talend | Connecting the Data-Driven Enterprise

Talend Data Mapper

User Guide

6.4.1

Adapted for v6.4.1. Supersedes previous releases.

Publication date: June 29, 2017

Notices

End User License Agreement

Table of Contents

Preface

1. General information

1.1. Purpose

This User Guide explains how to manage *Talend Data Mapper* functions in a normal operational context.

Information presented in this document applies to *Talend Data Mapper* **6.4.1**.

1.2. Audience

This guide is for users and administrators of *Talend Data Mapper*.

The layout of GUI screens provided in this document may vary slightly from your actual GUI.

1.3. Typographical conventions

This guide uses the following typographical conventions:

- text in **bold:** window and dialog box buttons and fields, keyboard keys, menus, and menu and options,
- text in **[bold]:** window, wizard, and dialog box titles,
- text in `courier`: system parameters typed in by the user,
- text in *italics*: file, schema, column, row, and variable names,
- The  icon indicates an item that provides additional information about an important point. It is also used to add comments related to a table or a figure,
- The  icon indicates a message that gives information about the execution requirements or recommendation type. It is also used to refer to situations or information the end-user needs to be aware of or pay special attention to.

2. Feedback and Support

Your feedback is valuable. Do not hesitate to give your input, make suggestions or requests regarding this documentation or product and find support from the **Talend** team, on **Talend Community** at:

https://community.talend.com/

Chapter 1. Overview

1.1. GUI of Talend Data Mapper

The *Talend Data Mapper* allows you to define and execute transformations (called maps) between data records or documents. For the purposes of this product, a document is the same thing as a data record.

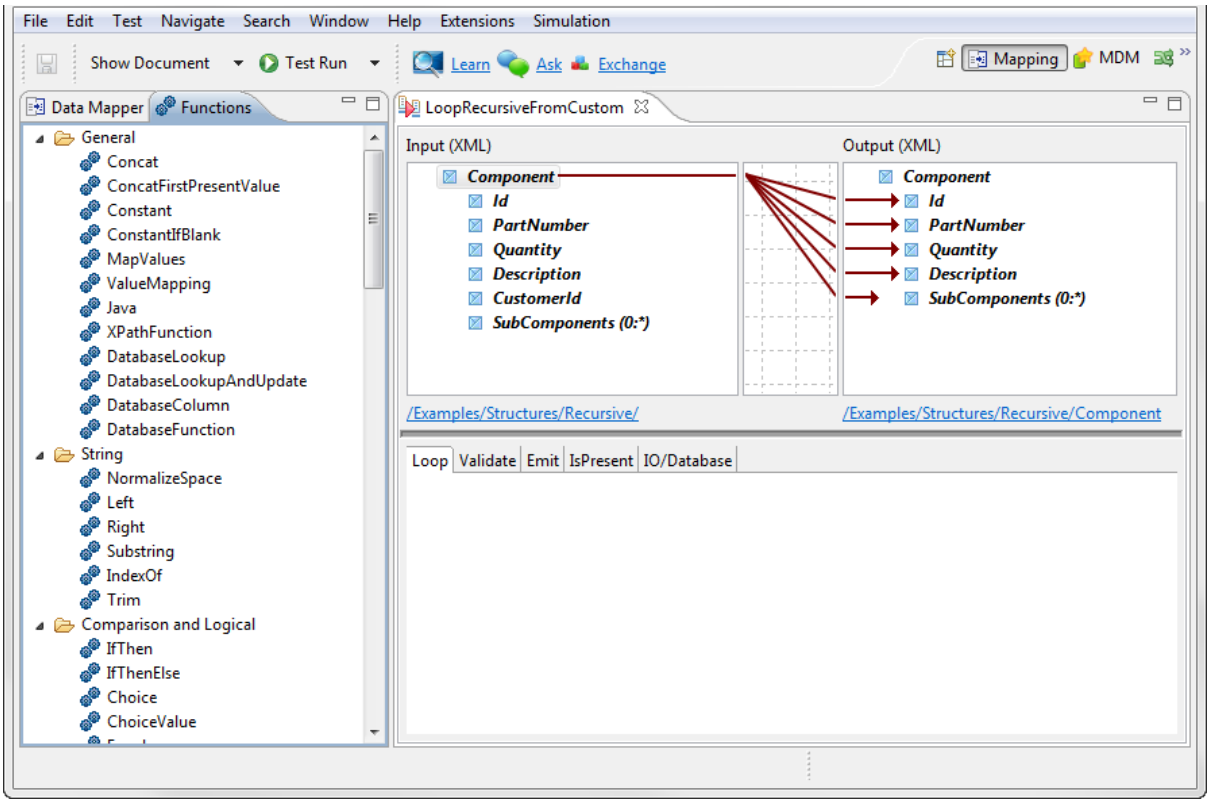
The *Talend Data Mapper* has three main parts:

- **Repository navigator** - The **Repository navigator** is a tree view in the left pane consisting of structures, maps, and namespace containers. To view one of these workspace objects, select it in the browser and open it.

  In the *Talend Studio*, the **Repository navigator** is located under the **Hierarchical Mapper** node, which can be found in the **Data Mapper** tab in the **Mapping** perspective; and in the **Repository** tab in the **Integration** and **Mediation** perspectives, under the **Metadata** node. When *Talend Data Mapper* is running in Eclipse without the *Talend Studio*, it is located in the **Project Explorer**.

- **Functions** - The **Functions** tab in the left pane has all of the functions that can be used to create expressions.

- **Editor Area** - The editor area contains an editor for each structure, map, or namespace container that is open. You can close an editor by clicking the **[X]** icon on the editor's tab.



To create a mapping, you drag and drop data from the input document to the output document. For more sophisticated mappings, you can create an expression that specifies the value of the output element. The expression is built from functions that operate on input (or output) values.

At any time, you can view all or a portion of your mapping using a sample document. You can also view all or a portion of the sample document. In addition, a powerful search capability is included that allows you to look for any text in your transformation maps, expressions, or document definitions.

You can import document definitions, also called structures, from specification files (like XML Schema, Guideline XML, or examples), or you can enter them manually. Structures can refer to other structures, allowing any part of a document definition to be reused. Furthermore, to handle cases where a structure is derived from another structure (like a specific implementation guideline of an EDI or XML specification), a structure can inherit from another structure.

Structures can have example document instances associated with them, which you can edit directly. Example document instances are also used when testing maps associated with the structure.

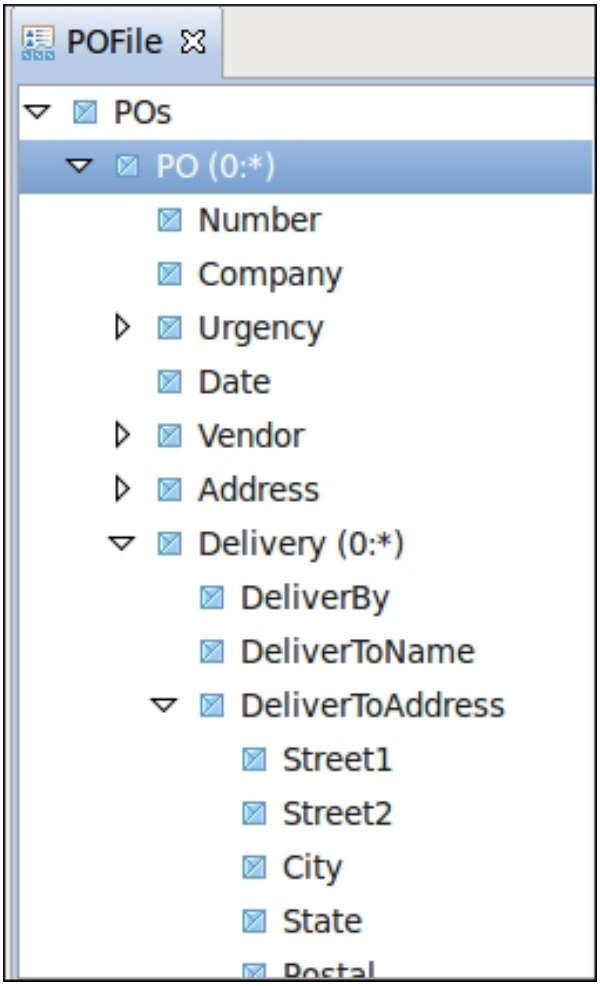
You can use various runtime projects (provided separately) in the environment where you can execute maps independently of the studio.

Chapter 2. Concepts

This chapter introduces some concepts necessary to understand how to use the studio.

2.1. Structure

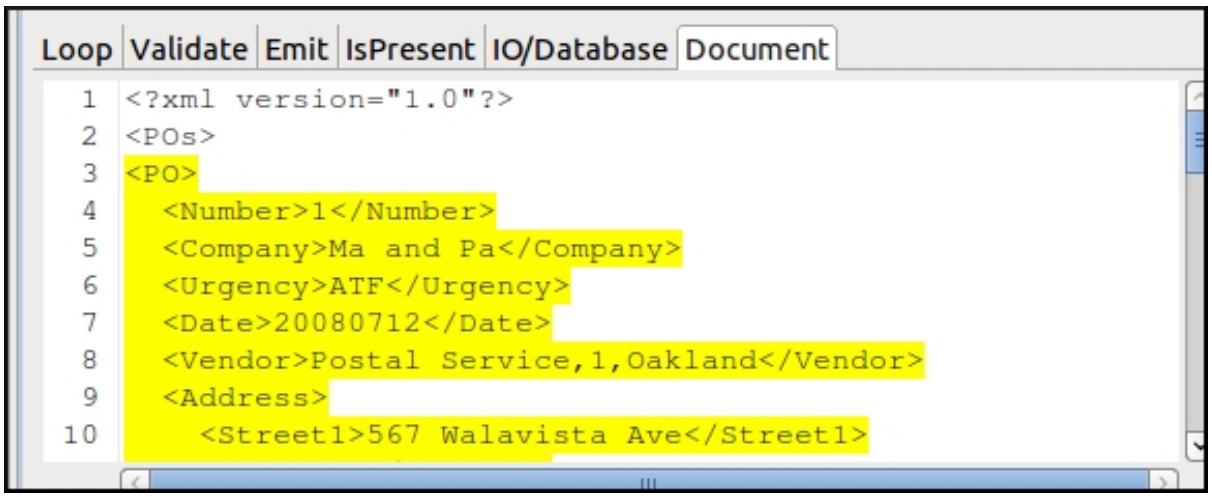
The structure mechanism defines the semantics of all types of data, including Java objects, EDI, XML, flat (delimited or positional) and database. Like an XML document, a structure contains a hierarchy of elements with one element as the root. An element of a structure is conceptually the same as an element in an XML document, and when the structure represents an XML document, structure elements correspond exactly with XML elements.



Typically, you create a structure by importing definitions created by other programs, such as a CSV Sample, XML Schema, or Guideline XML (gXML). You can also create and maintain structures manually.

To reuse definitions, structures can refer to other structures. This is called structure inheritance. For example, you could have one structure that defines an address. You could then create another structure that uses that address structure, such as a purchase order structure with both a bill-to and ship-to address. To use the address structure, you would simply drag it to the bill-to and ship-to elements in the purchase order structure. The structure inheritance mechanism is automatically used when creating structures from other specifications like XML Schema.

Structures can also refer to sample document instances, which can be used as test input documents when executing maps or as sample output documents to help define a map.



A structure can have multiple representations, which define the issues of conveying the structure's data in specific forms, such as Java Object, XML, and flat file. For example, a structure may be defined to be a certain flat file definition which can have two representations, one for XML and the other for the flat file.

2.2. Representation

A representation contains the information needed to manifest a structure for a specific concrete type of document, for example XML or Java.

Each structure can have an XML representation and one other non-XML representation. If there are no representations, XML is assumed.

2.3. Element

An element is a data field, that is, a portion of a structure. Elements in a structure correspond generally to elements in XML documents and columns in databases. Each element is associated with information like the length of the field, the number of occurrences, and its position in a record.

Elements are associated with both structures and maps (the elements in maps are always automatically derived from the elements in the structure). Each structure has a single root element that contains all other elements, and like an XML document, each element may contain other elements or just contain character data.

An XML attribute is defined as an element.

Code values are defined as specially marked elements. Typically structure inheritance is used to group a set of code values into a code table structure, and this code table structure is inherited by the required elements.

2.4. Map

A map is the definition of a transformation between two structures. A map can have one input structure and one output structure, and its elements correspond exactly to the elements associated with the structures of the map. A single map can process any number of input or output documents using the explicit document Input/Output mechanism.

Certain maps may have only an input structure, such as when a map is used for document validation.

The map contains the expressions that are used to create the output document and validate the input document. These expressions generate the value of each output element, define the rules for handling looping, define the rules for input element validation, and can constrain the creation/reading of output/input elements.

Maps can inherit from other maps, allowing you to create a base map that defines typical transformations between two types of documents and then customize this map without duplicating all of the transformations in the base map.

Maps that provide for expressions to translate an input to an output structure are called standard maps. There are other special purpose maps that perform functions other than translation. For example, an XSLT Report Map can run an XSLT script processing data from an input structure.

2.5. Function

A function is a unit of code that accepts a fixed or variable number of arguments and returns a single value. Functions are used to create expressions. The designer has many built-in functions.

2.6. Expression

An expression is an instance of a function or a reference to a map element used to define a transformation. Whereas a function defines an action (like adding two numbers), an expression is the use of that function as part of defining the value for an output map element, for example.

Expressions are arranged in trees where the result of each expression becomes an argument value for the parent expression. Expression trees are used to define the values for output map elements. The root expression of the output value expression tree is the value of the output map element.

2.7. Looping

If an element can occur multiple times in a single object, you set up looping for that element. The loop is an expression tree that defines how the loop for that element is handled in the output document. Typically, this is automatically calculated when you reference elements from the input document, and, in most cases, the element loops corresponding to some input element.

Chapter 3. Getting Started

3.1. Getting started with Talend Data Mapper

Here are the basic steps to map one type of document to another assuming you are starting from scratch. You can also go through the cheat sheets (tutorials) by going to **Help** > **Cheat Sheets** and selecting **Data Mapper**.

1. Import or Create the structure definitions by right-clicking in the **Repository navigator** and choosing **New** > **Structure**. Possible sources of structures are:

   a. **XML Schema (XSD)/DTD/XML Sample/WSDL** - Import any of these to process XML documents. See XML Representation for more details.

   b. **CSV Sample** - Import a CSV sample document. See Flat Representation for more details.

   c. **Manual** - You can manually create a structure by adding elements in the structure editor. Typically used for flat (delimited/positional) data. See Flat Representation for more details.

   d. **COBOL** - Import a COBOL copybook. See COBOL Considerations for more details.

   e. **Database** (deprecated) - Import table definitions directly from a database. See Database for more details.

   f. **Java** - Import Java classes or Jar files. See Java Representation for more details.

   g. **EDI** - Import EDI Guideline XML (gXML) files. See EDI Representation for more details.

   h. **Projects** - Import a previously exported project. Use the Import menu item for this.

2. Set up the map by right-clicking **New** > **Map** in the **Repository navigator** which executes the map creation wizard. Once the map is created, drag the structure corresponding to the input document to the input panel of the map's editor, and drag the structure corresponding to the output document to the output panel of the map's editor.

3. Map elements from the input panel to the output panel by dragging an input element to an output element. This causes the input element value to be copied to the output element value.

   Note that after you have dropped the input element to the output element, the output element is selected and the **Output Value** panel contains *Concat* followed by the name of the input element. This indicates the output element is created by copying the specified input element.

4. Test your transformation by choosing **Test** > **Test Run** specifying a test document. The output is displayed on the screen

There are also extensive Examples.

Chapter 4. Examples

Extensive examples are provided with the studio. These are in the *Examples* project which is a read only project that always appears in the workspace. It is automatically updated with updates to the studio (you may need to refresh it after you update the software by right-clicking the **Examples** project and selecting **Refresh**).

Since the examples project is read only, it's most effective to work with a copy where you can freely modify things. You can use a cheat sheet to copy the examples to another project.

In addition, further scenarios showing *Talend Data Mapper* in use can be found in the *Talend Components Reference Guide*, in the documentation for the data mapping components.

4.1. XML Examples

- *Maps/PayPal/POPayPal_PO2* - A simple example of a mapping of an XML structure (that actually corresponds exactly to the PayPal CSV data) to a different type of XML structure representing a simple purchase order.

4.2. Looping Examples

There are examples that correspond to all of the cases described in the Looping Cook Book. They are found in the **Maps/Looping** folder in the *Examples* project, and the *Looping Cook Book* section has the links to the examples corresponding to each case.

4.3. Java Examples

- *Maps/Java/AcmePOToJava* - Shows the mapping of a set of XML purchase orders to a corresponding set of Java objects representing invoices.
- *Maps/Java/JavaInvoiceToXML* - Shows the translation of the above Java objects that represent invoices into an identically corresponding XML. Note the output structure in this example, *Structures/Java/XML/XMLInvoices* actually inherits from the same structure that defines the Java invoices, *Structures/Java/Invoice*.

4.4. Flat (including CSV) Examples

- *Maps/CSV/POPayPalCsv_PO2* - A simple example of a mapping of CSV data of a PayPal purchase into an XML structure that defines a simple purchase order.
- *Maps/Simple/AcmePOToFlat* - A mapping of the Acme purchase order XML to a corresponding binary flat file. The structure of the flat file has a tag in the first four bytes that indicates the type of data, with PO indicating the beginning of a new purchase order, DEL a delivery, and ITEM an item. Note that the structure definitions are identical as shown in the map. If you look at the output structure (*Structures/Simple/AcmeRetailer/AcmePOFlat*) you can see that the elements have delimited initiators, for example the */POs/PO* element, so that the flat file contains tags that show the type of each record.

4.5. Multi-Representation Examples

- The *Maps/MultiInputOutput/PurchaseOrdersToMandP* map shows use of multiple representations within a single map.

  A common use case is processing multiple unrelated XML documents in a single input. In some cases you want to concatenate the documents, in others you might select a document based on what content is actually provided in the input.

  This example shows processing of two existing structures representing different XML documents and automatically choosing between them. This is accomplished by having creating an enclosing structure (the map's input structure) which has an element that inherits from each of the desired XML structures. The enclosing structure uses the *Flat* representation and has a choice (note the group type of the *Root* element). Each of the

members of a choice specifies an initiator string that is the initial part of the element of the member's XML document. A **ReadNested** function signals that the representation is to change to that of the enclosing structure (by default the representation does not need to be specified on the properties of the **ReadNested**).

This example also shows the use of Map Inheritance where the from the *AcmePOToMaAndPa* map is used between the *POFile* structure and the *MandPPOFile* structure. This allows you to build and test a map between the two XML documents and then completely reuse the map in other cases (like this one).

- Another example shows embedding flat content inside of XML. The */Examples/Maps/Simple/POMap example* illustrates this. If you look at the *POs/PO/Vendor* element, you can see that it has a **ReadNested** function as its I/O expression.

4.6. Report Example

The *Maps/Reports/POToHtml* report map shows the execution of a report map which runs an XSLT stylesheet against the input. This example shows the creation of HTML from one of the example purchase order files.

Chapter 5. General Studio Information

This chapter discusses general topics of the studio which apply to all types of objects.

5.1. General

The studio shares interface elements across all perspectives that allow you, for instance, to switch between perspectives, import and export items, or access help contents.

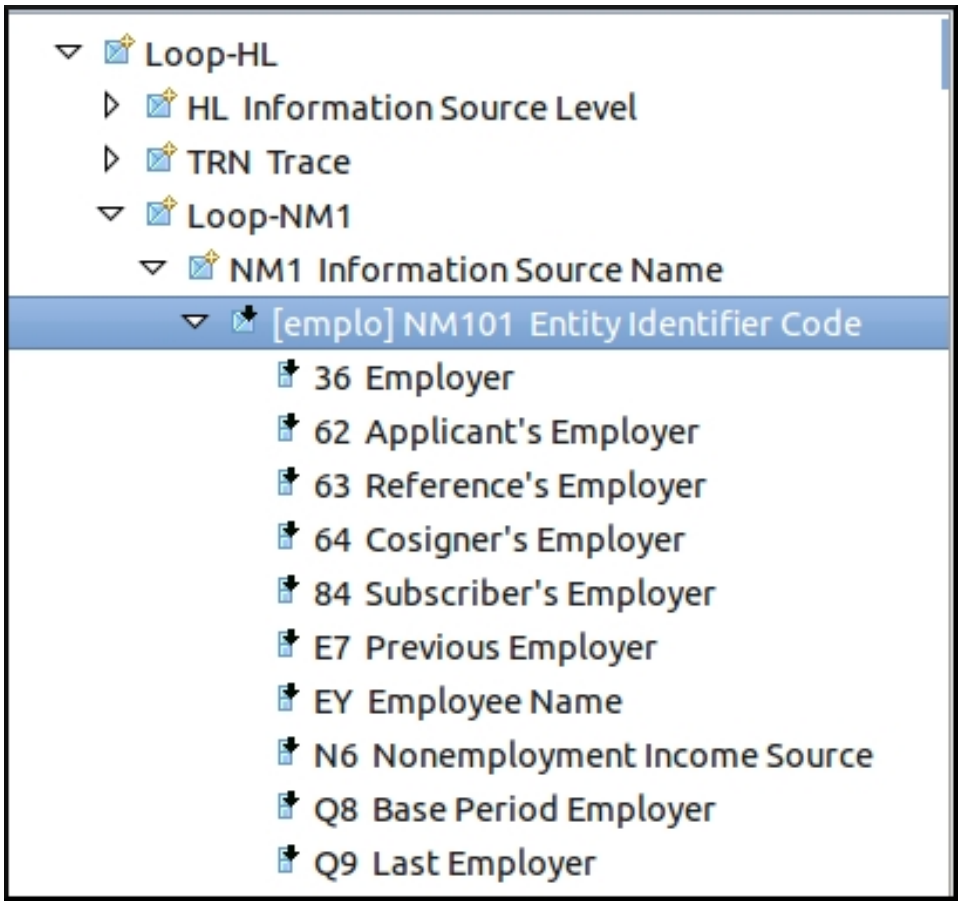
The following sections describe interface elements that are specific to studio.

5.1.1. Working with Trees

Trees provide an intuitive way of working and are used extensively in the studio. You can drag and drop nodes (structures and elements) in the tree, or cut/copy/paste nodes to the same tree or other trees, such as copying a collection of elements from one structure editor to another.

You can create a new node simply by right-clicking and selecting the new node type. When creating a new node, the cursor is positioned directly in the space where the new node appears so you can type in the name of the new node. If the name you specify is a duplicate or otherwise invalid (depending on the context), an error dialog will prompt you to enter a new name.

You can filter child elements in a tree by typing into the parent element. This is useful for situations where there are many child elements and you need to see only the elements whose name or description matches certain text. Each character typed is shown in brackets, for example typing *empl* would show *[empl]* in the parent element and then only children with the string *empl* contained in the element name or description would be shown. Use the **backspace** key to remove a single character, and use the **escape** key to remove the filter entirely.



5.1.2. Undo/Redo

You can undo any action you performed on a tree, including changing a property or deleting a node, by simply clicking the **Undo** button. You can undo multiple operations by clicking **Undo** multiple times. You can also click **Redo** to reinstate the last undone operation. To see which operation will be undone, click the **Edit** menu.

5.1.3. Back/Forward

The **Back** and **Forward** buttons allow you to switch easily between nodes. For example, if you are viewing a node and then click another node, you can return to the previous node by clicking **Back**, and then click **Forward** to switch to the other node again. This is useful when you are comparing two nodes.

5.1.4. Drag/Drop and Cut/Copy/Paste

When you drag and drop a node, it is cut from its original position and pasted in the new position. To copy a node instead of moving it, press and hold the **Ctrl** key before you start dragging the node, and a copy of the node is pasted in the new position. Throughout this documentation, drag/drop refers to both moving and copying a node. You can also cut, copy, and paste by right-clicking a node and choosing the relevant command from the popup menu.

