# The XmlDataSamples Package

All of the necessary material to run the samples is contained in this package.

The examples in this guide were developed and executed using Integration Server release 10.1. Partial implementations of the feature are available in earlier releases starting with release 9.9. This partial support is described in a chapter at the end of this guide. Some portions of the early support are incompatible with the full support in release 10.1. It is preferable to use the Java helpers in release 10.1 or higher.

The home page for the package contains a download link to this document, and also contains a link to the JavaDoc for the XmlData classes.

# Introduction to XmlData

The XmlData feature allows XML instance documents and XML schemas to be represented and manipulated using IData Documents and IS Document Types.

## The Legacy Approach

There has always been a limited ability to represent the XML infosets using IData and to represent XML schemas using Document Types. There is a wizard that converts XML schemas to Document Types and there is an informal set of conventions for representing the XML info using IData. There is a service that converts the XMLNode infosets into an IData structure using these conventions. In summary:

* Complex content is represented as an IData bag of key/value pairs. The IData key corresponds to an XML tag, and the IData value corresponds to the XML tag’s content. Schema complex content is modeled as an IS Document Type.
* Simple content is represented as a String value in an IData key/value pair. Simple content is modeled as a String field in a Document Type
* Repeating values for the same tag (maxOccurs > 1) are represented as arrays in IData instances. Repeating content is modeled a single field with values aggregated as a list in a Document Type. If the same tag is repeated in more than one place in a content model, all of the repeating occurrences are aggregated into a single list (unless the strict option is used).
* Complex types with attributes are represented as an IData with attribute tag names prefixed with @. Complex content member tags are peers with the attribute tags.
* Simple content with attributes is represented as an IData instance with the reserved \*body tag holding the simple value. The \*body key/value pair is a peer with the attribute pairs. It is modeled as a Document Type with a \*body String field and zero or more @-prefixed Strings fields.
* Namespace qualification is specified by prefixes that are defined externally from the instance data. Namespaces are defined either by an nsdecsl table, or a Document Type. It is the user’s responsibility to provide the proper Document Type or nsdecsl in order to map namespaces correctly. The table or Doctypes are used during conversion to and from the IData infoset.
* The XMLNode to IData conversion service assigns prefixes to tags based on the external namespace definition. (Because the namespaces are defined externally, the prefixes used in the converted XML may not match the prefixes in the input XML, though the converted IData will be logically equivalent).
* When creating Document Types from schemas, global complex types are created as distinct Document Types; complex type references are represented as Document Type references.
* There are no global element declarations.

The legacy approach has limitations:

* repeating and nested model groups cannot be correctly represented
* the relative ordering of repeating tag content is lost.
* Singly nested sequence, choice, and all groups are not properly represented unless the ‘strict’ option is selected during generation. Even when selected, repeating and nested models are still not properly represented.
* Depending on the situation, substitution groups, any wildcards, and anyType content may be either difficult or impossible to handle.
* The separation of namespace declaration from instance data can lead to very complex or impossible namespace handling. This is especially difficult when dealing with arbitrary payloads and content that is only partially extracted from an instance.
* The processContents option for ‘any’ wildcards cannot be handled.

## The XmlData Approach

XmlData expands on the legacy approach described above in the following ways:

* Fully qualified names are used in the instance data (in the form localName#namespaceURI).
* There is no external definition of the namespace, allowing preservation of namespace declaration in a wide variety of processing situations.
* Namespace declarations and xsi properties are maintained in the IData parent of the tag to which they apply. This makes it possible to preserve namespace declarations and instance properties without promoting a simple type to complex content IData representation.
* Model groups are preserved as group-tags with IData document content. This makes it possible to access and manipulate model groups from within a Flow Service. This is quite different from typical XML processors which “lose” the association between model groups and content after the XML infoset has been processed.
* Complex types with simple content are represented as a Document Type with a \*simple field for the simple content
* Attributes are maintained separately from content in an attribute-set. The attribute-set is an IData document paired with a \*attributes field that is a peer to either the group-tag or the \*simple field, or appears alone if the complexType only has attributes.
* A specialized structure is used to represent the “any” wildcards making it possible to access and manipulate wildcard name and value from a Flow service
* A specialized structure is use to represent substitution groups making it possible to access and manipulate substitution groups from within a Flow service.
* XmlData instances never contain the Java null. They consist exclusively of Strings, String[]s, IData, IData[]s, and Object[]s.
* It is possible to construct repeating content out-or-order. In other words, it is possible to set the value of the second instance of a tag before setting the value of the first instance, without the need to reorder the tags. This is possible because the particle-id uniquely identifies the particle reference.
* When Document-type directed conversion is used:
  + Elements with simple type are represented as a String
  + Repeating elements with simple type are represented as String[]
  + Elements with complex types are represented as an IData
  + Repeating elements with complex content are represented as IData[]
  + Specialized IData structures are used to represent any wildcards and substitution group heads that make it possible to reference element names and values from Flow
* When conversion is not Document Type directed:
  + Elements with simple types are represented as String
  + Elements with complex values are represented as IData
  + Repeating elements are represented as Object[]s containing String, IData, or both.
  + There are no groups, wildcards, or substitution groups because nothing in the input XML defines those constructs; they are defined by schema. In the absence of a Document type derived from a schema, the conversion processes uses the most primitive form of representation.

The net effect of these features results in a more fully functional XML infosets and schema representation. Virtually any construct that can be represented using XML and XML schemas can be processed using XmlData.

# XmlData Terminology

XmlData infosets are represented as IData instance containing tag-object-pairs. Again this is very similar to the legacy approach, though the legacy approach typically uses the term “key/value” pair.

XmlData has essentially four types of values:

* simple-value – a String; they represent schema simpleTypes. Empty simple-values are represented by an empty String, never by a Java null. Simple values are also used to represent attribute values.
* group-value – an IData containing tag-value-pairs; they are used to represent model groups or complex types that have no attributes or repeating top-level group. Empty group-values are represented by empty IData, never by a Java null. The tags used in group-values are a particular type of tag called content-tags.
* complex-value – an IData with a specialized structure; complex-values are used to represent schema complexTypes with attributes, and/or complexTypes that have a repeating top-level group. When a complexType has attributes, the complex-value has a \*attribute field paired with an IData that contains tag-object-pairs of attributes and their values. When a complexType has a top-level group, the complex-value has a \*group child than contains the complex content consisting of tag-object-pairs. The \*group value is therefore either an IData, or an IData array if the top-level group has maxOccurs > 1. When a schema complexType has only a non-repeating top-level group with no attributes, it is represented as group-value rather than a complex-value.
* indirect-item – an IData with a specialized structure; it is used as place-holders for any wildcards and substitution groups. Place-holders are necessary because the tag name and value type are not known at the time of Document Type generation. They become known at run-time. There are two kinds of indirect-items:
  + any-items – a specialized IData structure that has three child fields: ncName, namespace, and value. They allow a Flow to access the name and value of the XML instance element that corresponds to an “any” wildcard particle.
  + substitution-item – a specialized IData structure that contains a single tag-object-pair that corresponds to the XML instance element that appears as a replacement for the schema substitutionGroup head element.

XmlData has five types of tags. Each tag-type has a specific usage.

* element-tag – used to name an element. The String representation of an element tag is **{***particle-id***}***locallName***#***namespaceURI*. That seems rather cumbersome, but in practice the application never deals directly with the full name. Typically Flow uses map operations to reference named fields; the Java helper classes have constructors that simplify dealing with such cumbersome names. The components of the element-tag are:
  + *localName* – the unqualified portion of an XML instance tag
  + *namespaceURI* – full URI of the namespace
  + *particle-id* – an incrementing integer (starting at 1) that identifies the relative order of particle (i.e. the first appearance of a particle named “employee” in model is {1}employee, the second is {2}employee, etc.)
* group-tag – used to name a model group as it appears in the instance. The String representation of a group-tag is **{***particle-id***}\*group**.
* any-tag – used to name an any wildcard place-holder. The String representation of an any-tag is **{***particle-id***)\*any**
* substitution-tag – used to name the head of a substitution group. The String representation of a substitution-tag is **{***particle-id***}\*substitution-***locallName***#***namespaceURI*. The components of the substitution-tag are:
  + *localName* – the unqualified portion of an XML instance tag
  + *namespaceURI* – full URI of the namespace
  + *particle-id* – an incrementing integer (starting at 1) that identifies the relative order of particle (i.e. the first appearance of a particle named “\*substitution-employee” in model is {1}\*substitution-employee, the second is {2}\*substitution-employee, etc.)
* attribute-tag – used to name an attribute. The String representation of an attribute-tag is*loclalName***#***namespaceURI*. Attributes have no particle id because they are unique within an element and no explicit ordering.

There is a common super type for element-tag, group-tag, substitution-tags, and any-tags; it is the content-tag. The common behavior shared by all content-tags is that they have a particle Id and can be used as a tag in a group-value. This is in contrast to an attribute tag which has no particle id and can only be used in attribute tag-object-pairs.

There are actually two additional types of tags, but they are not described here because they never appear in instance data. They are the type-tag and the iterator-tag; they are described in a later section.

As might be expected, only certain pairings of tag and value are permitted.

* Element-tags may be paired with simple-values, group-values, and complex-values, or arrays of any of those value types.
* Group-tags may be paired only with group-values or arrays of group values.
* Any-tags may be paired with any-items or arrays of any-items
* Substitution-tags may be paired only with substitution-items or arrays of substitution-items.
* Attributes-tags may only appear within the attribute-set of a complex value as tag-object-pairs that represent an attribute and its value.

The definition of tags, values, and the permitted tag-value pairings may seem a bit overwhelming, but in practice, most schemas use only model groups and the typical XmlData document consists only of element-tags paired with various values (simple, group, and complex) and group-tags paired with group values.

**The user is never expected to explicitly create all of the structure described above. For the Flow user, the structures are explicitly defined by the XmlData Document Types created from schemas. For the Java user, the structures are created and updated using the XmlData helper classes.**

# Overview of XmlData Usage

The overall usage patterns for legacy IData XML info and XmlData infoset are quite similar, as illustrated in this table:

|  |  |  |
| --- | --- | --- |
|  | **XmlData** | **Legacy IData XML infoset** |
| Creating the infoset | pub.xmldata.domNodeToXMLData  pub.xmldata.queryXMLNode  XmlData (helper for Java service) | pub.xml.XMLNodeToDocument  pub.xml.queryXMLNode |
| Manipulating from Flow | Flow Map | Flow Map |
| Converting to String | pub.xmldata.xmDataToXMLString | pub.xml.documentToXMLString |
| Manipulating from Java | IData (base structure)  XmlDataCursor (navigation)  XmlData (construction)  **Advanced features:**  XmlDataTreeCursor  XmlDataMap  XmlDataCopy | IData (base structure)  IDataCursor (navigation) |

There are two ways to manipulate XmlData after it has been created: Flow Services, and Java Services.

Flow Services access and manipulate using Document Types generated from schemas, and the Flow Map operations such as copy, set, and drop. This is exactly the same approach used with the legacy IData XML infoset.

Java Service access is similar but there are formally defined APIs. These APIs include:

* XmlDataCursor – very similar to IDataCursor, but the API is formalized to support attributes, substitution groups, any wildcards, repeating and nested model groups, and namespace-qualified tags.
* XmlData – a helper class that assists in creating XmlData structures outside of the converter; that is, XmlData infoset that is created in an ad hoc fashion rather than from XML
* XmlDataTreeCursor – an extension of XmlDataCursor that facilitates navigation through multiple levels in an XmlData infoset.
* XmlDataMap – a helper class that supports retrieval and setting of an XmlData infoset using path expressions (rather than cursor navigation). It provides some of the same capabilities provided by Flow Map.
* XmlDataCopy – a helper that facilitates the copying parts of an XmlData infoset to another XmlData infoset using XmlDataMap specifications in conjunction with wild-cards.

# A simple example

This simple example focuses on the mechanics of using XmlData. The actual results for this simple sample are no different than those obtained using the legacy approach to IData-based XML infosets handling. As we explore more advanced features, difference between the approaches will emerge.

The following schema must first be converted to XmlData Document Types.

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="employee" type="personInfo" />

<xsd:complexType name="personInfo">

<xsd:sequence>

<xsd:element name="fullName" type="xsd:string" />

<xsd:element name="address" type="addressInfo" />

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="addressInfo" >

<xsd:sequence>

<xsd:element name="street" type="xsd:string" />

<xsd:element name="city" type="xsd:string" />

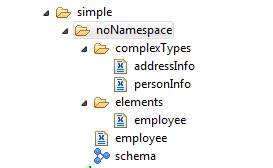
<xsd:element name="state" type="xsd:string" />

</xsd:sequence>

</xsd:complexType>

</xsd:schema>

This was already accomplished in the sample package using the Designer by selecting **File -> New -> XML Document Type** and then selecting the **xmldatasamples.simple** folder as the destination and then selecting the **File/URL** option and specifying the location of the schema file. The schema file is in the sample package in the pub folder and has a name of the form **<*server-directory*>/packages/WmXmlDataSamples/pub/samples/simple/schema.xsd.** Then **finish** was selected. Document Types reflecting the schema structure were created.



A separate sub-folder is created for each XML namespace. The input XSD file did not specify a namespace, so the items were created under the **noNamespace** sub-folder. Notice that a separate Document Type is created for each named complex type (in this case there are two) in the **complexTypes** sub-folder. Also notice that a Document Type is generated for each global element (in this case there is one) in the **elements** folder. In addition, a Document Type wrapper was created for each global element and has the same name as the global element that it wraps (though it is immediately under the namespace folder). In this case, it is the **elements** Document Typein the **noNamespace** folder**.** The wrapper document types are used to map the top-level IData instance that contains the root element (the document element) of a complete XML document (a well-formed XML document can have only one top-level element).

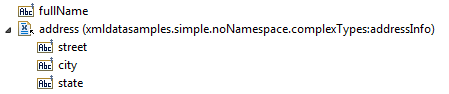
All of Document Types and Schemas created by an invocation of the wizard are permanently related to one another. If it is ever necessary to resolve a reference in one component by QNames to another component, it will always resolve to another component in the group of components that were created together (this is sometime called a Document Type Domain). While this is typically not needed for simple situations, the need for a domain becomes apparent when processing “anyType”s and “any” wildcards. The need is also apparent when applying type restrictions via the xsi:type constraints.

If we look at the generated **addressInfo** Document Type in the **complexType** folder we see the following:



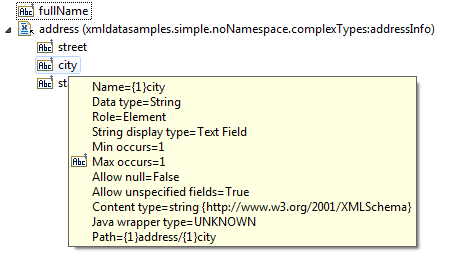
There is a String field for each of the local elements defined in the complex type.

If we look at the **employee** Document Type in the **element** folder we see the following:



The **employee** element is defined as a type reference to **personInfo** so we see the content of **personInfo** displayed. **personInfo** has a String field for simple type **fullName** field, and a Document Type Reference to the **addressInfo** type.

One very important feature of XmlData is that if fully supports multiple occurrences of the same tag within a content model. The legacy approach aggregated repeating tags into a single tag with an array of repeating values. The XmlData feature creates a separate tag corresponding to each element in a schema content model. In order to maintain uniqueness, each tag has a particle ID prefix that identifies its position relative to other tags of the same name. If we place cursor over a field in a Document Type, design displays the full name which includes the particle Id. If we look at the properties of city, we see that the first (in this case only) occurrence of city is actual named {1}city.



Once we have Document Types, we’ll want to use them in converting an XML instance into an XmlData-structured IData object.

