

ForX v0.7.1 alpha (for ActionScript 2.0)

# ForX Developers' Guide

# Contents

1. Introduction	3
1.1 What is ForX?	3
1.2 Main Features	4
1.3 Other Information	4
2. Examples	5
2.1 How To Use ForX	5
2.2 And What About That Dot Path Thing?	6
2.2 Formulation	8
2.4 Fetching Data	10
2.5 Case-Sensitivity Issues	11
3. Class Interface	12
3.1 ForX Constructor	13
3.2 getXML	13
3.3 getChildrenNames	13
3.4 getChildrenCount	14
3.5 getChildNode	14
3.6 getPathNode	14
3.7 getPathValue	15
3.8 createPathNode	15
3.9 extractPathNode	16
3.10 getMultiNodes	16
3.11 fetchData	17
3.12 getAttributes	17
3.13 getAttributeValue	17
3.14 getAttribute	18
3.15 getNCFTokens	18
3.16 getAddress	18
3.17 formulateNode	19
3.18 formulateXML	19
3.19 fillTemplate	19
4. Notations Used	20
4.1 DPN, Dot Path Notation	21
4.1.1 DPN/A, Dot Path Notation with an Attribute	22
4.2 NCF, Node Creation Formula	23
4.3 XCF, XML Creation Formula	24
4.4 CXF, Compact XML Formula	25
4.5 MFS, Mapping Formula String	26
5. Development Road	27
5.1 Future Releases	27
5.2 Known Bugs	28
6. Terms of Usage / Additional Information & Contact	29

# 1. Introduction

— “Utilizing ForX helps you level the spaghetti code.”

## 1.1 What is ForX?

- ForX is a utility class that brings additional possibilities when it comes to manipulating XML data in Flash ActionScript. However, it isn't intended as a replacement for standard `XML` and `XMLNode` classes already in ActionScript 2.0, so don't expect to find substitutes for the features these two classes already provide.
- ForX is written in ActionScript 2.0, which means that you have to run it under (Macromedia) Flash 7 (MX2004) or Adobe Flash 8. (And for that matter, this guide assumes that you already know how to work with objects in AS 2.0, which is actually very similar to working with plain `String` or `Array` data types.)
- You don't have to know a thing about the object-oriented programming (OOP), although I do recommend such knowledge, by all means.
- You do have to know how to make a program, however. It is merely impossible to learn how to use ForX without, at least, intermediate (but not advanced) ActionScript programming knowledge.
- ForX is a perfect choice for small to medium XML files (up to 100 kb) which are used for external Flash communication and/or Flash client data feed.
- ForX helps you greatly reduce the time spent during the development and debugging phases of any code related to Flash XML logistics, the process also known as “XML juggling” (XML organization in terms of structure and semantics, as well as XML data parsing routines are much more easier to implement and understand, etc.).
- However, ForX uses Flash XML object to achieve all of this, so it's fully compatible with any XML-based work you've already done. You can use it at any point in your program: just create an instance, pass XML object to it, and finally get all the data you need, in a couple of lines or even less. This practically enables you to concentrate on much more important areas of your application, instead to search for every node by using relativistic and very error-prone `firstChild/nextSibling` coding style.

Just leave all your XML spaghetti code behind.

Btw, ForX means Formulated XML, and it is pronounced “forks”.

## 1.2 Main Features

- ForX is easy to use, and easy to read, yet powerful enough to satisfy all your needs in any given one-way or bilateral Flash XML communication development. ForX code is ideal for sharing communication concepts and ideas with other developers, and you'll also find it very easy to manipulate during the code maintenance or refactoring.
- Per-instance or per-task case-sensitivity preference when searching for particular nodes or attributes.
- The XML nodes are searched for by their identity (by corresponding names and attribute names/values) and by their container nodes, rather than by their relative structural position. (I. e. any change on the server won't disrupt the communication flow.)
- Only [XML/XMLNode](#) object reference (depending on the usage) is needed to introduce ForX. It doesn't extend XML class, so you don't have to rewrite your existing code to use ForX.
- Each XML node can be translated into unique string address that defines it, and vice versa, so you can easily make smart XML editors, driven only by input boxes, able to create entire XML structures from just several lines of text.
- Any node can be tracked down with a couple lines of code and manipulated thereafter. Any legal XML structure is supported.
- Attribute-based querying model (allowing you to find a specific node among the others sharing the same name).
- You can work with NCF or XCF strings that are 5-20% harder to read than whitespaced XMLs, but take 10-60% less space on the screen, and consume >50% less memory.
- Intuitive dot path syntax XML node access.
- Serialized XML data acquiring based on a given map, and followed by an automatic data type conversion.
- Easier XML construction without [createElement](#) and [appendChild](#). You simply define a path and then select whether to create a new node at each step, or to reuse the existing one. The path node will be created if it doesn't exist, or if you're forcing it to be created.
- and many more ...

## 1.3 Other Information

This software is released under the MIT Licence.

## 2. Examples

All text in this chapter is written in ActionScript 2.0 code.

You can copy/paste it in the first frame of the Adobe Flash (v7 or v8) document timeline to try it on your own.

Currently there are no downloadable versions of these examples.