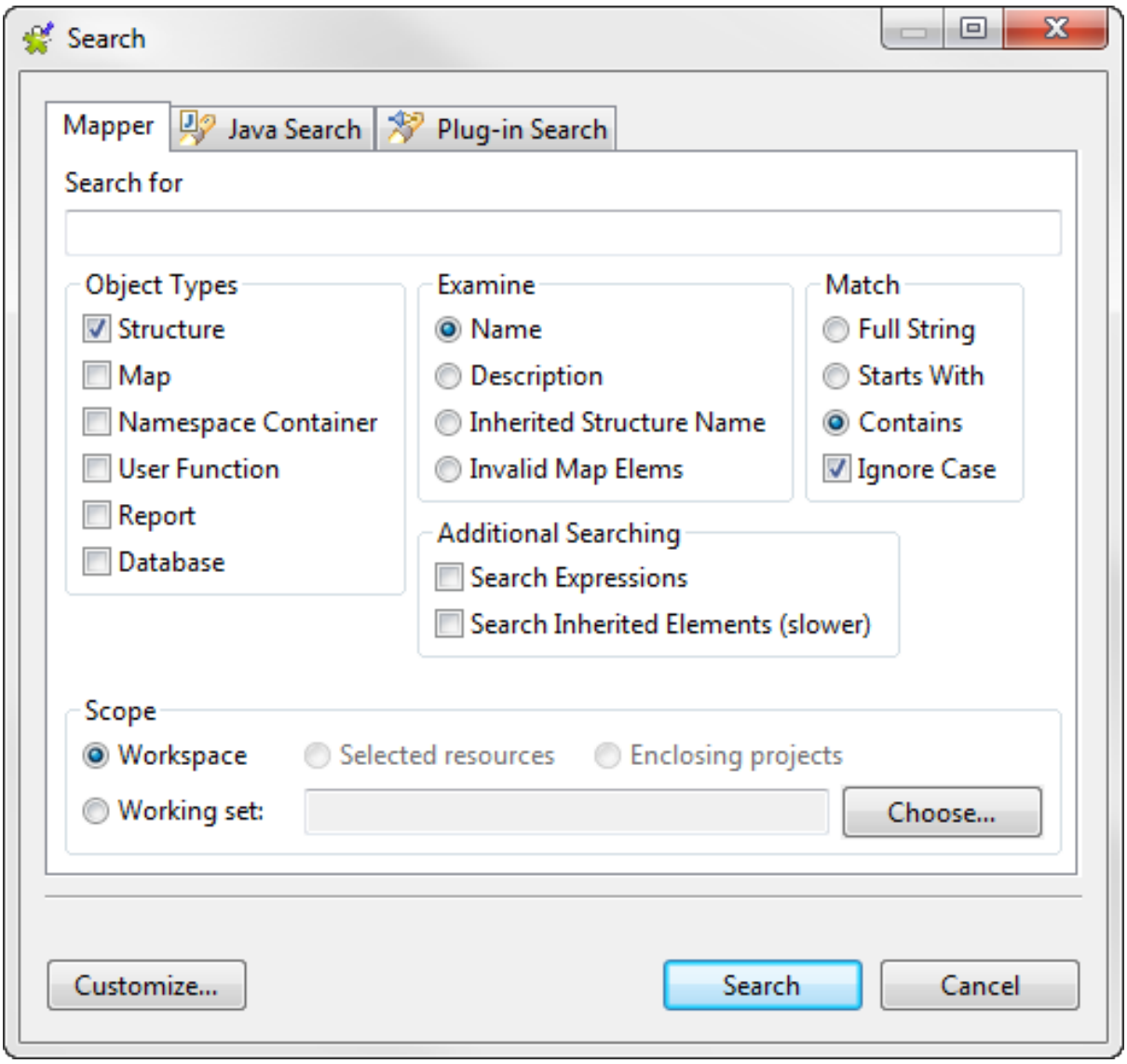
5.1.5. Properties

Most of the tree nodes have associated properties. You can access a node's properties by double-clicking the node or by right-clicking it and choosing **Properties**.

In some cases, you are not permitted to alter the properties of a node. For example, if you are viewing the properties of a map element, it is actually showing the properties of the corresponding structure element in read-only mode.

5.2. Searching

The studio provides a powerful search feature that allows you to find text in the names or descriptions of structures, maps, structure elements, map elements, and expressions. To search, choose **Search** > **Mapper...**. Following are descriptions of the various options in the **[Search]** dialog.



5.2.1. Object Types

Select one or more of the following object types to search:

- **Structure** - Search the elements and names in structures.
- **Map** - Search the elements and names in maps.
- **Namespace Container** - Search the namespaces and namespace container names.
- **User Function** - Search the user functions (in the **Functions** panel) and any expressions associated with user functions.
- **Report** - Search the report names.

5.2.2. Examine

Select one of the following:

- **Name** - Look only at names.
- **Description** - Look only at descriptions.
- **Inherited Structure Name** - Look only at elements that inherit from a structure and only at the full path name of the inherited structure. Useful to discover where a given structure is inherited from.
- **Invalid Map Elems** - Look for invalid map elements. See Validating a Map for more details on this.

5.2.3. Match

Select one of the following:

- **Full String** - Select only items whose name/description exactly matches the specified search string.
- **Starts With** - Select only items whose name/description starts with the specified search string.
- **Contains** - Select only items whose name/description contains the specified search string.

Additionally, you can click **Select Ignore Case** to ignore the case of letters when making the string comparison.

5.2.4. Additional Searching

Select one of the following:

- **Search Expressions** - Search any expression names associated with the structure and/or map elements being searched.
- **Search Inherited Elements** - By default the search considers only non-inherited elements. For example, if structure Child inherits from Parent, the elements in Child that come from the Parent structure are not searched when you search Child. If you want to search the elements inherited from Parent as well as those defined explicitly in Child, select this option.

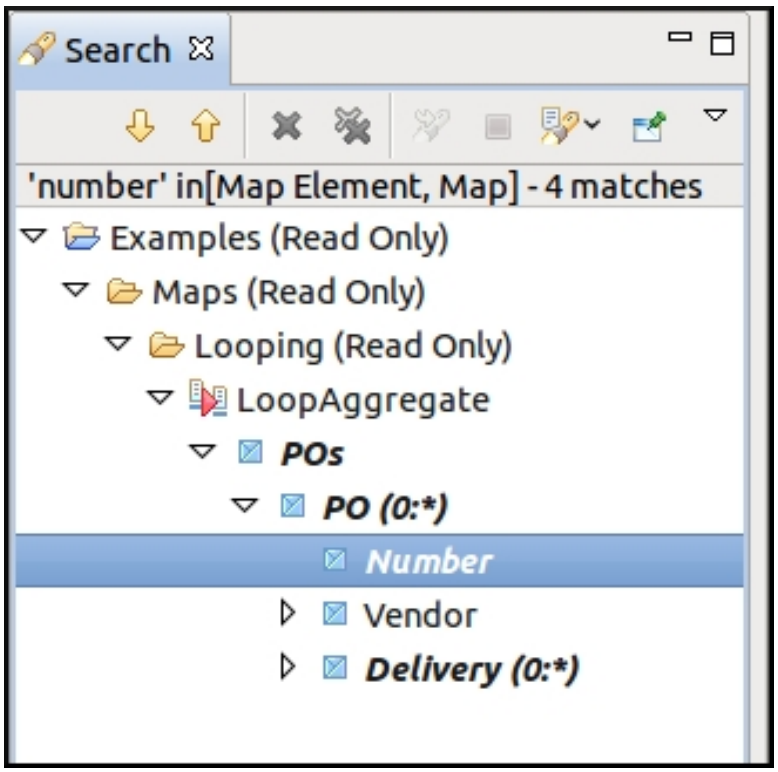
5.2.5. Scope

Select one of the following:

- **Workspace** - Search everything in the workspace (all projects).
- **Selected Resources** - Search only within the selected objects.
- **Enclosing Projects** - Search only within the project(s) enclosed by the selected objects.
- **Working Set** - Search only withing the selected working set.

5.2.6. Executing the Search

Click the **Search** button to execute the search. The results are shown in a **Search** view as shown below. If you double-click a result entry, the entry will be selected.



5.3. Dependency Management and Metadata

5.3.1. Dependency Management

Most objects (maps, structures, etc) have a dependency relationship with other objects. For example, a map depends on the structure(s) that define the input and output. These dependencies are managed automatically so that a change in a dependent object (the structure in the above example) would be reflected immediately in the map object, even without saving the structure object. The dependency management applies to all related objects, like structures that inherit from other structures, or a structure's use of a namespace container.

If, for example, you have a map editor open and one of the map's structures open in another editor and change and save the structure, when you switch back to the map editor it will detect the structure has changed and revalidate the map against the structure, reporting any problems encountered.

For performance reasons, the dependency information is tracked in separate metadata (so that it can be referenced without examining all possible objects). This metadata is automatically calculated and can be easily manually rebuilt if required.

5.3.1.1. Showing Dependency Information

Each object has a **Used By** and **Uses** menu item on the **Repository navigator** popup menu for the object. You can use these to discover the relationships between objects and quickly open the referenced object by double clicking on it in the dialog that appears.

5.3.1.2. Showing Objects Requiring Validation

Often you will change objects (mainly structures) and then you must validate and possibly change downstream dependent objects. For example, if you add a property to a Java class, you would reimport the Java classes and then you will need to open each of the maps that might use the structures that have changed to make sure the mapping information is correct and add new mappings as required.

To do this, you can use the **Show Objects Needing Validation** menu item in the **Repository navigator** popup menu for the project . Objects will show up as requiring validation until they have been opened and saved since the dependent object changed and have no warnings. So for example if you delete a property from a Java class and open the map that requires validation, if you don't remove the warning given in the map (by deleting the corresponding map element), then the map will still appear on the list of objects requiring validation.

5.3.2. Project Metadata

Maps, structures and the like are managed as resources in the workspace. These are essentially files that contain XML that defines the map or structure. In addition to these files there is a requirement for metadata that applies to the entire project. This data is kept in the project properties file for each project which is called *.settings/com.oaklandsw.base.projectProps* . The project properties file is automatically maintained and generally requires no intervention. If you are using a source control system, it's important that the project properties file be checked in when you change any objects in the project.

The project properties file consists of the following:

- **Project classpath** - The classpath associated with the project that is maintained in the project's *Data Mapper* Properties.
- **Dependency Information** - This is automatically calculated when objects are saved.
- **Folder Order Information** - The order in which the folders associated with the project are presented.

Generally, the project properties file is automatically built and updated as objects are saved. However there are some times when the dependency information might need to be rebuilt. You can do this manually by clicking on the **Rebuild Project Properties File** in the context menu associated with the project in the **Repository navigator**.

Chapter 6. Structures

6.1. Overview

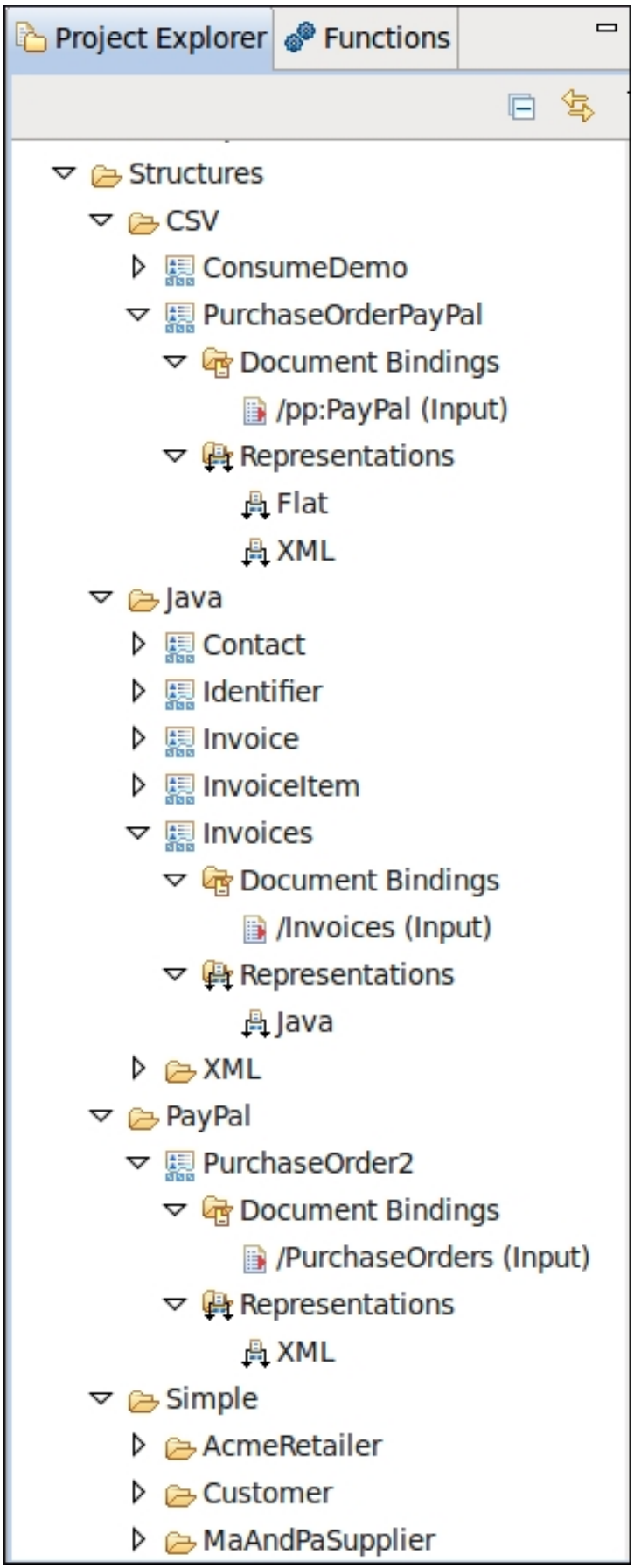
A structure contains a collection of elements that describe the contents of a type of document. You can connect structures through the inheritance mechanism, which allows an element to inherit its definition from another structure. The definition of a structure and its elements is abstract in that it is not dependent on the particular way the structure is represented in instance documents.

Generally structure definitions are created by importing some type of specification file (like XML Schema) or a sample document. You can also create elements manually, or use a map to create the definitions by mapping to the Importer Structure. This is convenient particularly for those positional structure definitions where you may have a spreadsheet that describes the elements.

Structures appear in the **Repository navigator** of the studio. Each structure has a name and can be contained in folders, similar to a file. When you expand a structure in the **Repository navigator**, you will see headings for **Document Bindings** and **Representations**.

The **Document Bindings** header contains the documents currently bound to the structure. Documents can be bound for different purposes, such as sample input, and they can also be bound to different parts of the structure if I/O functions are used and multiple document support is desired.

The **Representations** header contains the representations (sets of properties for different output formats) for the structure. You can right-click the **Representation** header to create a new representation.


Project Explorer
Functions
Structures
CSV
ConsumeDemo
PurchaseOrderPayPal
Document Bindings
/pp:PayPal (Input)
Representations
Flat
XML
Java
Contact
Identifier
Invoice
InvoiceItem
Invoices
Document Bindings
/Invoices (Input)
Representations
Java
XML
PayPal
PurchaseOrder2
Document Bindings
/PurchaseOrders (Input)
Representations
XML
Simple
AcmeRetailer
Customer
MaAndPaSupplier

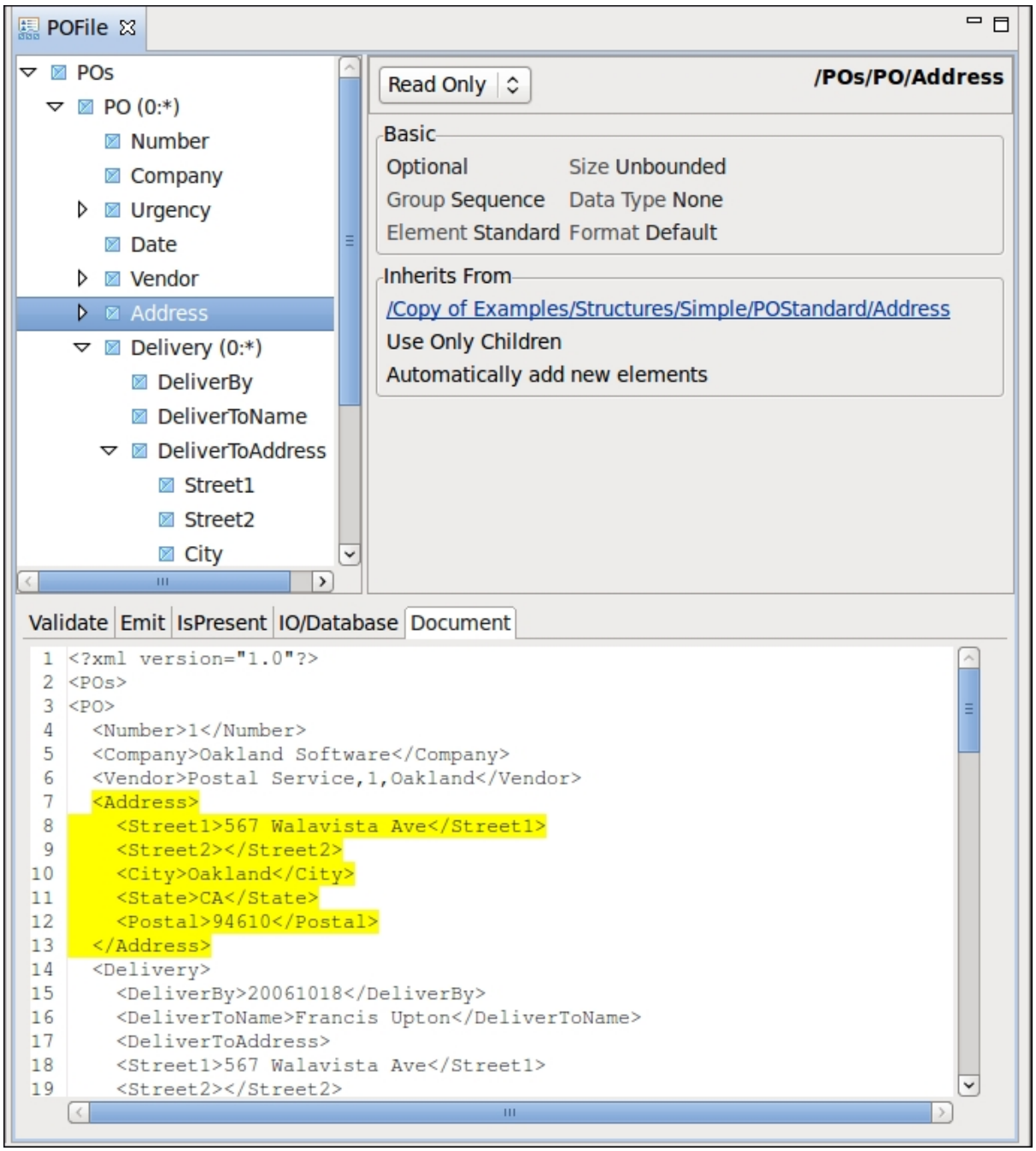
6.2. Working with Structures

To view or manipulate the contents of the structure, double-click it, or right-click it and choose **Open**. The structure editor appears, displaying the structure's elements and properties.

The top-left pane of the structure editor shows the hierarchy of elements. The top-right pane shows the properties for the currently selected element in that hierarchy. These properties are initially presented in a read-only view, which is more compact and legible. To change the element properties, you can change the view to editable by selecting it from the drop-down list.

The bottom portion of the structure editor can have the following tabs.

- **Loop** - Specifies the default loop expression to use for the selected element when mapping this structure.
- **Validate** - Specifies the default validation expression to use for the selected element when mapping this structure.
- **Emit** - Specifies the default emit expression to use for the selected element when mapping this structure.
- **IO/Database** - Specifies an input/output or database expression to use for the selected element when mapping this structure.
- **Consume** - Specifies the default consume expression to use for the selected element when mapping this structure.
- **Value** - Specifies the default value to use for this element when mapping this structure. When the structure is used as an output structure in a map, this is copied to the value expression of the map.
- **Document** - Shows a sample document that is associated with this structure. Depending on the product you are using and the preferences settings, the document can be opened automatically when the structure is opened. You can change the selected document.



6.2.1. Manipulating Elements

Elements can be added by right-clicking on an element and selecting **New Element**, which adds the element as a child. The standard tree operations apply to elements, allowing you to change their position using drag and drop or cut, copy and paste from the menus or the keyboard.

Generally when adding a new element, the properties associated with the new element are setup automatically and correctly. For example, when adding a new element to a leaf element, the group type property of the former leaf (now parent) element is automatically changed to *sequence* from *none*. The element type is changed to *none* since it now a grouping element. Sometimes you will need to go back and change these automatically set properties (suppose you want a group type of *choice* instead).

6.3. Representations

The representation defines the basic format of documents. Generally, structures have an XML representation (which can automatically work for any structure) and at most one other representation.

The following representations are supported:

- *XML* - Supports XML documents using a standard (JXerces2) parser. Import formats are XML Schema (XSD), DTD, WSDL, and an XML sample document.
- *JSON* - Supports JSON documents. You can import the JSON definitions based on a JSON sample document, and you can map JSON documents as input or output.
- *Java* - Supports Java objects (used with the runtime API). Import formats are Java classes, folders, or JAR files.
- *Avro* - Suppports the creation of structures by importing an Avro schema and the outputting of Avro files.
- *COBOL* - Suppports COBOL files in a way that has more restrictions than the flat representation but provides increased performance.
- *EDI* - Supports X12 and EDIFACT EDI. Import format is Edifecs Guideline XML (gXML) Version 1.0.
- *Flat* - Supports both positional and delimited (including CSV) flat files including binary and COBOL data types. Import formats are CSV sample and COBOL copybook.
- *Database* (deprecated) - Supports reading and writing with various databases. Import format is the database.
- *IDocs* - Supports reading and writing SAP IDoc files.
- *HL7v2* - Supports reading and writing HL7v2 documents.

6.3.1. Common Properties

These properties are common to all representations:

- **Trim whitespace on Input?** - When selecting this, any leading or trailing whitespace is removed automatically from the input data for all elements. The non-leading or trailing whitespace is not affected.
- **Pad to minimum length on output?** - When selecting this the output is padded to the minimum size with either spaces (character) or zeros (binary).
- **Import File/Version/Date** - These are read only and used only to record when the import was made and in some cases the version of the source that was imported.

6.4. Importing Structure Definitions

Typically, you create a structure by importing a definition file. The Editor supports the following import formats:

- **XML Schema (XSD)** - This can create many structures that are described by the XML schema definition.
- **XML DTD** - This can create many structures that are described by the XML data type definition.
- **Web Services Definition Language (WSDL)** - Uses the XSD portion of the WSDL file to create the structures.

- **EDI gXML 1.0** - The Edifecs format for describing structures with EDI representations. The root element of a gXML document is a *gXML* element that contains a *DocumentType* attribute. EDI documents must have the *DocumentType EDI*.
- **Database** (deprecated) - When connecting to the database, you import the database definition and create the connection properties using the database import mechanism. This creates the structures that correspond to the database tables.
- **Java class file or JAR file** - Used to create structure(s) based on a single class or set of classes contained in folders or a JAR file.
- **XML Sample document** - You can import an XML document to create a structure containing all of the elements of that document.
- **JSON Sample document** - You can import a JSON document to create a structure containing all of the elements of that document.
- **CSV** - You can import a CSV file that contains the field names in the first line to create a structure where each row contains all of the specified fields.
- **COBOL Copybook** - A COBOL record definition (or set of definitions). This creates a structure for each top-level record definition.
- **Avro schema** - You can import an Avro schema to create structures described by the schema definition.
- **SAP IDocs** - You can connect to an SAP system and import an IDoc file, then create a structure based on the content of the IDoc file.

6.4.1. Using a Map to Import Definitions

You can use a map to create a structure definition from any input. For example, suppose you have a positional structure that is described by a spreadsheet. The spreadsheet contains a list of element names, sizes, start columns, and further description. You can export this spreadsheet as a CSV file and create a map that maps the contents of the spreadsheet to the Importer structure.

The **Importer** structure is a predefined structure in the **Builtin** project (**Builtin/Structures/Executable Structures/Importer**). When it is executed in a map, the **Importer** structure can create one or more structures whose elements are defined by the mappings. To use it, create a standard map, and specify the **Importer** structure as the output structure. You can specify any input structure you like and map the elements to produce the desired elements in the structure. You can build and test your map in the studio as usual, producing the output of the map in XML for testing purposes. The **Test Run** menu will provide an additional option called **Test Run to Importer Structure**, which will actually create the structures. You can re-run the map as many times as you like, which creates the structures each time and replaces their contents.

6.5. Exporting Structures

Talend Data Mapper provides you with the possibility to export your Structures as CSV files, Java classes or Avro files, to enable you to work with the exported files in external tools.

The procedure for all three types of output file is more or less the same. Any particularlities based on the type of structure are specified below.

To export Structures as CSV files, Java classes or Avro files, do the following:

1. Click **File** > **Export** in the **Export** window that opens, expand **Data Mapper** and select **CSV Export**, **Java Export** or **Avro Export** as appropriate, then click **Next**.

2. In the window that opens, expand your project in the left-hand pane and select the **Structures** checkbox, then select the checkboxes in the right-hand window that correspond to the Structures you want to export.
3. In the **To directory** field, select the directory to which you want to export by clicking **Browse**, browsing to the target directory or clicking **Make New Folder** if the directory does not already exist, and then clicking **OK**.
4. [For Java classes] In the **Java package name** field, enter a name for the package you are going to create.
5. [For Avro files] In the **Avro namespace** field, enter a name for the Avro namespace.
6. Click **Finish** to complete the process.

