



Adding Intelligence to Media

# **ADOBE XMP LIBRARY FOR ACTIONSCRIPT**

## **PROGRAMMER'S GUIDE**

Copyright © 2010 Adobe Systems Incorporated. All rights reserved.

*Adobe XMP Library for ActionScript: Programmer's Guide.*

NOTICE: All information contained herein is the property of Adobe Systems Incorporated. No part of this publication (whether in hardcopy or electronic form) may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Adobe Systems Incorporated.

Adobe, the Adobe logo, Acrobat, Acrobat Distiller, Framemaker, InDesign, Photoshop, PostScript, the PostScript logo, and XMP are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. MS-DOS, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Apple, Macintosh, and QuickTime are trademarks of Apple Computer, Inc., registered in the United States and other countries. UNIX is a trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. All other trademarks are the property of their respective owners.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied, or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes, and noninfringement of third party rights.

# Contents

<b>Introduction .....</b>	<b>4</b>
<b>XMPCore ActionScript library .....</b>	<b>5</b>
Adding the library to your projects .....	5
<b>XMP namespaces and prefixes .....</b>	<b>6</b>
Namespace and QName objects .....	6
Accessing dynamic properties .....	6
<b>Parsing and serializing metadata .....</b>	<b>7</b>
Parsing RDF/XML data to create an XMP object .....	7
Serializing data from an XMP object .....	8
<b>Manipulating metadata .....</b>	<b>8</b>
Working with simple properties .....	8
Data types for simple properties .....	9
Working with structured properties .....	9
Working with array properties .....	10
Working with language alternatives .....	11
Working with qualifiers .....	12
Handling dates and times .....	12
Iterating XMP properties .....	12
<b>Implicit property creation .....</b>	<b>13</b>
Simple properties .....	13
Structures .....	14
Arrays .....	14
Qualifiers .....	14
<b>Error Handling .....</b>	<b>15</b>

# Programmer's Guide

The ActionScript XMP Library is intended to be used with Adobe Flash and Flex technology, to read, modify and create XMP packets in the RDF/XML format.

Readers of this document should be familiar with the XMP Technology. The *XMP Specification*, available from [Adobe Developer Center \(XMP\)](#), provides a complete formal specification for XMP. Before working with the XMP Library for ActionScript, you must be familiar with, at a minimum, the XMP Data Model.

The specification has three parts:

- ▶ *Part 1, Data and Serialization Models* covers the basic metadata representation model that is the foundation of the XMP standard format. The Data Model prescribes how XMP metadata can be organized; it is independent of file format or specific usage. The Serialization Model prescribes how the Data Model is represented in XML, specifically RDF.
- ▶ *Part 2, Standard Schemas*, provides detailed property lists and descriptions for standard XMP metadata schemas; these include general-purpose schemas such as Dublin Core, and special-purpose schemas for Adobe applications such as Photoshop. It also provides information on extending existing schemas and creating new schemas.
- ▶ *Part 3, Storage in Files*, provides information about how serialized XMP metadata is packaged into XMP Packets and embedded in different file formats. It includes information about how XMP relates to and incorporates other metadata formats, and how to reconcile values that are represented in multiple metadata formats.

## Introduction

This library is an ActionScript 3 implementation of the C/C++ XMPCore library. It is written entirely in ActionScript 3 and has no other dependencies. The XMPCore library is the component that is used to manipulate the contents XMP Packets using the XMP Data Model; it does not deal with how XMP Packets are embedded in files.

XMP metadata is represented in memory as an object tree that reflects the XMP data model, as described in the *XMP Specification: Part 1, Data and Serialization Models*. The XMPCore ActionScript Library allows direct access to every node of the tree. The root node represents the XMP Packet itself. Each tree node represents an XMP *property*. Properties can be *simple* (key-value pairs) or *complex* (structures). Complex properties can be nested to any depth; that is, structures or arrays can contain other structures or arrays. Properties can have *qualifiers* that describe their values; for example, the `xml:lang` qualifier contains the language associated with each item in an alternative text array.

You can use the API functions to iterate the nodes recursively, in order to traverse the complete tree if needed. The library assures that the data tree is always in a valid state and can therefore be serialized into a well-formed XMP Packet at any time. If you use an API function incorrectly, the library throws an exception.

The library comprises three main parts:

- ▶ The *data model* represents a set of XMP data in memory. The API allows you to create, modify, or delete XMP properties.
- ▶ The *parser* reads an XMP packet in RDF/XML format into the data model in memory.