### 2.1 How To Use ForX

```
import com.orionSyndrome.xml.Forx;
```

```
// Let's start from the beginning
```

```
// Let's make a sample XML
```

```
var xml:XML = new XML('<Farm><Owner>orionsyndrome</Owner><Backyard><Henhouse><Hen /><Rooster /><Chicken /><Egg /></Henhouse><Pighouse><Pig  
name="Matilda" /><Pig name="Donald" /></Pighouse></Backyard><House><Room type="living"><Furniture type="sofa" /></Room></House></Farm>');
```

```
// Let's trace it
```

```
trace(xml);
```

```
// This is the basic principle of getting XML in Flash.
```

```
import com.orionSyndrome.xml.Forx;
```

```
// And here we're going to instantiate ForX and make it aware of the XML
```

```
// Let's make a sample XML
```

```
var xml:XML = new XML('<Farm><Owner>orionsyndrome</Owner><Backyard><Henhouse><Hen /><Rooster /><Chicken /><Egg /></Henhouse><Pighouse><Pig  
name="Matilda" /><Pig name="Donald" /></Pighouse></Backyard><House><Room type="living"><Furniture type="sofa" /></Room></House></Farm>');
```

```
// We pass that XML to the newly created ForX
```

```
var forx:Forx = new Forx(xml);
```

```
// And now we trace that ForX to see its native output
```

```
trace(forx);
```

## 2.2 And What About That Dot Path Thing?

```
import com.orionSyndrome.xml.Forx;

// Basic Dot Path Notation sample

// Let's make a sample XML
var xml:XML = new XML('<Farm><Backyard><Pighouse><Pig name="Matilda" /><Pig name="Donald" /></Pighouse></Backyard></Farm>');

// We pass that XML to the newly created ForX
var forx:Forx = new Forx(xml);

// First, we try to use getPathNode() method and store that information
var pighouse:XMLNode = forx.getPathNode("farm.backyard.pighouse");

// Next, we try to get the pig node from that pighouse
var pig:XMLNode = forx.getPathNode("pig", pighouse);

// Let's see what we get
trace(pig);

// Note the casing of the words we used in getPathNode, and it worked!
// Also note that we got only one of the two pigs
// Check the next example for how to get the other pig as well
```

```
import com.orionSyndrome.xml.Forx;

// Dot Path Notation sample with attribute querying

// Let's make a sample XML
var xml:XML = new XML('<Farm><Backyard><Pighouse><Pig name="Matilda" /><Pig name="Donald" /></Pighouse></Backyard></Farm>');

// We pass that XML to the newly created ForX
var forx:Forx = new Forx(xml);

// The pigs in this sample have different names, although they are both pigs
// So here I will demonstrate how to get the other pig by using the attribute querying

// First, we use the getPathNode() method and store that information
var pighouse:XMLNode = forx.getPathNode("farm.backyard.pighouse");

// Next, we want to get that pig named Donald
var pig:XMLNode = forx.getPathNode("pig?name:Donald", pighouse);

// Let's see what we get
trace(pig);
```

```
import com.orionSyndrome.xml.Forx;

// Here's how to get a value

// Let's make a sample XML
var xml:XML = new XML('<Farm><Owner>orionsyndrome</Owner></Farm>');

// We pass that XML to the newly created ForX
var forx:Forx = new Forx(xml);

// Now, we use the getPathValue() method
var owner:String = forx.getPathValue("farm.owner");

// Let's see who's the owner
trace(owner);
```

```
import com.orionSyndrome.xml.Forx;

// And here's how to place George in the hen house
// Btw, George happens to be a pigeon

// Let's make a sample XML
var xml:XML = new XML('<Farm><Backyard><Henhouse><Hen /><Rooster /><Chicken /><Egg /></Henhouse></Backyard></Farm>');

// We pass that XML to the newly created ForX
var forx:Forx = new Forx(xml);

// Now, we use createPathNode() method to make a node
// Note the big 'G' in the word 'pigeon': it will remain that way
var pigeon:XMLNode = forx.createPathNode("farm.backyard.henhouse.piGeon");

// Next, we can reuse that pigeon variable which references the newly created node
// Let's use some standard tools to set the name
pigeon.attributes["name"] = "George";

// Let's see that hen house again
trace(forx.getPathNode("farm.backyard.henhouse"));
```

```
import com.orionSyndrome.xml.Forx;

// Feeding Donald is much more easier than before

// Let's make a sample XML
var xml:XML = new XML('<Farm><Backyard><Pighouse><Pig name="Matilda" /><Pig name="Donald" /></Pighouse></Backyard></Farm>');

// We pass that XML to the newly created ForX
var forx:Forx = new Forx(xml);

// Let's put some quality food in Donald
var food:XMLNode = forx.createPathNode("farm.backyard.pighouse.pig?name:dona!d.Food", "Hogwash");

// Now, we check out that pig house
trace(forx.getPathNode("farm.backyard.pighouse"));
```