6.6. Structure Elements

Each structure has a single root element that can contain any number of child elements, which correspond to XML elements. XML attributes are also treated as elements. Click an element to display its properties.

6.6.1. General Element Properties

Each element has the following general properties.

6.6.1.1. Name

The name of the element. When using XML, this is the name that appears in the XML document. Names for XML can specify a namespace prefix refering to a namespace in the structure's associated namespace container. Specify the namespace prefix in the usual XML syntax of *prefix:name*.

6.6.1.2. Description

A short description of the element. This description appears on the same line as the name of the element. If the name of the element is sufficiently descriptive, this may be omitted (as is the case with many XML element names). However, for EDI, for example, the name of the element is not descriptive, so the description is included.

6.6.1.3. Occurs

Specifies the number of times this element may occur in a document. There are two values: minimum and maximum. For example, to require that the element occur at least once, set the minimum value to *1*. If you do not want to restrict the maximum number of occurrences, set the maximum value to *-1*.

Following are the values for mandatory, optional, and loop elements:

	Minimum	**Maximum**
Mandatory	1	1
Optional	0	1
Loop	0	-1

6.6.1.4. Size

Like Occurs, Size has a minimum and maximum value, which specifies the constraints on the number of characters in the element. Either value can take *-1*, which means its size is unconstrained. By default the size specification is *-1* for the minimum and maximum. Specifying the element's size is useful if you use the validation feature.

6.6.1.5. Visible Group / Element tags do not appear in document

For grouping purposes it is often desirable to define elements that are part of the definition of the structure, but the element tags do not appear as part of the document (the element contents, i.e. the children of the element so). For example, suppose you want to specify that two XML elements repeat like this: *<a/><b/><a/><b/>*. To specify this, you create a *non-visible group* element that encloses `a` and `b`, and then set the maximum Occurs value to something greater than *1*. Your element definition would look like this:

```
abHolder (non-visible group, loops)
    a
    b
```

When you import an XML Schema, several types of non-visible group elements are created, including the *Attributes* element, which holds the XML attributes of a given XML element. Having the XML attributes grouped under a single element makes it easier to work with the structure.

This is only meaningful for representations that support the notion of tagged elements like XML and in some cases EDI.

6.6.1.6. Null

Indicates this element can have a null value. This is possible only for XML, Java and database elements. See Null Value Support for more details.

6.6.1.7. Group Type

Indicates how the children of this element are grouped.

- *None* - this element contains text only; it has no child elements that define additional structure.
- *Sequence* - all children must occur (if they occur) in the order specified.
- *Choice* - only one of the children can occur. If the element loops, a different child element can occur for each instance of the loop.
- *All* - all of the elements must occur, and they can occur in any order.

6.6.1.8. Element Type

Defines how this element is used in a document. This is mostly relevant for XML documents, in which elements can manifest as XML elements, XML attributes, text, etc.

- *Standard* - An element with no special treatment (e.g., a standard XML element). For XML documents, the value of the element refers to the value of all text included within the element, provided there are no child

elements. When you want to access mixed content, you must use an element type of XML Text to access the text between the child XML elements.

- *XML Attribute* - (XML documents only) The element appears as an XML attribute. Child elements of this type must be of type *value*.
- *XML Processing Instruction* - (XML documents only) The element appears as an XML processing instruction. Elements of this type cannot have child elements.
- *XML Text* - (XML documents only) This element accesses text values, which is necessary only when accessing text values for mixed XML content, where text is interleaved with XML elements. Elements of this type cannot have child elements.
- *Value* - This element's name is used to match the text value of its enclosing element. This is used to define possible code values for validation and mapping purposes. An element will have a child element with element type of *Value* for each possible code value. Elements of this type cannot have child elements.
- *Any* - The content of this element can be any collection of elements, which are completely unspecified. Elements of this type cannot have child elements.

6.6.1.9. Data Type

The data type defines the type of an element's text content. The data type applies only for elements with a group type of *none*. The following data types are defined:

- *String* - A character string.
- *Byte* - An unsigned 8-bit byte.
- *Short* - A signed 16-bit quantity.
- *Integer* - A signed 32-bit quantity.
- *Long* - A signed 64-bit quantity.
- *Float* - A 32-bit floating-point number.
- *Double* - A 64-bit floating-point number.
- *Date/Time* - A date and time value together. This includes the timezone information.
- *Date* - A date value.
- *Time* - A time value.
- *Duration* - A duration of time. This is in the ISO 8601 format of *PnYnMnDTnHnMnS*. It must begin with a *P*, and the remaining capital letters identify the type of the period. The capital letters can be omitted when the corresponding period is not used. Examples: *P4Y* is 4 years; *P6Y7M2D* is 6 years, 7 months, and 2 days; P30S is 30 seconds.
- *Boolean* - Either true or false.
- *Binary* - A binary value.
- *QName* - A qualified name with an optional prefix and local name. The prefix must be found in one of the namespace containers associated with the structure.

*This is presently not to be used as an element data type, it is used only in the **Constant** function.*

6.6.1.10. Data Format

The data format describes how the data type is to be concretely manifested. The *default* data format chooses the best format for the representation. For example, in the EDI representation the date data type and default format will choose the correct data format based on the date qualifier for input documents. The representation specifies the byte order to be used for the entire structure, so it is not necessary to specify the byte order data format individually. However, you can specify the byte order if you want to override it for a specific element.

The following data formats are defined (organized by the data types to which they apply):

- *Integers*
  - *Default* - Numbers are encoded as specified in the representation.
  - *Little Endian (PC)* - Numbers are encoded in binary in little endian byte format as used by the Intel processors.
  - *Big Endian* - Numbers are encoded in binary in big endian byte format.
- *Decimal*
  - *Default* - Numbers are encoded as specified in the representation.
  - *Character* - Numbers are encoded as ASCII characters.
  - *Packed Decimal* - Numbers are encoded in packed decimal.
  - *Zoned Decimal* - Number are encoded in zoned decimal.
- *Float/Double*
  - *Default* - Values are encoded using the IEEE 754 standard.
  - *IBM Floating Point* - Values are encoded using the IBM 360 mainframe standard.
- *Date and Date/Time*
  - *Default* - Dates are encoded as specified in the representation.
  - The remaining date and date/time formats use *CC* for century, *YY* for year of century, *MM* for numeric month of the year, *MMM* for text month (first 3 characters of the month in English), *WW* for week of year, *W* for weak of month, *DD* for day of month, and *DDD* for day of year. Other special designators are as noted. The week of year (*WW*) follows the ISO 8601 rules for week conversion. The date/time formats also use the abbreviations below for the *Time* type.
- *Time*
  - *Default* - Encoded according to the rules for the representation.
  - The remaining time formats use *HH* for hour, *MM* for minute, *SS* for second and *DD* for tenth of second.
- *Binary*
  - *Default* - Specifies that the data appears in binary format.
  - *Base64* - Encoded into characters with base-64 encoding. (Not currently supported)
  - *Hex* - Encoded into characters with hex encoding. (Not currently supported)
- *String*
  - *String (null term)* - A string whose value ends with a byte of *0*.

When converting dates from an encoding with no century to encodings that require a century, the Java standard rules for deriving the century are followed, which put the year in the century that begins 80 years before the current date.

6.6.1.11. Decimal Places

This is used for numeric data to indicate the number of implied decimal positions. For example, if this value is *2*, and the element contains the value *1234*, the value of the element is *12.34*.

6.6.1.12. Use Children Of

This is used to define a recursive element, which has the same content as its parent element. If an element is recursive, set the **use children of** property to the parent element.

6.6.1.13. Other Text Descriptions

The Text field allows you to provide complete documentation of the element and to facilitate capturing documentation from imported specifications like EDI. You can tag the text field with one of the following description types:

- *Comments* - Comments about the element as specified in the imported structure definition.
- *Long Description* - The long description about the element as specified in the imported structure definition.
- *Purpose* - The purpose of the element as specified in the imported structure definition.
- *Semantics* - The semantics of the element as specified in the imported structure definition.
- *User Notes* - Notes about the usage of this element.
- *Example* - Example values for this element. This is not to be confused with the use of sample/test documents that may be associated with the structure. The example values here serve only as documentation.

6.6.2. Flat Properties

The following properties are used only for flat structures.

6.6.2.1. Initiator

The initiator is the sequence of characters that signals the start of this element. The initiator can be used for an element at any level, or having any group type. For example, you might have a situation where one of several elements can occur, and each element has a certain character sequence that precedes it. In this case, you can define a parent element to these elements with a group type of *choice*, and then define the initiator characters in each child element.

The special characters apply here.

The initiator characters are not considered part of the text of the element depending on the setting of the **Include Initiator** property below.

6.6.2.2. Initiator and Terminator Special Characters

The following are special characters that may be specified in the **Initiator** or **Terminator** properties:

- *\n* - A single newline character(s), depending on the platform. This does not necessarily mean the actual newline character (ASCII 10), but specifies the newline character(s) as defined in the flat representation properties.
- *\t* - A tab character is expected and emitted an output.
- *\uXXXX* - A single Unicode character specified in hex.
- *\w* - A single whitespace character which can be a newline, space or tab. On output a single space is emitted.
- *\W* - One or more consecutive whitespace characters (newline, space or tab). Note that this will consume consecutive whitespace characters until a non-whitespace character. On output all spaces are emitted.
- ** - A single backslash character is expected and emitted on output.

6.6.2.3. Include Initiator

If set, the value of the initiator is included in the text value of the element. This is meaningful only for elements with a group type of none (that is, non-container elements).

This means that on input, the initiator value is added to the beginning of the text value of the element, and on output the initiator value is not written because it is assumed to be part of the value of the element.

6.6.2.4. Terminator

The terminator is the sequence of characters that signals the end of this element.

The special characters apply here.

The terminator characters are not considered part of the text of the element depending on the setting of the **Include Initiator** property below.

6.6.2.5. Include Terminator

If set, the value of the terminator is included in the text value of the element. This is meaningful only for elements with a group type of none (that is, non-container elements).

This means that on input, the terminator value is added to the end of the text value of the element, and on output the terminator value is not written because it is assumed to be part of the value of the element.

6.6.2.6. Start Offset

The start offset is used when working with positional formatted documents where the first character of this element does not immediately follow the last character of the previous element. This is the number of characters to skip before starting this element. If used in conjunction with the **Column** property, the characters are skipped after going to the specified column.

6.6.2.7. Column

Column indicates the column where the element starts. This causes the reader to advance to the next occurrence of the specified column, regardless of what row (line) you are on. For example, if the reader has finished the previous element on column 5 and column 20 is specified, it will go to column 20 on the same line. However, if the reader has finished the element and is at column 47, it will go to column 20 on the next line.

6.6.2.8. Quote Handling

Used for processing Comma Separated Value (CSV) files. This provides pre-defined options for handling possible quotation marks around elements. Select one of the following:

- **None** - No special handling for quotation marks.
- **Optional Quotes** - Double quotation marks may be present or absent for this element. If they are present, they will bound the element, and they will not be included in the data. If they are absent, the element is bounded by the normal initiator or terminator.
- **Required Quotes** - Same as **Optional Quotes**, except that the quotation marks must be present for each element. If the quotation marks are missing, an error will occur.

6.6.2.9. Release

Defines the single character that causes an initiator or terminator not to be recognized. For example, if the terminator character is a double quote, and you specify a backslash as the release character, you can include a double quote within the value of the element by preceding it with a backslash. If you the release character is a backslash and you want to include a backslash in the value, simply enter two backslashes.

6.6.3. Inheritance Properties

The following properties are used when an element inherits from another structure.

6.6.3.1. Inherits From

Specifies the structure from which this element inherits. The specified structure is known as the parent structure, and the structure containing this element is the child structure.

6.6.3.2. Inherited Root

Specifies how to reconcile the root element from the parent structure with this element. When you inherit from a structure, you might want element properties to come from the (current) element in the child structure, or from the root element of the parent structure, or some combination of the two. The following options are possible:

- **Use Only Children** - Only the properties from the child elements are included in the child structure under this element. The properties of the root element of the parent (inherited) structure are ignored.

- **Use Everything** - The element properties of the current element are ignored, and all of its properties are inherited from the root element of the parent (inherited) structure.
- **Use All Except Name/Occurs** - All properties of the root element of the parent (inherited) structure are inherited, except for the name, description, and occurs properties. These properties are defined by this element in the child structure. This corresponds to the type of inheritance defined in XML Schema by a particle.

6.6.3.3. Conflict

This property is set when an inheritance conflict is detected. It remains set (and gives a validation warning each time the structure is validated) until you reset it manually. Generally, when there is a conflict, you take some action to resolve the conflict correctly, and then reset the property. The validation warnings will then go away.

6.6.3.4. Ignore Inherited Adds?

Typically, when you modify elements in a parent structure, the changes are also applied to its child structure. However, if you have already made changes to a child structure, you might not want changes in the parent to affect it. For example, if you modify a child structure and remove codes that are not in use, you won't want the list of codes to be updated if you migrate the parent structure to a new version of a standard. To prevent these updates from happening to the child structure, check the **Ignore Inherited Adds** check box. This will automatically be checked if you delete elements with the type of **Value** in a child structure.

If the **Ignore Inherited Adds** check box is selected (on), elements added to the parent structure will also not be added to this element.

6.6.4. EDI Properties

The following properties are used for EDI.

6.6.4.1. Syntax Rules

The syntax rules are specified at the segment level to define the valid combinations of required elements for that segment. They are exactly the X12 syntax rule specifications, where each rule is separated by a semicolon (*;*). Even though the syntax rule specifications are defined by X12, they are usable for both EDIFACT and X12 EDI.

6.6.4.2. Sequence Id

Used only for EDI elements, this defines the sequence number, in the form *nn* of this element. Generally this will match the sequence number that is included in the name of the element. This is used only for validating the element data against the EDI syntax rules.

6.6.4.3. EDI Element Type

Defines the type of EDI element represented by this element. The possible values are:

- *None* - This is not an EDI element.
- *Transaction* - The root element of a transaction, this defines the transaction. The name of this element is the name of the EDI transaction. For example *832* or *PRICAT*.
- *Segment* - Defines an EDI segment. The name of this element is the segment and must be less than or equal to three characters. For example *DTM*.
- *Element* - Defines an EDI element. The name of this element is the name of the enclosing segment and is followed by the two digit sequence number of the within the segment. For example *DTM02*.
- *Composite* - Defines an EDI composite. If the parent of the composite is an EDI segment, the name of the composite follows the same convention as that for an EDI element. If the parent of the composite is an EDI element, the name of the composite is the name of the element followed by a hyphen and a two digit sequence number. For example *SEG05-01*.
- *Loop* - Defines an EDI loop. The name of this element must start with *Loop-*.
- *Code Value Part* - Used when a code value has two parts to it. One element defines each of the parts of the code value.

6.6.5. Expressions

Expressions may be specified in structure elements. The main use for this is to specify default values and validation expressions that are associated with the structure. In all cases the expressions in the structure are copied to the map when the structure is associated with the map, and unless overridden by an explicit specification of the expression in the map they are re-copied if they subsequently change in the structure. See Expressions for more details.

6.7. Inheritance

Structure inheritance is a powerful means of reusing structures within other structures. It is used for two principal reasons:

- Avoiding duplication - This is similar to the concept of a segment in EDI, or a complex type in XML Schema. For example, you can define an *Address* structure that is reused for a billing address and shipping address.
- Customization - In many business document definitions, you will often use only a small subset of the standard document. You can use structure inheritance to create a customized version of the original standard, and then delete (or modify) the unused elements.
- Version Migration - If you are referencing some standard structure (either an internal or external standard) which might later change to a newer version, you also use structure inheritance to create a custom structure. Then when you want to use a newer version of the standard, simple change the inheritance to point to the new standard structure and then reconcile any differences in the structure editor, then open and reconcile any affected maps (using the Show Objects Needing Validation mechanism).

Structure inheritance allows an element to inherit from a structure, so that the element's children come from the inherited structure's elements. If elements are added or changed in the inherited structure, the changes automatically appear in any inheriting structures.

Structure inheritance works by specifying that an element inherits from a structure. You can do this in three ways:

- New Structure Wizard - Creating a Custom Structure - This creates a new structure that customizes another structure. Do this using **New -> Structure** from the **File** menu or from the **Repository navigator**. Select the **Create a structure that is a customization...** item and specify the structure to be created and the structure

you are customizing, which is typically a standard of some sort. Initially your custom structure will appear as identical to the structure from which it inherits. You then add, delete or change elements in the custom structure to suit your needs (typically you would delete many elements and code values).

- Setting an element's inheritance properties - Click the browse **[...]** button next to **Inherits From** and specify the parent structure whose properties you want this element to inherit.
- Drag the parent structure to the element - You can drag the parent structure from the Repository navigator directly onto the desired element in the structure editor.

6.7.1. Element Propagation from Inherited Structure

When an element inherits from a structure, the structure's elements are used to create child elements of the inheriting element. The structure doing the inheriting is called the child structure, and the structure from which the elements were inherited is the parent structure.

An element inherited from another is indicated with a black arrow icon decoration.

If element(s) are changed in the parent structure, the changes are automatically propagated to all child structures. This propagation happens immediately, so for example you can have an editor open for both the parent and child structures, add a new element in the parent structure and switch to the editor for the child structure and the new element will be present.

If you modify an inherited element in the child structure, this will be indicated by a gold triangle in the icon (signifying a delta, or change). This indicates that the child element is different than the parent element. You can see what the specified differences are by right clicking on the child element and selecting the **Show Differences from Inherited Element** menu item.

If you add an element in the child structure (as a descendant of an inheriting element), a yellow star will appear indicating the element is new in the child structure and not present in the parent structure.

6.7.1.1. The Element at the Point of Inheritance

The element at the point of inheritance (the element that specifies the parent structure) requires special consideration. There are some cases where you want none, some or all of properties of the element to be from the parent structure's root element. The Inherited Root property of the element (available in the element's properties when in Editable mode), specifies this by declaring which of the properties parent structure's root element are to be used. The following options are available:

- **Use Only Children** - The element in the child structure is not modified by the parent structure at all. The only elements inherited from the parent structure are the child elements of its root and their descendants.
- **Use Everything** - All properties from the parent's root element are used for the child element.
- **Use All Except Name/Occurs (XSD)** - All properties come from the parent element except the element's name and minimum and maximum occurrences which remain those from the child element. This is used for XSD schema support.
- **Use All Except Name/Occurs/SeqId (EDI)** - All properties come from the parent element except the element's name, minimum and maximum occurrences, and sequence Id which remain those from the child element. This is used for EDI support.
- **Use All Except Occurs/SeqId (EDI)** - All properties come from the parent element except the element's minimum and maximum occurrences, and sequence Id which remain those from the child element. This is used for EDI support.

6.7.1.2. Conflicts - Changing Both Parent and Child

Structure inheritance combines the elements and their properties into the child structure. There are times however when the same property of an element has changed independently in both the parent and child structure. For example the maximum number of occurs may have changed in both places. When this happens, the conflict indicator is set in the element's properties and a warning is given anytime a map is executed using that structure. You will need to manually resolve this conflict possibly by changing the child structure, and then you can remove the conflict indicator in the element's properties.

Note that structure element conflicts can also occur with the addition of elements to a sequence in both the parent and child structure, as it's not clear what the final order of the elements should be. The new elements are marked as having a conflict.

6.8. Recursively Defined Elements

XML allows an element to be recursively defined, so that children of a given element are the same as the children of an ancestor element.

Recursion can be defined in the following ways (these are automatically determined when using the XSD or DTD import capability):

- **Use Children Of** - You can set the **Use Children Of** property to specify the element whose children are to be used.
- **Inheritance** - If some ancestor of the given element either directly inherits from a structure that contains the given element, or indirectly does so through an intermediate structure, the given element is treated as a recursive element. This is indicated in the element properties (when using the read only view) as **Children not shown - recursive inheritance at [link]**.

  By way of example, consider a Person structure that has an element for Department that inherits from the Department structure. The Department structure has an element called Manager that inherits from the Person structure. In this case, the Manager element is effectively recursively inheriting because there is an enclosing Person element that this is all part of.

Full support is provided for mapping recursive elements.

6.9. Runtime Instance Validation Against Structure Definitions

Structure validation is the act of checking the values of an instance document against the element specifications defined by the structure, such as checking that the number of characters inside of an element does not exceed its size constraints. Custom validation checks are possible using the validation expression.

Because the additional checks associated with structure validation can add significant processing time, especially in large structures, structure validation is turned off by default. You can use the preferences to enable structure validation.

Chapter 7. Maps

This chapter explains how you create, edit, and test maps.

7.1. Maps and Structures

A map refers to a single input structure and an output structure.

A map can be used to operate on multiple types of documents within a single structure using the Input/Output (Multiple Input/Output Documents) mechanism.

When working with a map, you select map elements from the input or output panel. These elements are normally identical to the structure elements, and they cannot be modified or deleted (you have to change the structure elements for that). There are some cases when they might vary from the structure elements which are described at Validating a Map, Unrolling an Element from a Loop, and Splitting a Loop.

The properties associated with map elements are the properties of the corresponding structure elements. However, you cannot change these corresponding properties when working with a map. To change structure element properties, you must do so in the structure editor. In addition, you cannot add or delete map elements. In short, you cannot use the map editor to alter the definition of the map elements, as they must correspond to the structure elements.

7.2. Creating a Map

You can create a map by right-clicking any resource in the **Repository navigator** and choosing the **New** > **Map** menu item. Alternatively, in the standalone version you can click the **New Map** button. You then specify the map type, enter a name for the map, and select the input and output structures for the map using the **[New Map]** wizard. Alternatively you can end the wizard after naming the map and drag the input and output structures to their respective panels in the map editor. Make sure the **Link with Contents** button  is not selected. Otherwise, when you click a structure you want to drag to the map, the editor window will switch from the map view to the structure properties view, so you will not be able to drag the structure to the map.

7.2.1. Creating and editing a Map from the command line

Talend Data Mapper comes with a scripting capability that can be used to create and edit structures and maps directly from a command line. It is available with any *Talend Data Mapper* API jar file (`tdm-api-<version>.jar` or `tdm-camel-<version>.jar`).

The scripting capability works in a workspace and a project designated by the user. It is recommended to use a different workspace from the one used by *Talend Studio*, as objects created in this way cannot be used directly by the Studio because they do not have the accompanying *.properties* files, and then import the objects you create into the Studio later.

To import any objects created by the *Talend Data Mapper* scripting capability into the Studio, you must import the relevant project from your local workspace into the Studio. When you import the project, take care to select the **Copy projects into workspace** checkbox, or else the project will be linked to the workspace.

For more information on how to import projects, see *Talend Studio User Guide*.

7.2.1.1. Example syntax for the Talend Data Mapper scripting capability commands

This table provides some examples of the syntax for the *Talend Data Mapper* scripting capability.

Purpose	Syntax	Example
Specify which workspace to use	`java -jar tdm-api-<version>.jar -editor data <path>`	`java -jar tdm-api-5.6.2.jar -editor -data c:/Talend/Studio/myworkspace`
Create a project in a workspace	`java -jar tdm-api-<5.6.2>.jar -editor -action create -project <project_name> -data <path_to_workspace>`	`java -jar tdm-api-5.6.2.jar -editor -action create -project myproject -data myworkspace`

The `-help` command provides a complete list of all available commands and their syntax.

7.3. Editing a Map

To work with an existing map, double-click the map in the **Repository navigator**, or right-click it and choose **Open Map**. A map editor opens and its input and output structures appear in the input and output panels.

As you select map elements in the input or output panel, any expression trees that are valid for that map element are enabled in the expression tree panel(s) at the bottom of the map editor. These are described below.

7.3.1. Drag/Drop

The drag/drop mechanism is used to specify most of the aspects of a map. The simplest operation is to drag an input map element to an output map element. If the output map element had no previous value expression, this creates a value expression that copies the input to the output.

Here are the different possibilities for drag/drop when editing a map:

- Drag from input map element to:
  - An output map element - If no value expression is present, creates a value expression copying input to output.
  - An empty value expression panel - Behaves identically to dragging to the output map element (assuming the value expression tab is enabled, which occurs when you have previously selected the output map element).
  - An expression - Adds the input map element as the last child of the expression. If the expression is a function argument, only one child to a function argument is allowed.
- Drag a function to:
  - An output map element - If no value expression is present, the function becomes the root of the value expression tree. Otherwise, has no effect.
  - An empty value expression Panel - Behaves identically to dragging to the output map element (assuming the value expression tab is enabled, which occurs when you have previously selected the output map element).
  - An expression - Adds the function as the last child of the expression. If the expression is a function argument, only one child to a function argument is allowed.
- Drag an expression to:
  - Another expression - Adds the source expression as the last child of the target expression, and removes the source expression. This is because a drag/drop is identical to a cut/paste operation. A copy/paste will do the same thing, except it will not remove the source expression.
- Drag from an output map element to:
  - An output map element - Creates a value expression in the destination output with a reference to the source output, unless a value expression is already present in the destination output.

