all whitespace, is not allowed.

Examples:

Consider a component with a functionProperty:Function property. All of the following

<my:Component>  
 <my:functionProperty>  
 <fx:Function>myFunc</fx:Function>  
 <my:functionProperty>  
</my:Component>  
  
<my:Component>  
 <my:functionProperty>  
 myFunc  
 </my:functionProperty>  
</my:Component>  
  
<my:Component functionProperty="myFunc"/>

set this property to myFunc.

### Array properties

A property tag for an Array property must have content specifying an Array value. The content can be either an Array tag (with optional leading or tailing whitespace) or character data which is interpretable as an Array value.

An attribute value for an Array property will be interpreted as a Array value.

The interpretation in both cases is the same as that which is applied to the character data of a Array tag, with the exception that emptiness, or all whitespace, is not allowed.

Examples:

Consider a component with an arrayProperty:Array property. All of the following

<my:Component>  
 <my:arrayProperty>  
 <fx:Array>  
 <fx:int>123</fx:int>  
 <fx:String>123</fx:String>  
 </fx:Array>  
 </my:arrayProperty>  
</my:Component>  
  
<my:Component>  
 <my:arrayProperty>[ 123, "123" ]</my:arrayProperty>  
</my:Component>  
  
<my:Component arrayProperty="[ 123, '123' ]"/>

set this property to [ 123, "123" ].

Recommendation for MXML 2012

Consider allowing Array properties to be set with comma-separated lists.

#### Implicit Array tags

For convenience, you can omit the Array tag when setting a property of type Array, and simply nest the Array-element tags inside the property tag.

Example:

<my:Component>  
 <my:arrayProperty>  
 <fx:int>123</fx:int>  
 <fx:String>123</fx:String>  
 </my:arrayProperty>  
</my:Component>

If you specify a single child tag inside the property tag, you get an Array of length 1.

TODO: Can you use mixed content in this case?

### Vector properties

A property tag for a Vector.<T> property must have content specifying a Vector.<T> value. The content must be a <Vector type="T"> tag or all whitespace.

Vector properties cannot be set with attributes.

Examples:

Consider a component with a vectorOfIntProperty:Vector.<int> property. The following

<my:Component>  
 <my:vectorOfIntProperty>  
 <fx:Vector type="int">  
 <fx:int>123</fx:int>  
 <fx:int>456</fx:int>  
 </fx:Vector>  
 <my:vectorOfIntProperty>  
</my:Component>

sets this property to new Vector.<int>[ 123, 456 ].

#### Implicit Vector tags

For convenience, you can omit the Vector tag when setting a property of type Vector, and simply nest the Vector-element tags inside the property tag.

Example:

<my:Component>  
 <my:vectorOfIntProperty>  
 <fx:int>123</fx:int>  
 <fx:int>456</fx:int>  
 </my:vectorOfIntProperty>  
</my:Component>

If you specify a single child tag inside the property tag, you get a Vector of length 1.

### Object properties

A property tag for an property of type Object must have content specifying an value of type Object. Since Object is the base class for other types, you can set properties of type Object to anything (except for undefined, which cannot currently be specified in MXML).

Examples: Consider a component with an objectProperty:Object property.

1. Setting a property of type Object to a Boolean value

<my:Component>  
 <my:objectProperty>  
 <fx:Boolean>true</fx:Boolean>  
 </my:objectProperty>  
</my:Component>

or

<my:Component>  
 <my:objectProperty>true</my:objectProperty>  
</my:Component>

or

<my:Component objectProperty="true"/>

sets objectProperty to true.

1. Setting a property of type Object to an int/uint/Number value

<my:Component>  
 <my:objectProperty>  
 <fx:int>7</fx:int>  
 </my:objectProperty>  
</my:Component>

or

<my:Component>  
 <my:objectProperty>7</my:objectProperty>  
</my:Component>

or

<my:Component objectProperty="7"/>

sets objectProperty to 7.

1. Setting a property of type Object to a String value

<my:Component>  
 <my:objectProperty>  
 <fx:String>abc</fx:String>  
 </my:objectProperty>  
</my:Component>

or

<my:Component>  
 <my:objectProperty>abc</my:objectProperty>  
</my:Component>

or

<my:Component objectProperty="abc"/>

sets objectProperty to "abc".

1. Setting a property of type Object to an Array value

<my:Component>  
 <my:objectProperty>  
 <fx:Array>  
 <fx:int>123</fx:int>  
 <fx:String>123</fx:String>  
 </fx:Array>  
 </my:objectProperty>  
</my:Component>

or

<my:Component>  
 <my:objectProperty>[ 123, "123" ]</my:objectProperty>  
</my:Component>

or

<my:Component objectProperty="[ 123, '123' ]"/>

sets objectProperty to [ 123, "123" ].

1. Setting a property of type Object to a Vector value

<my:Component>  
 <my:objectProperty>  
 <fx:Vector type="int">  
 <fx:int>123</fx:int>  
 <fx:int>456</fx:int>  
 </fx:Vector>  
 </my:objectProperty>  
</my:Component>

sets objectProperty to new Vector.<int>[ 123, 456 ].

1. Setting a property of type Object to a plain Object value

<my:Component>  
 <my:objectProperty>  
 <my:a><fx:int>123</fx:int></my:a>  
 <my:b><fx:String>123</fx:String></my:b>   
 <my:objectProperty>  
</my:Component>

sets objectProperty to { a: 123, b: "123" }.

1. Setting a property of type Object to a Date value

<my:Component>  
 <my:objectProperty>  
 <fx:Date fullYear="2010" month="3" date="22"/>  
 </my:objectProperty>  
</my:Component>

sets objectProperty to new Date(2010, 3, 22).

1. Setting a property of type Object to a Button value

<my:Component>  
 <my:objectProperty>  
 <s:Button label="OK"/>  
 </my:objectProperty>  
</my:Component>

sets objectProperty to an instance of Button whose label is ="OK".

### \* properties

A property tag for a untyped property can have its value set in the same way as a property of type Object.

In the future, if we have add an <fx:undefined> tag to MXML 2012, you will be able to write

<my:Component>  
 <my:untypedProperty>  
 <fx:undefined/>  
 </my:untypedProperty>  
</my:Component>

to set untypedProperty:\* to undefined. This would not be allowed on objectProperty:Object.

### XML properties

A property tag for an XML property must have content specifying an XML value. The content must be an XML tag; character data cannot be used to set the XML value.

An attribute cannot be used to set an XML property.