- The *serializer* converts the data model into RDF/XML format.

## XMPCore ActionScript library

These are the most important classes defined in the XMPCore ActionScript library (referred as *XMPCoreAS*), with their constructors, properties, class methods and instance methods, in order of their significance. Complete reference documentation for the library is provided in a separate document, *XMP Library API Reference* <<[get correct path to doc? if HTML, make this a link](#)>>

---

*missing Namespace, XML, and base class XMPNode*

---

XMPMeta	The XMP data-model object containing the top-level properties of a metadata tree.
XMPStruct	A structured property that stores named child fields.
XMPArray	A list of indexed child properties.
XMPProperty	A simple property that represents a literal value, a leaf node of XMP metadata tree.
XMPQualifier	A qualifier that describes an XMP property.
XMPDateTime	A date-time compatible with ISO 8601 (used by XMP).
ParseOptions	Options to configure the XMP parser.
SerializeOptions	Options to configure packet serialization.
XMPException	Encapsulates all exceptions occurring in the library.
XMPConst	Constants for namespaces.
XMPError	Constants for all error codes that can occur in an exception.

## Adding the library to your projects

The library is deployed as a compiled Flash library (SWC), `XMPCore.swc`, which you can add to Flex projects and Flash Authoring projects.

To use the library in a Flex project:

1. Create or open the Flex project in Flash Builder, or Eclipse with the Flash Builder plug-in.
2. If a `libs/` folder does not yet exist in the project's root folder, create it.
3. Copy the `XMPCore.swc` library into the `libs/` folder. Flash Builder automatically recognizes libraries in this folder.
4. To check it, open the project properties and select the "Flex Build Path" page. In the "Library path" tab, open the `libs/` folder and verify that the XMPCore library is available.

## XMP namespaces and prefixes

XMP uses XML namespaces for all of its properties and qualifiers throughout the XMP data model. Properties without namespace are not allowed in XMP. The XMPCore library uses E4X (ECMA Script for XML) functionality to provide a convenient API to the data tree.

Each XMP property that is added or requested must be an XML *QName*, consisting of a namespace and a local name. In XML representation, namespaces are always bound to a prefix, which is used in the tags as a shortcut for the whole namespace URI. This is a snippet from an XMP packet that shows the association of a namespace with the prefix `xmp`, and the use of that prefix in addressing an element:

```
<rdf:Description xmlns:xmp="http://ns.adobe.com/xap/1.0/">
  <xmp:CreatorTool="Photoshop">
    ...
</rdf:Description>
```

### Namespace and QName objects

The XMPCore library encapsulates URI and prefix information in *Namespace* objects. In an application that integrates this library, you do not use any namespace prefixes directly; instead, you use a variable whose value is a *Namespace* object. The double-colon operator (`:`) combines a namespace object with a local property name, to form a *fully-qualified name*, represented by an ActionScript *QName* object.

For example, for the RDF XML expression above, read access to the `CreatorTool` property looks like this:

```
var meta: XMPMeta = new XMPMeta();
var xmp:Namespace = new Namespace("http://ns.adobe.com/xap/1.0/");
meta.xmp::CreatorTool = "Photoshop";
```

Here, the variable `xmp` is a *Namespace* object whose value is the actual namespace URI. It is common practice and avoids confusion to use the default XML prefix as the name of this variable (as shown in the example), although this is not required.

- Predefined namespaces are stored as constants in the `XMPConst` class:

```
var xmp:Namespace = XMPConst.xmp;
```

- To define your own namespace:

```
var my:Namespace = new Namespace("my", "http://my.namespace/");
```

### Accessing dynamic properties

In the XMPCore library, the `XMPMeta` object represents the root node of a metadata tree. Use it to traverse the data, using either the dot operator (`.`) or the square brackets operator (`[...]`) to access simple values, nested values, and the items within array properties. Use the dot operator when the requested child property is constant, and square brackets when the child property name is a variable. You must always use brackets to access array items.

Each metadata property becomes a *dynamic property* of the XMP object, in a hierarchical tree structure. You access each node using its fully qualified name; that is, the namespace object and the property key name, with a double colon between them.

