Validating XML Element Names

By David Adams

Technical Note 05-40

Overview

The purpose of this technical note is to provide information on the rules for naming XML elements and document a method for testing element names for validity. This information and code are useful when producing custom XML from 4th Dimension.

XML Names: Overview of Rules

XML element names must follow the naming rules defined in the XML specification. The rules for XML names are similar but not identical to 4th Dimensions variable and method naming rules. Below is a summary of the key requirements and behaviors of XML names:

- XML names are case-sensitive, unlike 4th Dimension names. Therefore, the following are all different: `customer`, `Customer`, and `CUSTOMER`.
- XML names may begin with ideograms, letters, and the punctuation marks hyphen and period.
- XML names may contain ideograms, letters, digits, and the punctuation marks hyphen, period, colon, and underscore.
- The colon character is a legal XML name character, but its use is reserved for namespaces and must not be used in any other context.
- XML names may not contain whitespace characters of any kind, including spaces, non-breaking spaces, tabs, line feeds, and carriage returns.
- XML names beginning with XML, in any case combination, are reserved for use in current and future XML standards.
- There are no limits on the length of an XML name.

Case-Sensitivity in 4th Dimension

As mentioned, XML element and attribute names should always be treated case-sensitively. It's worth paying special note to this rule since the 4th Dimension language and database engine are normally case-insensitive. For information and code to add case-sensitivity to 4th Dimension, see Technical Note 05-41 **Case-Sensitive Operations** in 4th Dimension.

XML Names: Examples of Good and Bad Names

Below are some examples to flesh out the rules shown above.

Name	Discussion
`Species_Name`	This name is **legal**.
`1Species_Name`	This name is **illegal** because it begins with a number.
`Species_Name_1`	This name is **legal**. Numbers are allowed in the body of a name.
`:Species_Name`	Initial colons are only legal for namespaces. Avoid the initial colon
`Species Name`	This name is **illegal** because it includes a space.
`xml_ Species_Name`	This is a **legal** XML name that is **illegal** to use for custom XML because it begins with the characters `xml`. Names beginning with x m l, in any case combination, are reserved for current and future XML standards.

XML Names: Simplifies Rules for 4th Dimension

For most 4th Dimension programmers, the main rules for naming XML elements can be reduced to the following:

- Names are case-sensitive.
- Names should begin with letters in the range a-z or A-Z.
- Names should only contain numbers, letters in the range a-z or A-Z, and the punctuation marks period and underscore.
- Other punctuation characters should be avoided unless you know how and why to use them.

Simplified Rules: Legal First Character

Following the simplified rules listed above, XML element names may begin with any of the following characters:

ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char
58	:	73	I	82	R	95	_	105	i	114	r
65	A	74	J	83	S	97	a	106	j	115	s
66	B	75	K	84	T	98	b	107	k	116	t
67	C	76	L	85	U	99	c	108	l	117	u
68	D	77	M	86	V	100	d	109	m	118	v
69	E	78	N	87	W	101	e	110	n	119	w
70	F	79	O	88	X	102	f	111	o	120	x
71	G	80	P	89	Y	103	g	112	p	121	y
72	H	81	Q	90	Z	104	h	113	q	122	z

Remember that the : (ASCII 58) character is reserved for use in namespaces and should not be used otherwise.

Simplified Rules: Legal Body Characters

Following the simplified rules listed above, XML element names may contain any of the following characters:

ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char
45	-	57	9	74	J	85	U	101	e	112	p
46	.	58	:	75	K	86	V	102	f	113	q
48	0	65	A	76	L	87	W	103	g	114	r
49	1	66	B	77	M	88	X	104	h	115	s
50	2	67	C	78	N	89	Y	105	i	116	t
51	3	68	D	79	O	90	Z	106	j	117	u
52	4	69	E	80	P	95	_	107	k	118	v
53	5	70	F	81	Q	97	a	108	l	119	w
54	6	71	G	82	R	98	b	109	m	120	x
55	7	72	H	83	S	99	c	110	n	121	y
56	8	73	I	84	T	100	d	111	o	122	z

Remember that the : (ASCII 58) character is reserved for use in namespaces and should not be used otherwise.