The steps for creating the XmlData instance are illustrated in the Flow service **simple:convertInput**.

In the first step simple authentication is defined so that the sample XML file can be accessed.

Then we load the XML instance into the pipeline with **pub.xml.loadXMLNode** setting the **url** parameter to <http://localhost:5555/WmXmlDataSamples/samples/simple/input.xml>and setting **isXML** to true. This file contains:

<employee>

<name>John Smith</name>

<address>

<street>100 Main</street>

<city>Friendly</city>

<state>Confusion</state>

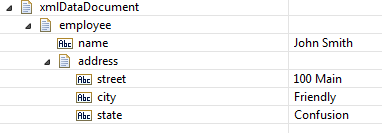
</address>

</employee>

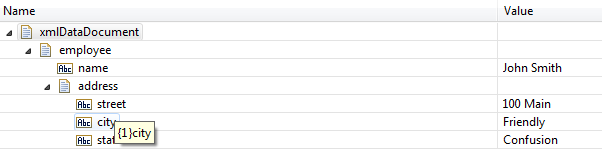
input.xml

The resultant XMLNode contained in pipeline variable **node** can then be converted to XmlData by using the **pub.xmldata.domToXMLData** service. We map the output **node** of loadXMLNode to the input of domToXMLData. We set the **conformsTo** parameter to the wrapper for the employee element **simple.noNamespace:employee** (though any Document Type from the generation step would work because the converter will find the appropriate Document Type in the domain).

After executing the service, the pipeline variable **xmlDataDocument** contains:

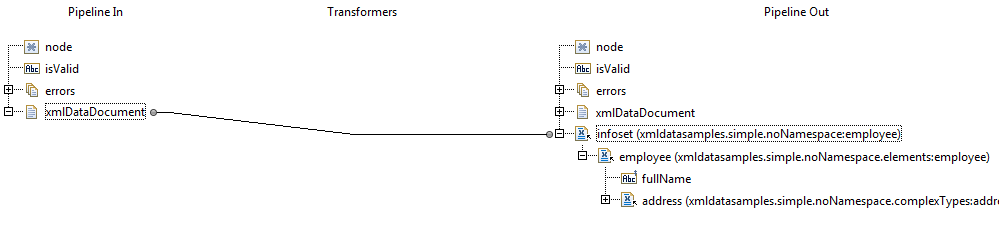


As with the Pipeline editor, if the cursor is placed on an XmlData field in the results panel, the actual tag name is displayed.



## Accessing from Flow

The next step in **simple:convertInput** maps the content of **xmlDataDocument** to a field (for example **infoset**) that is defined as reference to the wrapper for **employee**, we’ll have full Map access to the content.



We may get and set the various value parts of the infoset using Flow Map copy, set, and drop operations.

## Accessing from Java

The classes for XmlData processing are all contained in Java package com.wm.xmldata.

The last step in **simple:convertInput** invokes the Java service **simple:accessValues** which illustrates how XmlData instance are accessed from Java.

public static final void accessValues(IData pipeline) throws ServiceException {

// Retrieve the XmlData output from the conversion service

IData input = (IData) ValuesEmulator.get(pipeline, "xmlDataDocument");

// Obtain an XmlDataCursor

XmlDataCursor xdc = XmlData.getXmlDataCursor(input);

// Position to the first only element, "employee"

xdc.first();

// Retrieve its value which is an employee Document

IData empValue = (IData) xdc.getValue();

// obtain a cursor on the employee Document

XmlDataCursor xdc2 = XmlData.getXmlDataCursor(empValue);

// Position to the tag "name"

xdc2.first(new XmlDataElementTag("name"));

// Retrieve the String value

String nameValue = (String) xdc2.getValue();

// Put the retrieved name value into the pipeline

ValuesEmulator.put(pipeline, "result", nameValue);

// Release resources

xdc.destroy();

xdc2.destroy();

}

The overall flow of the program is very similar to that used with the legacy approach.

We obtain the input value from pipeline variable “xmlDataDocument” which must be an IData object is to cast to that class type. Next we obtain an XmlDataCursor on the input. Note that the getXmlDataCursor() method is a static method of the XmlData class and not an instance method of the IData class. That is because XmlData is not a new or extended implementation of IData, it is a structure of traditional IData objects. As with the legacy approach, the .first() method positions the cursor on the first item, and the .getValue() method returns the value which is stored there. The value stored must be another IData because this is the root element of the infoset, and it is cast to a class of that type.

Next we obtain another cursor on the value stored at the root, i.e. (the value stored at the “employee” node. We position to the first element named “name.” At this point, things look at bit different from the legacy approach. In the legacy approach we would simple invoke .first(“name”) to position to the element with the name. In XmlData, the tags are a bit more complex than simple Strings, and there are a variety of tags based on their usage. In the this case we want an element named “name” so we create an XmlDataElementTag using “name” in its constructor. This tag is then passed to the .first() method. This may seem a bit overdone for something as simple as positioning on an element with a particular name, but the need for this extra specificity will become apparent later.

After positioning to the “name” element, we retrieve its value and copy it into the pipeline variable named “result.’

Finally, we release any resources associated with the cursors that we obtained.

# Attributes

In this example, we investigate attributes.

The schema used in the prior example is expanded slightly to include a “**category**” attribute for the **addressInfo** complex type. The expanded type definition is shown below.

<xsd:complexType name="addressInfo" >

<xsd:sequence>

<xsd:element name="street" type="xsd:string" />

<xsd:element name="city" type="xsd:string" />

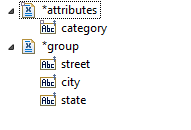
<xsd:element name="state" type="xsd:string" />

</xsd:sequence>

<xsd:attribute name="category" type="xsd:string" />

</xsd:complexType>

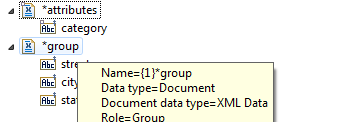
When we create the XmlData Document types from this new schema, all of the other components remain unchanged, but the **addressInfo** Document Type in **complexTypes** has a new structure:



The complex type is now comprised of two child Document Types: \*attributes and \*group. The \*attributes Document Type contains all of attribute definitions associated with this type The \*group Document Type contains the complex content model for the type.

In general, a complexType is modeled using the \*group Document Type only when attributes are present, OR when the top-level group in a complexType has maxOccurs > 1.

The \*group tags also have a particle ID. If we look at the properties of the \*group tag we see that its name is actually {1}\*group.



Later, we will see that all tags associated with complex content have a particle ID so that it can be uniquely identified. In fact, there are four types of complex content tag. We have already seen two: element tags and \*group tags. Later, we will also see \*any and \*substitution tags.

When we look at the results of running **xmldatasamples.attributes:convertInput** we find:



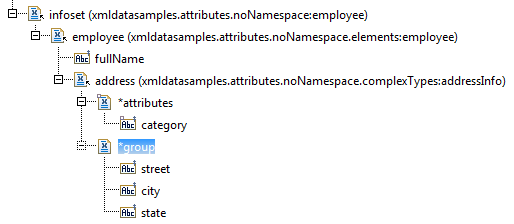
Similarly, when examining the \*group Document in the results panel, we see that the tag has a particle ID prefix.



The returned values of \*group are structured to match the Document Type for **addressInfo**.

## Accessing from Flow

If we map the content of **xmlDataDocument** to a field (for example **infoset**) that is defined as reference to the wrapper for **employee**, we’ll have full Map access to the content. This is illustrated in the fourth step of **attributes:convertInput**.



## Accessing from Java

If we modify the Java program from the previous example, we replace the fetching of “fullName” from address with the fetching of the category attribute value. This is illustrated in the Java service **accessValues** and it is invoked in the last step of **convertInput**.

// Position to the tag "address"

xdc2.first(new XmlDataElementTag("address"));

// Retrieve the value of the “category” attribute

String attrValue = (String) xdc2.getAttributeValue(new XmlDataAttributeTag("category"));

// Put the retrieved name value into the pipeline

ValuesEmulator.put(pipeline, "result", attrValue)

The big difference with the legacy approach is that attributes are not stored as peers of the other elements in the complex type. Additionally, rather than navigating to an attribute, there is a cursor method, getAttributeValue() that is used to retrieve the attribute value. As with elements, attributes must be named using a specific tag type; in this case it is an XmlDataAttributeTag.

When we run the **attributes:convertInputservice**, we note that the results panel shows a pipeline variable named “result” with the value “home” from the “category” attribute

# Repeating fields

With XML schema it is possible to specify the number of times an element or other particle may be repeated (adjacently). This is achieved with the minOccurs and maxOccurs property. By specifying minOccurs=0 the element becomes optional. When maxOccurs is greater than 1, the corresponding field in the Document Type is represented as List and the instance data is represented as a Java array of that type.

For this example, we’ll modify out first schema to include zero or more **project** elements (up to a maximum of five) in the **personInfo** complex type. This is shown below.

<xsd:complexType name="personInfo">

<xsd:sequence>

<xsd:element name="fullName" type="xsd:string" />

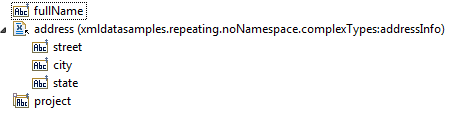
<xsd:element name="address" type="addressInfo" />

<xsd:element name="project" type="xsd:string" minOccurs="0" maxOccurs="5" />

</xsd:sequence>

</xsd:complexType>

When the XmlData Document Type for personInfo is created, it now has a field named **project** which is represented as a String list.



When we provide the following XML input

<employee>

<name>John Smith</name>

<address>

<street>100 Main</street>

<city>Friendly</city>

<state>Confusion</state>

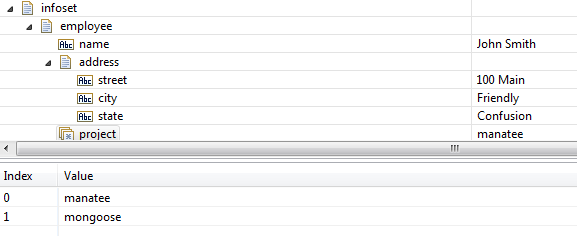
</address>

<project>manatee</project>

<project>mongoose</project>

</employee>

In the instance data, we can see that project is a String List.



# Model Groups

Model groups permit robust combinations of complex content. All of the complex content so far has been of type sequence. However, two other types are possible: choice, and all. Additionally, some model types cam be nested.

In this example we’ll further modify the **personInfo** complex type so that it includes a choice group. This choice group states that one of the specified elements must be present: **server**, **laptop**, or **tablet**.

<xsd:complexType name="personInfo">

<xsd:sequence>

<xsd:element name="fullName" type="xsd:string" />

<xsd:element name="address" type="addressInfo" />

<xsd:element name="project" type="xsd:string" minOccurs="0" maxOccurs="5" />

<xsd:choice>

<xsd:element name="server" type="xsd:string" />

<xsd:element name="laptop" type="xsd:string" />

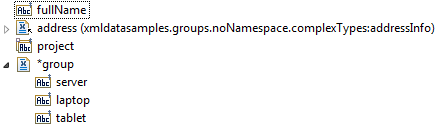
<xsd:element name="tablet" type="xsd:string" />

</xsd:choice>

</xsd:sequence>

</xsd:complexType>

When Document Types are created we see the inclusion of a nested Document Type named \*group that specifies the list of alternate elements.



We see the results below. The \*group Document contains one member, that member has the tag name and value of the choice group as present in the XML instance.



## Accessing with Flow

Since only one value can be present for the choice group, the Flow user can determine which is present by conditionally mapping on the individual fields in the Document Type. Typically, this test is performed using a Branch operation with evaluate-labels set to True. If a child operation of the Branch operation must have a label that matches the pipeline variable that is being tested. In this case the branch is specified as:



The full tag name must be used; in this sample it is infoset/{1}employee/{1}\*group/{1}/laptop. The pipeline variable **infoset** contains the document (note that top-level pipeline variable are never qualified with a particle Id). Within **infoset** the first **employee** is selected, within that the first \***group** is selected, and then within that the first **laptop** field is selected. Whenever working with a tag name in its liberal form, it must always use the full name syntax (it is only the Document Type editor, the Pipeline editor, and the Results panel that suppresses the particle Id in order to have a less-cluttered display. In this example, the pipeline variable **isLaptop** is set to true when the specified pipeline variable exists. In later example, when working with namespace-qualified tags, this literal approach can become quite cumbersome. An alternate method of perform this test will be described in those examples.

## Accessing with Java

When a Java program executes the getValue() method on the \*group field, it will return an IData. The program may then obtain an XmlDataCursor on the value, then invoke the first() method, and then invoke getTag() to discover which tag was present in the instance.

# Namespace declaration

Most XML documents employ XML namespace qualification. All of our previous examples have avoided namespace qualification in order to concentrate on the basic features of XmlData. Now that we have established that foundation, we can begin to explore more advanced features such as namespace qualification.

Let’s start with schema.xsd in the pub/sample/namespaces directory.

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="samples.xml.org"

xmlns="samples.xml.org" >

<xsd:element name="employee" type="personInfo" />

<xsd:complexType name="personInfo">

<xsd:sequence>

<xsd:element name="fullName" type="xsd:string" form="qualified" />

<xsd:element name="address" type="addressInfo" />

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="addressInfo" >

<xsd:sequence>

<xsd:element name="street" type="xsd:string" />

<xsd:element name="city" type="xsd:string" />

<xsd:element name="state" type="xsd:string" />

</xsd:sequence>

<xsd:attribute name="category" type="xsd:string" />

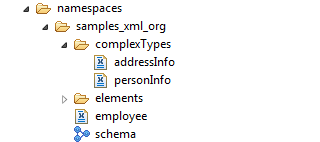
<xsd:attribute name="version" type="xsd:string" form="qualified" />

</xsd:complexType>

</xsd:schema>

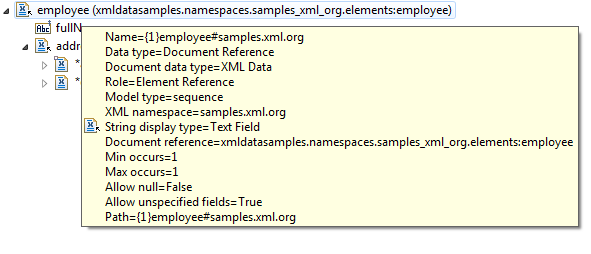
This schema is very similar to the schema in pub/sample/attributes, but with some important highlighted differences. The most important difference is the **targetNamespace** attribute on the **schema** element. This attribute declares that the schema describes content for a particular XML namespace (all proper examples have had no namespace association). When a schema is tied to a particular namespace, it means that all global elements are namespace-qualified in a conforming XML instance. In our sample schema, there is only one global element, employee. It must now be namespace-qualified when appearing in an XML instance. In addition, by using the targetNamespace as the XML default namespace in this schema document, we can reference global types and named complex types without using a namespace prefix.

When we create XmlData Document Types from this schema, we observe the following folder structure:



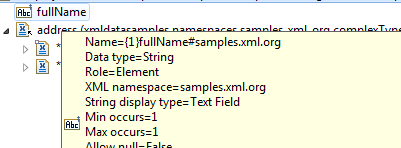
The structure is very similar to the attributes sample, but notice that the folder containing the Document Types and schemas is named **samples\_xml\_org**, the target namespace of the input schema, rather than **noNamespace**. All namespace objects related to a particular namespace are collected together into a common folder. In this sample, there is only one namespace, so there is only one folder. Also notice that characters that cannot be used in a Integration Server name are translate into an underscore character. Internally, the Integration Server retains the original XML namespace **samples.xml.org**.

If we open the wrapper Document Type **namespaces.samples\_xml\_org:employee** we see one small change.