## 2.2 Formulation

```
import com.orionSyndrome.xml.Forx;

// Hey, there's a house on this farm!
// We have a bed for it, but there's no bedroom...

// Let's make a sample XML
var xml:XML = new XML('<Farm><House><Room type="living"><Furniture type="sofa" /></Room></House></Farm>');

// We pass that XML to the newly created ForX
var forx:Forx = new Forx(xml);

// We need another room
// Pay special attention to that exclamation mark:
// It forces ForX to create a new Room node, instead to reuse the existing one
// Also, formulateNode method allows us to enter some attributes for that room as well
forx.formulateNode("farm.house.Room![type=bedroom;area=30]");

// Next, it's time to move the bed in there
forx.formulateNode("farm.house.room?type:bedroom.Furniture[type=bed]");

// But it seems that we didn't measured the living room area, so we might just add that
forx.getPathNode("farm.house.room?type:living").attributes["area"] = 40;

// Have a look at the house
trace(forx.getPathNode("farm.house"));
```



```
import com.orionSyndrome.xml.Forx;

// Let's try to work without any sample XMLs

// Here we make an empty XML
var xml:XML = new XML();

// We pass that XML to the newly created ForX
var forx:Forx = new Forx(xml);

// Let's create that XML from scratch by using formulateNode()
forx.formulateNode("Farm");

// Now, we add the owner
forx.formulateNode("Farm.Owner=syndrome");

// Then the backyard
forx.formulateNode("Farm.Backyard");

// And now we place an egg in the hen house
// We don't even have a hen house yet, but it will be created automatically
forx.formulateNode("Farm.Backyard.Henhouse.Egg");

// And now for the pigs
forx.formulateNode("Farm.Backyard.Pighouse.Pig[name=Matilda]");

// This one must have an exclamation mark to declare that this is a new Pig node
forx.formulateNode("Farm.Backyard.Pighouse.Pig![name=Donald]");

// Let's see what's done
trace(forx);
```

```
import com.orionSyndrome.xml.Forx;

// There is a more elegant way to create XMLs from scratch

// Here we make ForX without a reference
// (empty XML is created automatically)
var forx:Forx = new Forx();

// Let's create that XML from scratch, but now by using formulateXML()
forx.formulateXML("Farm,Farm.Owner=syndrome,Farm.Backyard.Henhouse.Egg,Farm.Backyard.Pighouse.Pig[name=Matilda]");

// Let's see what's done
trace(forx);
```

```
import com.orionSyndrome.xml.Forx;

// There is even more elegant way to this approach, without so much repeating

// Here we make new and empty ForX
var forx:Forx = new Forx();

// We'll use the CXF string here, and it's really great for this kind of stuff
forx.formulateXML("{Farm{Owner=syndrome+Backyard{Henhouse{Egg}+Pighouse{Pig[name=Matilda]+Pig![name=Donald]}}}}");

// Let's see what is done
trace(forx);
```

## 2.4 Fetching Data

```
import com.orionSyndrome.xml.Forx;

// This is an example of how XML-to-Array mapping should look like

// Let's make a sample XML
var xml:XML = new XML('<Farm><Owner>orionsyndrome</Owner><Backyard><Henhouse><Hen has_egg="yes" /><Rooster /><Chicken /><Egg /></Henhouse><Pighouse qty="2"><Pig id="0" name="Matilda" /><Pig id="1" name="Donald" /></Pighouse></Backyard></Farm>');

// We pass that XML to the newly created ForX
var forx:Forx = new Forx(xml);

// What we want here is the following information:
// - what's the owner's name (String),
// - what's the number of pigs in the pighouse (Number),
// - what's the name of the second pig (String),
// - did the hen lay the egg (Boolean),
// - the entire hen house node (XMLNode)

// And now we send a map and receive the results in an array
var data:Array = forx.fetchData("s=farm.owner,n=farm.backyard.pighouse@qty,s=farm.backyard.pighouse.pig?id:1@name,b=farm.backyard.henhouse.hen@has_egg,x=farm.backyard.henhouse"); // Note that since v0.7 we can omit all these type markers (s, n, etc.) and fetchData still returns properly typed information (i.e. did hen lay the egg? true). However, that's still not true for x (XMLNode).

// Let's trace the results
trace("owner's name: " + data[0]);
trace("number of pigs: " + data[1]);
trace("name of the second pig: " + data[2]);
trace("did hen lay the egg? " + data[3]);
trace("hen house population: " + forx.getChildrenNames(data[4].firstChild));
```

## 2.5 Case-Sensitivity Issues

```
import com.orionSyndrome.xml.Forx;

// Let's make a sample XML
var xml:XML = new XML('<Farm Area="270" />');

// We pass that XML to the newly created Forx
// Here, we explicitly demand that every string matching is case sensitive
var forx:Forx = new Forx(xml, Forx.CASE_SENSITIVE);

// And now, let's use getAttributeValue() method which offers some flexibility
// when compared to the standard 'attributes' array

// First, we try using the wrong word casing
trace("case sensitive match by default, wrong case: " + forx.getAttributeValue("aRea", xml.firstChild));

// Next, we try using the correct word casing
trace("case sensitive match by default, correct case: " + forx.getAttributeValue("Area", xml.firstChild));

// Finally, we explicitly ask for the case-insensitive match, with the wrong word casing
trace("explicit case insensitive match, wrong case: " + forx.getAttributeValue("aRea", xml.firstChild, Forx.CASE_INSENSITIVE));
```

## 3. Class Interface

In this chapter, every non-mandatory syntax part is surrounded by square brackets: [ and ].

Also, if you encounter a syntax that looks like { A | B }, that means you can select either A or B, but not both. Don't type these characters in your code, unless they represent the actual code (these are **red**, not black).