Example:

<my:Component>  
 <my:xmlProperty>  
 <fx:XML>  
 <employees>  
 <employee first="Gordon" last="Smith"/>  
 <employee first="Corey" last="Lucier"/>  
 <employees>  
 </fx:XML>  
 <my:xmlProperty>  
</my:Component>

### XMLList properties

A property tag for an XMLList property must have content specifying an XMLList value. The content must be an XMLList tag; character data cannot be used to set the XMLList value.

An attribute cannot be used to set an XMLList property.

Example:

<my:Component>  
 <my:xmlListProperty>  
 <fx:XMLList>  
 <employee first="Gordon" last="Smith"/>  
 <employee first="Corey" last="Lucier"/>  
 </fx:XMLList>  
 <my:xmlListProperty>  
</my:Component>

### Properties of other types

Previous subsections have discussed properties of specific builtin types. Properties of all other types can be set using standard instance tags, each of which can have its own properties.

Examples: Consider a component with properties dateProperty:Date and buttonProperty:Button.

<my:Component>  
 <my:dateProperty>  
 <fx:Date fullYear="2010" month="3" date="22"/>  
 </my:dateProperty>  
 <my:buttonProperty>  
 <fx:Button label="OK" click="trace('clicked')"/>  
 </my:buttonProperty>  
</my:Component>

### Implicit property tags

A class can designate one its properties (which must be implemented either as a public setter or as a public var) as its “default property” using [DefaultProperty] metadata. For example, the Spark Label class declare text as its default property:

[DefaultProperty("text")]  
  
public class Label …  
{  
 …  
 public set text(value:String):void  
 …  
}

When a class has a default property, its instances – and, since this metadata is inheriting, instances of its sublclasses (unless they specify a different default property) -- can omit that property tag or property attribute and specify the property value as . For example, instead of writing

<s:Label>  
 <s:text><fx:String>Hello World</fx:String></s:text>  
</s:Label>

or

<s:Label>  
 <s:text>Hello World</s:text>  
</s:Label>

or

<s:Label text="Hello World"/>

you can write

<s:Label><fx:String>Hello World</fx:String></s:Label>

or

<s:Label>Hello World</s:Label>

Note that in the latter two examples, the fact that you are setting the text property is no longer explicit.

TODO: Explain how default properties work when other properties are being set at the same time.

## Style tags and style attributes

In the same way that an instance tag can have property tags and/or property attributes that specify its properties, it can have style tags and/or style attributes that specify the CSS styles for that instance. For example, a style attribute such as fontSize="20" in a tag like

<s:Button id="myButton" fontSize="20"/>

specifies the value of one CSS style – fontSize – on the instance; it is the equivalent of the ActionScript code myButton.setStyle("fontSize", 20). (The id attribute will be explained later.)

The compiler understands what properties an instance can have based on its variables and setters, which are part of the core ActionScript language. By contrast, CSS styles are implemented in the Flex framework and are not a feature of ActionScript; they are declared by placing [Style] metadata on classes. An example is the metadata

[Style(name="fontSize", type="Number",  
 format="Length", inherit="yes")]

as declared on a superclass of the Spark Button.

The name and type attributes tell the compiler that, when it sees an attribute named fontSize, it should interpret it as specifying an style of type Number. Therefore, the call to setStyle() sets the value as 20, not "20".

Style values are interpreted based on their type in the same way that property values are. For example, if myStyle is declared with type=="Boolean" then the attribute myStyle="true" sets it to true.But if myStyle is declared with type=="String" then the attribute myStyle="true" sets it to "true".

Similarly, the rules for whitespace in style tags and style attributes are the same as for whitespace in property tags and property attributes.

And finally, the rules for namespaces for style tags and style attributes work the same way as namespaces for property tags and property attributes.

### Named colors

When a style of type int, uint, or Number is defined using [Style] metadata with a format="Color" attribute (note: the “C” in “Color” must be capitalized), then the compiler recognizes the following strings as color names and substitutes the corresponding numeric value:

|  |  |
| --- | --- |
| **Named Color** | **Value** |
| aqua | 0x00FFFF |
| black | 0x000000 |
| blue | 0x0000FF |
| cyan | 0x00FFFF |
| fuschia | 0xFF00FF |
| gray | 0x808080 |
| green | 0x008000 |
| haloBlue | 0x009DFF |
| haloGreen | 0x80FF4D |
| haloOrange | 0xFFB600 |
| haloSilver | 0xAECAD9 |
| lime | 0x00FF00 |
| magenta | 0xFF00FF |
| maroon | 0x800000 |
| navy | 0x000080 |
| olive | 0x808000 |
| purple | 0x800080 |
| red | 0xFF0000 |
| silver | 0xC0C0C0 |
| teal | 0x008080 |
| white | 0xFFFFFF |
| yellow | 0xFFFF00 |

The color names must be in lowercase with no leading or trailing whitespace. Specifying a string which is not in the above table is an error.

Recommendation for MXML 2012

Leading and trailing whitespace should be allowed, to permit formatting code like this:

<s:Button>  
 <s:color>  
 red  
 </s:color>  
</s:Button>

Recommendation for MXML 2012

The color names should not be hard-coded in the compiler's Java code. Instead, the flex-config.xml and air-config.xml compiler configuration files should have a section mapping color names to RGB values, giving developers control over this.

## Event tags and event attributes

An instance tag can also have event tags and/or event attributes that specify code for handling events. For example, an event attribute such as click="trace('I was clicked. ')" in a tag such as

<s:Button id="myButton" click="trace('I was clicked.');"/>

specifies that when this Button instance dispaches a "click" event, the code

trace("I was clicked")

should be executed.

Note that events are rather different from properties and styles: When you specify a property or style, you specify its ActionScript value. When you specify an event, you specify ActionScript code execute.

If you use an event tag to write the event handling code, by convention you should always wrap a CDATA block around the code, so that characters such as the left-angle-bracket (<) are not treated as XML syntax..

<s:Button id="myButton">  
 <s:click>  
 <![CDATA[

trace("I was clicked");  
 ]]>  
 </s:click>  
</s:Button>

The code that you specify in the event attribute or event tag becomes the body of an autogenerated event handler,with a name such as \_\_myButton\_clickHandler, as in

public function \_\_myButton\_clickHandler(  
 event:flash.events.MouseEvent):void  
{  
 trace("I was clicked");  
}

This event handler is registered with code such as

myButton.addEventListener("click", \_\_myButton\_clickHandler);

The autogenerated event handler is public.

Recommendation for MXML 2012

Autogenerated event handlers should be private so thata future compiler can omit their names from the constant pool..