When we hover over the employee field, we see that its full name is **{1}employee#samples.xml.org**. This is the fully qualified name for the first instance of **employee** in the **samples.xml.org** namespace.

Another change from the attributes example is the form=“qualified” attribute on the declaration of the **fullName** element. This asserts that even though fullName is not a global element, it must still be namespace qualified in an instance. If we look at the fullName field, we see that it also has a qualified name.



When we look at sample XML input that conforms to this schema we notice a few changes here as well.

<sam:employee xmlns:sam="samples.xml.org">

<sam:fullName>John Smith</sam:fullName>

<address category="home" sam: version="1.0">

<street>100 Main</street>

<city>Friendly</city>

<state>Confusion</state>

</address>

</sam:employee>

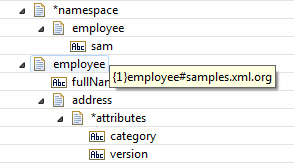
Because some elements must be namespace qualified, the XML instance contains a namespace declaration xmlns:sam="samples.xml.org”. This asserts that all tags belonging to that namespace are prefixed with “sam”. **Employee** is a global element, so it is prefixed with sam (both the start and end tags). **FullName** is a local attribute but it was declared with form=”qualified” so it too is prefixed. Finally, we added a new attribute **version** which is also declared with form=”qualified” so it is prefixed in the XML instance.

When we execute **namespaces:convertInput** we see the resultant XmlData Document. It is similar to the Document obtained in the **attributes** sample, but there are differences.

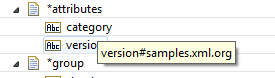


Notice that a new tag-object-pair has been added to the XmlData instance in xmlDataDocument. It has a reserved tag **\*namespace**. It contains a single IData instance with a single tag-value pair sam=”smaples.xml.org”. This structures indicates all elements that are peers or this **\*namespace** element have an associated namespace declaration of xmlns:sam=”smaples.xml.org”. It also applies to any children or attributes of the elements. In this case, \*namespace has only one peer, **employee**. If there had been other elements present, the declaration would apply to them as well. Note that the prefix sam will not appear in the tag names, tag names always consist of the local name followed by the # symbol and the actual namespace. The prefix is only used when converting XmlData Documents back to XML.

Also notice that the namespace qualification does not appear in the Results panel except when hovering over the field name.



This is true for namespace qualified attributes as well. Notice that attributes do not have a particle ID because they are order-independent and can occur only once.



It may seem a bit strange to store namespace declarations in the parent IData rather than in the item to which it applies, but the item to which it applies may not always be an IData. If a namespace declaration appears on a simple type, the XmlData instance contains a String value. There is no place to store the namespace declaration, so it is actually saved in the parent IData. When we look at XML instance properties (e.g. xsi:type) we will see a similar strategy employed.

## Namespaces with varying content

The fully qualified name can be rather awkward to work with in certain situations. Let’s slightly modify the schema used in samples/groups and retain it as pub/samples/namespaces/schema2.xsd

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="samples.xml.org/2"

xmlns="samples.xml.org/2" >

<xsd:element name="employee" type="personInfo" />

<xsd:complexType name="personInfo">

<xsd:sequence>

<xsd:element name="fullName" type="xsd:string" />

<xsd:element name="address" type="addressInfo" />

<xsd:choice>

<xsd:element name="server" type="xsd:string" form="qualified" />

<xsd:element name="laptop" type="xsd:string" form="qualified" />

<xsd:element name="tablet" type="xsd:string" form="qualified" />

</xsd:choice>

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="addressInfo" >

<xsd:sequence>

<xsd:element name="street" type="xsd:string" />

<xsd:element name="city" type="xsd:string" />

<xsd:element name="state" type="xsd:string" />

</xsd:sequence>

</xsd:complexType>

</xsd:schema>

Note that a different XML namespace is used so that items don’t conflict with the earlier sample in namespaces. In addition, the form=”qualified” attribute has been added to the members of the choice group.

The newly created XmlData items are created under folder namespace/sampes\_xml\_org\_2. That name is created by translating the special characters in samples.xml.org/2 to underscore.

### Accessing from Flow

When input2.xml is processed using Flow **namespaces:comvertInput2** we get the following result:



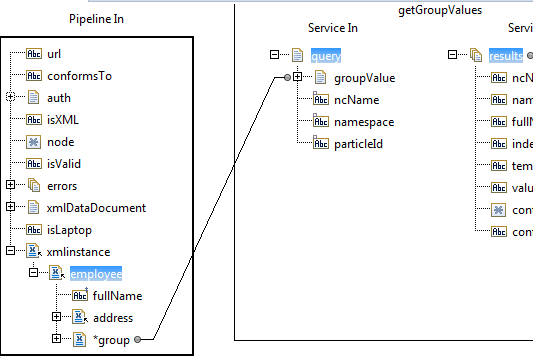
This output is similar to the output created for the groups sample, but there are two changes. There is the namespace declaration, and the laptop tag now has an associated namespace.

Any one of three tags may be present in the choice model of the \*group field. Determining which is present might be accomplished using the typical method of evaluating labels in a Flow Branch Operation.



Notice the particularly awkward expression for the path. It must include the fully qualifying namespace, and because the namespace includes the “/” character, any step with a qualified name must be enclosed in quotes. There is a helper service **pub.xmldata:getGroupValues** that can be used to more easily access the member of a choice group or substitution group. The principle argument, **groupValue**, is the IData containing the choice group or substitution group content. The primary outputs, ncName, and namespace contain the ncName, XML namespace, and value of the values present in the input IData.

In the following example, we map the content the choice group (\*group) to the groupValue field of the **getGroupValues** service. After execution, the results list contains a document for each value that describes the various properties of the values.



The results are show below. Notice that ncName and namespace specify the tag actually present in the choice instance, and value specifies the values associated with the tag. It is much easier to construct a Branch step using one or more of the properties returned by **getGroupValues**.



### Accessing from Java

When the accessValues Java service is executed, it positions to the first child of the choice group and invoke getValue() and getTag() and returns them in pipeline variables **resultValue** and **resultName**.



# The any Wildcard

XML Schema has the ability to represent content models using a wildcard. In particular, a wildcard is expressed using the schema **any** particle. The **any** particle matches content regardless of its tag. There are also options on the any particle that allows matching to be limited to selected XML namespaces.

Let’s start with a modified version of the schema used for the namespace sample.

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="samples.xml.org"

xmlns="samples.xml.org" >

<xsd:import schemaLocation="hobbies.xsd" namespace="localhost/hobbies" />

<xsd:element name="employee" type="personInfo" />

<xsd:complexType name="personInfo">

<xsd:sequence>

<xsd:element name="fullName" type="xsd:string" form="qualified" />

<xsd:element name="address" type="addressInfo" />

<xsd:any maxOccurs="10" namespace="localhost/hobbies" />

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="addressInfo" >

<xsd:sequence>

<xsd:element name="street" type="xsd:string" />

<xsd:element name="city" type="xsd:string" />

<xsd:element name="state" type="xsd:string" />

</xsd:sequence>

<xsd:attribute name="category" type="xsd:string" />

<xsd:attribute name="version" type="xsd:string" form="qualified" />

</xsd:complexType>

</xsd:schema>

Notice the **import** directive. It is used to reference definitions from a different XML namespace. It references a second XSD file. Also notice the addition of the **any** particle to the content model for **personInfo**. The particle specifies that up to 10 tags from namespace **localhost/hobbies** may appear at the end of the content. Looking at the imported schema we see:

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="localhost/hobbies"

xmlns="localhost/hobbies" >

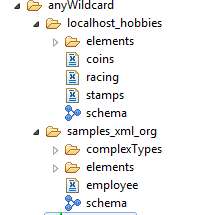
<xsd:element name="coins" type="xsd:string" />

<xsd:element name="stamps" type="xsd:string" />

<xsd:element name="racing" type="xsd:string" />

</xsd:schema>

When we generate Document Types and schemas using the above schemas we notice something new; we get two sets of Document types and schemas, a separate folder for each namespace.



When we load the following XML instance document

<sam:employee xmlns:sam="samples.xml.org" xmlns:hob="localhost/hobbies">

<sam:fullName>John Smith</sam:fullName>

<address category="home" sam:version="1.0">

<street>100 Main</street>

<city>Friendly</city>

<state>Confusion</state>

</address>

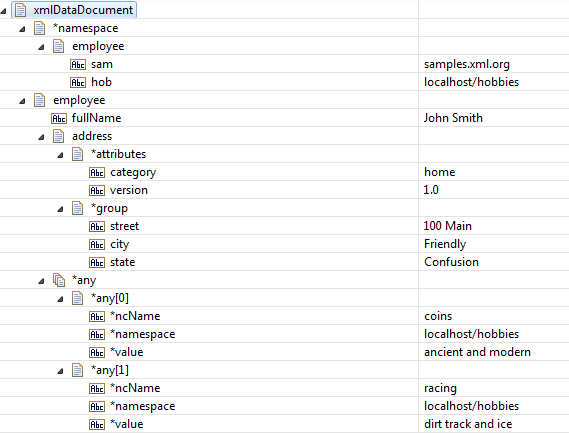
<hob:coins>ancient and modern</hob:coins>

<hob:racing>dirt track and ice</hob:racing>

</sam:employee>

## Accessing from Flow

We obtain the following pipeline results:



The tag and content corresponding to the any particle is represented as Documents with three key/values corresponding to the ncName, namespace, and value at that position.

## Accessing from Java

Accessing “any” content from Java is very simple. Simply position the cursor on the desired any tag and invoke getValue() to retrieve the value, of getValueTag() to retrieve the tag name. The getValue() and getValueTag() methods automatically determines whether a normal element, an any particle, or a substitution group is present and performs the necessary dereferencing to obtain the effective value of the tag. The sample service anyWildcard:accessValue illustrates how to obtain the value and tag for a particular instance of a repeating “any” particle.

# anyAttribute

XML Schema supports attribute wildcards as well as element wildcards. When Document Types are created from schemas, no explicit mapping structure is created for the attribute wildcard. That is, there is nothing that the user maps into or out of because there is content model for the arrangement of attributes (the order of attributes is undefined and duplicate attributes are not permitted).

When generating Document Types from the following schema:

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="employee" type="personInfo" />

<xsd:complexType name="personInfo">

<xsd:sequence>

<xsd:element name="fullName" type="xsd:string" />

<xsd:element name="address" type="addressInfo" />

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="addressInfo" >

<xsd:sequence>

<xsd:element name="street" type="xsd:string" />

<xsd:element name="city" type="xsd:string" />

<xsd:element name="state" type="xsd:string" />

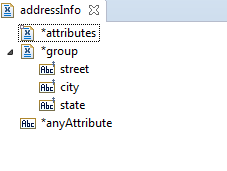
</xsd:sequence>

<xsd:anyAttribute />

</xsd:complexType>

</xsd:schema>

Notice that noNamespace.complexTypesaddressIfno has the following structure:



It is similar to the structure used when a complex type has declared attributes, but the any attribute wildcard is represented as a single field named \*anyAttribute. This denotes that the complex type may have attributes from any namespace. Also notice that there is a \*attributes parent even though the model has no declared attributes. That is necessary because at run time, the attribute/values are collected under the \*attributes parent just as when there are declared attributes. The user does not map into or out of the \*anyAttributes field because multiple attributes are possible, but the names of the attributes are not known in advance.

When the following XML is converted to XmlData

<employee>

<name>John Smith</name>

<address category="home" form="customary" status="active">

<street>100 Main</street>

<city>Friendly</city>

<state>Confusion</state>

</address>

</employee>

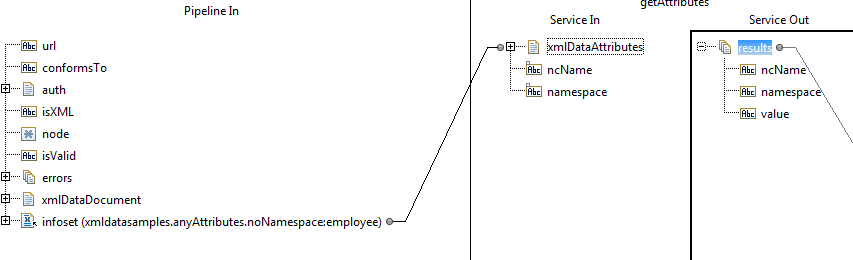
We observe the following results:



Notice that there is no \*anyAttribute tag present in the result, just the typical \*attributes document and the attribute key/value pairs.

## Accessing from Flow

In order to access these attribute from a Flow, it is necessary to use service **pub.xmldata.getAttributes**. With this service, the \*attributes parent is mapped into the service input **xmlDataAttributes**. The service returns a list of attributes and their values.



The service is invoked with no ncName argument, which causes all attributes to be returned. When the service is run, the following pipeline results are observed. In the results list, we can see the attribute name and value pairs.



## Accusing from Java

The XmlDataCursor class has a method to return the value of a specific attributes or the entire list of attributes. In the service **anyAttributes:accessValues,** the cursor is positioned on the address element and the **getAttributes**() method is invoked to return an IData of tag-object-pairss for all of the attributes. The result is assigned to pipeline variable **attributes**.

# Substitution groups

XML Schema supports the notion of substitution groups. They are somewhat like choice groups or any wildcards in that they make it possible for any one of a number of alternatives to appear in a content model. They differ from substitution groups in that they are not formal models. Instead a schema author merely asserts that some other global element can be substituted for another global element that is the target of the substitution. The target of the substitution is called the head of the substitution group. There is nothing obvious in the declaration of an element that it might be the head of a substitution group. Rather it is the substitution group reference by another element that makes it possible. This makes it possible for a schema author to leverage an existing schema definition without necessarily restructuring existing model groups.

Starting with the schemas used for the nay wildcard sample, a few changes are applied.

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="samples.xml.org"

xmlns="samples.xml.org" >

<xsd:import schemaLocation="hobbies.xsd" namespace="localhost/hobbies" />

<xsd:element name="activity" type="xsd:string" />

<xsd:element name="employee" type="personInfo" />

<xsd:complexType name="personInfo">

<xsd:sequence>

<xsd:element name="fullName" type="xsd:string" form="qualified" />

<xsd:element name="address" type="addressInfo" />

<xsd:element ref="activity" maxOccurs="10" />

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="addressInfo" >

<xsd:sequence>

<xsd:element name="street" type="xsd:string" />

<xsd:element name="city" type="xsd:string" />

<xsd:element name="state" type="xsd:string" />

</xsd:sequence>

<xsd:attribute name="category" type="xsd:string" />

<xsd:attribute name="version" type="xsd:string" form="qualified" />

</xsd:complexType>

</xsd:schema>

Very little has changed. In this model we have an activity global element that is referenced from a content model. However, the imported schema has changed somewhat.

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="localhost/hobbies"

xmlns="localhost/hobbies" xmlns:sam="samples.xml.org" >

<xsd:import schemaLocation="schema.xsd" namespace="samples.xml.org" />

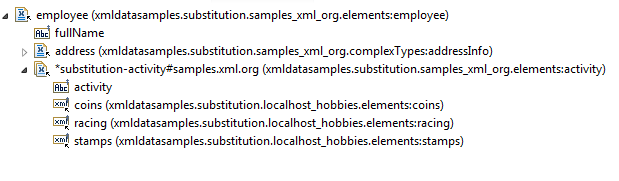
<xsd:element name="coins" type="xsd:string" substitutionGroup="sam:activity" */>*

*<xsd:element name="stamps" type="xsd:string" substitutionGroup="s*am:activity" />

<xsd:element name="racing" type="xsd:string" substitutionGroup="sam:activity" />

</xsd:schema>

The samples.xml.org namespace has been imported because we are declaring one of its global elements as the head of a substitution group. Then we change the three existing global element declarations in localhost/hobbies to be members of **activity**’s substitution group. When we look at the generated Document Type for employee we see that activity has become a substitution group document that includes the original definition for activity, and also element refs to the other members of the substitution group.