Name Validation Code: *xmlNameIsValid*

There are many ways to implement code to test XML string names for validity. This Technical Note describes a method called *xmlNameIsValid* that implements the simplified rules discussed above. Below is the code of the method:

```
C_BOOLEAN($0;$nameIsValid_b)
C_TEXT($1;$nameSource_t)

$nameSource_t:=$1

$nameIsValid_b:=False

C_TEXT($nameCleaned_t)
$nameCleaned_t:=""
$nameCleaned_t:=xmlNameClean ($nameSource_t)

Case of
    : ($nameCleaned_t="")
      $nameIsValid_b:=False

    : ($nameSource_t#$nameCleaned_t)
      $nameIsValid_b:=False

    Else
      $nameIsValid_b:=True

End case

$0:=$nameIsValid_b
```

As the code listing shows, the *xmlNameIsValid* method passes the test name to another method called *xmlNameClean* and then tests the results. If the results are empty or not equal to the original, then the test name is not valid. The XML name validation rules are hidden away in the *xmlNameClean* method. The *xmlNameIsValid* method is just a convenient way of calling the *xmlNameClean* method.

Name Utility Code: *xmlNameClean*

The XML name validation code could obviously have been put directly into the *xmlNameIsValid* method. However, breaking the rules out into a distinct method makes them easier to reuse. There are many situations where it is useful to take a string and clean it in an effort to generate a valid XML name. Below are two scenarios that make use of the *xmlNameClean* method:

End User XML Editor
Joan is chief programmer on the AcmeDesk program at ACME Black Dot in Utah. End-users now need to produce custom XML from existing field data and calculations. Joan is designing an editor that allows users to pick fields and calculations for export. Users are free to name the elements as they like so that they can generate exports that work with a variety of other programs. Early user testing with paper prototypes shows that, left to themselves, users routinely create illegal XML names. To avoid such problems, Joan realizes that names must be cleaned automatically before they are accepted, using code like the line shown below to remove illegal characters:

```
xml_name_entry_area:=XMLNameClean(xml_name_entry_area)
```

Field-to-Element Export
Bruce works for Joan at ACME Black Dot and has been assigned the task of building code to dump the contents of every record in the database into an XML file. To make the XML easier to read, Joan has asked Bruce to make the element names look like the original table and field names. Bruce is new to XML but very experienced with 4th Dimension. He writes some code that loops through the tables, loading records, and building XML element names using **Table name** *and* **Field name***. Unfortunately, Bruce quickly discovers that his export files are rejected by XML tools. There's nothing wrong with the text, but the XML element names are often illegal. Since XML is new at ACME, their existing naming conventions don't follow the restrictions imposed by XML. For example, field names routine include spaces, which are illegal in XML names. Bruce talks with Joan who lets him know about the* XMLNameClean *routine. Now, to produce an element from a field name, Bruce calls code a bit like this:*

```
$element_name:=xmlNameClean (Field name($field_pointer))
```