It always uses event as the name of its argument, so that you can write code such as

<s:Button id="myButton" click="trace(event.target);"/>

The compiler understands what events an instance can handle based on [Event] metadata on classes. An example is the metadata

[Event(name="click", type="flash.events.MouseEvent")]

as declared on InteractiveObject, a superclass of the Spark Button.

The name and type attributes tell the compiler that, when it sees an attribute named click, it should autogenerate an event handler whose argument is type flash.events.MouseEvent.

Whitespace in event tags and event attributes has the same meaning as in an ActionScript method body, since this is where the event handling code you write ends up.

The rules for namespaces for event tags and event attributes work the same way as namespaces for property tags and property attributes.

TODO: Attributes like click="clickHandler(event)" should be optimized since there is no need to wrap this event handler into another autogenerated event handler.

## id attribute

An id attribute is allowed on any instance tag. It is a attribute with special compile-time meaning, because it specifies an ActionScript identifier for that instance which can be used in other ActionScript code, such as Script tags and event handlers.

For example, consider the simple application MyApp.mxml:

<s:Application …>  
  
 <s:Label id="label1" text="Hello"/>  
   
 <s:Button label="Change Label"  
 click="label1.text = 'Goodbye';"/>  
  
</s:Application>

Since the Label instance specifies that its identifer is label1, the Button’s click handler can use this identifier to set the Label’s text property with the statement

label1.text = 'Goodbye';

The compiler accomplishes this by autogenerating an instance variable in the class that it autogenerates from the MXML file:

package  
{  
  
import …  
  
public class MyApp extends Application  
{  
 …  
  
 [Bindable]  
 public var label1:Label;  
  
 …

}  
  
}

The specified id must be a valid ActionScript identifier. It cannot have leading or trailing whitespace.

The type of the autogenerated variable is the class which the instance tag represents. (In this case, <s:Label> represents spark.components.Label).

The autogenerated variable is public

Recommendation for MXML 2012

We should introduce another special compile-time attribute – access, with possible values public, protected, private, internal, or any in-scope namespace identifier such as mx\_internal – to give developers control over the variable’s access specifier.

It is also marked as [Bindable], to allow databinding expressons such as {label1.text} to work.

Recommendation for MXML 2012

Bindability is too expensive to be the default behavior for every instance with an id. Most will probably never be involved in a databinding expression. We should change the default to make id’s nonbindable and introduce another special compile-time attribute – bindable, with possible values true or false – to enable binding.

It is not possible to use an id child tag to specify the ActionScript identifier for an instance; you must use an attribute.

The id attribute can either be in the default namespace (no prefix) or in the language namespace.

Some classes may have a property, style, or event named id. (For example, all UIComponents have an id:String property.) In this case, using the default namespace for the id attribute will create the ActionScript identifier and also set the property/style/event. If you want to only specify the identifier, you must use the language namespace. If you want to only set the property/style/event, you must use the tag’s namespace.

## Class definition tags

So far we’ve seen how MXML tags can create instances of classes, specify the values of properties and styles for instances, and specify event-handling code for instances.

Now it’s time to explain how certain MXML tags can define new ActionScript classes, which can then be used to create instances. We will call such tags *class definition tags*. There are three kinds of tags which define classes:

* root tags;
* the child tag of a Component tag;
* the child tag of a Definition tag.

Root tags will be explained in this section. Component and Definition tags will be discussed in the following section.

In each case, the tag does *not* specify the class being *defined*; instead it specifies the class being *extended* (i.e., the superclass of the class being defined). This is one of the less obvious things about MXML.

### Root tags

The outermost tag in an MXML file – either for an application or for a component – is known as the root tag. Root tags are the most important tag, because they are the only required tag in an MXML file.

Example:

The following MXML file defines the component com.myco.OKButton as a trivial extension of the Spark Button component in which all instances by default have their label set to "OK" and their fontSize to 24, and which trace out when they are clicked.

com/myco/BetterButton.mxml:

<?xml version="1.0" encoding="utf-8"?>  
<s:Button xmlns:fx="http://ns.adobe.com/mxml/2009"  
 xmlns:s="library://ns.adobe.com/flex/spark"  
 implements="com.myco.IFoo, com.myco.IBar"  
 label="OK" fontSize="24" click="trace('click')"/>

This example is essentially equivalent to writing

package com.myco  
{  
  
import spark.components.Button;  
public class BetterButton extends spark.components.Button  
{  
 public class OKButton()  
 {  
 super();  
 label = "OK";  
 setStyle("fontSize", 24);  
 addEventListener("click", \_\_OKButton\_Button\_click);  
 }  
  
 public function \_\_OKButton\_Button\_click(  
 event:MouseEvent):void  
 {  
 trace('click')  
 }  
}  
  
}

The key things to note are that

* the name of the MXML file without the .mxml extension becomes the name of the ActionScript class being defined;
* the directory path of the MXML file becomes the package in which this class is defined;
* the ActionScript class corresponding to the root tag – spark.components.Button – is the class being extended, not the class being defined;
* the class being extended can be any non-final class; it does not have to some kind of DisplayObject;
* attributes on the definition tag set properties, styles, and event handlers that are applied at instance construction time;
* the root tag must declare a language namespace (which in the above example is the MXML 2009 namespace, "http://ns.adobe.com/mxml/2009".

The mapping of a class definition tag to an ActionScript class follows the same rules as the mapping of an instance tag to a class.

#### Properties, styles, and events

Class definition tags can have property values, style values, and event handlers specified either by either child tags or by attributes, just as for instance tags. These autogenerate code within the autogenerated constructor, so that they get applied automatically to every new instance.

For some reason, probably unintentional, properties, styles, and events declared in the MXML file for the class being defined cannot have the defaults set on the root tag. The root tag can only set inherited properties.

Recommendation for MXML 2012

Allow the root tag to specify defaults for properties, styles, and events that are newly defined in the MXML class.

#### implements attribute

A class definition tag can use an implements attribute to specify one or more ActionScript interfaces that it implements. (Interfaces can only be defined in ActionScript, and not in MXML.)

Each interface must be specified as a fully-qualified name using dot notation, as in com.myco.IButton. Other notations such as com.myco:IButton, com.myco::IButton and com.myco/IButton are not allowed.

You can use a comma-separated list to specify multiple interfaces. There can be whitespace before and after the list, and before and after each comma.

#### Other special attributes

An id attribute is not allowed on a class definition tag, because an id attribute specifies an identifier that refers to an instance.

The state attributes includeIn and excludeFrom (which are explained later) are not allowed on class definition tags.

There is no way to declare that an MXML-generated class should be dynamic.

Recommendation for MXML 2012

Class definition tags should support a dynamic attribute which can be set to true or false, with leading and trailing whitespace allowed..