The input XML file is identical to the file used for the any wildcard sample.

<sam:employee xmlns:sam="samples.xml.org" xmlns:hob="localhost/hobbies">

<sam:fullName>John Smith</sam:fullName>

<address category="home" sam:version="1.0">

<street>100 Main</street>

<city>Friendly</city>

<state>Confusion</state>

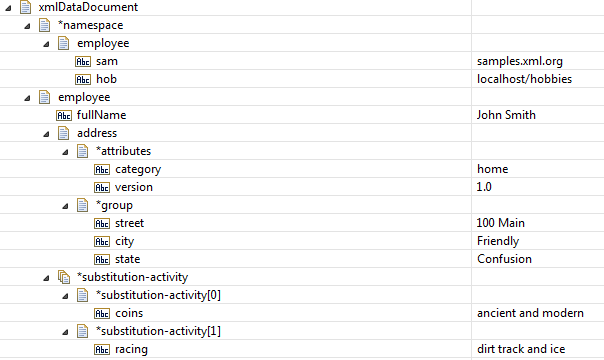
</address>

<hob:coins>ancient and modern</hob:coins>

<hob:racing>dirt track and ice</hob:racing>

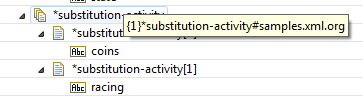
</sam:employee>

When the file is converted to XmlData format, we see the following results.

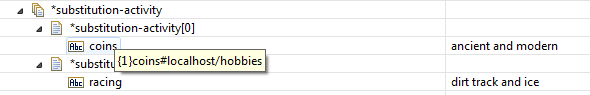


Notice that the substations for the activity element appear as an array with each member containing the single substituted element consisting of a tag and content pair.

Note that the head of the substation is namespace qualified.



Also notice that the members are namespace-qualified as appropriate. In this case the substation members are in namespace that is different from the head of the group.

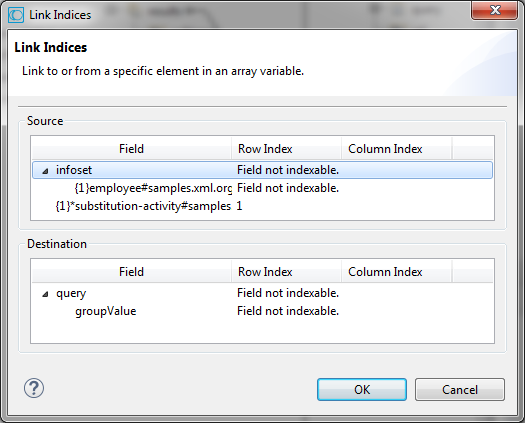


## Accessing from Flow

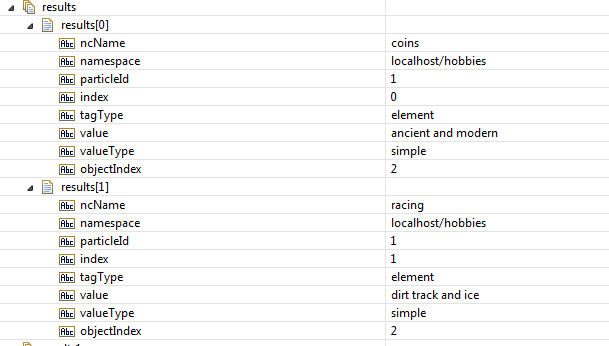
Looking at **substitution:convertInput,** we see two different techniques for determining the actual substitution tag and value. In the first invocation of **pub.xmldata:getGroupValues**, we map a specific member of the substitution group list (index = 1) to the query input. We map the output to results1 and observer the following:



The group value of the substitution head is an IData containing a single value with ncName “racing” and namespace “localhost/hobbies”. The query arguments expects a scalar IData so we must map the specific member of interest as shown below (right-click on the copy link).



However, looking at the second invocation of **pub.xmldata:getGroupValues,** we retrieve all of the child values of employee from **objectIndex** 2 (the substitution group head), which includes the two substitution members.



## Accessing from Java

The **substitution:accessValues** service positions to the substitution tag and retrieves the second value and the second tag, as shown below.



# anyType Content

With XML Schema, anyType literally means any well-formed XML. Within XmlData Document Types, anyType is represented by an Object field. The anyType is represented as an Object field because it is impossible to know in advance whether content will be simple (String) or complex (IData). Conversion of anyType is not Document Type directed.

The following XML input when process by the Flow service **anyType:convertInput**

<payload xmlns:ns1="nsone" xmlns:ns2="nstwo">

<able mode="off">

<ns2:delta>green</ns2:delta>

<gamma/>

</able>

<baker ns1:epsilom="23" />

<charlie>fox</charlie>

<ns2:zeta>five</ns2:zeta>

<ns2:zeta>

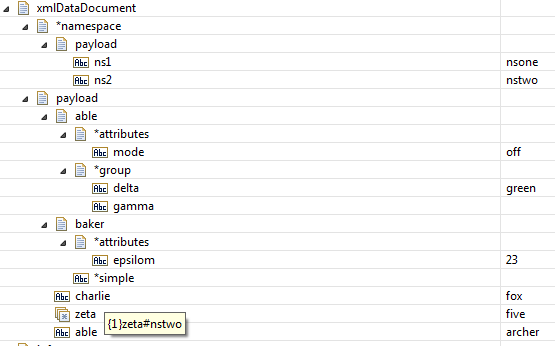
<one>xxx</one>

<two>yyy</two>

</ns2:zeta>

</payload>

produces the following output:



A few aspects of the conversion are worth noting.

* The namespace declarations on the document element, **payload**, are preserved in the \*namespace document
* The **zeta#nstwo** field is an Object[]; the first member is a String, the second is an IData.
* Field **{1}able** is an IData document, field **{2}able** is a String.

XmlData groups have a tag-object pair for each particle in the corresponding schema model. For example, if a schema sequence contains three element particles followed by an any particle, the corresponding XmlData document has a total of four fields corresponding to the four particles. The XmlData instance has four tag-object pairs, one for each field. If the field has maxOccurs > 1, then the instance object contains an array of values and the instance object is an array of values. When maxOccurs = 1, the object is a single value and the tag-object pair is actually a tag-value-pair.

During anyType processing, there is no schema, so an “ad hoc” model is produced from the input XML. The “ad hoc” model consists of a particle for each tag in the input, with repeating adjacent tags aggregated into a single particle with maxOccurs equal to the number of adjacent occurrences. Thus, the output XmlData contains array of repeating values that correspond to repeating adjacent input tags. “Ad hoc” anyType models never contain “any” tags or “substitution” tags. There is only a single top-level “group” tag, if needed. There are never relating groups.

The structure of anyType content is not known in advance. Therefore it is necessary to perform some analysis on an ad hoc basis. There are two services that can help with this analysis: **pub.xmldata:getGroupValues** and **pub.xmldata:getGroupObjects**. These services can be used on any XmlData input, but they are particularly using when analyzing anyType, any, substitution, and choice groups where the exact structure of the input isn’t known until runtime.

The input to **getGroupValues** specifies the values for which information is returned. This is specified by children of the query parameter.

* ncName – the local name; if not specified all ncNames are included in the query.
* namespace – the namespace; if not specified all namespaces are include; an empty String is used to specify no namespace.
* particle-id – the particle Id; if not specified all particle Ids are included
* objectIndex – the index of the object to be included in the query. If not specified, all objects are included.

The output is returned in the **results** field as an array of documents with the following child fields for each value in the group:

* ncName – the local name portion of the tag
* namespace – the namespace portion of the tag
* particle-id – the particle Id associated with the tag
* tagType – element or group
* index – the index of the value in a group of repeating values
* value – the value associated with the tag; if the value is an any-item or a substitution-item, the value stored indirectly in the item is returned.
* valueType – simple, group, or complex
  + simple – a String value
  + group – an IData with only tag-values-pairs (no attributes or a repeating top-level group)
  + complex – a value with attributes and/or a repeating top-level group-value
* content – the simple content, the group content, or the content portion of a complex value
* attributes – the IData containing the attribute tag-object-pairss. This filed is set only when type is complex and a \*attributes tag is present in the complex value.
* complexContentType – none, simple, or group.; set only when valueType is complex
  + none – there is no content, only attributes
  + simple – there is simple content associated with the \*simple tag
  + group – there is IData or IData[] associated with the {1}\*group tag
* complexContent – the content portion of a complex value.
* repeatingComplexContent – true is complex content is an IData[]; false otherwise
* objectIndex – the offset of this value in the object array of repeating values. When repeating values aren’t present, this value is always 0.

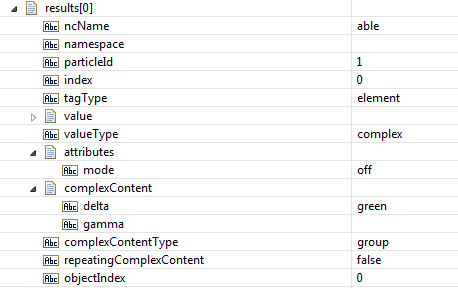
The input to **getGroupObjects** specifies the values for which information is returned. This is specified by children of the query parameter.

* ncName – the local name; if not specified all ncNames are included in the query.
* namespace – the namespace; if not specified all namespaces are include; an empty String is used to specify no namespace.
* particleId – the particle Id; if not specified all particle Ids are included

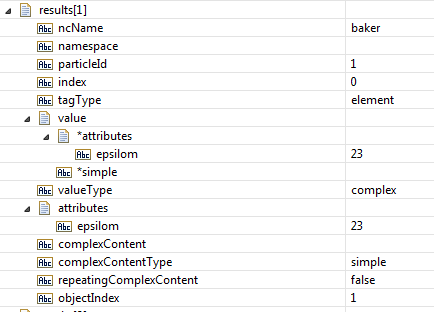
The output is returned in the **results** field as an array of documents with the following child fields for each object in the group:

* ncName – the local name portion of the tag
* namespace – the namespace portion of the tag
* particleId – the particle Id associated with the tag
* tagType – element, group, substitution, or any
* object – the object associated with the tag; it must me a String, String[], IData, IData[], or Object[]
* objectType – string, document, or object
* size – dimension of the array or 1
* hasRepeatingValues - true if the object is an array

If we invoke the service **xmldata:getGroupValues** on **payload**, we see that the **results** pipeline variable contains six members corresponding to the six child elements of payload.



**Results**[0] describes the element **able**. It’s in no namespace, so this entry contains the empty String. The **particleId** is 1, the first occurrence of the tag in the content model. The **index** is zero, it is the first and only member of able (because **{1}able** does not contain repeating values). The **tagType** is **element**. The value of the tag is an IData (but is collapsed in the image and not shown). The **valueType** is complex (because it contains attributes). The **attributes** field contains a reference to the \*attributes document and **complexContent** has the content of the top-level group. The value of c**omplexContentType** is **group** (it will always be either group or simple, but is only present when **valueType** is complex). **RepeatingComplexContent** is false (because the top-level group is not a repeating group). The objectIndex is 0 because song-and-dance-goes here.



**Results**[1] describes the element **baker**. It’s in no namespace, so this entry contains the empty String. The **particleId** is 1, the first occurrence of the tag in the content model. The **index** is zero, it is the first and only member of able (because **{1}baker** does not contain repeating values). The **tagType** is **element**. The value of the tag is simple with attributes.. The **valueType** is complex (because it contains attributes). The **attributes** field contains a reference to the \*attributes document and **complexContent** has the empty String. **ComplexContentType** is **simple** (it will always be either group or simple, but is only present when **valueType** is complex). **RepeatingComplexContent** is false (because simple content never repeats). The objectIndex is 1 because song-and-dance-goes here.



**Results**[2] describes the element **charlie**. It’s in no namespace, so this entry contains the empty String. The **particleId** is 1, the first occurrence of the tag in the content model. The **index** is zero, it is the first and only member of able (because **{1}charlie** does not contain repeating values). The **tagType** is **element**. The value of the tag is simple with attributes.. The **valueType** is simple. The **attributes, complexContent, complexContentType,** and **repeatingComplexContent** fields are not defined for simple values. The objectIndex is 2 because song-and-dance-goes here.



**Results**[3] describes the element **zeta#nstwo**. It’s in the nstwo namespace. The **particleId** is 1. The **index** is zero, it is the first of two members of **zeta#nstwo**. The **tagType** is **element**. The value of the **tag** is the simple value **five**. The **valueType** is simple. The **attributes, complexContent, complexContentType,** and **repeatingComplexContent** fields are not defined for simple values. The objectIndex is 3 because song-and-dance-goes here.



**Results**[4] describes the element **zeta#nstwo**. It’s in the **nstwo** namespace. The **particleId** is 1. The **index** is 1; it is the second of two members of **zeta#nstwo**. The **tagType** is **element**. The value of the tag is a group value containing two tags. The **valueType** is **group**. The **attributes, complexContent, complexContentType,** and **repeatingComplexContent** fields are not defined for group values. The objectIndex is 3 because song-and-dance-goes here.



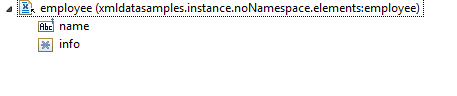
**Results**[5] describes the element **able**. It’s in no namespace. The **particleId** is 2 because this is the second occurrence of this tag in the model. The **index** is zero because it is a non-repeating value. The **tagType** is **element**. The value of the **tag** is the simple value **archer**. The **valueType** is simple. The objectIndex is 4 because song-and-dance-goes here.

It is also possible to obtain information about each object (rather than each value). If we invoke the service **xmldata:getGroupObjects** on **payload**, we see that the **results** pipeline variable contains six members corresponding to the objects in payload. (This step is disabled in the sample flow and must be enabled. This will replace the results in the pipeline returned by the previous step which invoked the service **xmldata:getGroupValues.**)

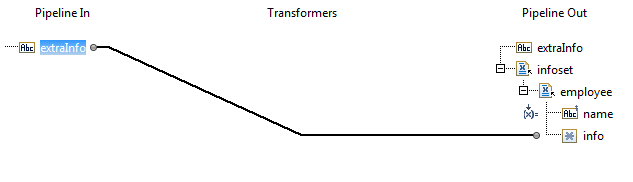
# Instance properties

XML Schema uses instance attributes to assign schema-related properties to XML content. These instance attributes are defined using XML namespace <http://www.w3.org/2001/XMLSchema-instance>. Typically it is defined using the prefix “xsi”, though the use of “xsi” is strictly by historical convention; any valid prefix can be used. XmlData uses an approach for instance properties very similar to the approach used for XML namespace declarations. The properties are expressed as name/value pair in the IData parent of the object to which they apply.

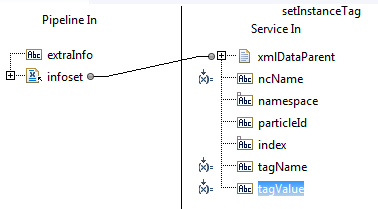
Using the XSD file in pub/samples/instance, we create XmlData Document Types. The employee document has two fields:



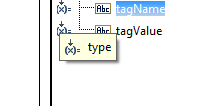
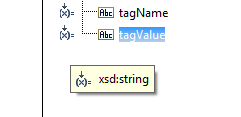
The info field is defined as anyType. This signifies that any type of valid XmlData structure can be assigned to info. In the following map operation we map a String to info that has been defined in the previous step:



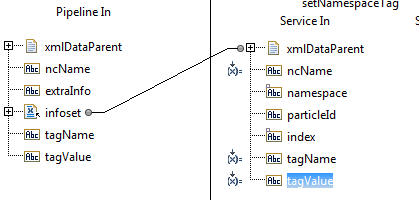
In order to create XML from this structure, it is necessary to assert the type of info using the schema instance type property. We can do this my invoking pub.xmldata.setInstanceTag. We set the tag name (type) to value “xsd:string”.