  - An empty value expression Panel - Behaves identically to dragging to the output map element (assuming the value expression tab is enabled, which occurs when you have previously selected the output map element).
  - An expression - Adds the output map element as the last child of the expression. If the expression is a function argument, only one child to a function argument is allowed.
- Drag from an value map element (input or output) to:
  - An output map element - Creates a *Constant* expression with the specified value.
  - An empty value expression Panel - Creates a *Constant* expression with the specified value.
  - An expression - Creates a *Constant* expression with the specified value.

No other drag/drop operations have any effect when editing a map.

7.3.1.1. Expanding Map Elements

As mentioned above, map elements correspond with the structure elements. In some cases though structures can be huge with thousands of elements most of which are not mapped. For performance reasons map elements are only materialized if they are actually used in mapping, which happens as you expand the map elements in the designer.

So when editing a map, and simply expanding a map element, you may see the map become unsaved (the asterisk appears in the editor tab); this is normal and indicates that new map elements have been materialized.

When searching through a map, all possible map elements (being searched) are temporarily materialized and then discarded at the end of the search so as not to have a large number of unused elements.

7.3.2. Value Expression Panel

This panel contains the expression used to provide the value for the output map element. It is valid only when an output map element is selected.

7.3.3. Loop Expression Panel

This panel specifies how the output map element is to loop. Typically, this specification is automatically generated as you map elements from the input to the output. The loop expression panel appears only when you select an output map element that can loop.

The root of the loop expression must be a loop function. The main loop function is **SimpleLoop** which requires the specification of an input map element that defines the looping for the output map element that contains the loop expression. See the Looping Expressions section for more details.

7.3.4. Copying Expressions (Mappings)

You can copy a map element's expressions to another map element. To do this, select the map element, right-click and select **Copy**, then select the target map element and select **Paste Expressions (Mappings)**. This copies all of the expressions, i.e. Loop expressions, Value expressions, etc. from the source map element to the target map element. If the expression is already present in the target element, it is not modified. Also, the expression copying

applies to all children of the source and target elements that are matched by map element name (in exactly the same way mapping from an input map element with children to an output map element with children is done).

This is particularly useful if the output structure has changed under the map, by adding or removing levels of elements, or moving a block of elements. You can easily move the mappings by copying the expressions from the old locations of the elements to the new ones.

7.3.5. Moving Expression References

In cases where an input structure of a map has changed and elements of the input structure have moved, you need to be able to fix the map to refer to the moved elements. You can easily do this using **Move Expression References** on the map element(s) that have moved. When you do this, all references that were original element are changed to the moved element. This also applies to the children of the original and moved elements which are matched up by name.

To move the map element's references, select the original map element you have moved (which is usually a red-circle invalid map element), right-click and select **Move Expression References**. This will present you with a dialog where you select the map element you wish to move to.

It will when search through all map element references to the original map element and delete them, and then add a new reference to the element you are moving to. If for some reason there is a problem moving an element, you will get an error dialog and nothing will be moved. Also, the expression moving applies to all children of the original and moved elements that are matched by map element name (in exactly the same way mapping from an input map element with children to an output map element with children is done).

7.3.6. Unrolling an Element from a Loop

There are times when it's more convenient to map an iteration of a loop as a non-looping element (so that you don't have to bother with specifying a loop expression). This is easily handled by unrolling an element from a looping element. Do this by right-clicking on the element and selecting **Unroll from Loop**. When you unroll an element from a loop, it creates a new non-looping element that you can map more easily. You can create as many of these as you like. The looping element is kept after all of the unrolled elements. If you no longer need an unrolled map element, simply delete it.

7.3.7. Splitting a Loop

In some cases, it's convenient to break a loop into two loops where you have one loop iterating according to one input element, and the second iterating according to a different input element. To do this, split the loop by right-clicking on the element and selecting **Split Loop**.

For example suppose your output has a loop for street addresses (of any type), where many different addresses are supported. And the input has a loop for business addresses and a different loop for personal addresses. To easily map this, you could split the output loop into two loops and map the first to the business addresses and the second to personal addresses. You can split a loop into as many loops as you like, and if you no longer need a split loop you can delete it.

7.3.8. Mapping Code Values

Code values (the set of possible values that an element can have) are specified by having elements with the element type of Value as children. If you right-click on one of these Value elements, you will see a **Map Value** option on

the context menu. Selecting this will create a **Constant** function mapping the enclosing element to the selected value. You can also use the drag/drop operations described above to map code values.

7.3.9. Mapping a Constant

To quickly map a constant value to a map or structure element, right-click on the element and select **Map to Constant**. This will add a **Constant** function and show its property sheet so you can specify the value.

7.4. Null Value Support

Certain representations support the notion of a null value, which is a special value indicating the explicit absence of data. These representations are XML, Java and Database.

- XML - A null value in XML is represented as the presence of the *xsi:nil* attribute being true. For example: *xsi:nil = "true"*.
- Java - A null value in Java is represented by the field or property of the object having a null value.
- Database - A null value in the Database is represented by using the SQL null value for the column in the row.

Null values are enabled for a particular element by setting the Null Property on the element. This is done automatically during the import process for the representations that support null values.

Null values are passed through the mapping transparently. So if you map a value from a Java field element to an XML element and they both support the Null property, the correct null value will appear in the output. You can explicitly specify a null value using the **Constant** function and you can test for null using the **IsNull** function.

There are also cases when you want container elements to be emitted as null, in this case you can use the Null Expression to control this. The Null expression is also automatically generated.

7.5. Automatic Generation of Expressions

When mapping an input element to an output element expressions are automatically generated to make the mapping work correctly. The automatic generation happens only if the expressions (that would be generated) were not previously specified. Though in most cases the automatically generated expressions will be what you want, this will not be true sometimes. In these cases you will need to manually alter them.

7.5.1. Loop Expressions

See Automatic Loop Calculation for these details.

7.5.2. Emit Expressions

The emit expression specifies the conditions for the emission of an optional element or a member of a choice. Emit expressions are generated automatically only for output map elements that are containers. This is used to make

sure the container element in the output appears corresponding to some container element on the input. These default expressions use the **IsPresent** function on the input element. So if the input element is present, then the corresponding output element will be emitted, otherwise it will not.

When you map an input to an output, the containing optional/choice members are examined in order starting from the parent of the output map element and the output is matched with the container of the input map element and so on up the line. For each matching containing map element, an **IsPresent** is generated.

There is a special case for emit generation when the output element is optional, but does not have the null property and the input element has the null property. In this case the emit expression is generated to emit the output element only if the input is not null, rather than if it's present as described above.

7.5.3. Null Expressions

The null expression is used only for containing elements (elements whose group type is not None), and only when the element has the null property set in the element properties. For leaf elements (elements with a group type is None), you don't need the null expression; if the input being mapped to the element is null, then the output will be null. If you wish to have special conditions for emitting null then you can use the **Constant** function to emit the null or you can test for null using the **IsNull** function.

Similar to the generation of the Emit expression, when you map an input to an output, the containing elements are examined in order starting from the parent of the output map element and the output is matched with the container of the input map element and so on up the line. For each matching containing map element, an **IsNull** is generated, so that the generation of the null for the output is controlled by its corresponding input.

7.6. Mapped and Unmapped Elements

Initially, none of the map elements are mapped. The word "mapped" is used as a short-hand expression: for output map elements, it means the map element has a value expression; for input map elements, it means that the map element is referred to in some output map element's value expression.

Mapped elements are shown in bold font, and unmapped elements are shown in normal font. A map element that contains a mapped map element is also shown in bold.

7.6.1. Finding Elements Mapped to an Input

You can easily find any of the output map elements that are mapped to an input map element by right-clicking on the input map element and choosing the **References to Output Elements** menu item. When you select the menu item corresponding to the desired output map element, the output map element is selected.

7.6.2. Finding Elements Mapped to an Output

When you click an output map element, the **Output Value expression** panel shows the expression used to create the output map element. To locate one of the map element references contained in the expression, right-click the map element reference expression and choose the **Show Map Element** menu item. This will select the associated input or output map element.

7.7. Data Type Checking and Compatibility

Each structure element is associated with a data type. Data types are also associated with function return values and function arguments.

When creating expressions, the editor checks that data types are compatible. Data types are compatible if it is possible to automatically convert from one type to another. For example, a string data type is compatible with the date data type because it is possible to convert a string representation of a date to a date representation (with the type of *Date*). However, a numeric data type cannot be converted to a date and is thus not compatible.

If the data type of a child expression (often a structure/map element) does not match the data type of its parent expression (and assuming they are compatible data types), code is automatically generated with the map to convert from the child's type to the parent's type.

A runtime error will occur if the data in the document is not in a valid format for type conversion. For example, suppose you are mapping a string to a date. If the string is not the valid date format, the runtime error is signaled.

7.8. Testing a Map

When working with a map, you can use a test document to execute all or part of the map and view the results. You can also examine the test document. To select a test document, use the **Test** > **Select Test Document** menu item. You can use the options below to view the test document and execute the map on the most recently selected test document. If no test document was selected, you are prompted to select one.

7.8.1. Viewing the Test Document

To view the entire test document, use the **Test** > **Show Test Document** menu item.

To view a portion of the test document, right-click the input map element you want to view (if this is a container map element, it is shown with all of its children), and then choose the **Show Test Input element** menu item.

7.8.2. Executing the Map/Viewing Output

To execute the entire map and view the entire output document, use the **Test** > **Test Run** menu item.

To execute only a portion of the map and view the relevant part of the output document, right-click the output map element you want to view (if this is a container map element, it is shown with all of its children) and choose the **Test Run element** menu item.

7.9. Inheriting from Other Maps

Often there is a set of standard transformations between two document types that are captured in a given map. If a small customization is required, it comes in handy to have one map reference another map. The parent map would be the standard map, and the child map is the map where the small customization must occur. This customization is done only in the child map, and any future changes to the parent map are automatically reflected in the child map. This avoids the maintenance problems of making complete copies of maps and having them subsequently change.

Another useful case is where you have small maps that transform a portion of a document, and you want to use these to build up a larger map. These small maps (for example corresponding to *EDI* segments) can be used to compose several different larger maps.

When a map inherits from another map, this forms a parent/child relationship between the maps.

A parent map and child map can both be associated with the same structure. In this case, both maps have exactly the same map elements. If there is no value expression associated with a map element in the child map, the value expression (if any) associated with the parent map is used. However, if there is a value expression in the child map, the *Output Value* expression in the parent map is not used.

It is also possible that the parent map is associated only with a portion of the structure of the child map. In this case, when associating a parent map, you specify the elements in the child map that anchor the parent map.

7.9.1. Adding a Parent Map

You can add a parent map in two different ways, depending on which is more convenient for the situation.

1. Be sure that the **Offer map inheritance/extract on drag/drop in map editor** preference is enabled (it's not enabled by default). Then in the map editor drag an element from the input to the output where the input and output inherit from structures corresponding to a map that you want to inherit from. If there are maps available with the same (or compatible due to structure inheritance) input/output structures, they will be offered for map inheritance.

2. Expand the child map in the **Repository navigator**, and then either drag the parent map to the **Inherits From** tree node that appears below the child map, or use copy/paste to place it there. When you do this you will be prompted for the input and output child map elements to anchor the parent map.

7.9.2. Inherited Map Elements

A map element that inherits from another man element is indicated with a black arrow icon decorator (the same decorator that's used to indicate inheritance of an element in the structure editor). You can go to the inherited map element by right-clicking on the child map element and selecting **Go to Inherited Map Element**.

If the map element in the child map is modified from the parent map, this is indicated by a gold triangle icon decorator (the same decorator used for the corresponding reason on the structure editor). If you wish to remove the changes in the child map element you can select it and right click and select **Revert to Inherited Map Element**. You can revert either the selected element, or all of the descendant map elements.

7.10. Validating a Map

The editor is designed to work in an environment with thousands of maps and structures. Therefore, it is possible for these structures and maps to change independently from each other. In the event of such changes, it is important that they be gracefully handled and reported.

You can force the synchronization of a map with its structures by right clicking the map in the Repository navigator and clicking "Synchronize with Structures".

Alternatively, whenever you open a map, access its properties, or execute it using the runtime API, the map is automatically validated. Any errors or warnings associated with the validation are displayed.

Any map being edited is also automatically revalidated if one of its structures or parent maps is being edited and changes.

Validating a map compares the current map elements to its input and output structures and to any parent maps.

7.10.1. Structure Comparison

If a structure has changed independently of a map, we have to consider elements added to the structure and not present in the map, and elements removed from the structure that are in the map. The validation process takes care of both of these cases.

In the case where elements have been reordered in the structure, the map elements are automatically adjusted with no notification.

If a structure element has been added, it is added to the map, and a warning is given once.

If a structure element has been deleted, its associated map element is marked invalid (which is indicated by the  icon). You can locate invalid map elements using the Search feature. Invalid map elements are ignored when executing the map, and they are the only map elements that can be explicitly deleted. The purpose of not automatically deleting them is to preserve any associated expression information. Once you are satisfied that there is nothing about the map element that you want to keep, you should delete it. A warning about the invalid map element will be given each time you open or execute the map until the element is deleted.

You can use the **Delete Invalid Map Elements** menu item on the map element to delete all of the invalid map elements at or below the selected map element. This is often useful if you know you don't have any mapping information you want to preserve for the invalid map elements.

In the case where elements in the structure have been moved, they will appear as added or deleted elements in the map. For the output side of the map you can copy and paste the map expressions to quickly move the expressions to their new location. For the input side of the map you can move the expression references to quickly adjust the map output to refer to the new location of the moved input elements.

7.10.2. Map Comparison

This is used to update the child map with the latest expressions from the parent map. This is fully automatic and requires no intervention.

7.11. Execution/Validation Results

When you run or validate a map, errors or warnings may occur. In the editor, these are displayed in the **Problems** view (**Mapper Problems** if you are using the product in an IDE). In the runtime portion of the product, an **ExecutionStatus** object is created with all of the errors/warnings. This is returned to the caller of the map execution or validation methods. You can translate this object to XML using the **ExecutionStatus.exportToXml()** method.

7.12. Map Runtime Considerations

When a map is executed at runtime (in the absence of the *Data Mapper*) it is normally not validated against its structures as all of the data associated with the structures are stored in the map's metadata. The reason for this is the validation against large structures can consume quite a bit of memory in the initial map startup.

However, for this optimization to work correctly, the validation results of the map with the structures must be completely clean: there must be no errors or warnings. If there are errors or warnings in the last validation of the map in the *Data Mapper*, the map is marked such that it will require validation against the structures at runtime to make sure any problems are correctly reported and the runtime execution results of the map are the same as they would be in the *Data Mapper*.

7.13. Streaming Execution

Streaming execution is used to process unlimited amounts of data. Without streaming execution, the entire input of the transformation is stored into memory before the transformation is executed, which limits the amount of data to be transformed to what may fit in the available memory.

Streaming execution works by accumulating chunks of input data and then executing the transformation on each chunk separately. Because of this, there are limitations on what may be specified in the transformation.

You specify that the transformation is to stream by checking the **Stream Input** property on the **SimpleLoop** function. When that is done the loop is automatically partitioned into chunks depending on how much memory is allocated per chunk and each chunk is processed separately. Because of this, the use of an aggregate function to span all occurrences of the loop is not permitted. It is possible to perform aggregation using the **GetVariable** and **SetVariable** functions, as these maintain their state across multiple transformation executions.

If you select **Stream Input** property on the **SimpleLoop** function, you cannot use sort keys, since the sort action cannot be performed while streaming.

If you select **Stream Input** property on the **SimpleLoop** function, and you also select a distinct child element, the input is already sorted by the child element such that the distinct calculation can be done without further sorting.

7.14. Map Execution Properties

You can specify properties (named values) when you execute a map using any of the mechanisms to invoke a map. These properties can be read during map execution using the **GetMapProperty** function.

If you are using an ESB to execute a map, the properties of the message triggering the map execution are automatically included as map execution properties.

The following properties are automatically specified. Properties starting with *oaklandsw* are reserved.

- *oaklandsw.source.url* - The URL of the source document on which the map is executing.
- *oaklandsw.result.url* - The URL of the result document on which the map is executing.

7.15. Special Purpose Maps

The following maps are provided for special purposes:

- *XSLT Report* - Processes data specified by the input structure using an XSLT report and emits the result to an output string. One typical use for this is to process the execution status from a map execution and format it into HTML. This can be done using an XSLT report map with the input structure of */Builtin/Structures/Status/ExecutionStatus* with a report of */Builtin/Reports/ExectionStatus/ExecStatusToHelp*. The output structure of the map contains only a single field, which is the string result of the XSLT execution. This can either be processed as the output of the map, or you can use an I/O function to send a message, etc.

7.16. Troubleshooting No Output Found

Sometimes when you execute a map no output at all will be produced. This mainly happens when the input document is XML and most commonly caused by all of part of the input document of the map not being recognized. Here are some things to have a look at:

- Use the Structure Editor - Switch over to the structure editor and have the sample document in question be the current document. Check that the highlighting is responding to the selected elements. You can also select an element and do right-click Show Sample to see if the element appears as desired. If the element does not appear, see the items below.
- Does the Structure Match? - XML requires that the hierarchy of elements matches exactly what is defined in the structure. Start with the root element in the structure definition and check that the elements in your sample document correspond exactly.
- Case Sensitivity - XML is case sensitive so make sure the case of the element matches that in shown in the map editor.
- Namespace Issues - If the document uses XML namespaces, follow the namespace troubleshooting instructions.
- Correct Visibility/Element Type/Group Type - Make sure these properties of each element are what you expect and match the actual document.
- Emit Expressions - In some kinds of structures (like those found in FpML version 4.x) the root is a non-visible choice. If none of the members of the choice have an Emit expression that returns true, you will not get any output.

Chapter 8. Sample Instance Documents

8.1. Overview

The studio has extensive support for sample documents which can be used to help understand and define structures and help create maps. Sample documents are normally associated with a structure and are requested when needed (to display output for example). By default, they are stored persistently in the *Sample Data* folder of the project that contains the structure. The path of the structure including the name of the structure is the path under the *Sample Data* folder contain the sample documents. For example, if you have a structure named *Structure1*, the sample documents for that structure would be in the *Sample Data/Structure1* folder of the project that contains the structure.

You can also set the path to the sample documents associated with a structure explicitly in the structure's properties. This is useful for cases where you wish to share a pool of sample documents among many structures.

For structures that are in read-only projects, like *Builtin* and *Examples*, it is not possible to associate the sample documents with the structure. In these situations, the sample documents are associated with the map.

Sample documents can be viewed in a variety of ways in both the structure and map editor. They can be shown either partially (associated with a structure or map element), or in their entirety. Also, sample documents can be created as the result of the execution of a map.

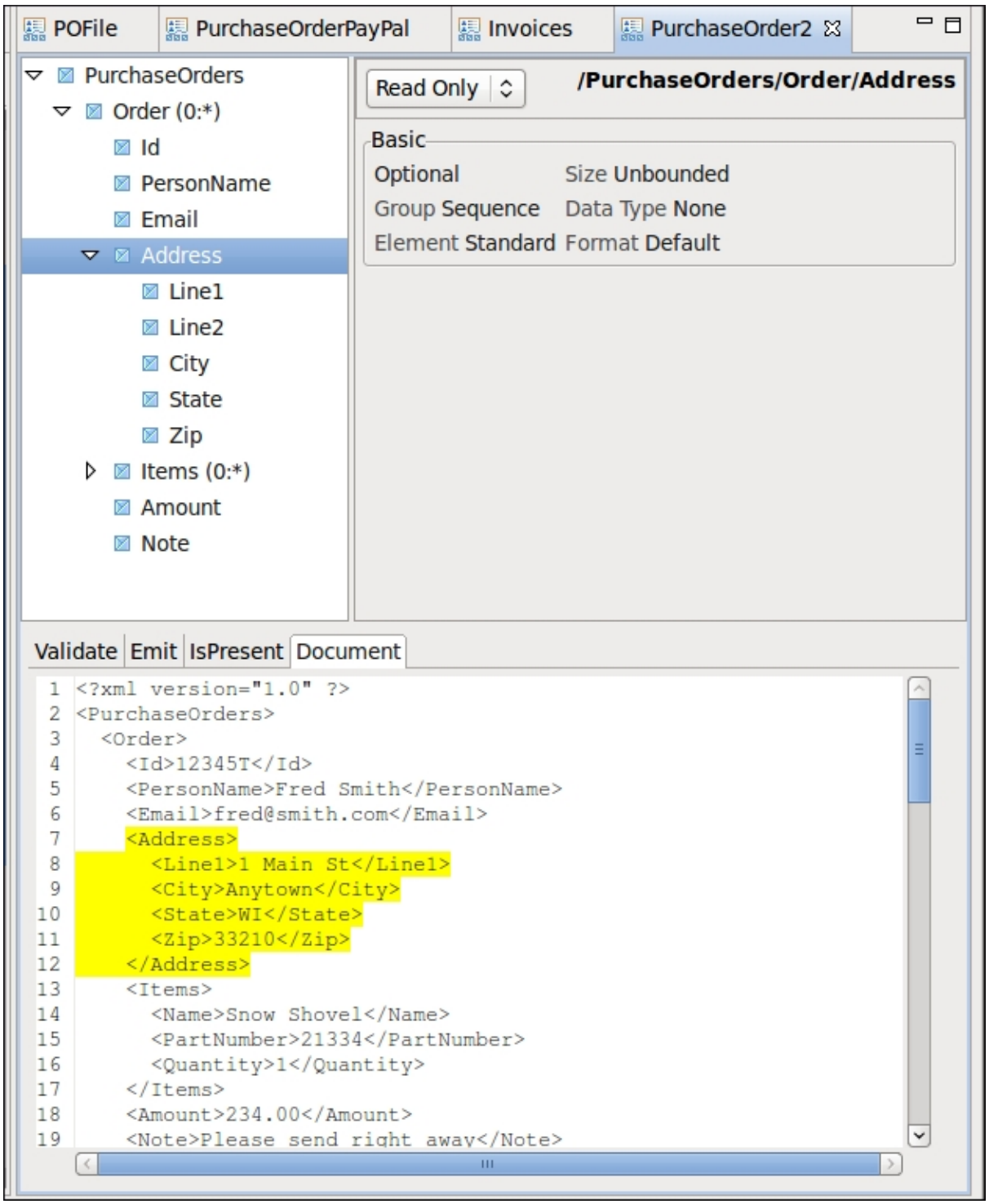
When testing maps associated with a structure, any sample documents associated with that structure are available to the map.

Sample objects are possible for Java as well. The Java Sample Data Objects described how you can create these.

8.2. Sample Documents and Structures

You can import and edit sample instance documents into a structure so that the documents are used when viewing or editing the structure elements. By default, these sample documents are stored in the workspace in the *Sample Data* folder (with the path to the structure as subfolders), so there is no need to refer to the original sample document files once they are imported. You can also override this default location of the sample documents by specifying the path in the structure's properties. The structure's instance documents can also be used when testing maps.

Any number of sample documents can be associated with a structure. Only a single sample document is the current document when you are editing or viewing a structure. Using the **Show** > **Select Sample Document** menu item, you can set the current document or import a new document. However, the first sample document shown under the "documents" header will be the default document that is opened when the structure is viewed or edited. The sample document appears on the **Documents** tab of the structure editor. As you click elements, they are highlighted in the sample document if you have the Automatically load the default sample document preference set.