#### Creating instances

Once you have defined a com.myco.OKButton class with an MXML file such as com/myco/OKButton.mxml, you can create instances of this class either in other MXML files or in ActionScript.

In MXML, the easiest way to use the component is with a package namespace. For example, you could put xmlns:m="com.myco.\*" on the root tag of another component and inside use the tag <m:OKButton>.

A more elegant way to use the component in MXML is to include it in a manifest file so that you can use a manifest namespace.

In ActionScript, you can use the OKButton class if you simply import it:

import com.myco.OKButton;

The import statement is not necessary if the file using OKButton is in the same com.myco package.

### Application tags

Root tags for applications are a special case:

* they give additional meaning to some standard Application properties;
* they can have additional special attributes;
* they autogenerate additional ActionScript code for application initialization.

Several application attributes specify settings for [SWF] metadata on the generated class. This metadata is processed later by the MXML compiler, but it isn't specific to MXML. For example, it can be written by hand even in ActionScript-only applications. Therefore that processing is not discussed here.

### Ordinary attribute with special meaning

TODO: Are the parsing rules for these attributes the same as for a property or style attribute?

#### backgroundColor

The attribute value is parsed as a uint and used to set the backgroundColor attribute in the [SWF] metadata on the autogenerated application subclass.

#### styleName

The attribute value is parsed as a String. TODO: How is it specially processed?

#### width and height

The attribute value is parsed as a Number or a percentage String and used to set the width and height attributes in the [SWF] metadata.

### Special attributes

There are certain attributes that may appear on the root tag, but which aren't properties, styles, or events.

You can put these attributes on any class definition tag, not just on a root tag for an application, but they are ignored in that case.

Recommendation for MXML 2012

Using these special attributes on a class definition tag that isn't the root tag for an application should be an error.

TODO: Are the parsing rules for these attributes the same as for a property or style attribute?

#### frameRate

The attribute value is parsed as a uint and used to set the frameRate attribute in the [SWF] metadata.

#### lib

TODO

#### pageTitle

The attribute value is parsed as a String and used to set the pageTitle attribute in the [SWF] metadata.

#### preloader

TODO:

#### rsl

TODO

#### scriptRecursionLimit

The attribute value is parsed as a uint and used to set the scriptRecursionLimit attribute in the [SWF] metadata.

#### scriptTimeLimit

The attribute value is parsed as a uint and used to set the scriptTimeLimit attribute in the [SWF] metadata.

#### theme

TODO

#### usePreloader

The attribute value is parsed as a Boolean and used to set the usePreloader attribute in the [SWF] metadata.

### Automatic imports

For convenience, the generated class has import statements for many – but not all – of the packages in playerglobal.swc or airglobal.swc:

import flash.accessibility.\*;  
import flash.data.\*; // AIR-only  
import flash.debugger.\*;  
import flash.desktop.\*; // AIR-only  
import flash.display.\*;  
import flash.errors.\*;  
import flash.events.\*;  
import flash.external.\*;  
import flash.filesystem.\*; // AIR-only  
import flash.geom.\*;  
import flash.html.\*; // AIR-only  
import flash.html.script.\*; // AIR-only  
import flash.media.\*;  
import flash.net.\*;  
import flash.printing.\*;  
import flash.profiler.\*;  
import flash.system.\*;  
import flash.text.\*;  
import flash.ui.\*;  
import flash.utils.\*;  
import flash.xml.\*;

The following documented packages are missing:

import flash.filters.\*;  
import flash.net.drm.\*;  
import flash.sampler.\*;  
import flash.security.\*;  
import flash.text.engine.\*;  
import flash.text.ime.\*;

Recommendation for MXML 2012

flash.text.engine and flash.text.ime, and perhaps others, should be added.

The following Flex packages are also automatically imported:

import mx.binding.\*;  
import mx.core.ClassFactory;  
import mx.core.DeferredInstanceFromClass;  
import mx.core.DeferredInstanceFromFunction;  
import mx.core.IDeferredInstance;  
import mx.core.IFactory;  
import mx.core.IFlexModuleFactory;  
import mx.core.IPropertyChangeNotifier;  
import mx.core.mx\_internal;  
import mx.filters.\*;  
import mx.styles.\*;

TODO: Explain why.

## Language tags

Language tags represent special language features of MXML. The most comon ones are Script and Style.

While most tags in MXML, such as <s:Button>, represent ActionScript instances (or their properties, styles, and events), language tags are an exception to this typical usage. Each language tag has its own special rules and is interpreted by the MXML compiler in a particular way; understanding one language tag does not help you to understand the others.

For each language tag, we will explain

* what it means;
* where it can be used;
* what attributes it can have;
* what child tags it can have;
* what character data it can have.

All language tags must be used in the language namespace. Some language tags can be used only with the MXML 2009 namespace.

All language tags are optional; none of them are required to be present in an MXML file.

### Binding tag

A Binding tag can be used instead of {…} or @{…} syntax to specify databinding. (The first syntax indicates “one-way databinding”; the second syntax indicates “two-way databinding”.)

Example:

<s:TextInput id="textInput"/>  
<s:Label id="label"/>  
  
<fx:Binding source="textInput.text"  
 destination="label.text"/>

Is equivalent to

<s:TextInput id="textInput"/>  
<s:Label id="label" text="{textInput.text}"/>

An MXML file can have multiple Binding tags. Each must be a child of the root tag.

A Binding tag can have three tag-specific attributes: source, destination, and twoWay. These cannot be written as child tags.

A Binding tag cannot have any child tags. Any character data must be whitespace.

The source and destination attributes are required and must be valid ActionScript expressions in the context of the ActionScript class being defined by the root tag. They can have leading and trailing whitespace.

The twoWay attribute is optional. It can have the values true or false and defaults to false. Setting it to true is the equivalent of using @{…} binding syntax.There can be leading and trailing whitespace around true but not around false.

Recommendation for MXML 2012

Allow leading and trailing whitespace around false.

The values true and false can be written with each letter in any case.

Recommendation for MXML 2012

Require true and false to be lowercase.

### Component tag

A Component tag can be used to define a “helper” component inside an MXML file, without having to put the code for it in a separate MXML file. The most common use case is for “inline” item renderers with list-based components.

Example:

<s:List dataProvider="{new ArrayCollection([1,2,3])}">  
 <s:itemRenderer>  
 <fx:Component>  
 <s:ItemRenderer>  
 <s:Button label="{data}"/>  
 </s:ItemRenderer>  
 </fx:Component>  
 </s:itemRenderer>  
</s:List>

The idea here is that if a component has a property of type IFactory, ClassFactory, Class, etc. then you can use a Component tag to specify a factory class from which instances can be created.