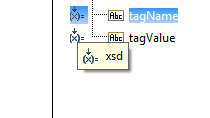
As illustrated in the tool tips

In addition to setting the type, we must now define the namespace for prefix xsd because that appears nowhere in the XmlData. We accomplish this by invoking **pub.xmldata.setNamespaceTag** with **tagNames** set to **xsd** and **tagValue** [**http://www.w3.org/2001/XMLSchema**](http://www.w3.org/2001/XMLSchema)**.** This is illustrated in the map:



And the tool tip hints:

When **pub.xmldata:XmlDataToXmlString** is invoked, we can examine the output which has been saved to pipeline variable **firstResult**.

<employee xmlns:xsd="http://www.w3.org/2001/XMLSchema"><name>expert</name><info xmlns:ns0="http://www.w3.org/2001/XMLSchema-instance" ns0:type="xsd:string">beknighted</info></employee>

Notice that no prefix definition is in scope for “XMLSchema-instance”, so the encoding service automatically creates one using default prefix **ns0**. In order to use the more traditional prefix **xsi**, the **setNamespaceTag** service can also be used to also provide a namespace declaration for the instance namespace. This results in the more traditional appearing encoding in pipeline variable **xmlString**:

<employee xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><name>expert</name><info xsi:type="xsd:string">beknighted</info></employee>

This extra step of defining the xsi prefix isn’t necessary, it is just a matter of personal preference because the default use of prefix ns0 is perfectly valid and equivalent to the explicit xsi prefix specification.

# Advanced XmlDataCursor

The XmlData processing for interface for Java is defined in the Java package com.wm.xmldata. There are a variety of interfaces as well as concrete classes. The two most typically used classes are XmlData and XmlDataCursor.

## Differences between basic XmlData and legacy IData infoset

The most basic methods of the XmlDataCursor are not too dissimilar from the traditional IDataCursor methods. There are some terminology differences. For example, IData is typically described as key/value pairs whereas XmlData is typically described as tag-object-pairs; this highlights XmlData’s close affiliation with the XML infoset.

Another example of a notable difference is that XmlDataCursor is more discriminating when working with an unpositioned cursor. For example, many IData implementations return a null when getValue() is invoked on an unpositioned cursor while XmlData throws an UnpositionedCursor exception. While this may seem inconvenient at times, the intent is to reduce the number of “accidental” implementations that might work simply because a user forgot to consider the unintended consequences of an unpositioned cursor.

Yet another example is that IData keys are simple strings, but the XmlData API uses various forms XmlDataTags as its keys. XmlDataTags incorporate both the local name and the namespace as part of tag. This eliminates the need for an “external” definition of namespace via a prefix defined by a Document Type or a namespace declaration table. External definitions can be difficult to maintain and often lead to conflicts and inconsistencies. Constructors are used to create these tags from Strings.

XmlDataCursor explicitly supports the concept of repeating values (arrays of values) and provides specific meth implementations that are array-aware. With IDataCursor, the user is responsible for implementing array-aware data.

## Summary of Basic methods

This table provides a comparison of XmlData versus IData. Features uniquely supported by XmlData are highlighted.

|  |  |  |
| --- | --- | --- |
| **Operation** | **XmlData** | **IData** |
| Obtain a cursor on an IData instance | XmlData.getXmlDataCursor(ido)  **ido** is an IData instance object | ido.getCursor()  **ido** is an IData instance object |
| Position to first, next, previous, or last pair | cursor.first(), cursor.next(), cursor.previous(), cursor.last() | cursor.first(), cursor.next(), cursor.previous, cursor.last() |
| Position to first, next, previous, last tag ( key) | cursor.first(tag), cursor.next(tag), cursor.previous(tag), cursor.last(tag)  **tag** is XmlDataTag | cursor.first(key), cursor.next(key), cursor.previous(key), cursor.last(ley)  **key** is a String |
| Retrieve value at the current tag (key) | cursor.getValue(), cursor.getValue(index)  **index** is an offset into an array of repeating values | cursor.getValue() |
| Store value at the current tag (key) | cursor.setValue(value), cursor.setValue(value, index)  **index** is an offset into an array of repeating values | cursor.setValue(value) |
| Retrieve the current tag (key) | cursor.getTag()  returns an XmlDataTag | cursor.getKey()  returns a String |
| Stores the current tag (key) | cursor.setTag(tag)  **tag** is an XmlDataTag | cursor.setKet(key)  **key** is a String |
| Make the cursor unpositioned | cursor.home() | cursor.home() |
| Release resources | cursor.delete() | cursor.delete() |
| Delete the current pair | cursor.deleteObject(), cursor.deleteItem(index)  **index** is an offset into an array of repeating values | cursor.delete() |
| Insert a pair after the current pair | cursor.insertObjectAfter(tag, object) cursor.insertItemAfter(value, index)  **index** is an offset into an array of repeating values | cursor.insertAfter(key, value) |
| Insert a pair after the current pair | cursor.insertObjectBefore(tag, object) cursor.insertItemBefore(value, index)  **index** is an offset into an array of repeating values | cursor.insertBeforer(key, value) |
| Obtain a cloned copy of the cursor | cursor.copyCursor()  return an XmlDataCursor | cursor.getCursorClone()  return an IDataCursor |
| Insert pair with a value that is a new nested container | cursor.insertObjectBefore(tag, XmlData.createGroupValue())  cursor.insertObjectAfter(tag, XmlData.createGroupValue()) | cursor.insertDataBefore(key)  cursor.insertDataAfter(key) |
| Non-disruptive preview of cursor.next() | NOT SUPPORTED | cursor.hasMoreData() |

## Error handling

IData has methods for interrogating the error status of a cursor. These include: setErrorMode(), getLastError(), and hasMoreErrors(). XmlData does not support these methods. XmlData throws an exception when erroneous use is encountered.

The XmlDataException class uses the following subclasses to describe error situations:

* UnpositionedCursor – the cursor was unpositioned even though the requested operation requires a positioned cursor
* IndexRequired – no index was specified for an operation on a repeating value
* IndexOutOfRange – an index was specified that exceeds the size of the repeating value.
* UnsupportedType – an attempt to insert a value whose type is not supported by XmlData. Generally, XmlData only supports String, String[], IData[], IData[], and Object[]. Object[] may only contain String or IData.
* MemberConflict – an attempt to set or insert a repeating value into an array of differing type, for example attempting to insert a String into an IData[].
* FormatError – an attempt to retrieve or update an XmlData infoset failed because it is not formatted properly.
* ArgumentError – an attempted update or retrieval failed because of an invalid argument
* TagAndContentConflict – an attempt to use an inappropriate tag and content pairing; for example, assigning a String directly to an any-tag.
* TagConflict – an operation was attempted that is not supported for the current tag.

There are some exceptions than can only occur when using the more advanced features of XmlDataMap and XmlDataCopy. They include:

* ModelConflict – the existing XmlData infoset conflicts with the mapped Document Type
* UnknownField – the requested field does not exist in the mapped Document Type.
* IndirectItemError – no content qualifying tag was specified for a map operation on an indirect-item.
* ImproperDocumentType – attempt to define a map using non-XmlData Document Types
* ReadOnlyWorkspace – there was an attempt to define a global component using a Workspace that is read-only. The IS Server permanent workspace is read-only. Temporary workspaces are read-write.

There are actually quite a few more exception types that are covered later, as needed.

## Differences between IData keys and XmlDataTags

As noted earlier, XmlData uses tags (keys) that are more structured than the legacy XML infoset representation. In particular, a namespace-qualified tag is represented as name#namespace. This eliminates the need for prefixes and external definition of prefixes. In addition to namespace qualification, XmlData also distinguishes among repeated usages of the same name in a schema model. It does this by prefixing a particle id to the name. So the first occurrence of a name in a model is prefixed by {1}, the second occurrence by {2}, etc. This reduces the likelihood of mapping operations inadvertently affecting the order dependence of values. For example, using the IData model, inserting a key of “employee” and then later inserting a second occurrence of “employee” might have unintended consequences if the first occurrence has been deleted (for example the second occurrences now becomes the first). By unambiguously mapping to a specific occurrence, for example {2}employee, the presence of absences of {1} employee does not affect the intended result.

Constructors are provided for tags.

* new XmlDataElementTag(name) creates an XmlDataTag for element with no namespace. The particle Id defaults to 1.
* new XmlDataElementTag(name, int) creates an XmlDataTag for element with no namespace and the specified particle Id.
* new XmlDataElementTag(name, namespace) creates an XmlDataTag for element with the specified namespace. The particle Id defaults to 1.
* new XmlDataElementTag(name, namespace, int) creates an XmlDataTag for element with the specified namespace and particle id.

This may seem like “overkill”, but in fact there are a number of different tag types and having formal types and constructors for each is necessary.

## Value types for XmlData

The legacy XML infoset representation has no formal API. There are a set of conventions for representing simple content, complex content, attributes, etc. Many edge conditions (e.g. empty content) have no formal definition and various components might represent these edge conditions differently and might even change at a given fix level. This leads to much confusion and frustration (as evidenced by the large number if wat parameters that govern these edge conditions).

XmlData has the follow value types:

* Simple-value – a String, nothing else. An empty value is represented by an empty String. There is no possibility of attributes. Namespace declaration and instance properties (xsi) can be specified through a mechanism described later. Simple values are paired with XmlDataElementTags.
* Group-value – an IData with zero or more pairs of tags and values. Group values are always paired with XmlDataGroupTags in their parent. Their content may be either group tags paired with group values, or element tags paired with simple values, complex values, or group values.
* Complex-value – a specialized IData structure that contains one of:
  + A collection of attributes with simple value content, group value content, or no content. The attributes are stored as an IData pair with a \*attributes tag. Group content is stored as an IData paired with a {1}\*group tag. Simple content is paired with the tag \*simple.
  + Repeating top-level group values (with or without attributes). The top-level IData is always a {1}\*group tag paired with the top-level group content.
* Indirect-item – a specialized IData structure used to represent the values of “any” wildcards and substitution group members. Technically, indirect items are not values, but contain a reference to a value. When XmlDataCursor methods like getValue() and setValue() are applied to indirect items, the operation is actually performed on the value referenced by the indirect item. A set of specialized cursor methods getItem(), setItem(), insertItemBefore(), and insertItemAfter() are used to access and manipulate the indirect item. These methods are typically not used in simple applications.

There are formal XmlDataTag constructors for groups. The constructors are:

* new XmlDataGroupTag() – create an XmlDataGroupTag with default particle id of 1.
* new XmlDataGroupTag(int) – create an XmlDataGroupTag with the specified particle id.

There is no API for creating \*simple or \*attribute tags. Instead, there is a formal API for manipulating complex-values and attributes.

Group-values correspond to schema model groups (sequence, choice, and all). Whenever a Document Type is created from a schema model, there is always a one-to-one correspondence between non-trivial schema groups and Document Type \*groups.

There is a formal XmlDataTag constructor for any wildcards. The constructors are:

* new XmlDataAnyTag() – create an XmlDataAnyTag with default particle id of 1.
* new XmlDataAnyTag(int) – create an XmlDataAnyTag with the specified particle id.

There is a formal XmlDataSubstitutionTag for substitution heads. They are of the form {x}substitution-name#namepace. The constructors are:

* new XmlDataSubstitutionTag(name) – create an XmlDataSubstitutionTag with specified name default particle id of 1.
* new XmlDataSubstitutionTag(name, namespace) – create an XmlDataSubstitutionTag with specified name and namespace and default particle id of 1.
* new XmlDataSubstitutionTag(name, int) – create an XmlDataSubstitutionTag with specified name and particle id.
* new XmlDataSubstitutionTag(name, namespace, int) – create an XmlDataSubstitutionTag with specified name, namespace and particle id.
* new XmlDataSubstitutionTag(tag) – create an XmlDataSubstitutionTag derived from the specified XmlDataElementTag.

|  |  |
| --- | --- |
| **Schema construct** | **Generated XmlData Document Type** |
| element with simple type (maxOccurs = 1) | XmlDataElementTag with String content |
| element with simple type (maxOccurs > 1) | XmlDataElementTag with String[] content |
| group model | A Document Type with one field for each particle in the model. Models may contain elements, group, any-wildcards, or references to substitution group heads. |
| complexType with no children or attributes | Empty Document Type |
| complexType with attributes | Document Type with \*attributes Document child; \*attributes child has one String child for each attribute |
| complexType with attributes and complexContent | Document Type with \*attributes child as described above; a single {1}\*group child with group content |
| complexType with attributes and simple content | Document Type with \*attributes child as described above, a \*simple child with String simple content |
| complexType with non-repeating top-level group | Document Type with corresponding to the group content (there is no {1}\*group top-level child in this situation) |
| complexType with repeating top-level group | Document Type with a single {1}\*group tag with children corresponding to the top-level group. |
| element with complex type (maxOccurs =1) | XmlDataElementTag with a Document for complexType as described above |
| Element with complex type (maxOccurs > 1) | XmlDataElementTag with a Document List for complexType as described above |
| any wildcard | Document Type with a specialized structure of three children:   * ncName * namespace * content   any wildcard models must be paired with an XmlDataAnyTag |
| substitution head | Document Type with a specialized structure; it has one child for each of its substitution members (including itself). It must be paired with an XmlDataSubstitutionTag. |

Summary of tag and value pairs supported by XmlData:

|  |  |
| --- | --- |
| **Tag** | **Associated value** |
| XmlDataElementTag | simple value, Group value, complex value, any-item, substitution-item |
| XmlDataGroupTag | group value |
| XmlDataSubstitutionTag | substitution-item. A substitution-item must be a group value (i.e. an IData) containing a single tag-object-pair that corresponds to the substitution member and its value |
| XmlDataAnyTag | Any-item. A specialized IData structure whose “ncName” and “namespace” tags correspond to an element tag or a substitution tag because an any-item is a place-holder for an element and its value. |

## The ‘item’ methods.

The item methods provide a way to directly access indirect items (rather than their values). Otherwise, the ‘item’ methods work exactly like the ‘value’ methods when accessing non-indirect-items such as simple values, group values, and complex value. When accessing indirect items, the ‘item’ methods return the actual any-item or substitution-item which contains a reference a value. Generally, the ‘value’ methods are more useful since they mask the existence of the indirect items from the use and work with values. The ‘item’ methods are used only when an application has a specific need to construct or modify the indirect item (rather than its value). This is an unusual situation.

The “item” methods include:

* getItem(index) – returns the repeating value at the specified index. It is an error to specify an index > 0 if the value is not a repeating value.
* setItem(item, index) -- sets the repeating value at the specified index. It is an error to specify an index > 0 if the value is not a repeating value.
* insertItemAfter(item, index) – insert an item into the repeating value array. Inserting at position 0 on a non-repeating value converts the value to a repeating value.
* insertItemBefore(item, index) – insert an item into the repeating value array. Inserting at position 0 on a non-repeating value converts the value to a repeating value.
* deleteItem() – removes the specified item from the repeating array by moving all items below the specified item up one position and shortening the array length by one. If the last item is removed, the entire tag-object-pair is removed (XmlData never creates tag-object-pairs with a zero-length array).

An application should **NOT** attempt to construct complex-values, any-items, or substitution-items directly using IData methods. There are static helper methods in the XmlData class that should be used for this purpose.

***Typically, most applications can be written with minimal or no use of the “item” methods.***

## The “object” methods

The “object” methods provide a way to manipulate the entire object associated with a tag. This makes it possible to get or set the entire array of values (or items) associated with a tag. SetObject requires that a fully and properly constructed object is provided.