For example, if you have assigned an `XMPMeta` object to the variable `meta`, and a `Namespace` object to the variable `my`, the following assigns a value to a field contained in a nested structure, using dot notation:

```
meta.my::struct.my::innerStruct.my::innerField1 = "field 1";
```

Similarly, you can access array values at any depth using square brackets.

```
meta.my::array[1].my::innerArray[1].my::innerItem1 = "item 1";
```

This example sets simple properties stored in an array:

```
var Iptc4xmpExt:Namespace = new
    Namespace("http://iptc.org/std/Iptc4xmpExt/2008-02-29/");

meta.Iptc4xmpExt:LocationShown[1].Iptc4xmpExt:City = "Hamburg";
meta.Iptc4xmpExt:LocationShown[1].Iptc4xmpExt:CountryName = "Germany";
```

## Parsing and serializing metadata

- The constructor for the `XMPMeta` object calls a parser that translates RDF/XML data into the memory model in an `XMPMeta` object. You can pass data from the RDF/XML file to the parser as an XML object, as a String, or as a ByteArray representation.

Compared to the native C++ XMP parser, some legacy functionality has been removed for the ActionScript library in favor of small footprint and performance:

- ▷ Alias functionality has been removed. Since all existing XMPCore implementations have not created aliased properties for some years, this change does not break existing XMP packets created with any newer application (approximately since CS1).
- ▷ Latin1 characters are not tolerated in XMP packets anymore. Only valid UTF-8 or UTF-16 (LE and BE) encodings are accepted.
- The serializer serializes the memory model to RDF/XML, as a String, ByteArray or XML object. The API provides options for configuring the XML output. The most commonly used options are for encoding and the absolute length of the resulting packet.

Prefixes are not stored in the data model. The parser collects the namespace prefixes, to be used when the same `XMPMeta` object is serialized after modification. The Adobe standard namespaces, defined in the *XMP Specification: Part 2, Standard Schemas*, are stored in a global namespace registry of the library. You can also register custom prefixes for your own namespaces; if the serializer does not find a registered prefix for a certain namespace, it generates prefixes of the form "ns1", "ns2" and so on.

## Parsing RDF/XML data to create an XMP object

- Create an empty XMP object:

```
var meta: XMPMeta = new XMPMeta();
```

- Create and fill an XMPMeta object by parsing an XMP packet from a String:

```
var metaStr: String = "<x:xmpmeta xmlns:x=\"adobe:ns:meta/\">" +
    ...
    "</x:xmpmeta>";
var meta: XMPMeta = new XMPMeta(metaStr);
```

- Create and fill an XMPMeta object by parsing an XMP packet from an XML object:

```
var metaXML: XML = <x:xmpmeta xmlns:x="adobe:ns:meta/">
    ...
</x:xmpmeta>;
var meta2: XMPMeta = new XMPMeta(metaXML);
```

- Create and fill an XMPMeta object by parsing from a ByteArray, which is useful when the packet is read from a file or stream.

## Serializing data from an XMP object

- The XMPMeta object offers serialization methods that transform its data tree into the normal RDF/XML format:

```
var meta:XMPMeta = new XMPMeta(...);
// ... modify metadata

var metaStr:String = meta.serialize();

var metaBuffer:ByteBuffer = meta.serializeToBuffer();

var metaXML:XML = meta.serializeToXML();
```

- Use the SerializeOptions object to set non-default serialization options, and pass it to one of the serialization methods. For example, to serialize in UTF-16LE encoding to a packet with the exact length 8192:

```
var options: SerializeOptions = new SerializeOptions();
options.encoding = SerializeOptions.UTF16LE;
options.exactPacketLength = 8192;
var buffer:ByteBuffer = meta.serializeToBuffer(options);
```

- To simplify debugging or write unit tests, use the dumpObject() method to serialize to a human-readable, debug format:

```
trace(meta.dumpObject());
```



# Manipulating metadata

This section describes some typical workflows, with source code examples. For details of functions and usage, see the [“XMPCore ActionScript Library Reference” on page 16](#).