ForX Class Interface can be divided in several main categories:

- ForX Constructor:  
`Forx(xml_reference:XML[, default_case:Number])`
- XML Child Node Methods:  
`getChildrenNames([node:XMLNode[, uniqueOnly:Boolean]]):Array`  
`getChildrenCount([ node:XMLNode[, recursion:Boolean] | [recursion:Boolean] ]):Number`  
`getChildNode(name:String[, node:XMLNode[, lettercase:Number]]):XMLNode`
- XML Path Node Methods:  
`getPathNode([path:String[, node:XMLNode[, lettercase:Number]]):XMLNode`  
`getPathValue([path:String[, node:XMLNode[, lettercase:Number]]):String`  
`createPathNode(path:String[, value:Any[, lettercase:Number]]):XMLNode`  
`extractPathNode([path:String[, node:XMLNode[, lettercase:Number]]):XMLNode`  
`getMultiNodes(path:String[, node:XMLNode[, lettercase:Number]):Array`
- XML Map Resolving Method:  
`fetchData({ mapString:String | mapArray:Array }[, node:XMLNode[, lettercase:Number]):Array`
- XML Attributes Methods:  
`getAttributes([node:XMLNode[, lettercase:Number]]):Array`  
`getAttribute(attribute:String[, node:XMLNode[, lettercase:Number]):String`  
`getAttributeValue(attribute:String[, node:XMLNode[, lettercase:Number]):String *DEPRECATED`
- XML Formulation Methods:  
`formulateNode(formula:String):XMLNode`  
`formulateXML({ formulaString:String | formulaArray:Array }):XML`
- Tokens & Paths & Queries:  
`getNCFTokens(node:XMLNode):Object *STATIC`  
`getAddress(node:XMLNode[, queryLimit:Number]):String *STATIC`
- Miscellaneous:  
`getXML(Void):XML`  
`fillTemplate(str:String, data:Array):String`
- Constants:  
`VERSION:String *STATIC`

### 3.1 ForX Constructor

`ForX([xml_reference:XML[, default_case:Number]])`

**Usage:**

Use this to construct a new ForX instance (by using the `new` keyword), and to pass XML reference to it. The referenced XML is optional, but if you pass it, almost all public ForX methods will be able to assume this XML to work on (this XML object is called the “Main XML” throughout this document), but are not required to. Note that some methods (such as `formulateXML`) cannot manipulate anything other than Main XML.

You can always call `getXML` method (see 3.2 below) to retrieve the reference back to your application. However, you are not required to use `getXML` in every occasion, because all subsequent changes made by ForX affect the original XML by reference, so if you already have an access to your original XML, that’s all you need.

After you’re done with ForX, you are free to destroy it. A destroyed ForX instance won’t destroy the XML it worked on. Conversely, you can have more than one ForX instances manipulating a single XML object. If you lose your original XML reference, and there is a ForX instance using it, that XML object won’t be deleted and you can still access it by calling `getXML` method (see 3.2 below).

**Parameters:**

<code>xml_reference:XML</code>	Optional. A reference to the existing XML object. If no reference is passed, a new XML instance is automatically created to work on.
<code>default_case:Number</code>	Optional. A case-sensitivity setting which will be used as a default preference for all other ForX methods. Use either <code>ForX.CASE_SENSITIVE</code> (0) or <code>ForX.CASE_INSENSITIVE</code> (1) static class constants. If omitted, the case-insensitive setting match is assumed by default (‘Jerry’ and ‘jerry’ are resolved as identical).

### 3.2 getXML

`getXML(Void):XML`

**Usage:**

Returns a reference to the Main XML object passed to the constructor. Use this if you are able to access the ForX instance that manipulates XML, but not the original XML itself. This way you can interact with the original XML class interface, which is still a powerful way to accomplish certain tasks.

Remember that ForX wasn’t designed to replace the original XML class interface, but only to make the XML juggling easier to implement and maintain.

### 3.3 getChildrenNames

`getChildrenNames([node:XMLNode[, uniqueOnly:Boolean]]):Array`

**Usage:**

Returns the array of all child nodes’ names in the specified node. If you don’t specify the node, the Main XML root node will be used.

**Parameters:**

<code>node:XMLNode</code>	Optional. The reference node. If omitted, the Main XML root node is used as a reference.
<code>uniqueOnly:Boolean</code>	Optional. Pass <code>true</code> to obtain only truly unique names in the list (without repeating). <code>false</code> by default.

### 3.4 getChildrenCount

1: `getChildrenCount([recursion: Boolean]): Number`  
or  
2: `getChildrenCount(node: XMLNode[, recursion: Boolean]): Number`

Usage:

Returns the number of all child nodes in the specified node. If needed, this method alternatively counts the number of all nested nodes, recursively. The value nodes are included.

Parameters:

<code>node: XMLNode</code>	Required (2). The reference node. If syntax 1 is used, the Main XML root node is used as a reference.
<code>recursion: Boolean</code>	Optional (1 or 2). Specifies whether to apply recursion or not. <code>false</code> by default.

### 3.5 getChildNode

`getChildNode(name: String[, node: XMLNode[, lettercase: Number]]): XMLNode`

Usage:

Returns the node's child node that bears the given name, as `XMLNode` object.

Parameters:

<code>name: String</code>	Required. The name of the child node to return. This method doesn't support Dot Path Notation (DPN), but supports the attribute query if needed in the following format:
---------------------------	--

`childname?query`

See 4.1 DPN, Dot Path Notation, for DPN and the attribute query.

<code>node: XMLNode</code>	Optional. The reference node. It doesn't have to be a member of the Main XML. If omitted, the Main XML root node is used as a reference.
<code>lettercase: Number</code>	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.

### 3.6 getPathNode

`getPathNode([path: String[, node: XMLNode[, lettercase: Number]]): XMLNode`

Usage:

Returns the `XMLNode` object on the specified XML path.

Parameters:

<code>path: String</code>	Optional. The path of the node to be obtained. This argument supports the Dot Path Notation (DPN). See 4.1, DPN, Dot Path Notation, for more. By omitting this argument or passing <code>null</code> or empty string to it, you can force the method to work with the top-level node in the Main XML. However, if you use <code>node</code> , that node will be used instead.
<code>node: XMLNode</code>	Optional. The reference node. It doesn't have to be a member of the Main XML. If omitted, the Main XML root node is used as a reference.
<code>lettercase: Number</code>	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.