8.3. Sample Documents and Maps

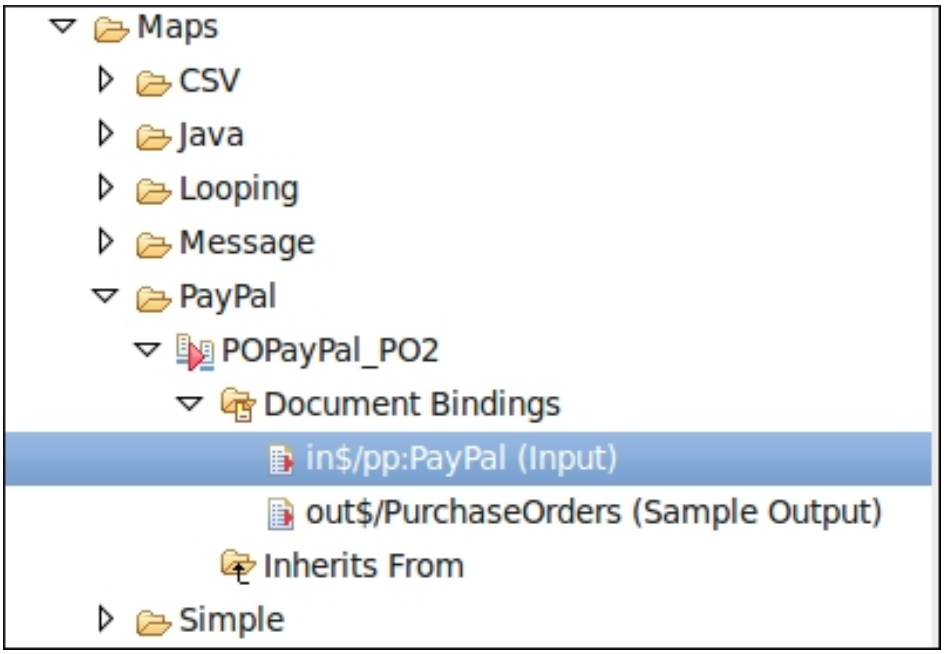
You can also associate sample documents with maps. When you add a sample document to the workspace, and the structure is read-only, the document is automatically associated with a map instead. Just as with structures, when the document is associated with the map, it is stored in the workspace in the *Sample Data* folder.

When documents are associated with a map, this is indicated in the map properties. As with structures, you may select the folder that contains the sample documents for each side (input or output) of the map. If you don't select a default folder, it is assumed to be *Sample Data/<map name>_<input|output>*.

8.4. Sample Document Bindings

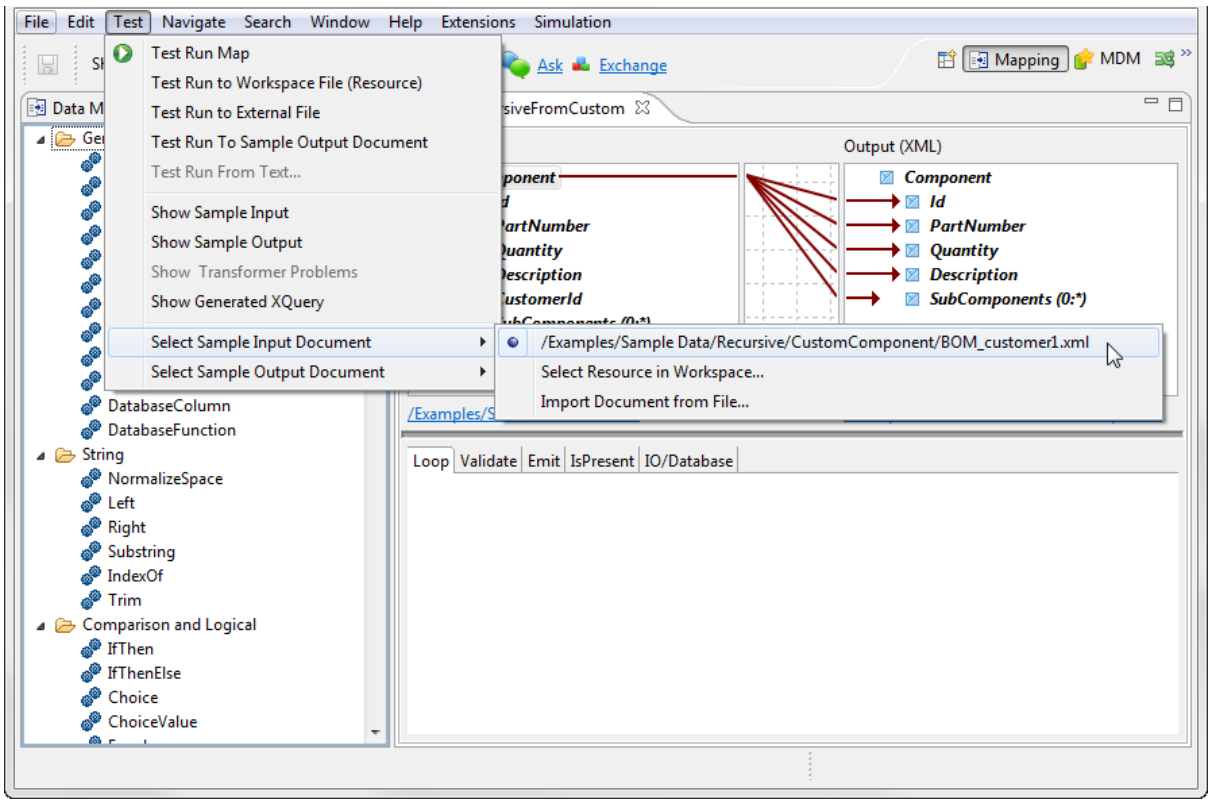
A structure or map can have multiple points at which it can be associated with a sample document. The simplest case is a structure that does not use the multiple document support available through I/O Functions; it will have only one document binding, for its root element. For maps, the documents may be bound for input purposed (that are associated with structure(s) on the input side) and sample output (that are associated with structure(s) on the output side).

The document bindings are shown in the **Repository navigator** under the structure or map . Each document bindings shows the element of the structure or map that is bound, what type of binding it is (either input or sample output), and the current sample document that is bound to that instance. The latter is shown by right-clicking on the document binding entry and selecting **Properties**.



8.5. Displaying a Sample Document

You can display the contents of the sample document using the **Sample** > **Select Sample Document** menu item. A structure contains a collection of elements that describe the contents of instance documents. Structures may be connected through the inheritance mechanism, which allows an element to inherit its definition from another structure. The definition of a structure and its elements is abstract in that it is not dependent on the particular way the structure is represented in instance documents.



8.6. Multiple Sample Documents

When using I/O functions for multiple document support, sample documents are supported as well. The sample document is associated with the structure that is inherited by the element that has the I/O expression. Therefore, when you show the sample document or test the map, you can provide a sample document for each I/O expression.

8.7. Java Sample Data Objects

You can create sample documents from Java object instances so that you can test maps that use Java objects as input. These documents are stored as XML as defined in the XStream project's default serialization. Well, almost the default serialization, the only difference is that it specifies the *ID_REFERENCES* option so that the object graph is preserved in the case of inter-object references.

You can produce these XML-serialized Java objects in the following ways:

- As an Output - If you have a map that writes to the desired Java object, this is the easiest way to go. Simply test run the map in the designer to a sample output document. Do this by selecting the **Test Run to Sample Document** in either the **Test Run** button's menu or the **Test menu** on the main menu bar. This will produce the XML document and associate it with the output structure of the map.

- Using the Runtime API - In the Runtime API use the *RuntimeEngine.addSampleData()* method to add a specified Java object instance to the desired structure's sample documents. There is an example program for this in the Java folder of the examples project called **com.mycompany.accting.AddJavaObjectSample**. You can modify this program to create your object, and then execute it as a Java application within the Data Mapper. In order to execute it, you will need to provide the JAR file containing the Runtime API in your classpath. You will also need to specify the *ODT_DEV_WORKSPACE* Java system property (you can do this in your Launch

Configuration) to be the file path or URL (using the file scheme) of the Eclipse workspace containing the project that gets the sample document.

- Using your own Standalone Code - The following snippet of code can be used to produce the sample document; this depends on the some XStream classes, but has no other dependency.

```
HierarchicalStreamWriter javaWriter;
StringWriter sw = new StringWriter();
javaWriter = new PrettyPrintWriter(sw);

XStream xsShow = new XStream(null);
xsShow.setMode(XStream.ID_REFERENCES);
xsShow.marshal(yourObject, javaWriter);
javaWriter.flush();
```

You put the string into a file and then copy the file into the *Sample Data* folder corresponding to your Java structure in the workspace.

Chapter 9. Expressions

Expressions are used to specify values, looping, conditions for emitting optional elements (forcing them to appear), and the condition for emitting a null value. These expressions essentially define the execution logic.

Expression panels of the above types are associated with structure and map elements. Each expression panel forms a tree where each expression is evaluated in terms of its subordinate expressions. Each tree node in the expression panel is an expression.

Expressions may be presented as tree nodes (as shown in the examples later in this chapter) or in a text format that you can directly manipulate by typing. The text format is also used as part of the export format and for printing.

You can choose either the text or tree representation for expressions at any time using the preferences.

9.1. Expression Basics

Expressions are powerful but simple. Every expression returns a single value, which is used by its parent expression. The value of the expression at the root of the expression panel is the final value used according to the purpose of the panel. For example, the value of the root expression in the value expression panel becomes the value of the output map element.

Example 9.1. Simple Expression Example

```
Transaction-856/Segment-ISA/Element-I01 ❶
```

❶ An input map element reference, provides the value of the input map element.

In this example for a value expression panel, the root expression is a map element reference expression that refers to *Element-I01* in the input panel. This is the expression tree you get if you drag an input map element to an output map element.

Example 9.2. More Complex Expression Example

```
Concat ❶
  Person/LastName ❷
  Constant ", " ❸
  Person/FirstName
```

❶ The concatenation function, concatenates all of the arguments. In this case there are three arguments, the last name, a constant comma, and the first name.
❷ An input map element reference, provides the value of the input map element.
❸ A constant expression, providing a constant value, in this case a comma followed by a space.

In this example for a value expression panel, the root expression is the concatenation function which has three arguments. The first and third are map element references to input elements. The second is a constant expression containing a comma. The result of this expression would be something like *Smith, John*.

9.2. Expression Trees

Each expression tree has a purpose associated with its map or structure element. The following expression trees are used:

1. *Value* - Defines the value of the element. The result of the root of the expression tree is the value of the output element in a map. If the value expression is specified for a structure element, it is copied to the corresponding map element as the default value. When the structure is use as an output structure in a map its value expression defines the default value for the output. When the structure is used for an input structure, the structure element value is used as a default value by some representation readers (like Java) to contain metadata. The input value expression is not intended to be modified in the map and is only present to reflect the value expression from the corresponding structure element.

2. *Loop* - For an output map element, defines how the element loops. The root of this expression tree must be a loop function.

3. *Validate* - Allows arbitrary validation logic to be associated with the element. When the result of this expression is *false*, a validation error is reported.

4. *Emit* - Defines the conditions where this element is emitted (forced to appear) when the element is optional or a member of a choice. An element is optional if it occurs between 0 and 1 times. If the result of this expression

is true, or if the expression is not specified, the element is emitted, otherwise it is not. This is useful when the optional element needs to be emitted based on some special condition. However, most of the time it is not necessary, since an optional element with an empty value will automatically not appear.

This is also useful in the case of a choice. For example when using XML and mapping to an XML Schema Type that has other types which extend from it. In this case the *xsi:type* attribute contains the type of the actual elements that appear in the input document. An *Emit* expression can refer to the *xsi:type* value. When elements from a member of the choice are mapped somewhere to the output the Emit expression from the input is automatically copied to the enclosing member in the output allowing the output member to be selected based on the type of the input. *Emit* expressions are automatically generated on the input side of maps for the purpose of copying to the output side to have the output sense the choice condition from the input (as explained above).

Emit expressions are automatically generated to associate output containing map elements with their corresponding input map elements when elements are mapped.

5. *IsPresent* - Defines the conditions where this element is considered to be present and therefore consumed by the input processor. If this is omitted or true, the element is consumed, otherwise (if specified and false) the element is ignored. This is used only for *Flat* (including COBOL) and *EDI*, which must process a stream of data that is not self describing and thus the reader needs additional information to describe which data to process and which to skip. Generally the **IsPresent** expression is not necessary for XML, Java, and database representations, as they are able to determine which elements to provide based on the input data. There are some restrictions on what can be specified in this expression tree:

   - Aggregate functions may not be used.
   - When referencing an element, it will be the value in the current iteration of a loop.
   - Any structure elements referenced must occur at or before the element containing the **IsPresent** expression in the element tree.

   Violations of these restrictions will result in an error when the map or structure is executed.

6. *Null* - Determines when to emit this output map element as a null value. If the result of this expression tree is true, the element is emitted as a null value. The *Null* expression tree is present only for elements that are containers (element group type other than *None*). Non-container elements can get null values simply by using the **Constant** function.

   Null values are carried through from input to output for those representations that support them (Java, XML and database). If the both the input and output elements have the *Null* element property set, and the input element's value is null, when mapped, the output element's value will be null.

   Null expressions are automatically generated to associate output containing map elements with their corresponding input map elements when elements are mapped.

7. *IO/Database* - Provides input/output operations, which for example redirect the reading or writing to another document instance. Also provides database functions, which control how the element interacts with the database. The **DatabaseJoin** and **DatabaseSelect** functions are used here.

8. *Util* - Utility expressions associated with the output map element. Whatever is in the Util tab is executed before the map element is executed, so you can use the **SetVariable** function (and combine it with other functions if required).

Expression trees exist for structure elements and map elements. When an expression tree is defined in a structure element, it is copied to the corresponding map element. This allows you to predefine loop expressions for example in the structure and have all maps that use that structure use the predefined loop expression. The map is also free to override that by providing its own loop expression tree for the given element.

When an input element is mapped to an output map element, the Loop and Emit expression trees are copied from the input map element to the output map element. This continues the idea of creating default expressions to be used for these categories that are associated with the input.

9.3. Expression Types

Most expressions are made from functions, such as the **Copy** function in the above example. Here are the types of Expressions:

1. *Functions* - Created when a function is dragged to the expression tree or to an output map element.

2. *Map Element Reference* - Created when a map element is an argument of a function expression. There can be references to either input or output map elements.

3. *Function Argument* - Represents a formal parameter (argument) to a function. This returns the result of its child expression, which is the value of the argument.

Functions can have a fixed or variable number of arguments. If the function has a variable number of arguments, there will be no function argument expressions when you drop the function on the expression tree. A variable argument function expression can have any number of child expressions.

A fixed argument function results in a function expression having a function argument expression for each of its arguments. Each of these arguments takes exactly one child expression, and if the child expression is not specified, the map cannot be executed (you will get an error explaining the problem).

9.4. Conditional Expressions

The **IfThen** (and **IfThenElse**) functions are used to make conditional execution of expressions. These function expressions are evaluated the same way as any other type of expression.

The **IfThen** function has two arguments:

1. *Condition* - Requires an expression returning a Boolean value. If the result of the Boolean expression is true, the result of the **IfThen** function expression is the *Then* expression. If the result is false, the **IfThen** function expression returns no result.

2. *Then* - The expression that provides the value to the **IfThen** function expression if the condition is true.

The **IfThenElse** function adds a third *Else* argument to the **IfThen** function. The value of the *Else* argument expression is returned if the condition argument evaluated to false.

9.5. Complex Expression Example

This section shows an example of how several expressions can work together.

Example 9.3. Complex Expression Example

```
Concat ❶
  Add ❷
    First Value ❸
      Transaction-856/Segment-ISA/Element-I01
    Second Value ❹
      Const "2" ❺
  Const "-" ❻
  LoopIndex ❼
```

```
    Looping Output Element ❽
       Transaction-856/Loop-HL ❾
```

❶ The root expression is a **Concat** function expression, which is a variable argument function expression.
❷ The first argument to the **Concat** function expression is an **Add** function expression. The second argument is an **LoopIndex** function expression. Both of these arguments are fixed argument functions, so each can have only one child expression.
❸❹❽ The function argument expressions for the fixed argument functions (**Add** and **LoopIndex**). Each of these can have only one child expression.
❺ A **Constant** function expression with the value *2* (as specified in the expression properties). This is the value of the second argument to the **Add** function expression.
❻ A **Constant** function expression with the value - (as specified in the expression properties). This is the value of the second argument to the **Concat** function expression.
❼ A **LoopIndex** function expression. This returns the index value of the *Loop* expression of the specified output map element. This is the value of the third argument to the **Concat** function expression.
❾ An output map element reference, the single argument to the **LoopIndex** function expression.

In this example, we want to construct a value that is a reference to an input map element plus the number *2*, followed by a -, followed by the loop index of the *Loop-HL* output map element. So if the value of *Element-I01* is *5*, and we are on the tenth iteration of the *Loop-HL*, the value of the expression tree is *7-10*.

One way to read an expression tree is from the top down. We see that the root does a **Concat**, at the next level down it is copying the values of the result of an **Add**, a **Constant** and an **LoopIndex**. We then look at each of those expressions to see how they are constructed to get their respective values.

9.6. Text Expression Representation

When text expression representation is selected in the preferences, each expression panel contains a text string that is the expression.

The definition for the text form of expressions is:

```
      Function = FunctionName [ '[' Properties ']' ]
               '(' Argument ',' Argument ',' ... ')'

      Properties = ( [ PropertyName '=' ] '"' PropertyValue '"' ',' ... )

      Argument = Function | ElementRef |

      ElementRef = ( 'in$:' | 'out$:' | 'elem$:' | 'inv$:' ) ElementName

      PropertyName = Name of the property of the function.

      PropertyValue = Value of the property of the function.

      ElementName = Path name of map/structure element. Example: /Transaction-856/BEG/
BEG01

      FunctionName = Name of built-in function, or full path name of user function.
```

Notes:

1. If there is a single property for a function, the *PropertyName* can be omitted.

2. For element references, *in* references an input map element, *out* references an output map element, and *elem* references a structure element (used in a validation expression associated with a structure). *inv* represents a reference to a map element that is not present (this can appear due to structure changes after a map has been defined).

3. If the function refers to a built-in function, only the name of the function need be specified, for example: *Concat*. However, if a user function is specified, the fully qualified name, including the directory name of the function, must be specified. For example: */userDir/userFunc1*.

Some examples:

- *Concat(in$:/a/b)* - Calls the Copy function with the map element */a/b* on the input of the map.
- *Equal(in$:/a/b,Constant["14"])* - Calls the Equal function with two arguments. The first is element */a/b* on the input of the map, and the second is the **Constant** function with a (value) property of *14*. This returns true if the value of input map element *a/b* is equal to *14*.

Chapter 10. Looping Expressions

Looping specifies how to handle elements that can occur multiple times (that is, *Occurs Maximum* is set to greater than *1*) when mapping them to the output elements. This includes looping corresponding to some input, filtering, and sorting. The looping mechanisms also deal with aggregation, which is used to count and sum elements, for example.

The specification for the looping is in the Loop Expression Panel and is automatically generated when you map an element from the input to the output. In most cases, the automatically generated looping is sufficient. However, if you require filtering or sorting, or if you want the output loop to loop in a different way, you must modify the loop expression.

To support aggregation, special aggregation functions are provided that contain their own loop expression, allowing them to process elements from anywhere. However, their default looping is within the enclosing output loop in which they are specified.

Loop Functions provide different ways of looping and provide the means for filtering and sorting.

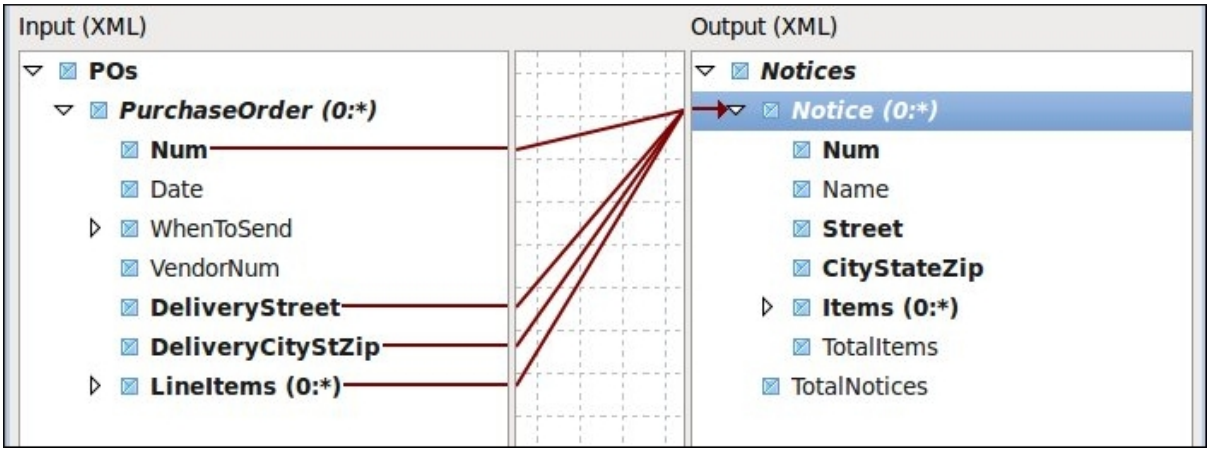
There are times when it's helpful to unroll a map element from a loop so that it can be mapped as if it were not looping. In other cases splitting a loop into multiple loops is helpful.

10.1. Looping Cook Book

This section describes some of the common looping issues and how to map them. Most of the cases described below have an example in the *Examples* project.

10.1.1. Matching Loops

Matching loops (shown in example *Maps/Looping/LoopSimple*) are the easiest looping case. In this case, there is a loop in the output and a loop in the input (at the same level of looping), and you want an output element for each input element. Most of the time the matching looping is correctly calculated automatically when you map an input value to an output value. Here, the *PurchaseOrder* element in the input matches the *Notice* loop in the output.

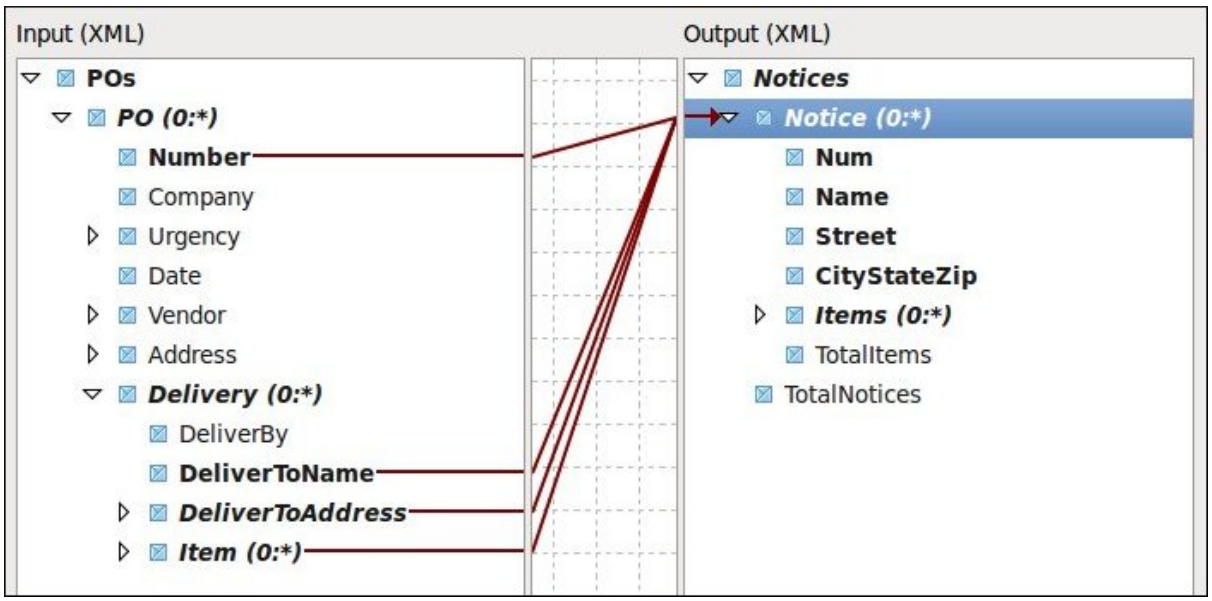


To make a loop in the output match a loop in the input, use the **SimpleLoop** function in the Loop expression tab of the looping output map element. The *Input Map Element* argument takes the looping input map element. As with most of the looping functions, you can add filters and sorting as desired.

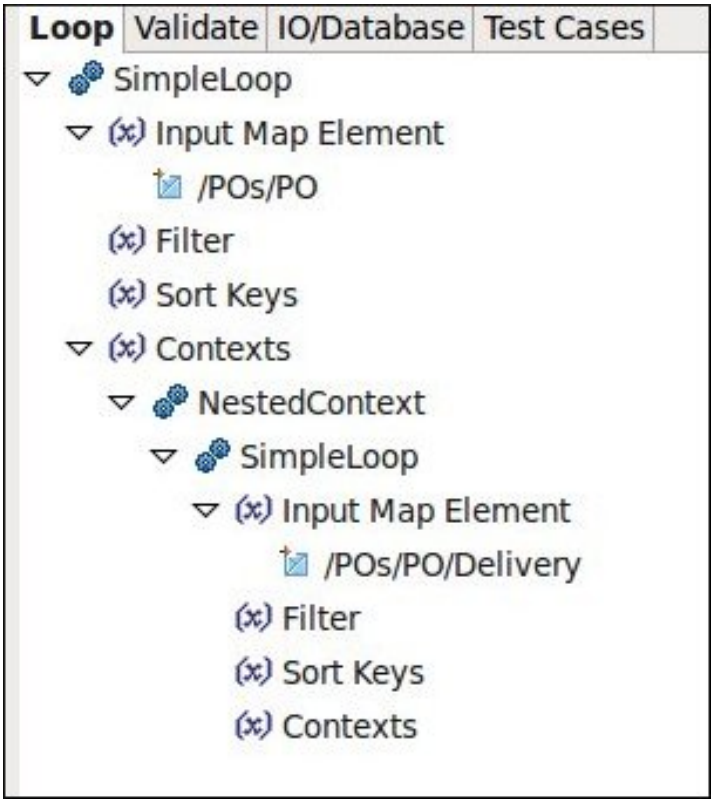
10.1.2. Nested Loops

A nested loop (shown in example *Maps/Looping/LoopSkipLevel*) is the case where a loop in the output refers to a loop in the input that is nested inside of one or more ancestor loops in the input. In this case, the *Notice* loop in the output wants to loop as the *Delivery* loop from the input. The *Delivery* loop is nested inside of the *PO* loop in the input.