This is similar to having a second MXML file, MyItemRenderer.mxml, with the component definition

<?xml version="1.0" encoding="utf-8"?>  
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"  
 xmlns:s="library://ns.adobe.com/flex/spark">  
 <s:Button label="{data}"/>  
</s:ItemRenderer/>

and writing the List as

<s:List dataProvider="{new ArrayCollection([1,2,3])}"  
 itemRenderer="MyItemRenderer"/>

A Component tag can also be used “non-inline” by placing it inside a Declarations tag.

Example:

<fx:Declarations>  
  
 <fx:Component className="OKButton">  
 <s:Button label="OK"/>  
 </fx:Component>  
  
</fx:Declarations>

In this case is it typical to use the optional className attribute of the Component tag to specify an unqualified class name for the helper class.

The original MXML file can create instances of this component as if it were in the same package, using the \* namespace:

<local:OKButton xmlns:local="\*"/>

ActionScript code can use this class by importing it:

import OKButton;

An MXML file can have multiple Component tags. Each can be used in two places: (1) anywhere a value of type IFactory, ClassFactory,or Class is espected; (2) as a child tag of a Declarations tag.

It can have one tag-specific attribute, className, to specify the unqualfied name of the secondary class. If this attribute is omitted, an autogenerated name is used.

Recommendation for MXML 2012

The className attribute should be required when the Component tag is used non-inline in the Declarations tag. Otherwise, there is no way to refer to it. It should also be code-hinted.

The Component tag must have a single child tag, which is treated as a root tag defining a helper class.

Any character data before or after this child tag must be whitespace.

Whether used inline or non-inline, the autogenerated helper class includes a bindable public variable named outerDocument, whose type is the ActionScript class being defined by the root tag.

Also, whether used inline or non-inline, the autogenerated helper class is public and can be accessed by any code.

Recommendation for MXML 2012

Make it an out-of-package helper class, visible only to the MXML file that defines it.

### Declarations tag

The Declarations tag was introduced in MXML 2009 and is not allowed in MXML 2006. A Declarations tag can be used to declare and initialize instance variables in the ActionScript class being defined by the root tag of the MXML file. It can also be used to declare helper classes, if you use a Component tag inside a Declarations tag.

Example:

<fx:Declarations>  
  
 <fx:String id="greeting">Hello</fx:String>  
  
 <fx:Number id="pi">3.14159</fx:Number>  
  
</fx:Declarations>

is basically equivalent to

<fx:Script>  
<![CDATA[  
  
 [Bindable]  
 public var greeting:String = "Hello";  
  
 [Bindable]  
 public var pi:Number = 3.14159;  
  
]]>  
</fx:Script>

TODO: Is it really? They don’t autogenerate the same code.

Note that each declared object (or at least each one with an id attribute) is made bindable by the MXML compiler; this means that declarations are expensive.

Recommendation for MXML 2012

Unless databinding becomes much cheaper, declarations should be non-bindable by default. You should have to opt-in to databinding. It isn’t clear whether [Bindable] metadata should be used on MXML tags, or whether a bindable="true" attribute would be preferable.

An MXML file can have multiple Declarations tags. Each must be a child of the root tag.

A Declarations tag cannot have any tag-specific attributes.

Each child tag inside a Declarations tag represents a declaration of an instance or a class, so the child tags must be instance tags or a Component tag.

Recommendation for MXML 2012

The compiler should warn if a child tag inside a Declarations tag doesn’t have an id; otherwise, the corresponding ActionScript variable has an autogenerated name and is not useful except for any possible side effects.

Any character data between child tags must be whitespace.

### Definition tag

The Definition tag was introduced in MXML 2009 and is not allowed in MXML 2006. A Definition tag can be used inside a Library tag to define a helper component. (In this sense, it is similar to the Component tag, but it cannot be used inline as in the item renderer example. )

Example:

<fx:Library>  
  
 <fx:Definition name="OKButton">  
 <s:Button label="OK"/>  
 </fx:Definition>  
  
</fx:Library>

An MXML file can have multiple Definition tags. Each must be a child of a Library tag.

A Definition tag has one tag-specific attribute, name. It is a required attribute and specifies the MXML tag name for the helper component. Whitespace is allowed around the name.

A component defined by a Definition tag can be used as an MXML tag in the language namespace, as in

<fx:OKButton>

Note that this is different from a component defined with a Component tag, which must be accessed in the \* namespace.

The compiler does not check whether multiple Definition tags have the same name attribute. If they do, the first one wins.

Recommendation for MXML 2012

Multiple Definition tags with the same name should be a compilation error.

The Definition tag must have a single child tag, which is treated as a root tag defining a helper class.

Any character data before or after this child tag must be whitespace.

The autogenerated helper class is public and can be accessed by any code.

Recommendation for MXML 2012

Make it an out-of-package helper class, visible only to the MXML file that defines it.

The name attribute does *not* specify the name of the autogenerated helper class. Instead, this class is given an autogenerated class name in the unnamed package. The name is only used to convert MXML to ActionScript.

Recommendation for MXML 2012

If we continue to generate a public class, it should have the name that is specified, so that it can be used in ActionScript as well as in MXML, and this name should be allowed to be a fully-qualified classname.

### DesignLayer tag

The DesignLayer tag was introduced in MXML 2009 and is not allowed in MXML 2006. A DesignLayer tag can be used to group instance tags into “design layers”, independently of their grouping in the parent/child hierarchy of DisplayObjects.

Example:

<s:Group>  
 <s:Button/>  
 <fx:DesignLayer id="layer1"/>  
 <s:Button/>  
 <s:Button/>  
 </fx:DesignLayer>  
</s:Group>

Here all three Buttons are children of the Group, but only two are the DesignLayer.

An MXML file can have multiple DesignLayer tags.

TODO: Describe where they can go.

A DesignLayer tag can have three compile-time attributes: id, alpha, and visible.

TODO: Provide details.

TODO: What about activate, deactivate, and layerPropertyChange?

Unlike most other language tags, a DesignLayer tag actually creates an instance of a class, mx.core.DesignLayer.

### Library tag

The Library tag was introduced in MXML 2009 and is not allowed in MXML 2006. A Library tag is a container for Definition tags, each of which defines a helper class.

Example:

<fx:Library>  
  
 <fx:Definition name="OKButton">  
 <s:Button label="OK"/>  
 </fx:Definition>  
  
</fx:Library>

An MXML file can have only one Library tag, and if present it must be the first child tag of the root tag, even if the root tag has property tags, style tags, or event tags.