The “object” methods include:

* deleteObject() – delete the entire object, for example an array of repeating values
* getObject() – get the entire object, for example an array of repeating values
* insertObjectAfter() – insert a tag-object pair, for example an array of re[eating values
* insertObjectBefore() – insert a tag-object pair, for example an array of repeating values
* setObject() – stores the object at the current tag. The object must be compatible with the tag. For example, if the current tag is a substitution tag, the object must be a substitution-item or an array of them. If the current tag is a group tag, the object must a group value or an array of them. Etc.

## Working with content

Complex values may contain both content and attributes. The content may be a simple content (a subordinate simple value), or group content (a subordinate group value). Simple values may only be Strings. Group values may not contain attributes. At times it may be preferable to work with the ‘net’ content of a value without making all of these distinctions with regard to the value type. There are some methods that can help in this situation

* getContent() – returns the ‘net’ content portion of a value:
  + For simple values or complex values with simple content, a String is returned
  + For group values or a complex values with non-repeating group content, an IData is returned
  + For complex values with a repeating group content, an IData[] is returned.
* setContent() – sets the content of the current value, leaving attributes (if any) undisturbed
  + For simple values and group values this method behaves exactly kike setValue().
  + For complex values, the current \*simple or {1}\*group content is replaced by the new content.
* getComplexValueContent() – a more restrictive method that only returns content if the value is in complex value format. For simple and group content, an exception is thrown
* setComplexValueContent() – a method that sets the simple or group content of a complex value. If the value is not in complex value format, an exception is thrown.
* isComplexValue() – returns true if getComplexValueContent() or setComplexValueContent() can be successfully invoked for the current value. Essentially this means that the value contains any of: attributes, simple content, or repeating top-level group content; or it must be a completely empty IData.
* isGroupVaue() – returns true if the current value is an IData which is either empty or is not a complex value.

Note that when an empty IData is tested with isComplexValue() or isGroupValue(), they both return true.

Note that when dealing with group values that are assigned to group tags, it is possible to use getValue() and setValue() directly.

|  |  |  |
| --- | --- | --- |
| Method invoked … | Current value is … | Result is … |
| setContent(object)  object may be String, IData, or IData[] | Complex | Any existing group or simple content is replaced with the new content; attributes remain unchanged |
| setContent(object) | Simple or group  (same result as setValue(object)) | The current value is replaced; assigning simple content to a group tag results in an exception |
| getContent() | Simple or complex with simple content | A String |
| getContent() | Group or complex with group content | An IData or IData[] |
| getComplexContent() | complex | String, IData, IData[], or null if no content present |
| getComplexContent() | Simple or group | Exception thrown |
| setComplexContent() | complex | Any existing simple or group content is replaced |
| setComplexContent() | Simple or group | Exception thrown |
| isComplexValue() | Must be IData; must contain any of \*attributes, \*simple, a solitary {1}\*group, or be completely empty | true |
| isGroupValue() | Must be IData; cannot contain \*simple, \*attributes, or solitary {1}\*group | true |

## Working with attributes

Attributes are not ordered and must be unique. They are accessed through the cursor and are addressed using an XmlDataAttributeTag. Attribute tags do not have a particle-id because attributes are unique within an element instance.

The constructors for XmlDataAttributeTags are:

* new XmlDataAttributeTag(name)
* new XmlDataAttributeTag(name, namespace)

Attribute methods apply to the element ta the current cursor position. They include:

* getAttributeValue(tag) – return the values of the attribute. Returns null if the attribute is not present.
* getAttributes() – return an IData list of attributes and their values
* getAttributeNames – return an array of XmlDataAttributeTags
* has Attributes() – return true if the element has attributes
* removeAttribute(tag) – remove the specified attribute
* setAttributeValue(tag, value) – sets the attribute value. Value may not be null.

## Working with tag names

Generally, the setTag() method can be used to change the tag associated with the current tag-object-pair. However, when working with indirect items the typical goal is to change the tag associated with the value, not necessarily the item tag. The setValueTag() method provides an easy to insulate an application from the need to handle tags for values referenced by indirect items differently from values that are directly associated with an element tag. The setValueTag() method changes the tag when positioned on an element, changes the ncName and namespace when positioned on an any-item, and changes the tag of the first child when positioned on a substitution-item.

* getValueTag()
* setValueTag()

## Working with namespaces

## Working with instance properties

## Other useful methods

* count() – returns the number of repeating values; returns 1 if not an array
* data() – return the IData object on which this cursor was obtained
* isAny() – return true if the current tag is an XmlDataAnyTag
* isElement – return true if the current tag is an XmlDataElementTag
* isGroup() – returns true if the current tag is an XmlDataGroupTag
* isRepeating() – returns true if the object at the current tag is an array
* isSubstitution – return true if the current tag is an XmlDataSubstitutionTag

## Specialized navigation

There are a variety of methods for specialized or simplified navigation. The “partial” methods position to the specified tag ignoring the particle id. Thus it is possible to position to the next, previous, first, or last occurrence of a tag name without regard for its association with a distinct particle.

* firstPartia() – first occurrence of a tag ignoring particle id.
* getValueCursor() – returns a cursor if the current value is complex and has group content, or if the value is a group. If the group is empty the cursor is unpositioned. If the current value content is not group (e.g. a String), a null is returned.
* isHome() – return true if cursor is unpositioned
* lastPartial – last occurrence of a tag ignoring particle id.
* nextPartial() – next occurrence of a tag ignoring particle id.
* previousPartial() – last occurrence of a tag ignoring particle id.
* getAnyTag()
* setAnyTag(tag)

## The XmlData helper class

An application should **NOT** attempt to construct complex-values, any-items, or substitution-items directly using IData methods. There are static helper methods in the XmlData class that should be used for this purpose.

### Indirect items

* createAnyItem() – various forms of this method are used to construct an any-item which can then be inserted into the infoset using a cursor method such as insertItem().
* createSubstitutionItem()
* getAnyItemTag() (specialized and not generally used)
* getAnyItemObject()
* getAnyItemValue()
* getAnyItemValueTag()
* getSubstitutionItemValue()
* getSubstitutionItemValueTag()
* setAnyItemObject()
* setAnyObjectValue()
* setAnyObjectValueTag()
* setAnyItemTag() (specialized and not generally used)
* setSubstitutionItemValue()
* setSubstitutionItemValueTag()

### Builders

The builders are used to create various types of values and items.

* createComplexValue() – various forms of this method are used to construct a complex-value which can be placed into the infoset using cursor methods such as insertItem(), insertObject(), setValue(), etc. Such constructed values can also be used by helper methods such as createAnyItem(), etc.
* createGroupValue() – various forms of this method are used to create group-values

### Attributes

There are methods to extract or set the \*attributes child of a complex-value.

* getAttributes(complex-value) – return the \*attributes IData that contains tag-object-pairs of attributes names and their values.
* setAttributes(complex-value, attributes) – create or set the \*attributes child of complex-value.

### Content

* getComplexValueContent() – returns the content portions of a complex-value. This is the value associated with the \*simple ot top \*group child. Returns null if no content is found.
* setComplexValueContent – sets the \*simple or top-level \*group child.
* getConent() – returns the content of the current item. For simple-values, the result is the same as getValue(). For complex-value, the result of getComplexValueContent() is returned, except in the case when the current value is an empty IData which means that it can be wither group-value or complex-value. In this case, the empty IData is returned.

### Utility

* getXmlDataCursor(idata) – return an XmlDataCursor for the specified IData object
* getXmlDataTreeCursor(idata) – return an XmlDataTreeCursor for the specified IData object
* getValueCursor(object) – return null if the argument is not an IData; otherwise return an IDataCursor positioned on the first tag-object-pair or return an unpositioned cursor if the IData object is empty.
* copy(idata) – return a copy of the XmlData structure. New Array object and new IData objects are created. Only String references are shared.
* createEmptyStringList(size) – create an empty String list of the specified size. Each member is initialized with an empty String.
* createEmptyGroupList(size) – create an empty IData list of the specified size. Each member is initialized with an empty IData.
* createObjectList(array) – convert a String[] ot IData[] to an Object[].
* equalTo(object) – tests two XmlData objects for equality. Tags must match, be in the order. Arrays must match member by member.

### Metadata

* setNamespacePrefix()
* getNamespacePrefix()
* getInstanceProp()
* setInstanceProp()

# Using XmlDataTreeCursor

Navigating through an XmlData tree using Java can be rather cumbersome. At each level of the tree, a cursor must be obtained on values that have IData content in order to navigate to a lower level. When finished at a given level, it is necessary to destroy the cursor. When iterating over the elements and groups, it is necessary to distinguish between scalars and arrays and iterate over individual items. This behavior has been encapsulated into a helper class called an XmlDtataTreeCursor. The tree cursor allows a Java program to easily move about the XML infoset when encoded as XmlData. The XmlDataTreeCursor also flattens out arrays so that repeating elements as accessed by successive method invocations. This model move closely resembles DOM in which does not make a distinction between arrays and scalars.

A tree cursor is obtained using the static XmlData method getXmlDataTreeCursor(). The tree cursor implements all of the methods of XmlDataCursor, so at any given level in the tree, an XmlDataCursor method may be used to navigate within that level of the tree. There are a few minor differences in the behavior of the XmlDataCursor methods as implemented by tree cursor; these differences are noted later in this section.

The sample Java service **xmldatasamples.treeCursor:first** illustrates basic usage of the tree cursor. The tree cursor is especially useful when the structure of the infoset is not known in advance, such as when processing anyType content or when not using a Document Type to manage conversion from XMLNodes to the XmlData infoset. The calling service does not employ a Document Type, so the infoset is in the alternate format in which repeating values are always represented as Object[]s. The file first.xml contains:

<root>

<able>one</able>

<able>two</able>

<baker>three</baker>

<charlie>four</charlie>

<charlie>five</charlie>

<charlie>six</charlie>

</root>

Running Flow service **xmldatasamples.treeCursor:runFirst** creates the following XmlData infoset:



Note that “able” and “charlie” are repeating values and represented as Object[]s. The **xmldatasamples.treeCursor:first** service retrieve the infoset from the pipeline and creates a StringBuilder where results are accumulated.

IData input = (IData) ValuesEmulator.get(pipeline, "xmlDataDocument");

// create a buffer for console output

StringBuilder sb = new StringBuilder();

Next, a tree cursor is obtained. The XmlDataTreeCursor implements all of the XmlDataCursor methods and is mostly identical (differences are noted elsewhere). The cursor is positioned on the first tag (in this case “root”). The firstChild() method updates the tree cursor to point at the first child of the current item.

// Obtain an XmlDataTreeCursor

XmlDataTreeCursor xdc = XmlData.getXmlDataTreeCursor(input);

// Position to the first and only element, "root"

xdc.first();

// position on the first child

xdc.firstChild();

The following code simply iterates opver the infoset by performing a nextItem() and appending the tag of the current item value to the StringBuilder.

// iterate over items

sb.append(xdc.getTag().toString() + " " + xdc.getValue() + "\n");

while(xdc.nextItem())

sb.append(xdc.getTag().toString() + " " + xdc.getValue() + "\n");

sb.append("\n");

Then position on the first item again, but now retrieve the raw objects and append their value to the StringBuilder.

// position on first object again

// iterate again using the XmlDataCursor methods

xdc.first();

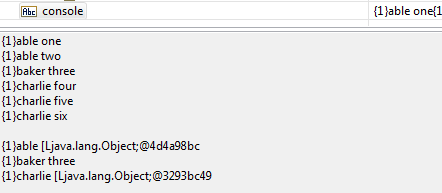
sb.append(xdc.getTag().toString() + " " + xdc.getObject() + "\n");

while(xdc.next())

sb.append(xdc.getTag().toString() + " " + xdc.getObject() + "\n");

ValuesEmulator.put(pipeline, "console", sb.toString());

Examining the output returned in the “console” variable both the output of iterating as items and as objects can be seen.



The tree cursor also provides a simplified way to navigate the values of an infoset in “Document Order”. This is essentially the order in which tags appear in an XML document. This is illustrated by **xmldatasamples.treeCursor:runSecond,** which iterates over the infoset created from:

**<root>**

**<able>**

**<baker>**

**<charlie>**

**<delta>xxx</delta>**

**<epsilon>yyyy</epsilon>**

**</charlie>**

**<theta/>**

**<charlie>four</charlie>**

**</baker>**

**<baker/>**

**</able>**

**<able/>**

**</root>**

The Java service **xmldatasamples.treeCursor:second** illustrates the ordered traversal. The infoset is retrieved from the pipeline and an output StringBuilder is obtained. An XmlDataTreeCursor is obtained on the infoset.

IData input = (IData) ValuesEmulator.get(pipeline, "xmlDataDocument");

// create a buffer for console output

StringBuilder sb = new StringBuilder();

// Obtain an XmlDataTreeCursor

XmlDataTreeCursor xdc = XmlData.getXmlDataTreeCursor(input);

The nextInOrder() method is used to iterate over the infoset. This rather compact snippet of code reveals several features of “in order” navigation. Notice that the nextInOrder() method returns an Integer. This value indicates whether the tree cursor moved up a level (+1), stayed at the same level (0) or moved down a level (-1) when progressing to the next item. A null value indicates that there are no more items in order. The iteration continues as long as nextInorder() does not return null. Additionally, the change in level is saved. It is displayed along with the tag name and the index of the item (which indicates its position in a list of repeating values (if any).

// iterate over items

Integer change;

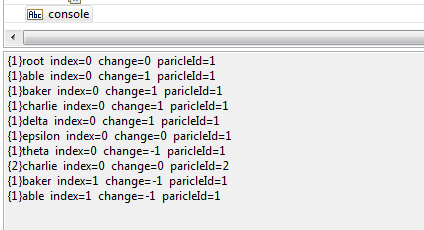
while((change = xdc.nextInOrder()) != null)

sb.append(xdc.getTag().toString() + " index=" + xdc.currentIndex() +

" change=" + change + " paricleId=" + xdc.getTag().getParticleId() + "\n");

ValuesEmulator.put(pipeline, "console", sb.toString());

The output is returned in pipeline variable “console”:



Reviewing the output:

* The first item is “root”; it is the only occurrence; there was no change in level
* The next item is “able”; it is the first occurrence; there was a change in level of 1
* The next item is “baker”; it is the first occurrence; there was a change in level of 1
* The next item is “charlie”; it is the only item; there was a change in level of 1
* The next item is “delta”; it is the only item; there was a change in level of 1
* The next item is “epsilon”; it is the only item; there was a change in level of 0
* The next item is “theta”; it is the only item; there was a change in level of -1
* The next item is “charlie”; it is the only item; there was a change of level of 0. This not the second occurrence because there is the intervening “theta” item. The “charlie” items are not consecutive, and hence not repeating; the particleId is 2.
* The next item is “baker”; it is the second consecutive occurrence; the level change is -1
* The next item is “able”; it is the second consecutive occurrence; the level change is -1

## The “item” navigators

The tree cursor maintains a current index that reflects progress through repeating values. When positioned on a non-repeating value, these navigation methods operate similarly to the “object” navigator methods such as next() and last(). However, when positioned on repeating values, the current index into the repeating value is updated as appropriate. This makes it possible to iterate over the infoset one item at a time without regard to whether the underlying object is a single value or a repeating value. These methods maintain the current index:

* nextItem() – updates the index to reference the next item in a repeating value array. If already positioned on the last item or on a non-repeating value, the cursor is positioned on the next IData pair and the index is positioned on the first item in that pair (if that pair is also a repeating value).
* nextItem(tag) – positions to the next item with the specified tag (as would be returned by getTag()). The match is based on the partial name (particle id is ignored).
* nextValueTag() – positions to the next item with the specified value tag as would be returned by getValueTag()). The match is based on the partial name (particle id is ignored). For elements and groups, nextItem(tag) and nextValueTag(tag) behave identically. For indirect-items, the value tag is extracted from the indirect value.
* previousItem() – updates the index to reference the previous item in a repeating value array. If already positioned on the first item or on a non-repeating value, the cursor is positioned on the previous IData pair and the index is positioned on the last item in that pair (if that pair is also repeating value).
* previousItem(tag) -- positions to the previous item with the specified tag (as would be returned by getTag()). The match is based on the partial name (particle id is ignored).
* previousValueTag() -- positions to the previous item with the specified value tag as would be returned by getValueTag()). The match is based on the partial name (particle id is ignored). For elements and groups, previousItem(tag) and previousValueTag(tag) behave identically. For indirect-items, the value tag is extracted from the indirect value.
* firstItem() – positions the cursor on the first pair and sets the index to reference the first item if the pair is a repeating value.
* lastItem() – positions the cursor on the last pair and set the index to reference the last item in the pair if the pair is s repeating value.