As with matching loops, nested loops are also automatically calculated as you map values from the input to the output.



To map a nested loop, use the **SimpleLoop** function in the **Loop expression** tab of the looping output map element. The *Input Map Element* argument takes the top-most looping input map element, then for each nested looping input map element, use the **NestedContext** function and then another **SimpleLoop** for the nested looping input map element. As with most of the looping functions, you can add filters and sorting as desired.



10.1.3. Mapping Non-Looping to Looping

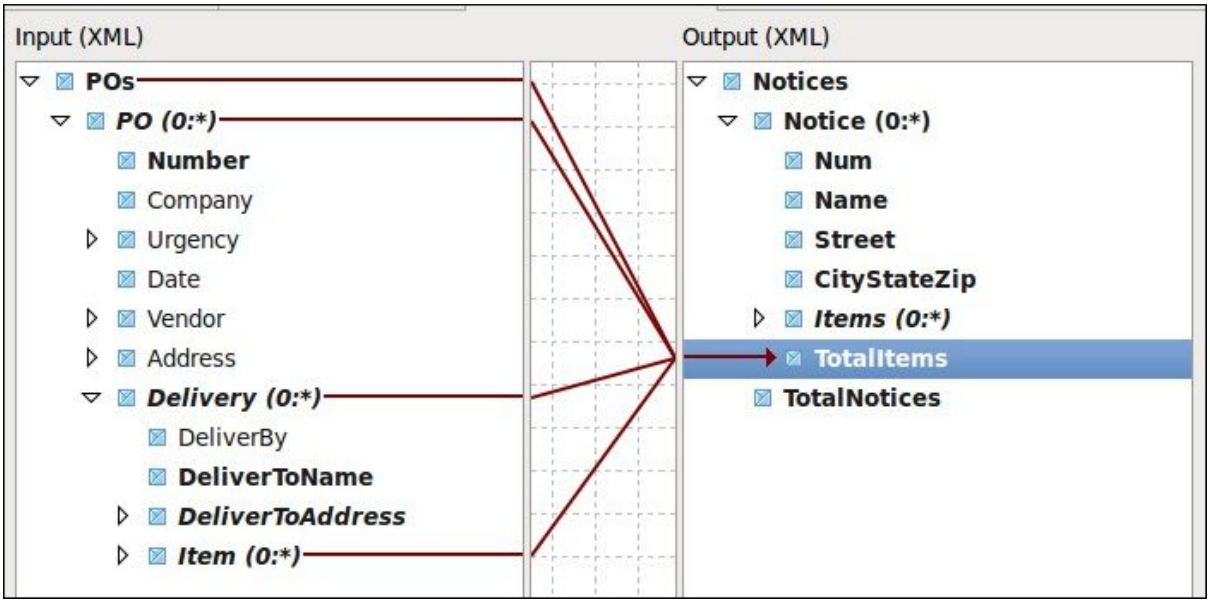
In this case, you probably do not want the loop in the output to loop, as you have only one value in the input to map to it. You have two choices here:

- Unroll the loop - Unrolling the loop is done by right-clicking on the output looping map element and selecting **Unroll Loop**. This makes a separate element that does not loop, and then you can map it normally. The loop still exists in the original map element, which you do not need to map.

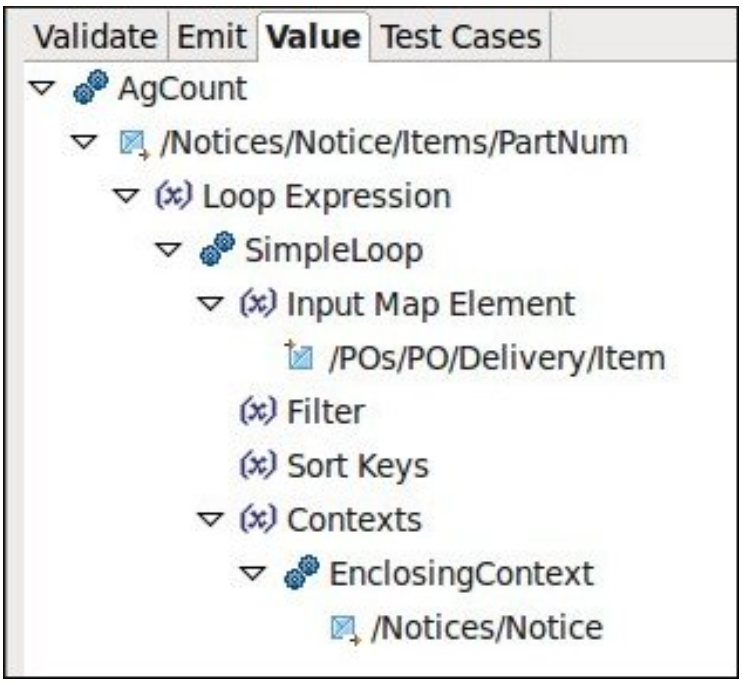
- Use **FixedLoop** - This loop function emits the loop a fixed number of times, by default once. Just use this as the loop expression for the output element and then map the elements normally.

10.1.4. Mapping Looping to Non-Looping

Aggregate functions (shown in example *Maps/Looping/LoopAggregate* - look at the */Notices/Notice/TotalsItems* element) are used in this case where you most often want to map a sum or a concatenation of many values. Other uses for aggregate functions are selecting a particular instance from the input loop. To use an aggregate function, simply map it to the value expression in the output map element. In the case below, we want the *TotalItems* in the output to be the count of all Items (represented by *PartNumber*) in the input.

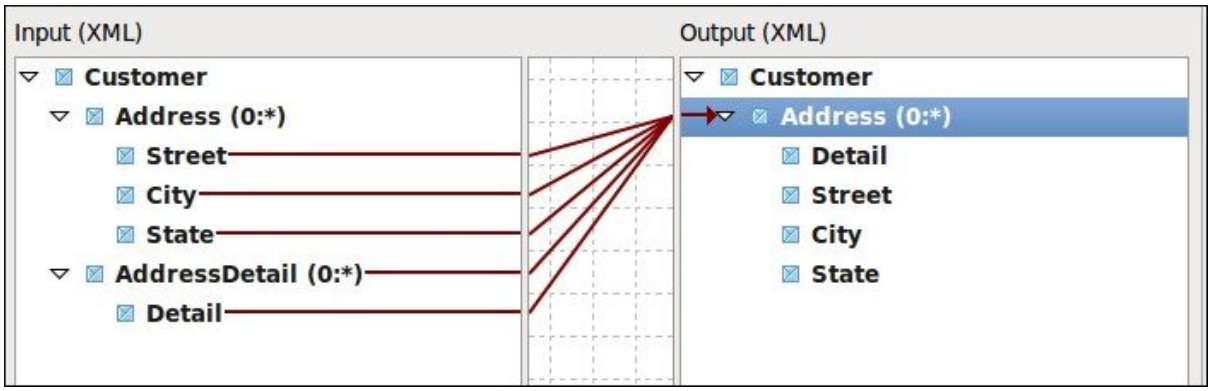


You can see below that for each argument of the aggregate function, a loop expression is required. This loop expression provides the context for its argument. In this case, we are just doing a simple loop to get all of the values of *PartNumber* in the *Item* loop. The context for this loop is implicitly the loop context of the enclosing output map element, which in this case is Notice. You can change the context to use a higher-level loop (if you wanted to count all of the Items for all POs, for example) by using the **EnclosingContext** function.

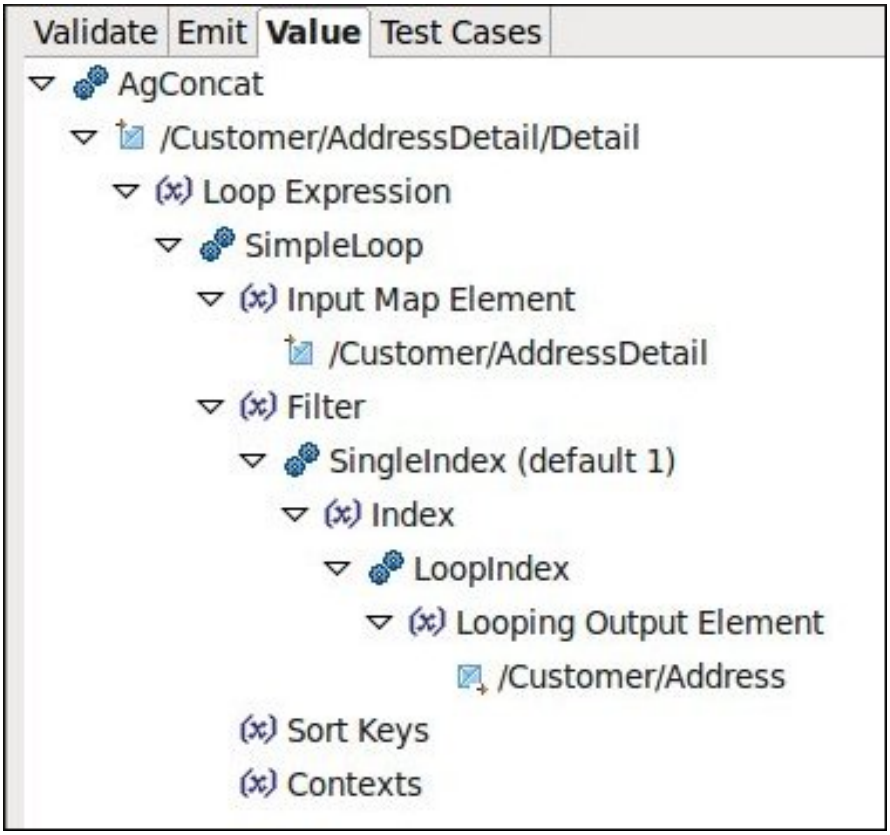


10.1.5. Merging Two Loops into One

In the *Maps/Looping/LoopMergeTwoLoops* example you can see the case where the input side has two corresponding loops to be merged into a single loop in the output. To do this, map the first loop (the one with more elements) normally as you see with *Street*, *City* and *State*.

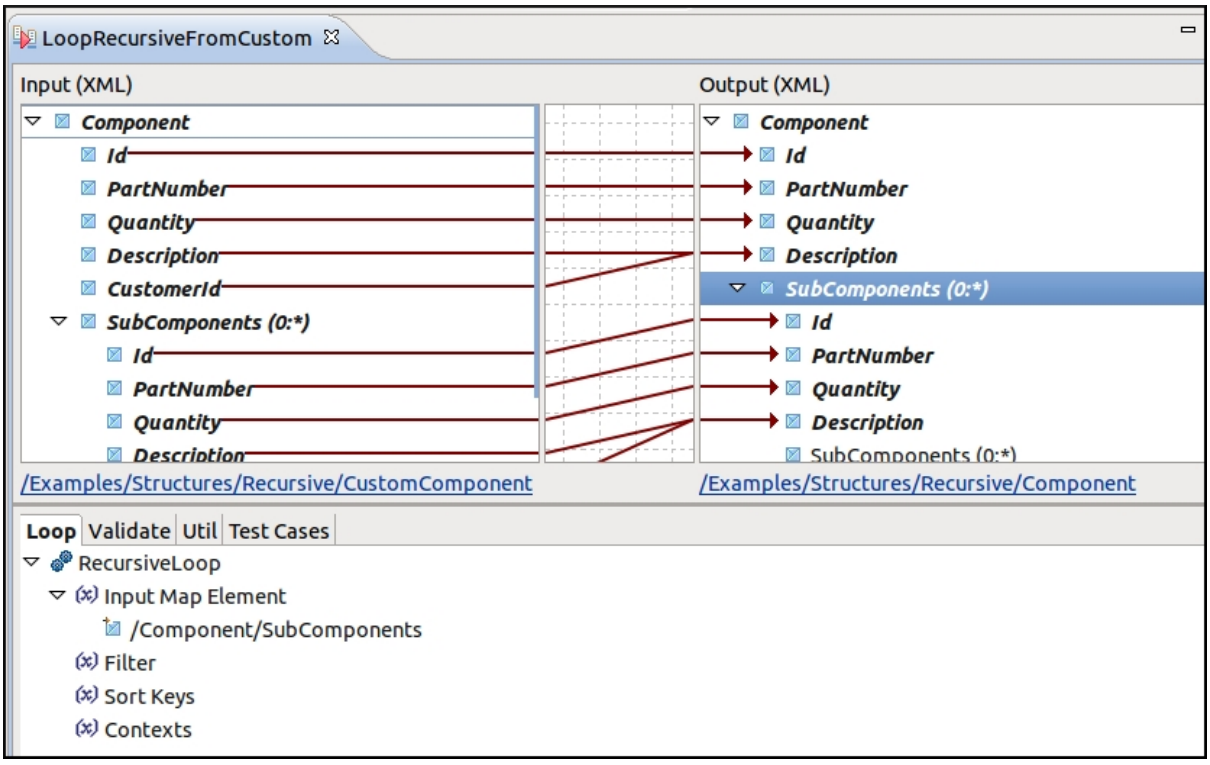


Then for the elements in the 2nd loop (*Detail*) you need to use an aggregate function to pull the value for the desired index from the input loop as shown below.

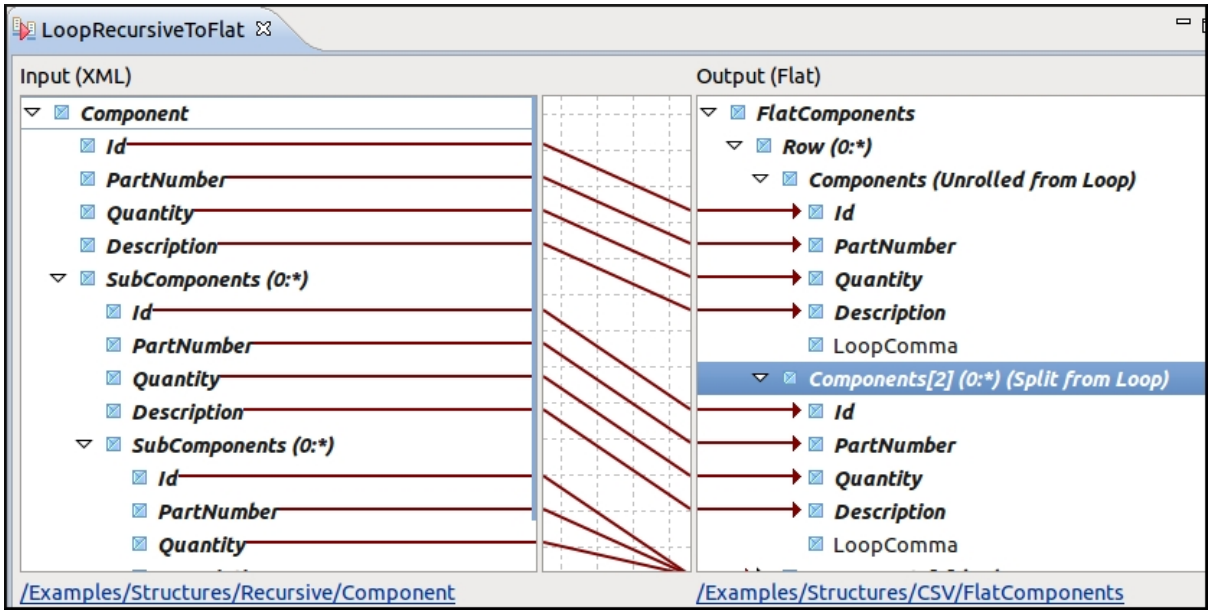


10.1.6. Recursive Loops

In the *Maps/Looping/LoopRecursiveFromCustom* example you can see the user of the **RecursiveLoop** function to easily map the components that recurse to any depth.



The *Maps/Looping/LoopRecursiveToFlat* example shows flattening a recursive structure (using only the first three levels) into a delimited (CSV) structure. The input structure corresponds to only a single row in the output CSV structure. The output structure has a loop for each CSV row and also a loop for the components, as each row is just a set of components.



10.2. Automatic Loop Calculation

When you map an input map element to an output map element, and the input map element is contained within (or is) a looping map element, a loop expression for the nearest containing looping *output* map element is automatically constructed, if it has not been explicitly specified. This output loop expression uses the **SimpleLoop** function to loop through the input map elements.

This process is repeated, going up to the next containing looping input map element and matching that with the next containing output looping map element until a loop expression is found for an output looping map element. By default, a loop expression is created for the root output map element, as it is assumed that there is exactly one output document for every input document.

In the case where the input side has more containing loops than the output the generated loop expression will include the extra loops as nested loops. Where the input side has fewer containing loops, the generation of the output loop expressions stops when the input loops are exhausted and the remaining containing looping output elements are left without a loop expression which will cause a warning when the map is executed. This warning tells you to specify a loop expression for the output (which could be something like a **FixedLoop** function).

10.3. Filtering and Sorting

All of the loop functions have a *Filter* argument that takes a single Boolean expression. This can be used to provide any criteria for filtering the loop. To extract only a single instance of a loop by index value, use the **SingleIndex** function.

Sorting is done using the *Sort* argument. The Sort argument takes a variable number of **AscendingSort** and **DescendingSort** functions. Each of these functions takes a value that is the actual sort key.

10.4. Looping Contexts: Enclosing and Nested

A loop expression specifies how a given output map element loops. It refers in some way to an input map element. Therefore, any input map element that is used in any subordinate (descendant) element of this output map element refers to a value from the looping input map element.

Looping contexts are specified using the **NestedContext** or **EnclosingContext** functions with the *Contexts* argument of the loop function.

10.4.1. Nested Context

When referencing a looping input map element in the output, it is necessary for all ancestor looping map elements in the input to have an associated output map element loop expression. Most often this occurs naturally if your output looping corresponds to the input looping. However, there are times when there is a gap in the correspondence, and this gap can be filled in using the **NestedContext** function by providing the instructions for handling the looping input elements that were not referenced by output elements.

To create a nested loop expression:

- Create the outer loop expression - Drag the desired loop function to the loop expression tree, and then drag the desired input map element to the *Input Map Element* argument.
- Create nested context - Drag the **NestedContext** function to the *Contexts* argument of the loop expression.
- Specify the nested loop - Drag the desired inner loop function to the **NestedContext** expression, and then drag the desired inner input map element to the *Input Map Element* argument.

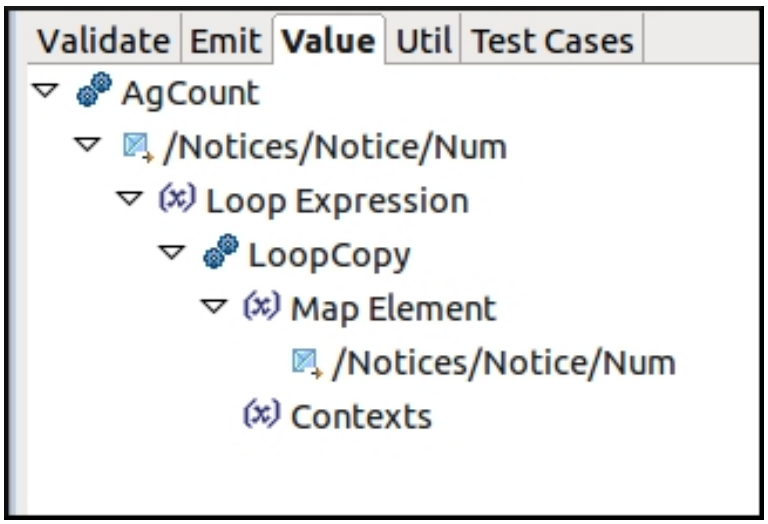
10.4.2. Enclosing Context

The Enclosing context is used by loop expressions associated with aggregate functions (those functions starting with "Ag"). Normally this type of looping is done within the current output map element's looping. For example, if you have a loop in the output for an invoice, and you want to calculate the total of all of the line items, you use the **AgSum** function and simply specify a loop on the line items. It will automatically assume you want only line items from the enclosing invoice. However, if you want to include line items from other invoices, use the enclosing context to specify a loop higher than the invoice (the root output element in this case), and then the **AgSum** function will look at all line items of all invoices.

10.5. Aggregate Looping

Non-aggregate functions execute in the looping context of their enclosing expression and output map element. Sometimes it is necessary to access data from an unrelated loop, for example, you might want to take the sum of all values from the input. Use aggregate functions for this. Each aggregate function allows you to specify a loop expression for each of its arguments, creating a new loop context for each argument.

The loop expression for an aggregate function is always the first argument of each argument of the function. In the **LoopAggregate example**, the value expression for *TotalNotices* counts the number of *Notices/Notice/Num* elements in the output.



Note the loop expression below the map element reference to *Notices/Notice/Num*. That is where the loop expression is placed. In this case, the loop expression is a reference (**LoopReference**) to the loop expression of the output element *Notices/Notice/Num*.

10.6. Loop Compatibility

Let's go through a couple of examples to demonstrate loop compatibility. Suppose you have an input structure that looks like this:

```
Document
Element1
LoopA (0:-1)
ElementA1
ElementA2
LoopB (0:-1)
ElementB1
```

In this structure, *LoopA* and *LoopB* loop, and the other elements don't. Now suppose you create a map that has this structure as both the input and output. What happens if you map input *ElementA1* to output *ElementA1* but map input *ElementB1* to output *ElementA2*? This does not make any sense, because when you initially map input *ElementA1* to output *ElementA1*, the Editor calculates that output *LoopA* loops using input *LoopA*. Then when you map *ElementB1* to *ElementA2*, the Editor cannot know which values of *ElementB1* to select. This is an instance of two map elements that have incompatible loops. If you attempt to make such a mapping, the Editor will give you an error and not allow the mapping.

Let's consider another case. Suppose you first map input *Element1* to output *ElementA1*. The Editor will calculate that output *LoopA* loops with the *Document* element, since *Element1*'s nearest enclosing looping element is *Document*. Now you map input *ElementA2* to output *ElementA2*. What you intend is for the output *LoopA* to loop using the input *LoopA*, but currently the output *LoopA* is looping with the input *Document*. In this case, the Editor will point out this fact, and ask if you want to change the output *LoopA* to loop as the input *LoopA*.

Loops are compatible when input map element *A* is enclosed in looping map element *AI*, and input map element *B* is enclosed in looping map element *BI*, and if *BI* is either the same map element or an ancestor map element of *AI*, map element *A* is loop compatible with map element *B*. Otherwise, they are not loop compatible.

10.7. Recursive Element Mapping/Looping

In XML, it is possible to define recursive elements, that is, an element whose children are the same children as some ancestor of that element. Recursive elements can be mapped in any of the following ways:

- Recursive Expansion - In the map editor, each recursive element has a menu item called **Expand Recursive Elements**. Clicking on this will add one level of expansion of the children of the given element. These can be mapped normally, and you can expand as many levels as desired. If the expansion is no longer needed, use the **Remove Recursive Child Elements** menu item.

  Use this in the case where you have a fixed depth that you need to map your recursive elements to.

- Recursive Mapping - In this case you have a recursive element in the output that needs to be mapped to some recursive element on the input. You want the recursive elements to dynamically correspond. To do this, use the **RecursiveLoop** function. This dynamically creates a level of dynamic recursion in the output as it discovers the recursion levels in the corresponding input.

- Flat to Recursive Mapping - In this case you have a non-recursive input that you want to map to a recursive output. The input has a level number (or one can be computed based on the input state) that defines the depth of the output recursive element. To do this, use the **FlatToHierarchyLoop** function.

10.8. Loop Functions

All of the loop functions appear in the *Loop* header in the functions view. Only these loop functions can be root expressions of the loop expression tree.