Recommendation for MXML 2012

The Library tag should come after any property tags, style tags, or event tags of the root tag.

A Library tag cannot have any tag-specific attributes.

The Library tag can only have Definition tags as children. The character data before, between, and after the Definition tags must be whitespace.

An empty Library tag, or one whose content is all whitespace, is allowed, but is pointless.

### Metadata tag

A Metadata tag can be used to specify metadata for the ActionScript class being defined by the root tag of the MXML file.

Example:

<fx:Metadata>  
 [Event(name="open", type="com.whatever.MyEvent")]  
 [Event(name="close", type="com.whatever.MyEvent")]  
 [Style(name="fooColor", type="uint", format="Color")]  
 [Style(name="barColor", type="uint", format="Color")]  
</fx:Metadata>

Typically this metadata would be [Event] and [Style] to specify which events and styles the class supports, so that when you write an MXML tag representing this class the MXML compiler can recognize attributes and child tags as being for events and styles. [DefaultProperty] and [Bindable] metadata are other common use cases.

The metadata for the class is specified as character data inside the Metadata tag. All of the character data must be parsable as well-formed ActionScript metadata, but it is legal to have any kind of metadata that you want, such as [Foo] or [Event(abc="def")], just as in ActionScript.

Recommendation for MXML 2012

The compiler should warn if inappropriate metadata is found, such as [Foo] metadata anywhere, [Event] metadata on a variable, [Event] metadata with an unknown attribute, etc.

An MXML file can have multiple Metadata tags. Each must be a child of the root tag.

A Metadata tag cannot haveany tag-specific attributes. Nor can it have any child tags.

The character data is simply copied into the autogenerated class above the class declaration (just above the autogenerated [Frame] metadata), in the same say that the character data of the Script tag is copied into the class body.

### Model tag

A Model tag can be used to declare hierarchical data models in MXML.. The data model is specified by XML inside the Model tag, but it gets translated by the compiler into a hierarchy of mx.utils.ObjectProxy instances.

Example:

<fx:Model id="info">  
 <root>  
 <employee>  
 <name first="Gordon" last="Smith"/>  
 <address>  
 <city>San Francisco</city>  
 <state>CA</state>  
 </address>  
 <email>gosmith@adobe.com</email>  
 </employee>  
 <employee>  
 <name first="Paul" last="Reilly"/>  
 <address>  
 <city>Boston</city>  
 <state>MA</state>  
 </address>  
 <email>preilly@adobe.com</email>  
 </employee>  
 </root>  
</fx:Model>

is similar to writing a nested data structure like

public var info =  
{  
 employee:  
 [  
 {  
 name: { first: "Gordon", last: "Smith" },  
 address: { city: "San Francisco", state: "CA" },  
 email: "gosmith@adobe.com"  
 },  
 {  
 name: { first: "Paul", last: "Reilly" },  
 address: { city: "Boston", state: "MA" },  
 email: "preilly@adobe.com"  
 }

]  
};

except that instead of constructing a hierarchy of plain Objects (which don’t support databinding) the compiler constructs a hierarchy of mx.utils.ObjectProxy instances (which do).

Once you have defined a model, you can write expressions such as info.employee[1].address.state (which in this case would evaluate to "MA").

A Model tag can have two tag-specific attributes, id and source.

The id attribute works the same as the id attribute on other tags, and specifies the identifier through which the data model can be accessed.

The source attribute can be used to tell the compiler to read the XML describing the data model from another file, as in

<fx:Model id="info" source="info.xml"/>

where info.xml contains

<root>  
 <employee>  
 <name first="Gordon" last="Smith"/>  
 <address>  
 <city>San Francisco</city>  
 <state>CA</state>  
 </address>  
 <email>gosmith@adobe.com</email>  
 </employee>  
 <employee>  
 <name first="Paul" last="Reilly"/>  
 <address>  
 <city>Boston</city>  
 <state>MA</state>  
 </address>  
 <email>preilly@adobe.com</email>  
 </employee>  
</root>

The MXML file has a file-dependency on the file specified by the source attribute, so that if the source file changes, the MXML file gets recompiled.

The contents of the Model tag (or the source file) must be well-formed XML. It must have a single root tag, with leading and trailing whitespace allowed.

If the data tags have namespace prefixes, these prefixes must be defined. However, the namespaces of the data tags are completely ignored in the process of turning the tags into ObjectProxy instances.

A data tag like

<name first="Gordon" last="Smith"/>

or

<name>  
 <first>Gordon</first>  
 <last>Smith</last>  
</name>

autogenerates code such as

name = new ObjectProxy();  
name.first = "Gordon";  
name.last = "Smith";

Values that look like they should have type Boolean, int/uint/Number, and Array (rather than String) are interpreted as such.

TODO: Provide the details, including whitespace rules, casing rules, etc.

When a data tag has more than one child tag with the same name, as in

<root>  
 <employee>…</employee>  
 <employee>…</employee>  
</root>

they are treated as elements of an Array.

If you use the source attribute, the character data is ignored, so it should be only whitespace. Currently the compiler warns if it isn’t.

Recommendation for MXML 2012

A Model tag with a source attribute and non-whitespace character data should be a compilation error.

### Private tag

The Private tag was introduced in MXML 2009 and is not allowed in MXML 2006. Application developers and tools can use it to store design-time information, such as the author and revision number of the MXML file.

Example:

<s:Application xmlns:fx=http://ns.adobe.com/mxml/2009  
 xmlns:s="library://ns.adobe.com/flex/spark">  
  
 <s:Label text="Hello World!"/>  
  
 <fx:Private>  
 <Author>John Smith</Author>  
 <Revision>1.3.1</Revision>  
 </fx:Private>  
  
</s:Application>

An MXML file can have only one Private tag, and if present it must be the last child of the root tag.

A Private tag cannot haveany tag-specific attributes.

The contents (character data and child tags) of the Private tag are ignored by the compiler and are not available at runtime; however the contents must be well-formed and valid XML to preserve the integrity of the document.

### Reparent tag

The Reparent tag was introduced in MXML 2009 and is not allowed in MXML 2006. It is used with MXML states to move a component from one parent to another when the state changes.

Example:

<s:Group id="g1”>  
 <s:Button id="b1" includeIn="state1"/>  
</s:Group>  
<s:Group id="g2”>  
 <fx:Reparent target="b1" includeIn="state2"/>  
</s:Group>

makes Button b1 move from Group g1 to Group g2 when the state changes from "state1" to "state2".

A Reparent tag can have three tag-specific attributes: target, includeIn, and excludeFrom..