## The “item” manipulators

These methods are similar to the “item” manipulators of the XmlDataCursor, but they implicitly operate on the current item rather than on an item specified by an index, though the option always exists to specify a specific index.

* deleteForward() – deletes the current item and updates the cursor and index to reference the item after the current item
* deleteBackward() – deletes the current item and updates the cursor and index to reference the item before the current item.
* deleteItem() -- deletes the current item and follows the cursor positioning rules for XmlDataCursor deleteItem().
* getItem() – returns the item at the current position
* getValue() – returns the value at the current position
* insertItemBefore() –
* insertItemAfter() --
* insertValueBefore() –
* insertValueAfter() –
* setItem() – sets the item at the current position
* setValue() – sets the value at the current position

## The hierarchy navigators

There are methods that allow an application to move up and down in the hierarchy without explicitly obtaining cursors.

* firstChild() – positions to the first pair in the child content of the current item and returns true. Returns false if the current item has no children.
* lastChild() – positions to the last pair in the child content of the current item and returns true. Returns false if the current item has no children.
* parent() – returns to the parent item from which a firstChild() or lastChild() was perform and returns true. Returns false if already at the top parent.
* depth() – returns the nesting depth; initially zero
* isTop() – returns true if the current data has no parent owner
* top() – returns to the top-level cursor, destroying all intervening cursors
* nextInOrder() – positions to the next content-tag in document order.
* previousInOrder() – positions to the previous content-tag in document order.

# Using the XmlData static constants

Using all of the various constructors and helpers in XmlData can lead to very syntax-heavy code. There is a class XmlDataConstants that has a collection of static members that can be used as short-cuts, greatly simplifying the code syntax when using XmlData. By including:

import static com.wm.xmldata.XmlDataConstants.\*;

A variety of static methods become known in the Java class.

## Tag short-cuts

The following methods are available for tag creation. There are actually several forms for each tag, corresponding to the various forms available for each tag constructor. For example, there is an eTag(name), eTag(ncName, namespace), eTag(ncName, namespace, particleId) for the corresponding constructors.

* eTag() – element-tag
* eTagname() --
* gTag() – groups-tag
* gTagName() --
* aTag() – attribute-tag
* aTagName() --
* sTag() – substitution-tag
* sTagName()
* anyTag() – any-tag
* anyTagName() --
* tTag() – type-tag
* iTag() – iteration-tag

## Data short-cuts

Building the various XmlData constructs is simplified through short-cuts. They include:

* groupValue() – build a group value
* groups() – an array of group values
* complexValue() – build a complex value
* attributes() – construct a populated attributes list
* strings() – an array of Strings
* objects() – and array of objects
* anyItem() – an any-tem
* subsItem() – a substitution-item

## Compact expression syntax

This example illustrates how importing the static constants described above can lead to more compact notation.

This code fragments creates a complex-value containing attributes and repeating group-value content:

IData complex = XmlData.createComplexValue(

createAttributes(new XmlDataAttributeTag("one"), "123"),

new IData[]{

XmlData.createGroupValue(new XmlDataElementTag("abc"), "333"),

XmlData.createGroupValue(new XmlDataElementTag("xyz"), "999"),

}

);

Using the static methods can lead to more compact and readable code:

IData complex = complexValue(

attributes(aTag("one"), "123"),

groups(

groupValue(eTag("abc"), "333"),

groupValue(eTag("xyz"), "999")

)

);

# Using XmlDataMap

The XmlDataMap helper class allows a Java program to easily update an XmlData tree while preserving the content model as specified by XmlData Document Types.

In order to use XmlDataMaps, the application must also use XmlData Document Types. An application creates a map by associating an XmlData instance with a Document Type. The Document Type defines the structure of the XmlData instance (much like a traditional Document Type describes the contents of the pipeline in a Flow Service). After defining one or more maps, the application can then invoke a method such as getValue() on a source map in order to retrieve a value, and then invoke a method such as setValue() on a destination to tore the value.

## Obtaining Document Type references

The XmlDataWorkspace provides methods for obtaining references to XmlData Document Types based on their role or by raw namespace name. The class is located in Java package com.wm.app.b2b.server.ns. There are two types of Workspace. The permanent workspace accesses namespace Objects that are permanently persisted in the server namespace. The temporary namespace allows transient namespace objects to be created without being formalized in the server namespace.

The following code fragment obtains read-only access to the Server’s workspace:

XmlDataWorkspace ws = XmlDataWorkspace.getServerWorkspace()

While this code fragment creates a red-write temporary workspace:

XmlDataWorkspace ws = XmlDataWorkspace.getTempWorkspace()

The Flow service xmldatasamples.map:runFirst illustrates the basic use of the workspace. The schema in pub/samples/map/schema.xsd has a very simple infoset description.

<xsd:element name="alpha" >

<xsd:complexType>

<xsd:sequence>

<xsd:element name="one" type="xsd:string" />

<xsd:element name="two" type="xsd:string" />

<xsd:element name="three" >

<xsd:complexType>

<xsd:sequence>

<xsd:element name="able" type="xsd:string" />

<xsd:element name="baker" type="xsd:string" />

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

There is a global element named “alpha” with three children “one”, “two”, and “three”. The child “three” has two its own children “able” and “baker”. This is schema is used to generate the Document Types in folder xmldatasamples.map.sampleMaps (the XML namespace is sampleMaps).

An XmlData infoset is created from input file first.xml.

<alpha xmlns="sampleMaps">

<one>aaa</one>

<two>bbb</two>

<three>

<able>ttt</able>

<baker>five</baker>

</three>

</alpha>

The Java service xmldatasamples.map:first illustrate the use of workspaces. A workspace is created and it uses “xmldatasamples.map” as its base folder because that is where the XmlData Document Types were created.

XmlDataWorkspace ws = XmlDataWorkspace.getServerWorkspace();

ws.setFolder("xmldatasamples.map");

Next, the wrapper Document Type for “alpha” is retrieved. A wrapper Document Type has a single element ref to a global element. It describes the top level of an XML infoset because an XML infoset has a single document element at the top level. The wrapper has a single ref to element “alpha” and so describes the XML infoset. The create() method establishes an association between the input XML infoset and the wrapper Document type. It binds the Document Type to the IData instance.

NSField rec = ws.getWrapper(new XmlDataElementTag("alpha", "sampleMaps"));

// Create a map association between the wrapper for “alpha” and the input infoset.

XmlDataMap map = XmlDataMap.create(input, (NSRecord) rec);

After the Document Type has been bound to the IData instance, various operations are possible using the names of the child content; there is no need to obtain a cursor and navigate to the desired element, or handle the situation in which the desired element does not exist and must be created. Any operation on the map is relative to this binding (the association between the input IData and the smapleNames:alpha wrapper Document Type.

When getValue() is invoked for the “alpha” element, it’s content is returned.

Object answer = map.getValue(new XmlDataElementTag("alpha", "sampleMaps"));

The sample program assigns this value to pipeline variable “answer” and it can be seen that the entire infoset associated with “alpha” is returned.



Next, the sample invokes the map() method on “alpha” which binds its children to the Document Type that defines alpha. The getValue() invoked for child “one” returns its value and it is assigned to pipeline variable answer2 where the value is viewed and is “aaa”.

Next invoke map() on “three” which binds the value of that element with the Document Type that describes it. Then invoke getValue() on the element “baker” and assign its value to answer3 and observe in the output that its value is “five”. When the method parent() on the map, we return to the previous level, where we had mapped the children of “alpha”. If we invoke getValue() for element “two” we observe that its value is “bbb” and it is assigned to pipeline variable answer4.



# Using Paths

Just as using a cursor to navigate through a nested XmlData structure can be very tedious, so can mapping through several levels of Document Types using maps. Just as the tree cursor facilitates simplified access of nested XmlData, XmlDataPaths can make it easier to map through nested Document Types. An XmlDataPath is simply an array of XmlDataContentTags. When a method supports paths (arrays of tags), the method treats each tag as a step in navigating to the item that is set or retrieved.

There is a helper class XmlDataPath that can assist in the creation of paths. There is also a constructor that can use a script-like notation to construct the path from a String specification.

The sample Java service xmldatasamples.path:first illustrates the basic use of paths. The schema is slightly expanded from the map sample, and another level of children is added.

<xsd:element name="alpha" >

<xsd:complexType>

<xsd:sequence>

<xsd:element name="one" type="xsd:string" />

<xsd:element name="two" type="xsd:string" />

<xsd:element name="three" >

<xsd:complexType>

<xsd:sequence>

<xsd:element name="able" type="xsd:string" />

<xsd:element name="baker" type="xsd:string" />

<xsd:element name="charlie">

<xsd:complexType>

<xsd:sequence>

<xsd:element name="electron" type="xsd:string" />

<xsd:element name="proton" type="xsd:string" />

<xsd:element name="quark" type="xsd:string" />

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

The XML file is also augmented with the extra children.

<alpha xmlns="sampleMaps">

<one>aaa</one>

<two>bbb</two>

<three>

<able>ttt</able>

<baker>five</baker>

<charlie>

<electron>green</electron>

<proton>blue</proton>

<quark>mauve</quark>

</charlie>

</three>

</alpha>

Rather than invoking the map() method multiple times to navigate down the infoset tree and then invoking the getValue() method, the map() method is invoked with an array of XmlDataContentTgas, rather than a single tag.

map.map(new XmlDataContentTag[]{

new XmlDataElementTag("alpha", "sampleMaps"),

new XmlDataElementTag("three", "sampleMaps"),

new XmlDataElementTag("charlie", "sampleMaps")

});

The getValue() method is invoked on the “quark” child at the current map level, and the value is assigned to pipeline variable “answer” .

ValuesEmulator.put(pipeline, "answer", map.getValue(new XmlDataElementTag("quark", "sampleMaps")));

The sample Java service xmldatasamples.path:second illustrates the use of a script-like specification for a path. It also illustrates that methods like getValue() and setValue() can also employ paths. Unlike the map() method whose path ends at a Document Type (because only Documents can be mapped), methods like getValue() and setValue() can include the “leaf” name (in this case “quark”) because they are operating on a value that is mapped by the preceding map steps.

Object o = map.getValue("{ns sampleMaps}/ns:alpha/ns:three/ns:charlie/ns:quark");

The path is specified as a String. The path String has two components: a namespace declaration, and asequence of map steps. The namespace declaration “{ns sampleMaps}” and specifies that the “ns” prefix is used in the path steps to specify the “sampleMaps” namespace. The sequence of map steps “ns:alpha/ns:three/ns:charlie/ns:quark” specifies the successive map steps are need to navigate to the value. When no prefix is used, the element tag is not namespace qualified.

## XmlNamespaceMaps

A namespace map is simply a specialized HashMap that establishes a one-to-one correspondence between prefixes and namespaces. These maps can be used in the construction of XmlDataMaps and workspaces that contain transient XmlData Document Types. The formal XmlNamespaceMap is alternative to using the literal form of the namespace declaration (e.g. “{prefix namespace}”) that is also supported by XmlDataMaps and the XmlDataWorkspace. It’s strictly a matter of personal preference and convenience whether to use the literal form of namespace declaration or formal XmlNamespaceMaps.

The XmlNamespace map is similar to the HashMap with similar methods. The key is used as prefix and value is used as namespace. There are some additional methods:

* getKey(namespace) -- returns the prefix associated with a namespace. If the namespace is not yet defined, a prefix is generated and associated with the namespace, and returned.
* getDeclaredKey(namespace) – returns the prefix associated with the namespace. Returns null if the prefix is undefined.
* putValue(namespace) – puts the namespace into the map and generates a unique prefix if namespace is not defined
* removeValue(namespace) – remove key/value pair for the specified namespace.
* createFromPairs(string) – creates from key/value pairs specified as a string
* createFromList{string[]) – create from string array containing alternating keys and values
* getAsPairs() – returns a string of key/value pairs
* getAsDeclaration – return as a string of key/value pairs enclosed in { braces }.

The relationship between key and value is one to one. Attempting to add more than one value for a particular key results in an Exception.

# Using Workspaces

In a previous section, the XmlDataWorkspace was used to retrieve Document Type definitions from the server namespace. These definitions were used in conjunction with XmlDataMap to retrieve and update the content of an XmlData structure. It is also possible to create ad-hoc definitions of XmlData Document Types. These temporary definitions can be used with XmlDataMap just as the permanent definitions. At present, there is no ability for temporary definitions to reference permanent definitions.

## Builder methods

Various methods are invoked on the workspace in order to create definitions in a temporary namespace maintain in the workspace. These methods include:

* all() – create an **all** content model
* any() – create an **any** particle
* anyType() – create an **anyType** content model
* attribute() – create a local or global **attribute**
* attributeList() – create a list of **attribute** definitions
* attributeRef() – create an **attribute** **ref**
* choice() – create a **choice** content model
* complexType() – create a named or anonymous **complexType**
* complexTypeRef() – create a **ref** to a named **complexType**
* element() – create a local or global **element**
* elementRef() – create a **ref** to a global **element**
* sequence() – create a **sequence** content model
* simpleType() – create a simpleType; currently limited to a String
* substitution() – create a **substitutionGroup**
* wrapper() – create a root wrapper Document for an element; the root wrapper is a Document Type containing a single **element** **ref** to a global **element**

## Accessor methods

There are a variety of methods used to address members of a workspace.

* getAttribute()
* getAttributeNSName()
* getElement()
* getElementNSName
* getFolder()
* getNodeByFullName() – find a namespace object using its full name. This allows access to nodes that might not use the standard naming convention.
* getWrapper()
* getWrapperNSname()
* getType()
* getTypeNSName()
* sertFolder()

## The schema script compiler

The workspace also provides methods that can create a collection of XmlData Document Types from a script specification (see the JavaDoc for the XmlDataSchemaConverter class in package com.wm.xmldata.xmldocumenttype). The specification is a String containing keywords and values in a hierarchical structure. The following sample is presented a series of indented Strings, but the actual schema argument is a single String in which whitespace is ignored.

{x org.xml.samples w webmethods.com}

element{x:invoice complexType{

sequence{w:name w:address}

}

wrappers{x:invoice}

The first line establishes two namespace prefixes. The element token declares a global element “invoice” in namespace “org.xml.samples”. Its content is complex with a sequence containing two elements “name” and “address” from namespace “webmethods.com”. The elements default to simpleType. The wrappers token is a directive that specifies a wrapper Document Type for x:invoice. The wrapper document is simply a Document Type containing a single element ref. The wrapper maps the XML infoset for a complete document (i.e. an XML infoset contains a single document element, and the corresponding wrapper contains a single element ref that maps the document element).