## Working with simple properties

For these examples, we will use an existing namespace from the `XMPConst` class (named `xmpNS` to distinguish it from our `XMPMeta` object), and also define a custom namespace named `my`:

```
var xmpNS:Namespace = XMPConst.xmp;
var my:Namespace = new Namespace("my", "http://my.namespace/");
```

- Create an empty XMP object:

```
var meta:XMPMeta = new XMPMeta();
```

- Add simple properties to the XMP object in the two namespaces:

```
meta.xmpNS::Rating = 4;
meta.my::simpleProp = "value";
```

- Read the simple properties back:

```
trace(meta.xmpNS::Rating); // shows "4"
trace(meta.my::simpleProp); // shows "value"
```

- To overwrite existing properties, assign new values:

```
meta.xmpNS::Rating = 5;
meta.my::simpleProp = "new value";
```

- Delete properties by setting the value to null, or use the `delete` operator:

```
meta.my::simpleProp = null;
```

—or—

```
delete meta.my::simpleProp;
```

- Use the `exists()` method or the standard `toString()` method to find out whether a property currently exists in an XMP object:

```
trace( meta.xmpNS::Rating.exists() ); // true
trace( meta.my::simpleProp.exists() ); // false
trace( meta.my::simpleProp.toString() ); // null
```

## Data types for simple properties

You can assign all kinds of literal data types to simple properties; however, they are all converted to `String` before the value is added.

- Boolean values are converted to "True" or "False" (with a leading capital letter)
- Date objects are converted to ISO 8601 format (see [“Handling dates and times” on page 12](#)).

The `XMPMeta` object provides convenience methods for retrieving String property values as literal types:

```
toBoolean()
toInteger()
toFloat()
toDate()
```

## Working with structured properties

Structured properties build a node in the data tree that contains other structured properties, arrays or simple properties. They are represented by the `XMPStruct` object.

- Continuing the example using the `XMPMeta` object (`meta`) and `Namespace` object (`my`), we add a structure with two simple property fields:

```
meta.my::struct = new XMPStruct();
meta.my::struct.my::structField1 = "value 1";
meta.my::struct.my::structField2 = "value 2";

// Retrieve a struct field
trace (meta.my::struct.my::structField1);
```

- Structured properties can be nested. In this example, we add a nested struct named `my::innerStruct` to `my::struct`, which contains two additional simple-property fields.

```
meta.my::struct.my::innerStruct = new XMPStruct();
meta.my::struct.my::innerStruct.my::innerField1 = "field 1";
meta.my::struct.my::innerStruct.my::innerField2 = "field 1";

// Retrieve an inner struct field
trace(meta.my::struct.my::innerStruct.my::innerField2);
```

- ▷ You can accomplish the same thing by first creating the inner `XMPStruct` object, and add it to the `XMP` object afterward:

```
var struct: XMPStruct = new XMPStruct();
struct.my::innerField1 = "field 1";
struct.my::innerField2 = "field 1";

// add the structured property to the XMP object
meta.my::struct.my::innerStruct = struct;
```

- ▷ For convenience, you can also create nested structures implicitly, simply by adding the first field to it:

```
meta.my::implicitlyCreated.my::field1 = "value 1";
meta.my::implicitlyCreated.my::field2 = "value 2";
```

See [“Implicit property creation” on page 13](#) for more details of this technique.

- Delete structures in the same way as simple properties, by setting them to null or using the delete operator. All direct and nested child properties are also deleted:

```
meta.my::struct = null

—or—

delete meta.my::struct
```

## Working with array properties

Array properties contain a list of child properties, which can be of any type including nested arrays. The three types of arrays can be created by factory methods of the `XMPArray` class:

```
meta.my::bagArray = XMPArray.newBag();
meta.my::seqArray = XMPArray.newSeq();
meta.my::altArray = XMPArray.newAlt();
```

To add items, use the square brackets, "[...]" operator, or the `append()` and `insert()` methods of the `XMPArray` object.

- ▶ Arrays start with index 1 and must be filled from start to end.
- ▶ You cannot add an item at an index that is beyond the index of the last existing member plus one. For example, an array with the length 5 accepts indexes [1..5] for reading and [1..6] for writing; the index value of 6 appends a new item.
- ▶ The `last()` method accesses the last item; you can use the function as an index value, ["last()"].

Here are some examples of array access:

```
meta.my::seqArray[1] = "one";
meta.my::seqArray[1 + 1] = "two";

var index = 3;
meta.my::seqArray[index] = "three"; // use a variable as an index

meta.my::seqArray.append("five");

meta.my::seqArray.insert(4, "four");

trace(meta.my::seqArray[5]); // prints "five"
trace(meta.my::seqArray["last()"]); // prints "five"
```

Arrays can contain other arrays or structures as items. The property types of the items can be mixed, although this is not commonly used in XMP.