### 3.7 getPathValue

```
getPathValue([path:String[, node:XMLNode[, lettercase:Number]]]):String
```

Usage:

Returns the node value on the specified XML path. Alternatively, it can be used to obtain the attribute value, as well. The returned type is always `String`, unless the method fails to find the indicated path or that particular path doesn't contain a value. In that case it returns `undefined`.

Parameters:

<code>path:String</code>	Optional. The path of the node or attribute which value is to be obtained. This argument supports the Dot Path Notation with an Attribute (DPN/A). See 4.1.1, DPN/A, Dot Path Notation with an Attribute, for more.  By omitting this argument or passing <code>null</code> or empty string to it, you can force the method to work with the top-level node in the Main XML. However, if you use <code>node</code> , that node will be used instead. Additionally, if you use the attribute part (see 4.1.1, DPN/A) you can choose to omit the first part of the string (leaving only <code>@id</code> , for example).
<code>node:XMLNode</code>	Optional. The reference node. It doesn't have to be a member of the Main XML. If omitted, the Main XML root node is used as a reference.
<code>lettercase:Number</code>	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.

### 3.8 createPathNode

```
createPathNode(path:String[, value:Any[, lettercase:Number]]):XMLNode
```

Usage:

Creates the node according to the specified path, and attaches a value node to it, if provided. Returns a reference to the newly created `XMLNode` object, to which you can immediately apply attributes, if needed. This method works with the Main XML only.

All non-existing parent nodes in the path string will be created in the process (note that regardless of the case sensitivity preference, the nodes will be named as is, according to the names provided in the path). If the target node already exists, `createPathNode` won't create a new one. Using the exclamation mark suffix as stated in 4.1, DPN is a good workaround, if this is the case. Similarly, you can use that suffix on every other node in the path that needs to be created by "force."

Parameters:

<code>path:String</code>	Required. The path of the node to be created. The path must be absolute (i.e. it must define every node from the topmost one all the way down). This argument supports the Dot Path Notation (DPN). See 4.1, DPN, Dot Path Notation, for more.
<code>value:Any</code>	Optional. The value node of the target node. If omitted, no value node will be created, and the node will remain empty. (Type declarator "Any," shown in the syntax above, doesn't really exist in ActionScript 2.0. It is presented here just to illustrate that this parameter virtually accepts <u>any</u> type of data. However, embedded ActionScript type converter will automatically cast this information to <code>String</code> . Try to use the primitive data types, as a general rule.)
<code>lettercase:Number</code>	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.

### 3.9 extractPathNode

```
extractPathNode([path:String[, node:XMLNode[, lettercase:Number]]]):XMLNode
```

Usage:

Extracts the node from the specified XML path and returns it. The extracted node is permanently removed from the XML structure it belongs to.

Parameters:

path:String	Optional. The path of the node to be extracted. This argument supports the Dot Path Notation (DPN). See 4.1, DPN, Dot Path Notation, for more. By omitting this argument or passing <code>null</code> or empty string to it, you can force the method to work with the top-level node in the Main XML. However, if you use node, that node will be used instead.
node:XMLNode	Optional. The reference node. It doesn't have to be a member of the Main XML. If omitted, the Main XML root node is used as a reference.
lettercase:Number	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.

### 3.10 getMultiNodes

```
getMultiNodes(path:String[, node:XMLNode[, lettercase:Number]]):Array
```

Usage:

Unlike all other path methods, `getMultiNodes` returns all nodes that fulfill the given requirements, and not just the one first encountered. If you use asterisk (\*) in the path as the last `nodename` (See 4.1, DPN, Dot Path Notation, for more), this method will return all child nodes within the last indicated parent node. The returned array contains `XMLNode` objects.

Parameters:

path:String	Required. The path of the nodes to be obtained. The path might include attribute queries to narrow down the list. This argument supports the Dot Path Notation (DPN). See 4.1, DPN, Dot Path Notation, for more.
node:XMLNode	Optional. The reference node. It doesn't have to be a member of the Main XML. If omitted, the Main XML root node is used as a reference.
lettercase:Number	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.



### 3.11 fetchData

```
1: fetchData(mapString:String[, node:XMLNode[, lettercase:Number]]):Array
   or
2: fetchData(mapArray:Array[, node:XMLNode[, lettercase:Number]]):Array
```

Usage:

Obtains the values from the reference node according to the string map provided. Each value is then converted in the appropriate data type specified in the map, and the entire array of these values is returned as a result.

Parameters:

mapString:String	Required (1). The serialized scheme of values that are to be extracted from the XML. This string map utilizes comma-separated MSF strings. See 4.5, MSF, Mapping String Format, for more.
mapArray:Array	Required (2). The array must be comprised of strings that utilize MSF (Mapping String Format). See 4.5, MSF, Mapping String Format, for more. Each value is one array element and you cannot use comma-separated MSF strings.
node:XMLNode	Optional. The reference node. It doesn't have to be a member of the Main XML. If omitted, the Main XML root node is used as a reference.
lettercase:Number	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.

### 3.12 getAttributes

```
getAttributes([node:XMLNode[, lettercase:Number]]):Array
```

Usage:

Returns the array of all attributes attached to the given reference node. The elements of this array are objects, containing the properties called name and value. If you want to obtain an associative array of values, use `XMLNode.attributes` instead.