FixedLoop
FlatToHierarchyLoop
IndexRangeLoop
LoopReference
SimpleLoop
RecursiveLoop

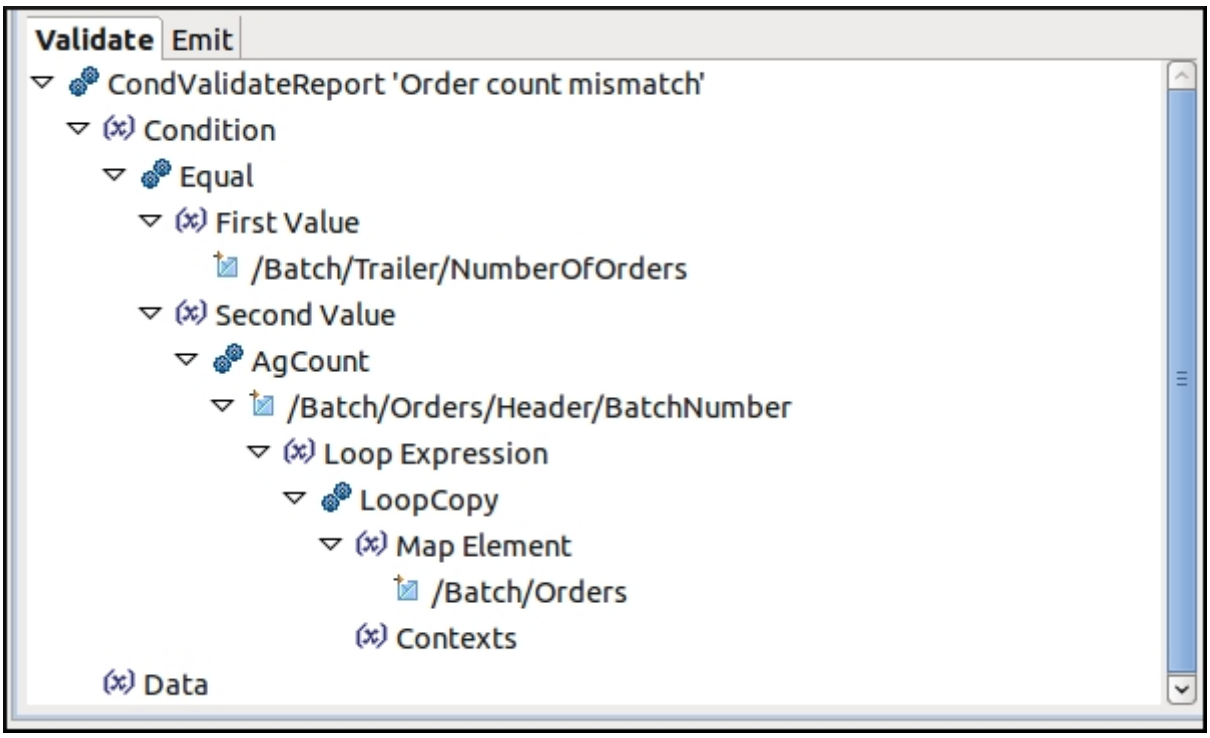
Chapter 11. Validation

11.1. Validation

Validation is done at runtime to ensure that documents pass certain constraints. This chapter shows how to make the constraints and also shows how the validation reporting works.

11.1.1. Validation Constraints

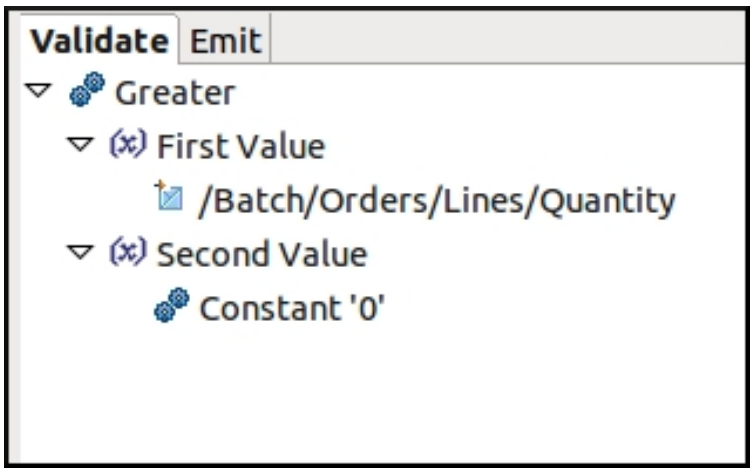
Validation constraints are implicitly specified in the definition of a structure. This includes things like the data type, number of occurrences, and element length. Validation constraints can also be specified either in the structure or map definition in terms of expressions in the **Validate** expression tab for each map element.



Validation constraints can be specified either in a structure or a map. When you specify constraints in a structure, they are copied into the map at the time the structure is mapped. If the validation constraint is updated in the structure subsequently, it has no effect on the map.

11.1.1.1. Validation Expressions

The validation expression is associated with both map and structure elements. You specify the expression in the **Validate** expression tab. These expressions require a Boolean value. If the value is true, the element is consider valid. If false, a validation report is generated. The validation report is generic in that it only indicates the validation of the element failed and gives the location of the failure.



If you want to provide more information, use the **CondValidateReport** function, which you can put at the root of the validation expression.

11.1.1.2. Validation Report Functions

The **ValidateReport** function unconditionally indicates a validation failure and can be used in any type of expression. The **CondValidateReport** is the same, except that you can specify a condition that indicates a validation failure if false.

With both of these functions, you can provide a severity, error number, and error message text. In addition, you can provide any other data necessary to provide context for the report, such as the value of a map element. All of this will be captured for the validation report.

11.1.2. Grouping Validation Errors

Suppose you are processing a batch of *Order* elements, and you want to reject any *Order* element that has a validation error but process normally those that have no error or have just a validation warning. To do this, you must provide a grouping for any validation reporting at the *Orders* level by using the **ValidateGroup** function. Specify this function in the **Validate** expression tab of the element you want to be the group (the *Orders* element in this case). Any validation reports coming from subordinate elements will trigger this validation group and also cause a validation report for the group (which is associated with the triggering reports).

You can use the validation group as a filter in a loop using the **IsValid** function.

11.1.3. Validation Reporting

Validation reporting deals with what happens when the validation constraints are not met.

Whenever a map is executed, an execution status object is created that contains any sort of exception information associated with the map execution. This includes any validation errors. The execution status object is defined by a built-in structure *ExecutionStatus* and is automatically provided to an execution status map to handle the error processing upon completion of the executed map. The execution status map can be specified for each map or globally for the entire runtime.

Chapter 12. Input/Output (Multiple Input/ Output Documents)

This chapter discusses how to read and write files from maps and how to handle multiple data sources (both input and output) and multiple messages within a map.

12.1. General

Maps that read from one source (typically a file or the payload of a message) and write to one result generally do not have to have any I/O expressions at all. They simply read and write from the documents that are given at map execution times.

You use I/O expressions only if you need to process multiple input or output documents or objects in a single map, you want to "wire in" the URL of documents in the map, or you want to handle differing representations (e.g., flat embedded in XML) within a map.

You can use I/O expressions to do the following:

- Read and write from/to multiple data sources (files, objects, or messages) in a single map
- Specify the location of data to be read/written within a map, for example, specifying the name or URL of a file
- Switch to another representation within a single map

A map is composed of exactly one input structure and one output structure, although these structures may incorporate other structures using the structure inheritance mechanism. By default (in the absence of the explicit use of I/O expressions), when a map is executed, the source data is specified and the output result is specified. For example, when you test run a map, you specify a source document. and the result is displayed in a window or to the test output document.

I/O expressions are allowed only in elements that inherit from structures, so each document instance corresponds to a single structure (which can inherit from multiple other structures).

The I/O expression is processed only for the element on which it is declared. When that element is finished, processing reverts to the I/O expression of its nearest enclosing ancestor. This means that you could process different subordinate document instances inside of a single outer document instance, continuing to process the outer document instance in between the subordinate instances.

Using I/O expressions allows you to have multiple sources (that is, read from multiple files) or produce multiple results in the execution of a map. To specify an I/O expression, drag a *Read* or *Write* function (such as **ReadURL**) to the I/O expression panel. When you specify an I/O expression on an input map element, that map element and all of its children will be read using the specified *Read* function. When you specify an I/O expression on an output map element, that map element and all of its children will be written using the specified *Write* function.

If you specify any I/O expressions for a side (input or output) of a map, all of the I/O associated with the map is done through those expression(s), and any elements not encompassed by an I/O expression will not be read or written.

12.1.1. Default I/O (No I/O Expressions)

If no I/O expressions are specified for a side (input or output) of a map, it is treated the same as if a **ReadMapInput** or **WriteMapOutput** function was specified as the I/O expression at the root. A single document, specified by the runtime, is read, and a single document is written.

12.1.2. Using the URL Expressions

If you want to process multiple documents, use the **ReadUrl** or **WriteUrl** functions. You must provide the URL either as a property or argument of the function. This allows you to calculate the URL at runtime if desired.

12.1.3. Using Multiple Source or Result Objects

To process multiple *Source* or *Result* objects use the **ReadUrl** or **WriteUrl** functions. In this case, it's not necessary to specify the URL. You can assign the *Source* or *Result* objects using the *MapExecutionContext.addInputSource()* or *MapExecutionContext.addOutputResult()* methods. This allows multiple Java objects to be read or written in a single map execution for example.

12.1.4. Embedding Multiple Representations

Within the same document instance, it is possible to switch to a different representation and therefore handle embedded data. This is particularly useful for example when processing multiple unrelated XML documents in a single document instance. Even though the XML documents may have different sets of namespaces they can be processed as if they were a single document by enclosing them in a structure with a flat representation.

The mechanism for embedding representations is to use the **ReadNested** at the element that inherits from the structure of the desired representation to switch to. Look at the first example for how this is done.

When using the **ReadNested** function, you must use another I/O function enclosing it. For maps that don't have any special I/O handling beyond **ReadNested**, use the **ReadMapInput**.

The **ReadNested** function can also be nested. In the first example the outer representation is flat, which contains two XML structures one of which contains another structure switching back to flat to handle a small portion of CSV embedded in the XML. The second example shows this in isolation.

12.2. I/O Functions

The I/O functions are shown in the function tab under **Input Output**. There are two types of I/O functions: *Read* functions, which can be used only for input map elements, and *Write* functions, which can be used only for output map elements.

12.2.1. Read Functions

ReadURL - Reads a document instance at a URL.
ReadNested - Processes the element using the representation of the inherited structure.
ReadMapInput - Reads the document specified as the input of the map.
ReadMessage - Reads a document instance from an ESB message.

12.2.2. Write Functions

WriteURL - Writes a document instance at a URL.
WriteMapOutput - Writes the document specified at the output of the map.
WriteMessage - Writes the document specified at the output of the map.

Chapter 13. Mapping Avro

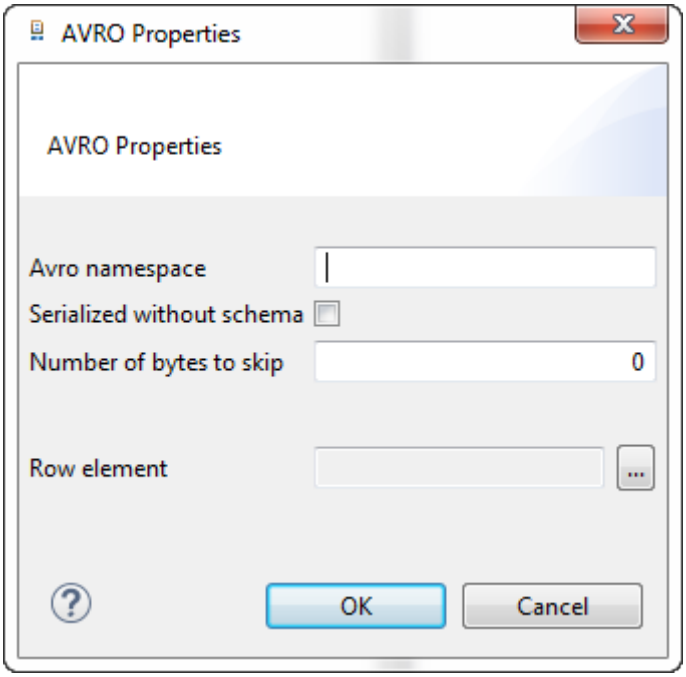
13.1. Overview

The Avro representation allows you to read and write Apache Avro files. Avro serializes data in a compact binary format, and is used mainly with Apache Hadoop.

13.2. Properties

Properties of the Avro representation:

- Avro namespace: This is similar to the XML namespace and can be used to further qualify the Avro schema generated by Talend Data Mapper.
- Serialized without schema: If you select this checkbox, the raw Avro data is produced instead of the more traditional serialization method, in which the Avro schema comes first, followed by the data.
- Number of bytes to skip: This lets you specify a number of bytes to skip at the beginning of the payload. This may be needed when the Avro schema provided matches only part the payload, past the number of bytes to skip.
- Row element: This lets you specify which element Talend Data Mapper should use to identify where a new row begins.



Chapter 14. Mapping COBOL

14.1. Overview

The COBOL representation is intended as an alternative to using the Flat representation when working with binary EBCDIC files.

Using the COBOL representation provides better performance but has some restrictions compared to the Flat representation.

In particular, the following restrictions apply to the COBOL represenation.

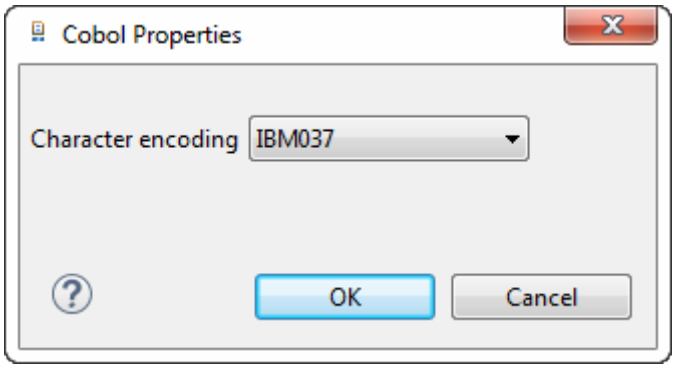
- It does not support Initiators/Terminators (they are ignored if you set them).
- It does not support offsets.
- It does not support non binary input files.
- It does not support newline characters.

For this reason, the COBOL representation is not the default representation for working with COBOL data types. When you import a COBOL copybook, the default representation remains Flat.

14.2. Properties

Properties of the COBOL representation:

- Character encoding: Specify the character encoding for the data to be processed. See Character Encodings for more information..



Chapter 15. Mapping Database Tables and Database Lookup

15.1. Overview

Database support is deprecated starting with version 6.4. There will be no further developements and features using JDBC drivers, and you are encouraged to use other components instead. Database support will be removed in a future release of the product.

The database support allows you to read and write from databases using a map. To use the database support, you must first import the database using the Import mechanism (**File -> Import**) in the studio. This reads all of the tables of the database into a set of structures, one for each table. Each database structure has a loop (with an element called *Row*) that corresponds to each selected or written row.

To read from the database, use the structure corresponding to the table as the input structure of a map. If you are reading from multiple tables using a database join, you can create a new structure representing the joined tables using the **[New Structure]** wizard (select the create a database join table option). When creating the map you can specify a SQL select statement using a **DatabaseSelect** function. You can also use **DatabaseJoin** function(s) to specify conditions on join(s) as required. All database functions are placed in the IO/Database expression tab.

To write to a database, use a structure corresponding to the table as the output structure of a map. Use a **DatabaseInsert** or **DatabaseUpdate** function as required.

You can read from or write to any number of database tables in a single map by creating an enclosing structure that inherits from the desired database table structures.

Lookup functions are also provided to find and update values from database tables. To use these, import the database as described below, and then use the functions in your mapping expressions, specifying the required database and tables.

15.2. Importing from a Database

- Select **File -> Import** and open the **Data Transformation** folder and select **Database**.
- Select the directory to contain the database table structures. The default selection will put the database under the name of the database at the top level structure folder.
- Fill in the database properties required to connect to the database. Once you have filled this in, press the **Test Connection** button to verify the connectivity to the database. If there is a problem connecting, resolve it by specifying the correct database name, user and password information, or resolving any underlying connectivity problem. Press next to continue the import. All of the tables will be imported.

15.3. Database Drivers

Both and studio and runtime provide all of the JDBC drivers for the supported databases with the one exception being MySQL. For MySQL you will need to download the driver Jar file from the MySQL site and put it in the directory identified by the studio which is shown when you attempt a Test Connection in the database properties. You will need to do this only once for your installation of *Data Mapper*. You will also need to include this Jar file in your runtime classpath.

Custom database drivers are also possible (see below), and they require the using the project classpath mechanism and including the Jar file in the runtime classpath.

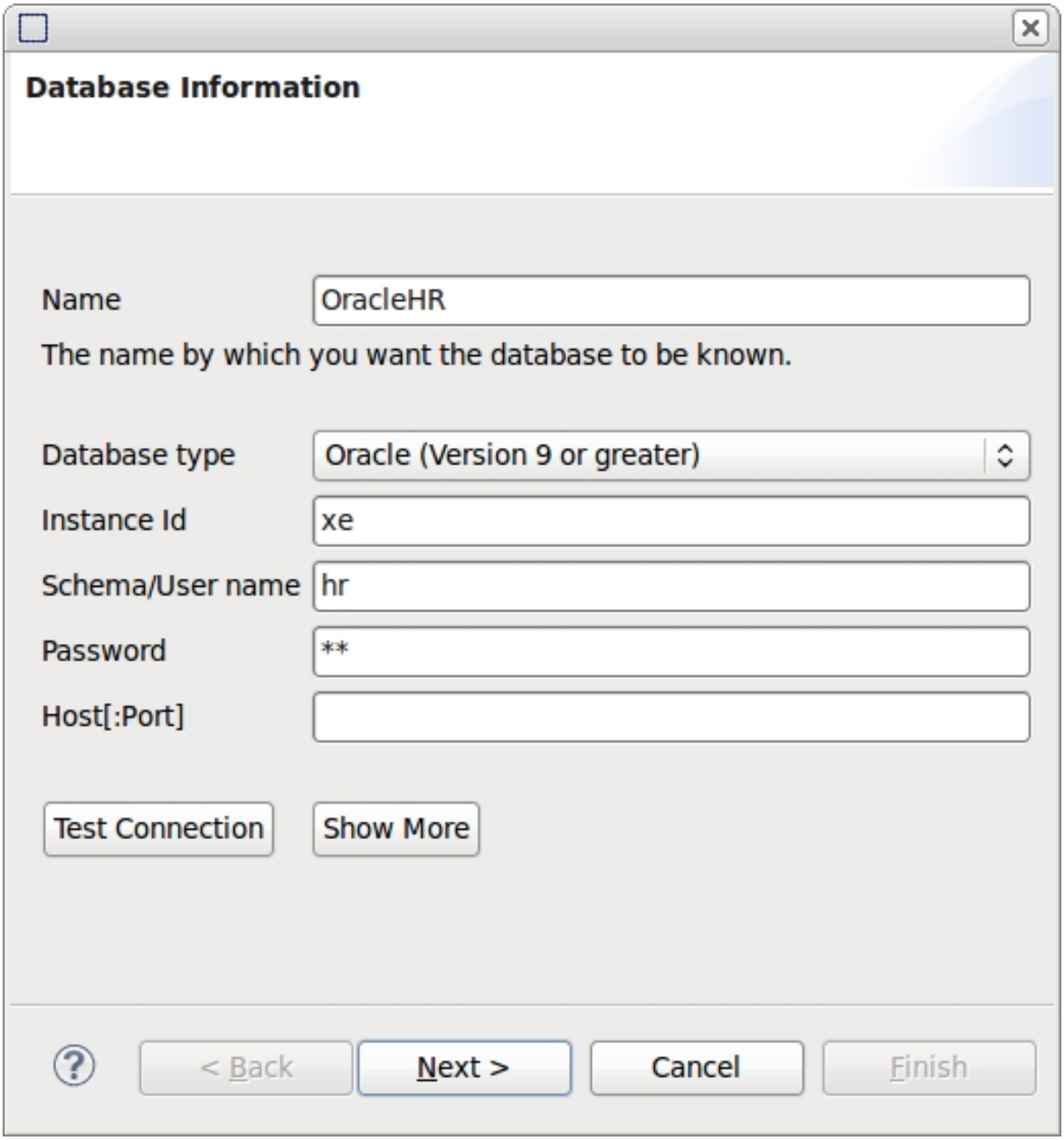
15.4. Database Information

The database is described using a database object which appears in the **Repository navigator** and a set of structures that describe each of the tables that were imported from the database.

15.4.1. Database Properties

This property sheet describes the database connection information and allows you to test the connection to the database. This is used both when you are importing the database and when you show the properties of the database in the **Repository navigator**.

15.4.1.1. Basic Properties

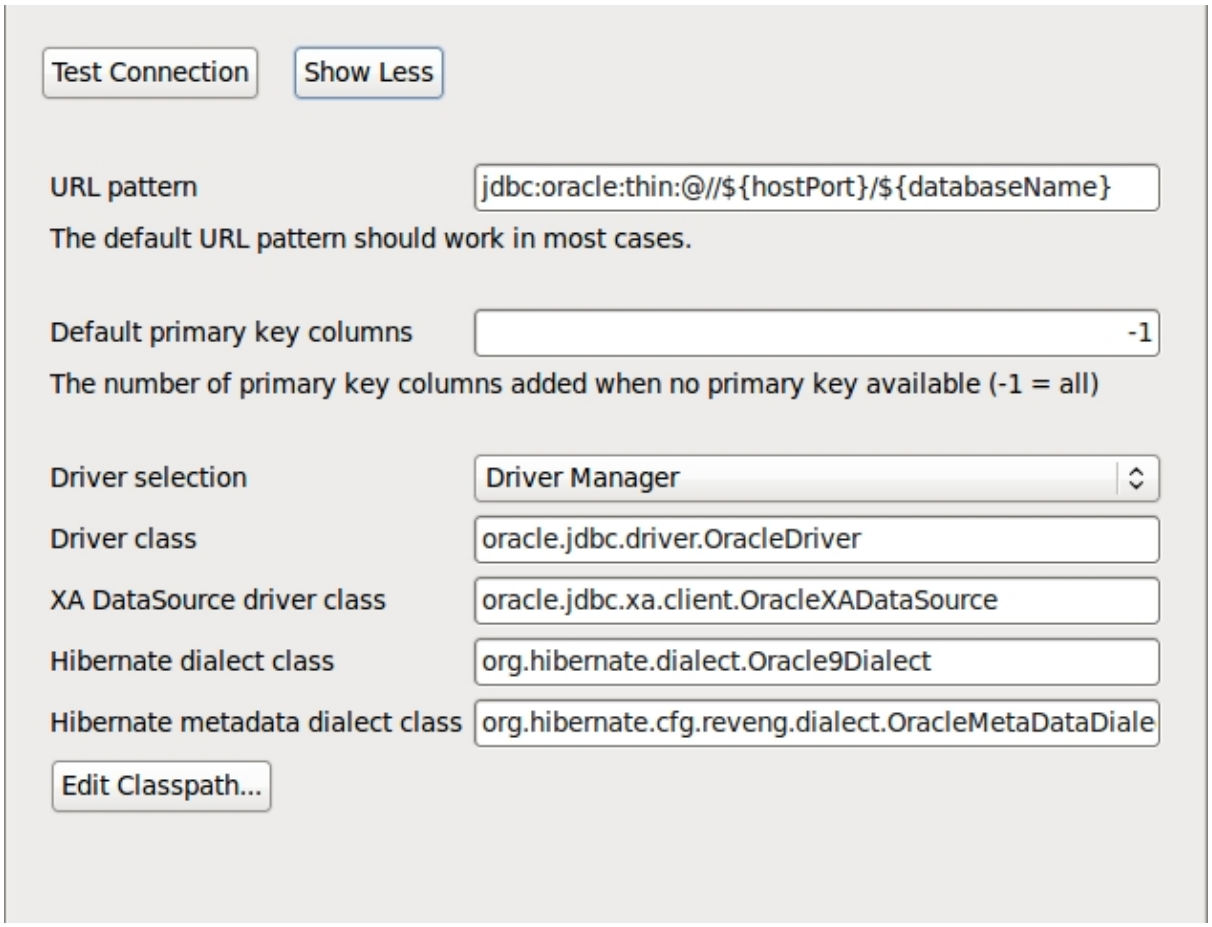


Database information basic properties:

- **Database Name** - The name of the database as you wish to refer to it. This will be the name of the object that is visible in the **Repository navigator**.
- **Database Type** - Specify the type of database using the list.
- **Database Name or Instance Id** - Specify the name of the database or the instance Id of the server. The label on this will change depending on the type of database selected.
- **Schema/User Name** - For Oracle this is the Schema name, for all other databases this will appear as the user name.
- **Password** - The password of the above user.