Here is a more complex example:

```
meta.my::twoDimensionalArray = XMPArray.newBag();
meta.my::twoDimensionalArray[1] = XMPArray.newBag();
meta.my::twoDimensionalArray[2] = XMPArray.newBag();
meta.my::twoDimensionalArray[1][1] = "value 1-1";
meta.my::twoDimensionalArray[1][2] = "value 1-2";
meta.my::twoDimensionalArray[2][1] = "value 2-1";
meta.my::arrayOfStructs = XMPArray.newAlt();
meta.my::arrayOfStructs[1] = new XMPStruct();
meta.my::arrayOfStructs[1].my::structField = 4711;
```

Arrays can also be created implicitly by adding the first item; see [“Implicit property creation” on page 13](#). For instance, the last example can be created simply by adding the nested structure field; the outer array and the structure itself are created automatically:

```
meta.my::arrayOfStructs[1].my::structField = 4711;
```

## Working with language alternatives

Language alternatives contain localized strings; they are technically normal Arrays of type `alt`. Each array item contains a different translation and the corresponding locale is stored in an `xml:lang` qualifier. The first array item is considered the default language.

The API provides a convenience accessor, `lang[]`, to handle localized text. Use it as if it were a nested array in the alt-text array (although it is not an actual property); use the language code as the index variable to access language-specific items. For example, to set values:

```
meta.my::localizedText = XMPArray.newAlt();
meta.my::localizedText.lang["en-US"] = "English"; // default
meta.my::localizedText.lang["de-DE"] = "German";
meta.my::localizedText.lang["fr-FR"] = "French";
```

When serialized, this creates the following RDF/XML construct:

```
<dc:title>
  <rdf:Alt>
    <rdf:li rdf:lang="en-US">English</rdf:li>
    <rdf:li rdf:lang="de-DE">German</rdf:li>
    <rdf:li rdf:lang="fr-FR">French</rdf:li>
  </rdf:Alt>
</dc:title>
```

To retrieve a particular translation:

```
trace("German translation: " + meta.my::localizedText.lang["de-DE"]);
```

For the format of the locale, use a String that conforms to the RFC 3066 notation, with two or three two-character codes for the language, country, and optional dialect, in the form `xx-YY-zz`. The provided locale is automatically normalized to the described format.

Many applications work with only one default language, that has no specific locale. In this case, the constant `"x-default"` is used as locale.

```
dc::title.lang["x-default"] = "unspecific language"; // default
trace(dc::title.lang["x-default"]);
```

If the `"x-default"` is requested from an array that does not have this entry, the first array item is returned.

## Working with qualifiers

A qualifier is a meta-property that adds additional information to another XMP property. It is represented in the API by the `XMPQualifier` class. The qualifier can be of any type; it can, in fact, be as complex as another XMP Packet.

The API provides a convenience accessor, `qualifier`, to access the qualifiers associated with a node. Use it as if it were a property of `XMPProperty`, `XMPArray` and `XMPStruct` (although it is not an actual property), which itself has dynamic properties for the qualifiers. Access these by their fully-qualified names, in the same way that you access the dynamic properties of the `XMNode` objects.

```
meta.my::myProp.qualifier.my::myQualifier = "description";
```

This example adds a simple qualifier to a simple property:

```
meta.my::number = 17;
meta.my::number.qualifier.my::kind = "This property contains a Prime Number";
```

To retrieve the qualifier:

```
trace(meta.my::number.qualifier.my::kind);
```

## Handling dates and times

XMP stores date and time values in the ISO 8601 format. The `XMPDateTime` class handles conversions between the ISO format and the ActionScript `Date` object, as well as the serialization of ISO dates. Typically, the class does not need to be instantiated, it is used internally by `XMPProperty` when you work with date-time values. For example, to add the current time to an XMP property:

```
meta.my::dateTime = new Date();
```

This serializes to a string like this:

```
1999-06-17T10:07:12+01:00
```

Retrieving the value as a String returns the ISO representation of the date. To retrieve the date as ActionScript `Date` object, call the `toDate()` method of the `XMPProperty`:

```
var date:Date = meta.my::dateTime.toDate();
```

Additional utility methods in the `XMPDateTime` class are described in the [“XMPCore ActionScript Library Reference” on page 16](#).

## Iterating XMP properties

You can iterate over the root node `XMPMeta` and the complex properties `XMPArray` and `XMPStruct` using the `for (... in ...)` or the `for each (... in ...)` syntax; use the base class `XMPNode` for the iteration.

The `for` command iterates through the contained property *names*, while the `for each` command iterates through the property *values*. Note that the properties returned by the iterator can themselves be structures or arrays.