# The XmlDataCopy helper

The XmlDataCopy class is a helper class that provides a simple means of copying multiple XmlData structures using a symbolic name iterator or index iterator. Using the copy helper can often eliminate the need to write code that iterates over names or indexes. A new type of content-tag is introduced: XmlDataIteratorTag. The iterator-tag is a symbolic place-holder used to represent a wildcard name or a wildcard index while copying one XmlData structure to another. The iterator-tag is used in place of other content-tags when specifying a path for XmlDataCopy class. The presence of an iterator-tag causes XmlDataCopy to iterate over all instances that match the tag in the copy input path. Multiple iterators may be present. For each input instance that matches the iterator, , XmlDataCopy finds the corresponding output item that matches the output path. Any iterator-tags in the output path are substituted with the value of the same-named iterator in the input path.

The Java service **xmldatasamples.copy:first** illustrates using the copy helper.

<xsd:complexType name="props" >

<xsd:sequence>

<xsd:element name="cost" type="xsd:string" />

<xsd:element name="weight" type="xsd:string" />

<xsd:element name="color" type="xsd:string" />

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="normal">

<xsd:sequence>

<xsd:element name="car" type="props" />

<xsd:element name="truck" type="props" />

<xsd:element name="boat" type="props" />

</xsd:sequence>

</xsd:complexType>

When the Document Types produced from this schema are used to process this XML instance:

<info>

<car>

<cost>123</cost>

<weight>xyz</weight>

<color>---</color>

</car>

<truck>

<cost>456</cost>

<weight>abc</weight>

<color>\*\*\*</color>

</truck>

<boat>

<cost>789</cost>

<weight>qrs</weight>

<color>$$$</color>

</boat>

</info>

The following XmlData is produced:



Now imagine that this structure must be inverted: The cost, weight, and boat tags are moved up in the hierarchy and car, truck, and boat tags are move down.

New schema structures are defined for the new structure:

<xsd:complexType name="vehicles">

<xsd:sequence>

<xsd:element name="car" type="xsd:string" />

<xsd:element name="truck" type="xsd:string" />

<xsd:element name="boat" type="xsd:string" />

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="inverted">

<xsd:sequence>

<xsd:element name="cost" type="vehicles" />

<xsd:element name="weight" type="vehicles" />

<xsd:element name="color" type="vehicles" />

</xsd:sequence>

</xsd:complexType>

The XmlDataCopy helper can be used to copy the old values into the new structure.

The output of the conversion service is retrieved from the pipeline using the xmlDataDocument key.

IData doc = (IData) ValuesEmulator.get(pipeline, "xmlDataDocument");

The XmlData.getValueCursor() method obtains a cursor on the data and also positioned the cursor on the first item (in this case the only item, the “info” document element). The getValue() method then retrieve the content of the “info” element.

IData input = (IData) XmlData.getValueCursor(doc).getValue();

Next, a workspace is obtained so that the relevant Document Types can be retrieved. The current folder is set to the folder for this sample.

XmlDataWorkspace ws = XmlDataWorkspace.getServerWorkspace();

ws.setFolder("xmldatasamples.copy");

Next, fetch the two relevant records

NSRecord recA = (NSRecord) ws.getType(new XmlDataTypeTag("normal"));

NSRecord recB = (NSRecord) ws.getType(new XmlDataTypeTag("inverted"));

Then create an empty output document

IData output = XmlData.createComplexValue();

Perform the copy using the help. The arguments are:

* input -- the input XmlData
* recA – record describing the input
* "/!a/!b" – the input path
* output – the output XmlData
* recB – record describing the output
* "/!b/!a" – the output path

The helper performs a number of steps for the invoker including creating XmlDataMaps and constructing paths, allocate an XmlDataCopy instance, and then performs the copy operation. When copy iterates over the input, it assigns the tags of the first step to the symbolic tag “!a” and assigns the tags of the second step to “!b”. While iterating over the output, the first step tag is retrieved from symbolic tag “!b” and the second step tag is retrieved from the symbolic tag “!a”, thus inverting the hierarchy.

XmlDataCopy.copy(input, recA, "/!a/!b", output, recB, "/!b/!a");

The input and output are assigned to pipeline variables.

ValuesEmulator.put(pipeline, "input", input);

ValuesEmulator.put(pipeline, "output", output);

The inverted output is shown:



The first example illustrated name iterators. This next example illustrates index iterators. The schema contains definitions for repeating structures.

<xsd:complexType name="typeBlist">

<xsd:sequence>

<xsd:element name="baker" type="xsd:string" maxOccurs="5" />

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name="typeAnest">

<xsd:sequence>

<xsd:element name="ableTop" maxOccurs="5" >

<xsd:complexType>

<xsd:sequence>

<xsd:element name="ableItem" type="xsd:string" />

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

</xsd:complexType>

When the Document Types produced from this schema are used to process this XML instance:

<top>

<ableTop>

<ableItem>aaa</ableItem>

</ableTop>

<ableTop>

<ableItem>bbb</ableItem>

</ableTop>

<ableTop>

<ableItem>ccc</ableItem>

</ableTop>

</top>

The following XmlData is produced:



An application might need to aggregate the individual “ableItems” into a String list. The output structure is the simple schema type:

<xsd:complexType name="typeBlist">

<xsd:sequence>

<xsd:element name="baker" type="xsd:string" maxOccurs="5" />

</xsd:sequence>

</xsd:complexType>

The first few lines of **xmldatasamples.copy:second** are identical to **copy:first** and not described here. The relevant records are retrieved.

NSRecord recA = (NSRecord) ws.getType(new XmlDataTypeTag("typeAnest"));

NSRecord recB = (NSRecord) ws.getType(new XmlDataTypeTag("typeBlist"));

Then create an empty output document

IData output = XmlData.createComplexValue();

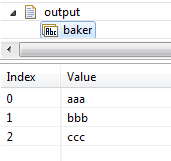
Perform the copy using the help. The arguments are:

* input -- the input XmlData
* recA – record describing the input
* "/ableTop[!x]/ableItem" – the input path
* output – the output XmlData
* recB – record describing the output
* "/baker[!x]" – the output path

When copy iterates over the input, it assigns the index of the first step “/ableTop” to the symbolic tag “!x”. The second step “/ableItem” is not indexed, but it does address the copied value. While iterating over the output, the first and only step uses the symbolic iterator index “!x” to index into the output array “baker”.

XmlDataCopy.copy(input, recA, "/ableTop[!x]/ableItem" output, recB, • "/baker[!x]"

The results are assigned to “output”.



# Using pub.xmldata:queryXMLNode

The XmlData variant of queryXMLNode is quite similar to the legacy implementation. There are additional arguments that allow the namespace and particle-id to be specified directly for each field in the output.

The top-level input to the service has an additional field: defaultNamespace. If present, the default namespace is applied to each element output field that doesn’t have an explicit namespace argument, and doesn’t employ a prefix.

The children of the top-level field argument have one additional field: childTags, which specifies the types of tags that are in the field’s children. This may be ‘attribute” or ‘element’.

The field children of the top-level field may also contain the fields ncName, namespace, particleId, and childTags. If name is used it may contain prefixed names, but if ncName is used, then name is ignored.

# Tag summary

The various tags that are described throughout this introduction have a number of characteristics that are summarized here:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tag type** | **Has a particle Id**  **(is a content-tag)** | **Can be indexed** | **Can be namespace qualified** | **Can employ an indirect-qualifier** |
| element | yes | yes | yes | no |
| attribute | no | no | yes | no |
| group | yes | yes | no | no |
| type | no | no | yes | no |
| substitution | yes | yes | yes | yes |
| any | yes | yes | no | yes |
| iterator | yes (ignored) | no | no | no |

# Glossary

any-item – an indirect-item; a specialized value used to indirectly refer to the value that corresponds to an “any” wildcard. It is an IData object with an \*ncName, \*namespace, and \*value field.

any-tag – a specialized content-tag that may be paired only with any-items. It may be constructed using the XmlDataAnyTag class. Any-tags are always paired with any-items.

attribute-tag – a tag used to name attribute values. They appear only as tags in the IData associated with \*attributes, the attribute-set. They are never prefixed with a particle-id and their associated value is always a simple String. Their String representation has no provision for an index suffix.

complex-value – an IData structure used to represent values that have attributes and/or repeating top-level group content. Attributes are represented as tag-object-pairs in an IData paired with the \*attributes tag. Complex content is represented as an IData or IData[] associated with a group-tag. Simple content is represented as a \*simple tag with simple-value pair.

content – the portion of a value that does not include the \*attributes field or top-level group-tag. It is roughly analogous the XML content of an element. It may be simple (a String), a group (an IData) or a repeating group (an IData[]). At present, XmlData does not support mixed content.

content-tag – a tag used to name the children of a group-value. Content-tags are prefixed by a particle-id because there could be more than one occurrence of the same tag in a content model. Element-tags, any-tags, substitution-tags, and group-tags are all content-tags. Attribute-tags and type-tags are not content-tags. They do not have a particle-id prefix. The XmlDataContentTag class implements the base behavior of content-tag. The XmlDataElementTag, XmlDataAnyTag, XmlDataSubstitutionTag, and XmlDataGroupTag classes extend XmlDataContentTag.

group-tag – a content-tag associated with a group-value. It is constructed using the XmlDataGroupTag class.

group-value – an IData structure used to represent content as tag-object-pairs. Group-values do not directly contain attributes. Rather, it is their parent complex-value that contains the attributes as a \*attributes field of the complex-value. The tags may be any kind of content-tag along with their associated value.

index-iterator – an iterator tag used as a placeholder for the index path of path-step.

indirect-item – a specialized IData structure that is used to indirectly refer to a value that corresponds to either a substitution group (substitution-item) or an “any” wildcard (any-item). Any-items (or arrays of any-items when repeating) are always paired with any-tags. Substitution-items (or arrays of substitution-items when repeating) are always paired with substitution-tags.

indirect-iterator – an iterator-tag used as a placeholder for the indirect-qualifier part of a path-step.

indirect-qualifier – a content-tag used to explicitly name the value-tag in an indirect-tem. Indirect-qualifiers only apply to substitution-tags and any-tags.

iterator-tag – used only in path specifications, they never appear in instance data or in Document Types. Iterator-tags are symbolic place-holders used by the XmlDataCopy helper. They indicate which parts of a path are iterated over during the copy operation. Iterator-tags always begin with the ‘!’ character. They have no index, and if specified, any particle-id is ignored

name-iterator – an iterator-tag used as a place-holder for a field name part of a path-step.

path – a String of one or more path-steps separated by the ‘/’ character. A path specifies how to navigate from a parent in the XmlData Infoset, to a child at a lower level in the tree. Paths may be constructed from arrays of XmlDataTags using the XmlDataPath helper class. Paths are used by the XmlDataMap and XmlDataCopy helpers.

path-step – a String used to describe an entry in a group-value. The path-step is of the form: {particle-id}ncName#namespace[index]{indirect-qualifier}

repeating-value – an array of values. For XML infosets that are created by conforming to XmlData Document Types, repeating values will be either String[]s or IData[]s. For XML infosets that do not conform to XmlData Document Types, or values that correspond to anyTypes, or values that corresponding to any with processContents=none, repeating values are represented as Object[]s. This is true even if the content is homogenous (e.g. all String or all IData) members).

simple-value – a String that represents the content of a simple type. Element-tags may be paired with simple-values or arrays of simple-values. Attribute-tags are always paired with simple-values.

substitution-item – a specialized value used to indirectly refer to the values that is a substitute for the head of a substitution group. Substitution-items (or arrays of substitution-items if repeating) are always pair with a substitution-tag.

substitution-tag – a specialized content-tag that may be paired only with substitution-items. It may be constructed using the XmlDataSubstitutionTag class. Substitution-tags are always paired with substitution-items.

tag-object-pair – a tag and its associated value. A pair is always a member of an IData. The IData containing the pairs is either associated with a \*attributes field, a \*group-tag, an element-tag, or possibly the \*value field of an any-item. The value may be an array of values. Arrays of values are called repeating-values and correspond to Document Type fields that have maxOccurs > 1.

type-tag – a qualified name used to identify a global type. Like attribute-tags, type-tags do not have a particle id. Also, their String representation has no provision for an index suffix.

value – a String or IData object used to represent the attributes and content of a XML element. There are three types of values: simple, group, and complex. In addition, an indirect-item holds a reference to a value and acts as a proxy for the value. Simple values are always Strings. All other value types are represented using IDatas.

value-tag – the content-tag associated with a value. Typically, the value-tag is the same as the content-tag in tag-object-pair. For example, the element-tag associated with a value is also the value-tag. However, in the case of an indirect-tem, the content-tag might be an any-tag or a substitution-tag. In this case, the value-tag is the element-tag encapsulated in the any-item or substitution-item.

XmlData – an IData object with a specific structure used to represent an XML infoset. There are two distinct models: Doctype-based and free-form. There are no XmlData instances; rather tere are IData instances that conform to the XmlData infoset model. It is also a helper class with static members used to help in the construction and manipulation of XmlData infosets.

XmlDataCursor – an Object used to navigate over the tags and values stored in a group-value

XmlDataMap – an object used to directly modify an XmlData document using path specification somewhat similar to those used by Flow Map.

XmlDataPath – a helper class used to construct paths.

XmlDataTreeCursor – an object used to navigate over a hierarchy of nested XmlData documents.

XmlDataWorkspace – an object used to reference permanent Document Types or to create temporary Document Types. It is typically used in conjunction with an XmlDataMap which use Document Types as data descriptors.

XmlNamespaceMap – an instance of this class specifies a one-to-one relationship between prefixes and namespace URIs. It can be used the XmlDataMap class and the XmlDataWorkspace class for namespace declaration.

# Implementation History

The XmlData feature consists of two distinct components: the base feature and the Java helpers feature.

## Base features

The base feature includes the ability to generate XmlData Document Types from schemas, built-in services that convert to and from the XmlData format, and Flow Map support for updating XmlData structures. The base feature has been available since Integration Server release 9.9. One small update in the 10.1 implementation changes the name of the \*content field to \*value in any-item structures. . The changed format is only an issue if an existing Floe service that updates the \*content (now \*value) field of an any-item.

## Java helpers

The Java helpers have been partially available since 9.10, but have only been fully implemented since release 10.1. There are some differences between the preliminary 9.10 release and the final 10.1 release. Java programs written for the 9.10, 9.12, and 10.0 preliminary releases will require change to run in the 10.1 environment.

The Java helpers in release 9.10 included the Java classes: XmlData, XmlDataCursor, XmlDataElementTag, XmlDataGroupTag, XmlDataAttributeTag, XmlDataSubstitutionTag, and XmlDataAnyTag. All of these classes were in package com.wm.data. These helpers were of a very tentative nature.

Many incompatible changes to the helpers were introduced in release 9.12 (some existing methods implemented new behaviors, some existing methods were renamed). The previous version of these implementations remained available in com.wm.data.obsolete, while the new implementations were available in com.wm.data. Any Java code written for the 9.10 release may be compiled and run on release 9.12 or 10.0 by importing the obsolete versions of XmlData and XmlDataCursor in com.wm.data.obsolete. Dong so, results in the 9.10 behavior.

Release 10.0 implementations in com.wm.data remain unchanged from 9.12.

Release 10.1 included a major restructuring of the Java helpers and many new features such as XmlDataTreeCursor, XmlDataMap, and XmlDataCopy. It would have been very complex and very confusing to preserve compatibility between the tentative early releases and the final 10.1 release. In addition, the XmlData helpers have been moved to their own package com.wm.xmldata. In order to run code written for the 9.10, 9.12 and 10.0 releases, it is necessary to change the import from com.wm.data to com.wm.xmldata. In addition, some methods have been renamed in going from release 10 to release 10.1.