The target attribute is required and must specify the id of a component in the same MXML document. You cannot use a databinding expression to specify the target.

Recommendation for MXML 2012

Investigate fixing this. It is inconsistent with how targets are specified for Effects.

Either the includeIn attribute or the excludeFrom attribute is also required, but both cannot be used at the same time. Which one you use must specify one or more states in which the reparenting is to happen, or not happen.

The Reparent tag can only have whitespace as content.

### Script tag

Recall that each MXML file defines an ActionScript class. The <fx:Script> tag allows you to place arbitrary ActionScript code inside the autogenerated body of that class.

TODO: Exactly where inside?

Typically this code consists of import statements, use namespace statements, and var and function declarations. The variables and function in this Script tag, together with the variables autogenerated from tags with id s, define the API of the class defined by the MXML file.

Loose statements in Script tags are unusual, but they are allowed because they are legal in a class body; such statements are executed at class initialization time, not instance initialization time.

Recommendation for MXML 2012

Loose statements should not be allowed in Script tags, because developers do not understand how they work. If they are allowed, they should become part of the iinit rather than part of the cinit.

A CDATA block

{{<![CDATA[

]]>

is normally placed around the character data of a Script tag, because characters such as the left-angle-bracket character (<) and the ampersand character (&) in ActionScript code would otherwise be interpreted as having their meaning in XML.

A Script tag can have a source attribute that specifies a file from which the ActionScript code should be read, instead of using the character data in the tag. If the character data is anything other than whitespace, the compiler warns that the character data will be ignored.

Recommendation for MXML 2012

It should be an error to have a source attribute and also non-whitespace character data.

The file can be specified as described in the “File references” section. It can have any extension.

The MXML file has a file-dependency on the file specified by the source attribute, so that if the source file changes, the MXML file gets recompiled.

TODO: What encoding is assumed for the source file?

An MXML file can have multiple Script tags. They are normally placed as children of the root tag, but this is not enforced by the compiler; it appears to allow them anywhere.

Recommendation for MXML 2012

Require that the Script tag be a child of the root tag.

### Style tag

A Style tag can be used to specify CSS selectors for styling components. The character data of the tag is interpreted according to the rules for CSS as used in Flex, which are outside the scope of this MXML specification.

<fx:Style>  
  
 @namespace s "library://ns.adobe.com/flex/spark";  
  
 s|Button { fontSize: 30 }  
  
</fx:Style>

A Style tag can have a source attribute that specifies a file from twhich the CSS should be read, instead of using the character data in the tag. If the character data is anything other than whitespace, the compielr warns that the character data will be ignored.

Recommendation for MXML 2012

It should be an error to have a source attribute and also non-whitespace character data.

The file can be specified as described in the “File references” section. It can have any extension, but the convention is to use .css.

The MXML file has a file-dependency on the file specified by the source attribute, so that if the source file changes, the MXML file gets recompiled.

TODO: What encoding is assumed for the source file?

An MXML file can have multiple Style tags. They must be children of the root tag, but can be mixed in with property tags, default property tags, etc.

# States

An important concept in Flex is state-based components. Although states are a runtime concept, the MXML language has syntax to support state-based programming. In particular, you can use MXML to

* declare states and state groups;
* specify that an instance tag is to be included in, or excluded from, particular states or state groups;
* specify the property values, style values, and event handlers with state-specific attributes.

## Declaring each State

Each state is represent at runtime by an instance of the mx.states.State class, and is declared at compile time with a State tag mapped to this class. (For example, the Spark manifest maps <s:State> to this class.)

Example:

<s:State name="state1"  
 stateGroups="stateGroup1"/>  
<s:State name="state2"   
 stateGroups="stateGroup1,stateGroup2"/>  
<s:State name="state3"  
 stateGroups="stateGroup2"/>

Because states have special significance in MXML, this tag has various compile-time quirks:

### name attribute

This attribute is required. The name of a state is its most important property, because you switch states with code like

myComponent.currentState = "state2";

Although name is a runtime property, it cannot be written as a property tag instead of as an attribute.

Its value must be a valid ActionScript identifier, without leading or trailing whitespace. This rules out state names such as 123, but true, false, NaN, Infinity, etc. are acceptable (which is probably unexpected).

Having multiple State tags with the same name is a compilation error.

### stateGroups attribute

This attribute is optional. It specifies a list of state groups to which the state belongs.

Note that state groups are implicitly declared while states are explicitly declared; there is no StateGroup tag.

Although stateGroups:Array is a runtime property of mx.states.State, it cannot be written as a property tag instead of as an attribute.

Its value must be a comma-separated list of one or more identifiers. There can be whitespace around each identifier. The compiler turns this comma-separate list into an Array.

Recommendation for MXML 2012

Introduce metadata so that any Array or Vector property or style can be set with a comma-separated list.

A state identifier cannot be used in the value of any stateGroups attribute, because this would implicitly define a state group with the same name as a state.

### basedOn attribute

This attribute is optional. It specifies the name of another state on which the state being defined is based.

Although basedOn:String is a runtime property of mx.states.State, it cannot be written as a property tag instead of as an attribute.

Its value is not checked to be a valid state name.

TODO: Is whitespace allowed?

Recommendation for MXML 2012

Setting basedOn to a non-existent state should be a compilation error.

### overrides attribute

This attribute is deprecated.

### Other attributes

The mx.states.State class supports enterState and exitState events; these must be written as attributes and cannot be written as event tags.

## Declaring the states of a component

Typically, states are declared using multiple <s:State> tags to set the <s:states> property of the root tag.

Example:

<?xml version="1.0" encoding="utf-8"?>  
<s:Application xmlns:fx=<http://ns.adobe.com/mxml/2009>  
 xmlns:s="library://ns.adobe.com/flex/spark">  
 <s:states>  
 <s:State name="state1" stateGroups="stateGroup1"/>  
 <s:State name="state2"  
 stateGroups="stateGroup1,stateGroup2"/>  
 <s:State name="state3" stateGroups="stateGroup2"/>  
 </s:states>  
  
 …  
  
</s:Application>

This makes states named state1, state2, and state3 – and state groups named stateGroup1 and stateGroup2 – known to the compiler when it is compiling this component.

Note that state groups – which are simply sets of states – are declared implicitly, by being mentioned in the stateGroups attribute.

## Using state-dependent tags

You can use includeIn and excludeFrom attributes on instance tags to specify which states the instances are present in.

Example:

<s:Button includeIn="state1,state3" label=="OK"/>

TODO: Explain what this means.

These are special compile-time attributes which do not correspond to runtime properites. They cannot be set with property tags.