Parameters:

node:XMLNode	Optional. The reference node. It doesn't have to be a member of the Main XML. If omitted, the Main XML root node is used as a reference.
lettercase:Number	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.

Example:

```
trace(forx.getAttributes(someXmlNode)[0].name);
```

### 3.13 getAttributeValue

```
getAttributeValue(attribute:String[, node:XMLNode[, lettercase:Number]]):String
```

Usage:

This is a deprecated name for this method. See 3.14, `getAttribute`.

### 3.14 getAttribute

`getAttribute(attribute:String[, node:XMLNode[, lettercase:Number]]):String`

Usage:

Returns the value of the specific attribute. Always returns `String`.

Parameters:

<code>attribute:String</code>	Required. The name of the attribute.
<code>node:XMLNode</code>	Optional. The reference node. It doesn't have to be a member of the Main XML. If omitted, the Main XML root node is used as a reference.
<code>lettercase:Number</code>	Optional. Used to override the default case-sensitivity preference specified in the constructor. If omitted, the default preference is used.

### 3.15 getNCFTokens

`Forx.getNCFTokens(node:XMLNode):Object`

Usage:

Static method. Returns the string tokens required to recreate the entire node in the NCF (Node Creation Formula) for later use in `formulateNode`, for example. The resulting `Object` contains the following information: `formula`, `path`, `attributes`, and `value`. Formula is made by concatenating path, attributes, and value tokens together. See 4.2, Node Creation Formula, for more.

Parameters:

<code>node:XMLNode</code>	Required. The reference node to derive tokens from.
---------------------------	---

### 3.16 getAddress

`Forx.getAddress(node:XMLNode[, queryLimit:Number]):String`

Usage:

Static method. Returns the node's path address in the DPN (Dot Path Notation) for later use in `getPathNode`, for example. The resulting `String` will include attribute queries needed to resolve any ambiguity between nodes that share the same name, unless you change `queryLimit`. See 4.1, Dot Path Notation, for more on paths and attribute queries.

Parameters:

<code>node:XMLNode</code>	Required. The reference node to derive address from.
<code>queryLimit:Number</code>	Optional. Level of path uniqueness. To select one of three modes of behavior, you can pass any of the following static class constants: <code>Forx.NO_QUERIES</code> (0), <code>Forx.PARENT_QUERIES</code> (1), <code>Forx.ALL_QUERIES</code> (2, default). Depending on the XML structure, node's DPN might vary from queryless to queryful, in order to unambiguously pinpoint the node. If you select Parent Queries, only the last (target node's) attribute query won't be shown in the address. Similarly, if you select No Queries, DPN will remain completely queryless. For example, <code>getNCFTokens</code> uses Parent Queries to define the node's path token.

### 3.17 formulateNode

`formulateNode(formula:String):XMLNode`

Usage:

Creates a node (and all of its containers, if needed), allowing a creation of multiple attributes and even the value node in the same process. This method works with the Main XML only, and returns a reference to the `XMLNode` object that was just created.

Parameters:

`formula:String` Required. The Node Creation Formula, following the NCF pattern. See 4.2, NCF, Node Creation Formula, for more.

### 3.18 formulateXML

1: `formulateXML(formulaString:String):XML`

or

2: `formulateXML(formulaArray:Array):XML`

Usage:

Creates multiple nodes (and all of their containers, when needed), and allows creation of multiple attributes per each node and even the value node assigned in the same process. This method works with the Main XML only, and returns a reference to the Main XML object.

Parameters:

`formulaString:String` Required (1). The string formula, following the XCF or CXF patterns. See 4.3, XCF, XML Creation Formula, and 4.4, CXF, Compact XML Formula, for more. Both notations can be used with this method's usage, but only one at a time.

`formulaArray:Array` Required (2). The array of NCF strings. See 4.2, NCF, Node Creation Formula. You cannot use XCF or CXF if you pass an array.

### 3.19 fillTemplate

`fillTemplate(str:String, data:Object):String`

Usage:

Finds all special tags in the template string provided, and replaces them with the corresponding object data. This is particularly useful when you are creating a dynamic XML full of variables. The resulting string is returned.

Parameters:

`str:String` Required. A template string. You can use XCF or CXF patterns (see 4.3, XCF, XML Creation Formula, and 4.4 CXF, Compact XML Formula, for more), but this method is not restricted to any of these. You can use it for anything else, as well. The tags must be in `<name>` format, where 'name' stands for the name of the property defined inside data object. Enclosing characters `<` and `>` are removed on replacement, however, their presence implies that you cannot use this method for regular XML or (X)HTML strings, at least without some workaround.

`data:Object` Required. Pass this object to feed the method with the appropriate values. All values are converted into `String` before being embedded in the template. Object data is used as an associative array.

Example:

```
import com.orionSyndrome.xml.Forx;
var forx:Forx = new Forx();
var data:Object = new Object();
data = { op: "action", id: 5, childnode: "RqChild", childattr: false };
var cxf:String = forx.fillTemplate("{Rq[op=<op>;id=<id>]{<childnode>[attr=<childattr>]}}", data);
trace(forx.formulateXML(cxf));
```

## 4. Notations Used

Throughout this chapter, all optional parts are surrounded by these characters: < and >.

That means if you omit that enclosed part of the string, parser will be able to consume it properly.

Don't use these characters in the actual strings (even if you want them that bad, these characters are not compliant with the XML specification, and should be substituted with these sequences: &lt; and &gt;).

On the other side, all other special characters are necessary, and without them you'll most probably encounter problems and weird errors.