For example:

► Iterate through all property *values* of a structured property:

```
for each (var prop: XMPNode in meta.my::struct)
{
    trace(prop);
}
```

► Iterate through all property *names* of a structured property:

```
for (var propName: String in meta.my::struct)
{
    trace(prop);
}
```

- Iterate through the names of all top-level properties:

```
for (var propName: String in meta)
{
    trace(prop);
}
```

- Iterate through all items of an array property:

```
for each (var prop: XMPNode in meta.my::array)
{
    trace(prop);
}
```

## Implicit property creation

The API offers a convenient way to create complex XMP data structures with few lines of code. All properties are created implicitly when a value is assigned or a child added; you need not to create each element explicitly with constructor calls.

- Simple properties are created when a value is assigned to them.
- Structure and array properties are implicitly created with all of the implied elements when a first child is added to them at any level.

This feature offers a more intuitive and usable API, but use it carefully and make sure you understand the mechanism to make the best use of it.

These examples illustrate implicit creation of the various types of properties, comparing it with explicit creation.

### Simple properties

- Without implicit property creation:

```
meta.my::simpleProp = new XMPProperty();
meta.my::simpleProp = 5;
```

- Creating a simple property implicitly by assigning a value:

```
meta.my::implicitSimpleProp = 5;
```

### Structures

- Without implicit property creation:

```
meta.my::manuallyCreated = new XMPStruct();
meta.my::manuallyCreated.my::manuallyCreated2 = new XMPStruct();
meta.my::manuallyCreated.my::manuallyCreated2.my::field1 = "value 1";
meta.my::manuallyCreated.my::manuallyCreated2.my::field2 = "value 2";
```

- With implicit property creation, both the outer and nested structure are created automatically:

```
meta.my::implicitlyCreated.my::implicitlyCreated2.my::field1 = "value 1";
meta.my::implicitlyCreated.my::implicitlyCreated2.my::field2 = "value 2";
```

## Arrays

Arrays can be created by adding the first item, either with the square bracket operator, `[1]`, or the `append()` method of `XMPArray`. You cannot implicitly create an array with any other index but one:

- ▶ Without implicit property creation, you must create the `XMPArray` object explicitly:

```
meta.my::array = XMPArray.newBag();
```

With implicit creation, this line is not needed.

- ▶ With implicit property creation, the array object is created when you add the first item:

```
meta.my::array[1] = "first";
meta.my::array[2] = "second";
```

Implicit arrays are always `bag`-type arrays, but you can change the type after creation. For example, this creates the array initially as `bag`, then changes it to a sequence:

```
meta.my::sequence.append("first");
meta.my::sequence.append("second");

meta.my::sequence.setType(XMPArray.SEQ);
```

This complex example creates a two-dimensional array of structures:

```
meta.my::Coordinate[1][1].my::Point.my::PointColor = "red";
meta.my::Coordinate[1][2].my::Point.my::PointColor = "green";
meta.my::Coordinate[2][1].my::Point.my::PointColor = "blue";
```

## Qualifiers

You *cannot* create a property implicitly by defining a qualifier for a property that does not yet exist.

For example, this does create a property with a qualifier:

```
meta.my::definedProp = 5;
meta.my::definedProp.qualifier.my::x = "value";
```

This, however, does not create a property, but instead throws an exception:

```
meta.my::undefinedProp.qualifier.my::x = "value";
```

## Error Handling

All errors that occur while using the API create and throw an `XMPException` object that contains an error ID (defined in `XMPError`) and a human-readable message. Exceptions that are thrown by `ActionScript` itself are also encapsulated in `XMPException`.

The best practice is to put independent sections of XMP processing code into `try ... catch` blocks that handle only the `XMPException`, as shown in the following example.

The `XMPMeta` object is not transaction-based; properties that have been created explicitly before the exception occurs are not deleted. If, however, a processing exception occurs after one or more properties have been created implicitly, these are automatically removed.

In this example, where a structure property would be created implicitly if no error occurred, nothing is added to the XMP object:

```
try {  
    meta.my::struct.my::nestedStruct.my::array[1234] = 5;  
} catch (e: XMPEException) {  
    trace (e); // out of bounds...  
}
```

In this case, the implicit creation process temporarily creates `my::struct`, `my::nestedStruct` and `my::array`. However, because the index 1234 does not exist in the array, the command throws an "Out of bounds" (with `errorId = XMPEError.BADINDEX`) exception, and deletes the temporary nodes before actually adding them to the XMP object.