Below is the code for the *xmlNameClean* method:

```
C_TEXT($0;$result_t)
C_TEXT($1;$source_t)

$source_t:=$1

$result_t:=""

C_LONGINT($source_length)
$source_length:=Length($source_t)

C_BOOLEAN($continue_b)
$continue_b:=True

` Verify that source name is not empty.
If ($source_length=0)` Test for empty string (invalid)
   $result_t:=""
   $continue_b:=False
End if

` Verify that first character is legal.
If ($continue_b)
   ` Valid first characters for an XML name are (Inclusive)
   ` A-Z a-z _ :
   ` The colon has a very specific use and meaning so don't use it unless you know how and why.
   C_LONGINT($firstChar_ascii)
   $firstChar_ascii:=Ascii($source_t[[1]])

   Case of
     : ($firstChar_ascii>=65) & ($firstChar_ascii<=90)` A-Z are valid first characters.
       $result_t:=$source_t[[1]]

     : ($firstChar_ascii>=97) &( $firstChar_ascii<=122)` a-z are valid first characters.
       $result_t:=$source_t[[1]]

     : ($firstChar_ascii=95) | ($firstChar_ascii=58)` Underscore (95) and Colon (58) may be valid.
       $result_t:=$source_t[[1]]

     Else
       $result_t:=""
       $continue_b:=False` Bad first character, terminate.
   End case

End if

` Remove any illegal following characters.
If ($continue_b)
   ` Valid body characters for an XML name are (Inclusive)
   ` A-Z a-z _ : - . 0-9
   ` The colon has a very specific use and meaning so don't use it unless you know how and why.
   C_LONGINT($index)
   C_LONGINT($ascii)

   For ($index;2;$source_length)
     $ascii:=Ascii($source_t[[$index]])

     Case of
       : ($ascii>=65) & ($ascii<=90)` A-Z are valid body characters.
         $result_t:=$result_t+$source_t[[$index]]

       : ($ascii>=97) & ($ascii<=122)` a-z are valid body characters.
```

```
            $result_t:=$result_t+$source_t[[$index]]
         : ($ascii=95) | ($ascii=58)` Underscore (95) and Colon (58) are valid body characters.
            $result_t:=$result_t+$source_t[[$index]]
         : ($ascii=45) | ($ascii=46)` Hyphen (45) and Period (46) are valid body characters.
            $result_t:=$result_t+$source_t[[$index]]
         : ($ascii>=48)&($ascii<=57)` 0-9 are valid body characters.
            $result_t:=$result_t+$source_t[[$index]]
         Else
            ` Bad character, don't include it.
      End case
   End for
End if
` Verify that name doesn't start with XML.
If (Length($result_t)>=3)
   ` Note: 4D's string comparisons are not case-sensitive, so the same result
   ` could be achieved with this line of code:
   ` If ($result_t="xml@")
   ` I've taken this more pedantic approach to make the rule explicit.
   C_LONGINT($charOne_ascii)
   C_LONGINT($charTwo_ascii)
   C_LONGINT($charThree_ascii)
   $charOne_ascii:=Ascii($result_t[[1]])
   $charTwo_ascii:=Ascii($result_t[[2]])
   $charThree_ascii:=Ascii($result_t[[3]])
   If ($charOne_ascii=88) | ($charOne_ascii=120)` 'X' or 'x'
      If ($charTwo_ascii=77) | ($charTwo_ascii=109)` 'M' or 'm'
         If ($charThree_ascii=76) | ($charThree_ascii=108)` 'L' or 'l'
            $result_t:=""` Name is bad. Would need to call routine recursively after trim if didn't clear here.
            $continue_b:=False
         End if
      End if
   End if
End if
$0:=$result_t
```

Implementation Notes: *xmlNameClean*

There are many ways to implement the functionality embedded in the *xmlNameClean* method. An obvious example is to replace the hard-coded character value tests with a data-driven solution. In such an implementation, the valid ASCII codes for initial characters would be stored in, for example, arrays before the method is run. Testing the initial and body characters would then look like the pseudo-code below:

```
 ` Verify that first character is legal.
If ($continue_b)
   C_LONGINT($firstChar_ascii)
   $firstChar_ascii:=Ascii($source_t[[1]])

   C_LONGINT($element)
   $element:=Find in array(array_of_legal_first_character_ascii_codes;$firstChar_ascii)
   If ($element>0) ` Good character.
     $result_t:=$source_t[[1]]
   Else
     $result_t:=""
     $continue_b:=False ` Bad first character, terminate.
   End if
End if

 ` Remove any illegal following characters.
If ($continue_b)
   C_LONGINT($index)
   C_LONGINT($ascii)

   For ($index;2;$source_length)
     $ascii:=Ascii($source_t[[$index]])

     C_LONGINT($element)
     $element:=Find in array(array_of_legal_body_character_ascii_codes;$ascii)
     If ($element>0)
       $result_t:=$result_t+$source_t[[$index]]
     Else
       ` Bad character, don't include it.
     End if

   End for

End if
```

There are several pros and cons to using data-driven solutions, like the one illustrated in the pseudo-code above:

Con The code isn't 100% clear unless you've got access to the contents of the arrays.

Pro Since the data is stored rather than embedded (hard-coded), the data can be read by other code for independent testing, printing, display, export, or any other purpose.

Pro If the allowed characters change, only the data (arrays) need to be changed. No new code needs to be added and no new coding bugs can be introduced.

Pro The method is shorter, simpler, and easier to read.

Pro The method requires fewer control structures and is less complex, reducing the chances of making a mistake from mis-nested or illogically combined control structures.

Some programmers might worry about the speed and memory differences of the two different approaches. Given that XML name strings are small, any such speed/memory differences on modern equipment will be meaningless in the real world.

Summary

This technical note reviews the basic rules for naming XML elements, offers a simplified rule-set for 4th Dimension programmers, and provides sample code to clean and test element names. Adding this code, or code like it, to your projects can help avoid problems when producing XML.