There are several notations used to describe the desired XML information in ForX:

- 1) Dot Path Notation (DPN) and Dot Path Notation with an Attribute (DPN/A)  
Very much similar to the way object referencing is done in ActionScript and other similar languages supporting so-called "dot notation".
- 2) Node Creation Formula (NCF)  
A way of defining entire XML nodes in a single line, including their name, attribute(s), and value node, but without the inner structure.
- 3) XML Creation Formula (XCF)  
Serialized NCF string, providing room for a bit more complex XML structure definition.
- 4) Compact XML Formula (CXF)  
A different and complex approach to generating the XML structure, but without the XML itself, and including the support for nesting.
- 5) Mapping String Formula (MSF)  
Used to precisely define maps of the values that are to be extracted from the sample XML. In reality, this is an expanded version of DPN/A.

## 4.1 DPN, Dot Path Notation

1: `nodename<declarator>`

or

2: `nodename1<declarator1>.nodename2<declarator2>. ... .nodenameN<declaratorN>`

Note that syntax 2 elements are delimited by dots (.).

Parts:

`nodename` Required. The name of the node to look for.

`declarator` Optional. A declarator can be any of these characters:

- `!`, force new (makes sense only if that particular node is about to be created)
- `?`, query expression, one of the following:

1: `?attribute:queryvalue`

or

2: `?attribute1:queryvalue1&attribute2:queryvalue2& ... &attributeN:queryvalueN`

Use a colon (:) to separate the name/value pair. The both parts are mandatory.

Note that syntax 2 elements are delimited by ampersands (&).

Note: You can precede `attribute` with a caret symbol (^) to create a logical NOT.

Example:

`Root.Branch?id:1.Sector?^color:red.Member!`

[FIND Root] [GO\_INSIDE] [FIND Branch WHERE id IS 1] [GO\_INSIDE] [FIND Sector WHERE color IS NOT red] [GO\_INSIDE] [FIND Member NEW]

Applicable to:

`<Root><Branch id="0" /><Branch id="2"><Sector color="red" /><Sector color="blue" /></Branch></Root>`

Used in Methods:

`getPathNode` (see 3.6), `getPathValue` (see 3.7 and 4.1.1 for the exact notation used in this method), `createPathNode` (see 3.8), `extractPathNode` (see 3.9), `getMultiNodes` (see 3.10), `getAddress` (see 3.16)

### 4.1.1 DPN/A, Dot Path Notation with an Attribute

`<path><@attribute>`

Parts:

`path` Optional. The ordinary DPN string. See 4.1, DPN, Dot Path Notation, for more on this.

`@attribute` Optional. The name of the node's attribute. It must have a leading @.

Example:

`Root.Branches.Branch?id:2.Sector@color`

[FIND Root] [GO\_INSIDE] [FIND Branches] [GO\_INSIDE] [FIND Branch WHERE id IS 2] [GO\_INSIDE] [FIND Sector] [GET\_ATTRIBUTE color]

Applicable to:

`<Root><Branches><Branch id="1"><Sector color="blue" /></Branch><Branch id="2"><Sector color="red" /></Branch>  
</Branches></Root>`

Used in Method:

getPathValue (see 3.7). Also, MFS notation is very similar to DPN/A (see 4.5)

## 4.2 NCF, Node Creation Formula

`path<[attributes]><=nodevalue>`

Parts:

`path`

Required. The ordinary DPN string. See 4.1, DPN, Dot Path Notation, for more on this.

`[attributes]`

Optional. The entire `attributes` construct must be enclosed with square brackets: `[` and `]`. This part is comprised of the following:

1: `[attribute=setvalue]`

or

2: `[attribute1=setvalue1;attribute2=setvalue2; ... ;attributeN=setvalueN]`

Use the equals sign (=) to separate the name/value pair. The both parts are mandatory. Note that syntax 2 elements are delimited by semicolons (;).

`=nodevalue`

Optional. The XML node value. Must have a leading =.

Example:

`Root.Branches.Branch[id=0;size=10]=logistics`

[FIND Root] [GO\_INSIDE] [FIND Branches] [GO\_INSIDE] [FIND Branch] [ATTRIBUTES id=0; size=10] [VALUE logistics]

Applicable to:

`<Root><Branches><Branch id="1" size="5">marketing</Branch></Branches></Root>`

Used in Methods:

`formulateNode` (see 3.17), `getNCFTokens` (see 3.15)

### 4.3 XCF, XML Creation Formula

- 1: `ncformula`  
or  
2: `ncformula1,ncformula2, ... ,ncformulaN`

Note that syntax 2 elements are delimited by commas (,).

Parts:

`ncformula` Required. See 4.2, NCF, Node Creation Formula.

Example:

`Root.Branches.Branch![id=3],Root.Branches.Branch?id:3.Sector![id=0]=marketing`  
[FIND Root] [GO\_INSIDE] [FIND Branches] [GO\_INSIDE] [FIND Branch NEW] [ATTRIBUTES id=3] [NEXT]  
[FIND Root] [GO\_INSIDE] [FIND Branches] [GO\_INSIDE] [FIND Branch WHERE id IS 3] [FIND Sector NEW] [ATTRIBUTES id=0] [VALUE marketing]

Applicable to:

`<Root><Branches><Branch id="1">logistics</Branch><Branch id="2">  
<Sector id="0">creativity</Sector></Branch></Branches></Root>`

Used in Methods:

`formulateXML` (see 3.18)



## 4.4 CXF, Compact XML Formula

- 1: {*cxfnode*}
- or
- 2: {*cxfnode*1+*cxfnode*2+ ... +*cxfnode*N}

Note that syntax 2 elements are delimited by pluses (+).  
The entire *cxfnode* construct must be embraced by curly braces: { and }.

