DFDL Trimming

### Author: Avery Bibeau

# Explanation

Trimming removes unwanted text or characters while parsing. For example, extra whitespace characters between elements may be unwanted and can be removed during parsing. Trimming is useful for parsing data formatted with unnecessary whitespace. DFDL provides capabilities for text trimming using several properties discussed in this tutorial.

# Problem

This tutorial explores trimming in email headers. The email headers are composed of two major components, the sender’s name and address. The address is made of a local part and a domain name.

The following email header will be deconstructed by the parser and extra whitespace will be parsed

From: “John Doe” <john@doe.com>

In this example, an unknown amount of whitespace exists before the display name and before the address. To remove this whitespace, DFDL properties for trimming will be used.

# Solution

DFDL uses several properties to trim text. Properties defining the character used for trimming for various element types including strings, numbers, Booleans, and calendars can be defined. Additionally, the justification for each of these element types can be defined.

1. The first DFDL property required for trimming is the ‘textTrimKind’, setting this property to ‘padChar’ indicates that trimming is based on the character defined for each element type.
2. To define the ‘padChar’ used for trimming, the properties ‘textStringPadCharacter’, ‘textNumberPadCharacter’, ‘textCalendarPadCharacter, and ‘textBooleanPadCharacter’ must indicate the character used for trimming during parsing and padding during unparsing. These properties are useful for removing excess characters from fixed length data.
	1. DFDL provides character entities to specify a single Unicode character. Character entities are preceded by a ‘%’ character. The character entity for spaces is simply ‘SP;’. This character is used in the ‘textStringPadCharacter’ property to trim spaces.
3. Justification indicates from which side characters should be trimmed. Setting ‘textStringJustification’ to ‘left’ indicates that characters should be trimmed from the right of the element. On the other hand, setting ‘textStringJustification’ to ‘right’ will trim characters from the left of the element.

Consider the following schema and input; by utilizing text trimming, excess ‘0’ characters can be removed by defining the ‘textTrimKind’ property and setting ‘textStringJustification’ to ‘right’ to remove characters from the left:

06:23:09

<xsd:element name="time" >

<xsd:complexType>

<xsd:sequence dfdl:separator=":" dfdl:separatorPosition="infix">

<xsd:element name="hours" type="xsd:string" dfdl:textTrimKind="padChar" dfdl:textStringPadCharacter="0" dfdl:textStringJustification="right"/>

<xsd:element name="minutes" type="xsd:string" dfdl:textTrimKind="padChar" dfdl:textStringPadCharacter="0" dfdl:textStringJustification="right"/>

<xsd:element name="seconds" type="xsd:string" dfdl:textTrimKind="padChar" dfdl:textStringPadCharacter="0" dfdl:textStringJustification="right"/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

The resulting XML infoset looks like this:

|  |  |
| --- | --- |
| Before | After |
| <time> <hours>06</hours> <minutes>23</minutes> <seconds>09</seconds></time> | <time> <hours>6</hours> <minutes>23</minutes> <seconds>9</seconds></time> |

Similarly, trimming may be used for fixed length data such as number and Boolean elements. The following examples demonstrate various ways trimming properties may be used to accomplish a variety of tasks.

|  |  |
| --- | --- |
| Before | After |
| *Justification: left*<number\_stream>54387000</number\_stream>*Justification: right*<number\_stream>00054387</number\_stream>*Justification: left*<condition>True </condition><condition>False</condition>*Justification: right*<condition> True</condition><condition>False</condition>  | <number\_stream>54387</number\_stream><number\_stream>54387</number\_stream><condition>True</condition><condition>False</condition> <condition>True</condition><condition>False</condition>  |

The resulting XML infoset for the email header example looks like this:

<header>

 <display\_name>John Doe</display\_name>

 <address>

 <local\_part>john</local\_part>

 <domain>doe.com</domain>

 </address>

</header>

# Schema

Here’s a complete schema using string trimming:

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:dfdl="http://www.ogf.org/dfdl/dfdl-1.0/">

 <xs:annotation>

 <xs:appinfo source="http://www.ogf.org/dfdl/">

 <dfdl:format separator="" initiator="" terminator="" leadingSkip='0' **textTrimKind="padChar" textStringPadCharacter="%SP;" textStringJustification="left"** lengthUnits="characters" initiatedContent="no"

 alignment="implicit" alignmentUnits="bits" trailingSkip="0" ignoreCase="no" separatorPolicy="suppressed" separatorPosition="infix" sequenceKind="ordered" occursCountKind="parsed" emptyValueDelimiterPolicy="both" representation="text" textNumberRep="standard" lengthKind="delimited" encoding="ASCII"/>

 </xs:appinfo>

 </xs:annotation>

 <xs:element name="header" dfdl:initiator="From:%WSP\*;">

 <xs:complexType>

 <xs:sequence>

 <xs:element name="display\_name" type="xs:string" dfdl:initiator="&quot;" dfdl:terminator="&quot;"/>

 <xs:element name="address" dfdl:initiator="%WSP\*;&lt;" dfdl:terminator="&gt;">

 <xs:complexType>

 <xs:sequence dfdl:separator="@" dfdl:separatorPosition="infix">

 <xs:element name="local\_part" type="xs:string"/>

 <xs:element name="domain" type="xs:string"/>

 </xs:sequence>

 </xs:complexType>

 </xs:element>

 </xs:sequence>

 </xs:complexType>

 </xs:element>

</xs:schema>