TODO: Comma-separated list is allowed.

TODO: Either states or state groups can be specified.

## Using state-dependent properties, styles, and events

You can “.state” or “.stategroup” notation to specify in what states property values, style values, and event handlers are to apply.

Example:

<s:Button label.state1="OK">  
 <s:label.state2>Cancel</s:label.state2>  
</s:Button>

sets the label of the Button to "OK" in state1 and to "Cancel" in state2. Notice that the dot notation can be used with either a property attribute or a property tag.

Similarly, you can have state-dependent style attribute, style tags, event attributes, and event tags.

Code such as

<s:Button label.state1="OK" label.stateGroup1="Cancel">

causes a compilation error. Since state1 is in stateGroup1, this code is specifying two different values for the label in state1.

TODO: The above explanation of states is for MXML 2009. Explain how states work in MXML 2006.

# File paths

File paths can be relative or absolute. If relative, they are relative to the directory containing the MXML file with the file path.

They can use either the forward-slash character (/) or the backslash character (\) as a path separator, and these can be mixed. TODO: Is \ legal on Mac?

# Compiler directives

MXML compiler directives are particular token sequences of the form @Whatever(…) that look like a function call starting with an at-sign character (@). They have special meaning to the MXML compiler.

The at-sign character can be escaped with an immediately preceding backslash if you don’t want it to be understood as the beginning of a compiler directive.

The parentheses are required. The tokens inside the parentheses are parsed in the same way as the tokens inside the square brackets of an ActionScript metadata expression.

Directives may have attributes that are required to be present. Unexpected attributes, such as foo="bar", do not cause compilation warnings or errors.

Recommendation for MXML 2012

Unexpected attributes should cause a compilation warning.

Only four directives are recognized: @Clear, @ContextRoot, @Embed, and @Resource. Any other causes a compilation error.

## @Clear directive

This directive is available in MXML 2009 but not in MXML 2006. It can be used as a state-specific value for a property, style, or event, either in a child tag or in an attribute. (It is intended for use with styles and events, but unfortunately also work with properties.) It cannot be used to specify a non-state-specific value.

Example:

<s:Button color="red" color.over="@Clear()"/>

For a style, it generates code that calls clearStyle() to remove the style value on the instance and allow the style to be determined by the rest of the style chain, such as a class selector or a type selector.

For an event, it generates code that calls removeEventListener() to remove the event handler for that event on that instance.

For a property, it generates code that assigns the value undefined to the property; this is then coerced to the default value for the type of the property. Note: This is often not the same as the component’s default value for that property, so it isn’t what developers expect.

Recommendation for MXML 2012

This directive should not be allowed as a property value, unless we re-implement it to restore the component’s original value for the property, which components do not currently keep.

TODO: Inside ()?

## @ContextRoot directive

This directive is obsolete. It was used by the webtier compilers.

Recommendation for MXML 2012

This directive should no longer be recognized.

## @Embed directive

An @Embed directive can be used anywhere that a value of type Class is expected. The effect of this directive is twofold:

* To cause an asset — such as image, sound, or font —to be embedded in the SWF or SWC.
* To cause an ActionScript class representing that asset to be autogenerated, so that ActionScript code can access the asset.

Example:

<mx:Image source="@Embed('assets/flag.jpg')"/>

is essentially equivalent to

<fx:Script>  
<![CDATA[ [Embed("assets/flag.jpg")]  
 private var flag:Class ;]]>  
<fx:Script>  
  
<mx:Image source="@Embed('assets/flag.jpg')"/>

The @Embed directive has six common attributes for all assets. Specific types of assets support additional attributes. The attributes of the @Embed directive simply become the attributes of the autogenerated [Embed] metadata on the autogenerated variable that stores a reference to the class representing the asset. The allowed attributes and their meaning are discussed in the Metadata Specification.

## @Resource directive

A @Resource directive can be used in an attribute value, or in character data, to provide a value for a property or style from a localized ResourceBundle in the ResourceManager. Such resource values are typically strings, as in

<s:Button label="@Resource(bundle='foo', key='bar')"/>

<s:Button>  
 <s:label>@Resource(bundle="foo", key="bar")</s:label>  
</s:Button>

A @Resource directive can have leading and trailing whitespace but otherwise must be the entire value.

It must have attributes named bundle and key. The value of the bundle attribute must be the name of a resource bundle; a compilation error occurs if the resource bundle can’t be found. The value of the key attribute is expected to be the name of a particular resource in that bundle, but its existence is not checked by the compiler.

A @Resource directive autogenerates code such as

button1.label = resourceManager.getString("foo", "bar");

The ResourceManager method call that gets generated – getBoolean(), getInt(), getUint(), getNumber(), getString(), getClass(), or getObject() – depends on the type of the property of style being set.

# Databinding expressions

Databinding is a way to tie the value of something in MXML to the value of something else. With a “one-way” binding, when the “source” value changes, the “destination” value is updated as well. With a “two-way” binding, the same thing happens but, in addition, the source is updated when the destination changes. This magic is accomplished by generating lots of ActionScript that handles events that get dispatched when things change; databinding is expensive.

Databinding can be indicated by using databinding expressions or by Binding tags. The tags have already been discussed.

Databinding expressions are indicated by the use of curly braces in certain contexts. For example,

<s:TextInput id="ti1"/>  
<s:TextInput id="ti2" text="{ti1.text}"/>

Here the value ti2.text (the destination) is being bound to ti1.text (the source). When you type into ti1, the text in ti2 stays in sync with ti1.

A two-way binding is indicated by an at-sign (@) before the curly braces:

<s:TextInput id="ti1"/>  
<s:TextInput id="ti2" text="@{ti1.text}"/>

Here, typing in either TextInput makes the other change in sync.

The expression inside the curly braces must be a valid ActionScript expression in the context of the autogenerated class. For example, if you write the databinding expression {foo.bar} then foo.bar must be an expression that is legal to return from a method of the class.

Each property or method in the subexpression must be declared as being bindable with Bindable metadata. (Such metadata is discussed in a separate specification. It basically indicates which event will be dispatched when the subexpression changes.) Otherwise, the compiler will warn that changes to the expression cannot be detected.

In the case of two-way bindings, the expression must be a legal r-value, because an assignment to it will get autogenerated.

Whitespace is allowed inside the curly braces. No whitespace is allowed between the at-sign and the left curly brace.

TODO: Explain concatenated databinding such as in a the attribute  
 text="The answer is {foo.bar}".

TODO: Explain where databinding expressions are allowed.

TODO: Explain what ActionScript code [Binding] metadata and databinding expressions cause to be autogenerated.