Parts:

*cxfnode* Required. *cxfnode* is comprised of several parts:

*nodename*<*declarator*><[*attributes*]><*content*>

See Sub Parts below, for more details on *cxfnode* construct.

Sub Parts:

*nodename* Required. The name of the node.

*declarator* Optional. Character ! ("force new"). See 4.1, DPN, Dot Path Notation, for more on this character.

[*attributes*] Optional. The attributes construction is described under 4.2, NCF, Node Creation Formula. Must be enclosed by square brackets: [ and ].

*content* Optional. The content can be any of these constructs:

- =*nodevalue*, The XML node value. Must have a leading =.
- *cxfnode*, allowing these parts to be nested within each other. The main CXF syntax applies (1 or 2).

If the content part is omitted, the node remains empty.

Step-By-Step Example:

```
{Root}
{Root{Branches}}
{Root{Branches{Branch![id=3]}}}
{Root{Branches{Branch![id=3]+Branch![id=4]}}}
{Root{Branches{Branch![id=3]{Sector[color=red]=marketing}+Branch![id=4]}}}
```

Resulting XML:

```
<Root>
  <Branches>
    <Branch id="3">
      <Sector color="red">marketing</Sector>
    </Branch>
    <Branch id="4" />
  </Branches>
</Root>
```

Used in Method:

formulateXML (see 3.18)

## 4.5 MFS, Mapping Formula String

- 1: `<type=>valuepath`  
or  
2: `<type1=>valuepath1,<type2=>valuepath2, ... ,<typeN=>valuepathN`

Note that syntax 2 elements are delimited by commas (,).

If you passed an Array to `fetchData`, you cannot use syntax 2.

Parts:

`type=` Optional. A data type letter. Currently, any of the following lowercase characters are available:

- `a`, automatic type casting (default if omitted)
- `s`, primitive `String` type
- `n`, primitive `Number` type
- `b`, primitive `Boolean` type
- `x`, complex `XMLNode` type

If `type` is used, the trailing `=` is mandatory.

`valuepath` Required. This is essentially a Dot Path Notation with an Attribute. See 4.1.1, DPN/A, Dot Path Notation with an Attribute, for more information on this construct.

Example:

```
s=Root.Titles.Sector?color:red,n=Root.Branches.Branch?id:2.Sector@membersnum
```

String: [FIND Root] [GO\_INSIDE] [FIND Titles] [GO\_INSIDE] [FIND Sector WHERE color IS red] [GET\_VALUE] [NEXT]

Number: [FIND Root] [GO\_INSIDE] [FIND Branches] [GO\_INSIDE] [FIND Branch WHERE id IS 2] [GO\_INSIDE] [FIND Sector] [GET\_ATTRIBUTE membersnum]

Applicable to:

```
<Root><Titles><Sector color="red">logistics</Sector><Sector color="blue">creativity</Sector></Titles>  
<Branches><Branch id="1"><Sector membersnum="8" /></Branch><Branch id="2"><Sector membersnum="16" /></Branch>  
</Branches></Root>
```

Used in Method:

`fetchData` (see 3.11)

## 5. Development Road

Here you can see what I plan to do in the future, and if I (or someone else) noticed something that doesn't work properly. For the versioning history, consult the external file `ForX-Changelog.txt`.

Please contact me if you have any comments on ForX or if you need further information regarding ForX features/implementation/bugs etc.

### 5.1 Future Releases

Although the code still(!) cries for some serious feedback, I was able to pinpoint some minor issues to work on for the following releases.

Code optimization in the future.

A caching system for complex node querying which will seriously improve the attribute query performance.

I will probably have to implement a support for escape sequences in case of some special characters.

I might as well add a whitespace support for CXF strings. That would be really cool, to make them more human-readable.

One other thing I'm considering is to enable passing an `XMLNode` argument to the constructor. Currently, only XML object can be passed to it, making that feature quite rigid in design (and very impractical in some cases).

In the next version, I'll try to introduce a new, "smart" and serialized fetch method based on a given key attribute (i.e. "id"). Also expect `getPathValue` and `getAttribute` to receive automatic type casting on their own.

## 5.2 Known Bugs

None so far :)

However, the class hasn't been tested with XMLs that use namespace prefixes.

If you test it under this circumstance, please feel free to send me a feedback (see the next chapter for Contact Info). Thanks!

## 6. Terms of Usage / Additional Information & Contact

This Software is released under the MIT Licence (<http://www.opensource.org/licences/mit-licence.php>).  
The terms and conditions for using it can be found in ForX-Licence.txt and at the beginning of the source code file (Forx.as).

If it appears to you that some files are missing, write to [feedback@orionsyndrome.com](mailto:feedback@orionsyndrome.com) and I will send you the missing files.

Probably there are many products such as this on the Flash scene (XPath?!), but I can guarantee you that neither of them is as comprehensible and easy-to-implement as this one. Alas, this project is still in alpha, which means that some features are still missing or they don't even work in some cases (and I am either unaware of any such issue at the moment, or you can find that issue in the Development Road chapter, under Known Bugs).

If you need something done, and it seems there's no way to do that using ForX, please write to me.

Also, it would be nice if you could send me some comments and (detailed) bug reports, or just any other feedback you find appropriate. I'll try to respond to you as soon as possible.

I sincerely hope this AS2.0 class saved your..... Yeah, bottom :)

Cheers,  
orionSyndr[o]me