- **Host** - The host name to connect to. Leave blank for the current host.
- **Port** - The port number to connect to. Leave as zero for the default port number for the database.
- **Schema** - Some databases (SQL Server and Sybase) have the notion of a schema within the database. You can specify a schema value here (like *dbo*) which will be used to filter the tables extracted from the database.
- **Test Connection** - Use this to immediately connect to the database and report any errors.
- **Show More** - Press this and you will see the advanced properties (below).
- **Structure Folder** (not shown on screenshot) - The folder that contains the structure definitions for the database that were generated on the import. This appears only when viewing the properties of the database object. It does not appear in the import wizard.

15.4.1.2. Advanced Properties



The advanced properties are normally things you will not need to change; they are automatically set up depending on the choice of database type. However you can change these as required, even to the point of supporting a different database type.

- **URL pattern** - A pattern used to construct the connection URL. In most cases you will not need to change this, however you can customize it as required. The *${databaseName}* and *${hostPort}* place holders will be substituted with actual values at runtime. This is automatically set to work with the default driver class using the driver manager mechanism (see below). If you wish to use a DataSource, set this to be the URL used to lookup the DataSource in JNDI.

- **Default primary key columns** - In the (rare) case where there is no primary key specified by the database, you can set the primary key based on the leading columns in the table. For example, if you specify a 1, the primary key will be set to the first column in the table (when not specified otherwise by the database). If you specify *-1* (the default), then all of the columns in the table will be considered the primary key.

- **Driver selection** - Specify Driver Manager to use a JDBC driver class that is on the classpath or built in. Specify Data Source if you wish to use a JNDI lookup using the specified URL pattern to get the DataSource.

- **Driver class** - The class name of the JDBC driver to use. Each database type has a default driver class, and the Jar file that contains the driver is automatically included in the product, both at design and runtime. If you wish to use your own driver, modify the project's classpath (see **Edit Classpath**) below so that you can test it in the designer. At runtime, make sure the driver class is available on the classpath (the project's classpath is used only at design time).

- **XA DataSource driver class** - The class name of the XA data source that is provided when distributed transaction support is required. Like the driver class, this is set by default to a class automatically included.

- **Hibernate dialect class** - The class name of the Hibernate dialect class that is used to handle SQL translation for the database.

- **Hibernate metadata dialect class** - The class name of the Hibernate metadata dialect class to handle issues when reading database metadata.

- **Edit Classpath** - Use this to change the classpath that is associated with the project containing the database. You can also change the project's classpath in the project properties (from the **Repository navigator**). You would change the classpath if you want to test with a database driver for example that is not built in.

15.4.2. Database Structures/Tables

When a database is imported, two structures are created for each table in the following categories (each represented by a folder):

- **Tables** - The structure which contains a loop representing the rows of the table. You use this structure directly in a map. You can also open this structure in the structure editor and view the database data using the **Show Document** button or **Show Sample** menu item.

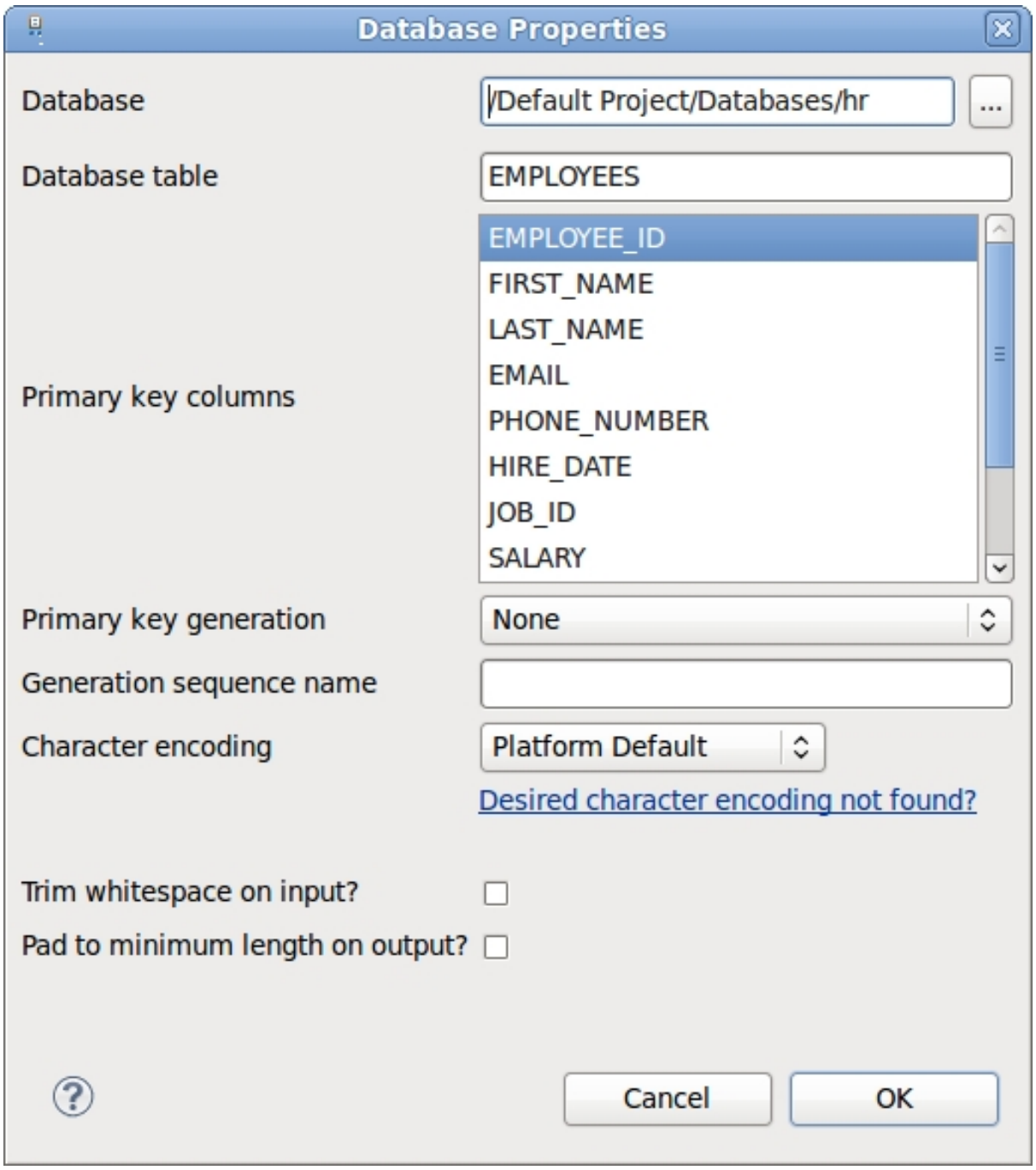
  If you are reading from or writing to the database, this is the one to use. When reading, use the **DatabaseSelect** function in the *Row* element. When writing use either the **DatabaseInsert** or **DatabaseUpdate** function in the *Row* element.

- **Single Row Tables** - The structure which defines the columns of the table. The Tables structure inherits from this one. In addition, each of these structures has a Database representation which defines the properties associated with that table, like primary key and automatic ID generation. Finally, this is also used to construct joined tables.

- **Joined Tables** - This is initially empty and is where tables that represent a join are placed by the join wizard.

15.4.3. Database Representation Properties (Single Row Table Structures)

The database representation describes properties associated with each table.



The following properties are available:

- **Database** - Refers to the database node which contains the global information for the database.
- **Database Table** - The table in the database corresponding to this structure.
- **Primary Key Columns** - Select the column(s) that form the primary key for this table. These will be automatically set on the import.
- **Primary Key Generation** - This controls how the primary key is to be generated for this table. Select one of the following options:
  - **None** - Does not generate anything for the primary key. The value is expected to be mapped.
  - **Native** - Automatically select the primary key generation most appropriate for the database.

- **Identity** - Supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The used for identity columns whose type is long, short or int.
- **Increment** - Generate a sequential number by incrementing a local value before an item is inserted.

  This is not suitable for an multi-user (including multi-threaded) environment where more than one process/thread may be inserting into the same table at a time.

- **Sequence** - Uses a sequence in DB2, PostgreSQL, or Oracle. The used for identity columns whose type is long, short or int.

- **Generation Sequence Name** - Used in the case where **Primary Key Generation** is either *Sequence* or *Native* (and the native implementation uses *Sequence*). This specifies the sequence name to be used. If not specified, *hibernate_sequence* is used.

See also the common properties associated with representations.

15.4.4. Reloading Definitions from a Database

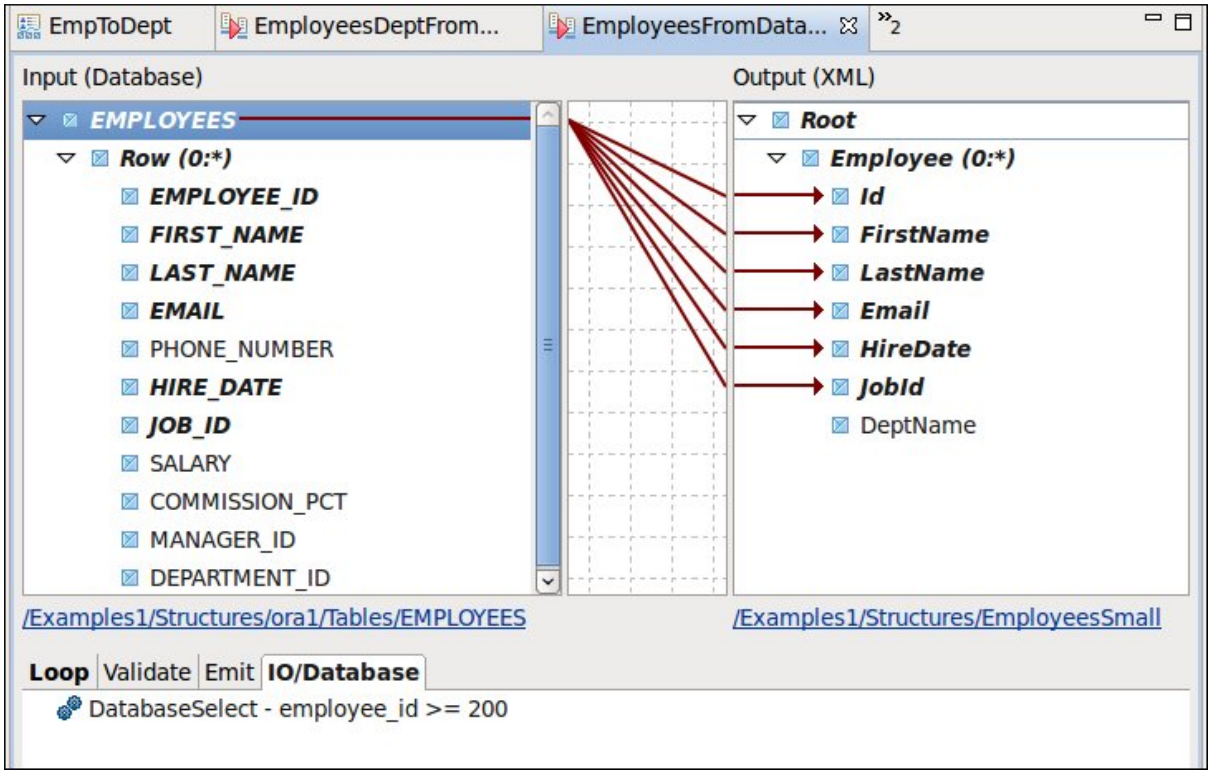
Often the database will have changed and you need to get the current table definitions. To do this, simply right-click the database object and select **[Re]Load Table Definitions**. This will recreate all of the database structures from their current values. It will recreate the structures in place so it will not disturb any other structures or maps that depend on these. If there are changes, these changes will be automatically seen by the maps that depend on them.

15.5. Database Lookup Functions

The database lookup functions allow a value of a single column for a single row to be looked up based on a specified condition. The **DatabaseLookup** returns a value and is read only. The **DatabaseLookupAndUpdate** is used to do a database update at the same time. This is used for the common case of processing a row and then updating a value indicating the row has been processed.

The database lookup functions can be used whether or not you read and write database tables.

15.6. Reading from a Database



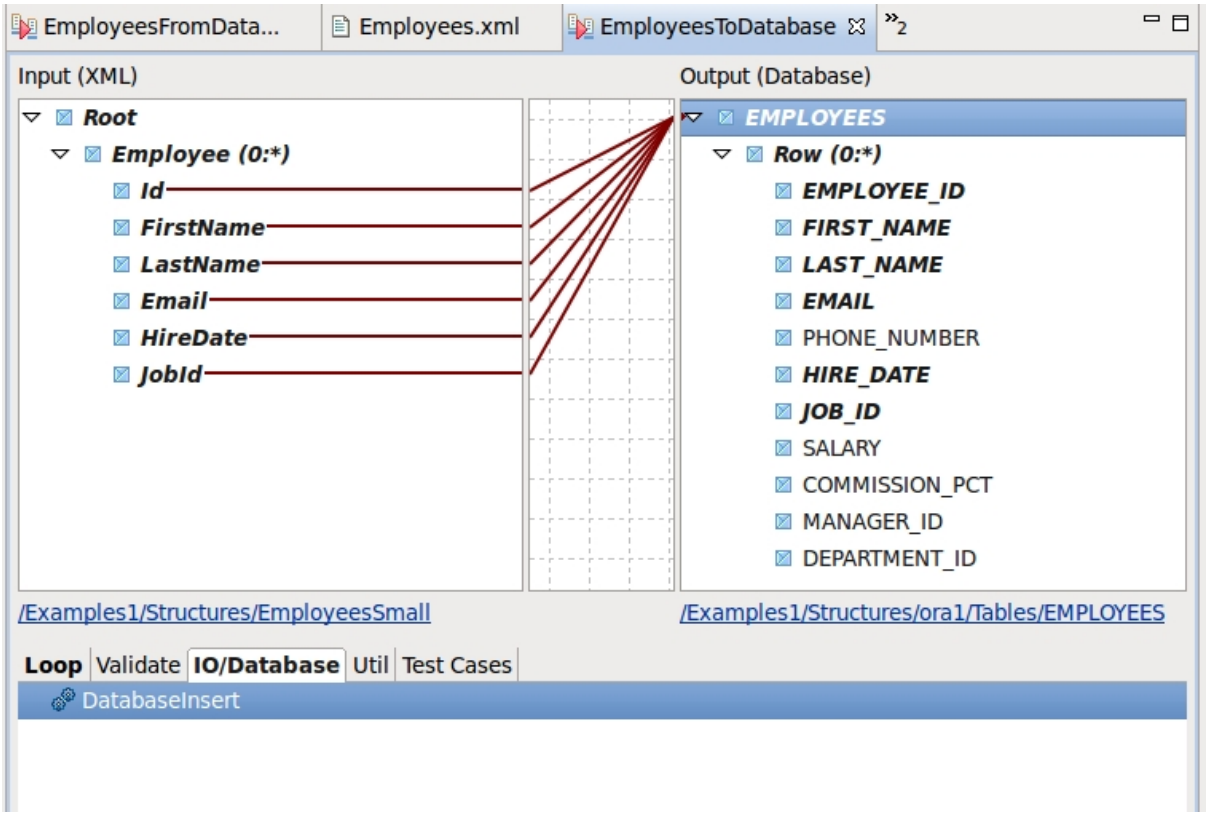
When reading a table from a database, use the structure in the Tables as the input to your map. By default, it will read all of the rows from the table. If you wish to select less than that, use the **DatabaseSelect** on the root element of the table (the element about the *Row* element).

If you wish to read multiple unrelated tables (in the same or different databases) as an input to a map, create an enclosing structure with an element for each table to read. And in each table element, inherit from the Tables structure and then specify the **DatabaseSelect** as usual.

If you wish to read from multiple tables related by a join, see the join handling section below.

In the above screenshot you can see that it is reading all rows from the *EMPLOYEES* table whose *EMPLOYEE_ID* is greater than or equal to *200*. Those elements are then mapped to an XML structure for the output.

15.7. Writing to a Database



To write to a database, use the structure in Tables. You can either insert new rows into the database or update rows already present. When updating, it will find the row to update by the primary key column(s) specified for the table (which is normally detected in the database import).

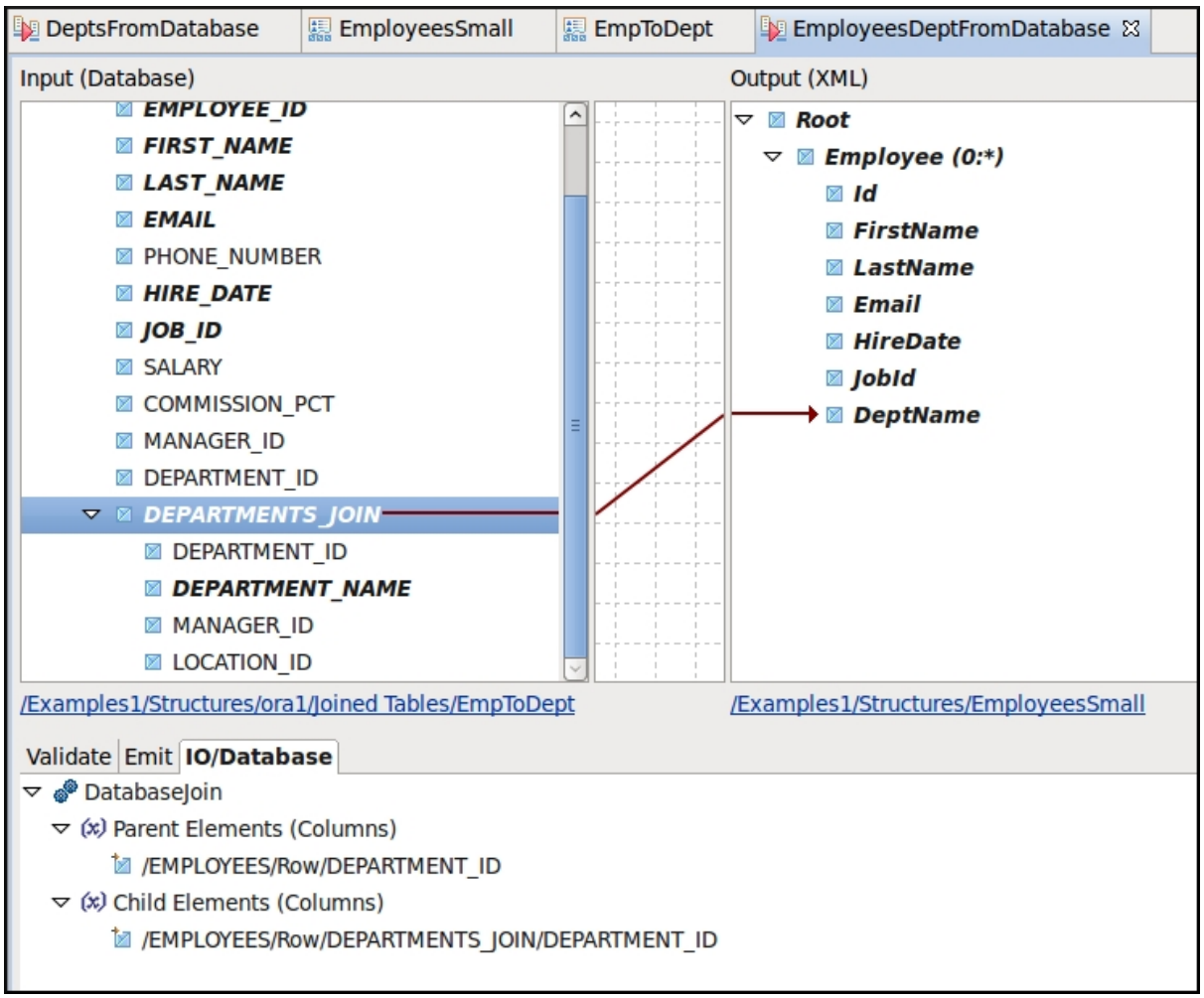
You can insert into or update multiple tables in the output of the map by inheriting from the structure in Tables to insert/update. Then specify the **DatabaseInsert** or **DatabaseUpdate** function as required at the root element of the Tables structure, which is the element that is the parent of the Row element.

If the representation of the output of the map is database and the output structure is a structure in Tables, it will insert into the database by default (no function is needed).

15.8. Join Handling

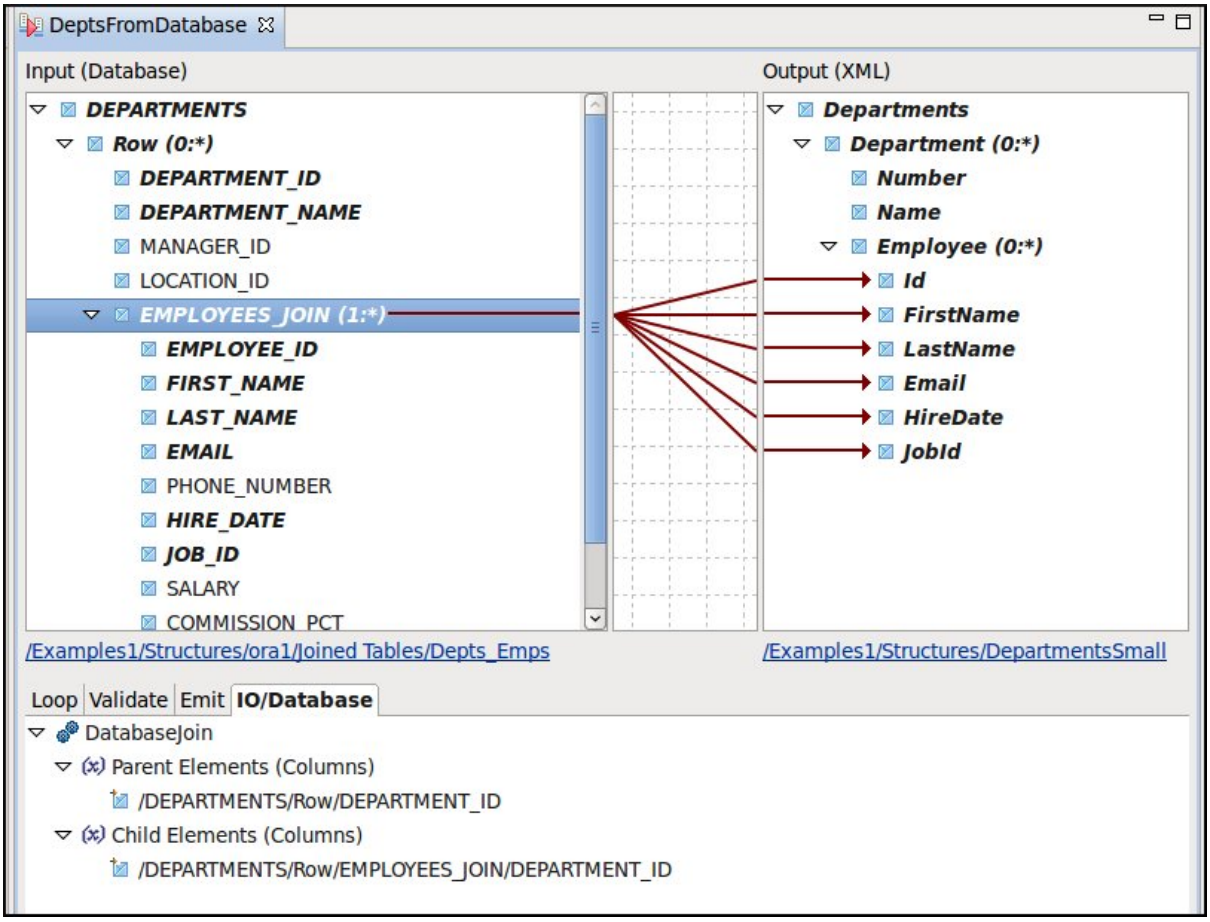
Join handling extends the mechanism of reading from the database to join multiple tables and have them related using the database join mechanism as you would in an SQL query. To do this, you need to create a join structure that contains both the table being selected and the table being joined. As with SQL, you can join as many tables as you like by adding them to the join structure. By convention, the join structures live in the *Join Structures* folder associated with the database.

Creating a join structure is automated using the **[New Structure]** wizard and selecting **Create a database join structure**. It will prompt you for the tables involved and create the structure automatically.



In the above figure we are joining each employee to their corresponding department (a many to one join). This is done by having a join structure that first inherits from the structure to select from in Tables *EMPLOYEES* in this case), and then has a new element (called *DEPARTMENTS_JOIN*) that inherits from the Single Tables Structure to which we are joining (*DEPARTMENTS*). The IO/Database expression for this new element uses a **DatabaseJoin** function which specifies the connection between the columns in the selecting structure (*EMPLOYEES*) and the joined structure (*DEPARTMENTS*).

Additional constraints on the join may be specified in the properties of the **DatabaseJoin** function (double click on it in the expression tab to get the properties) in a similar manner as with the **DatabaseSelect** function.

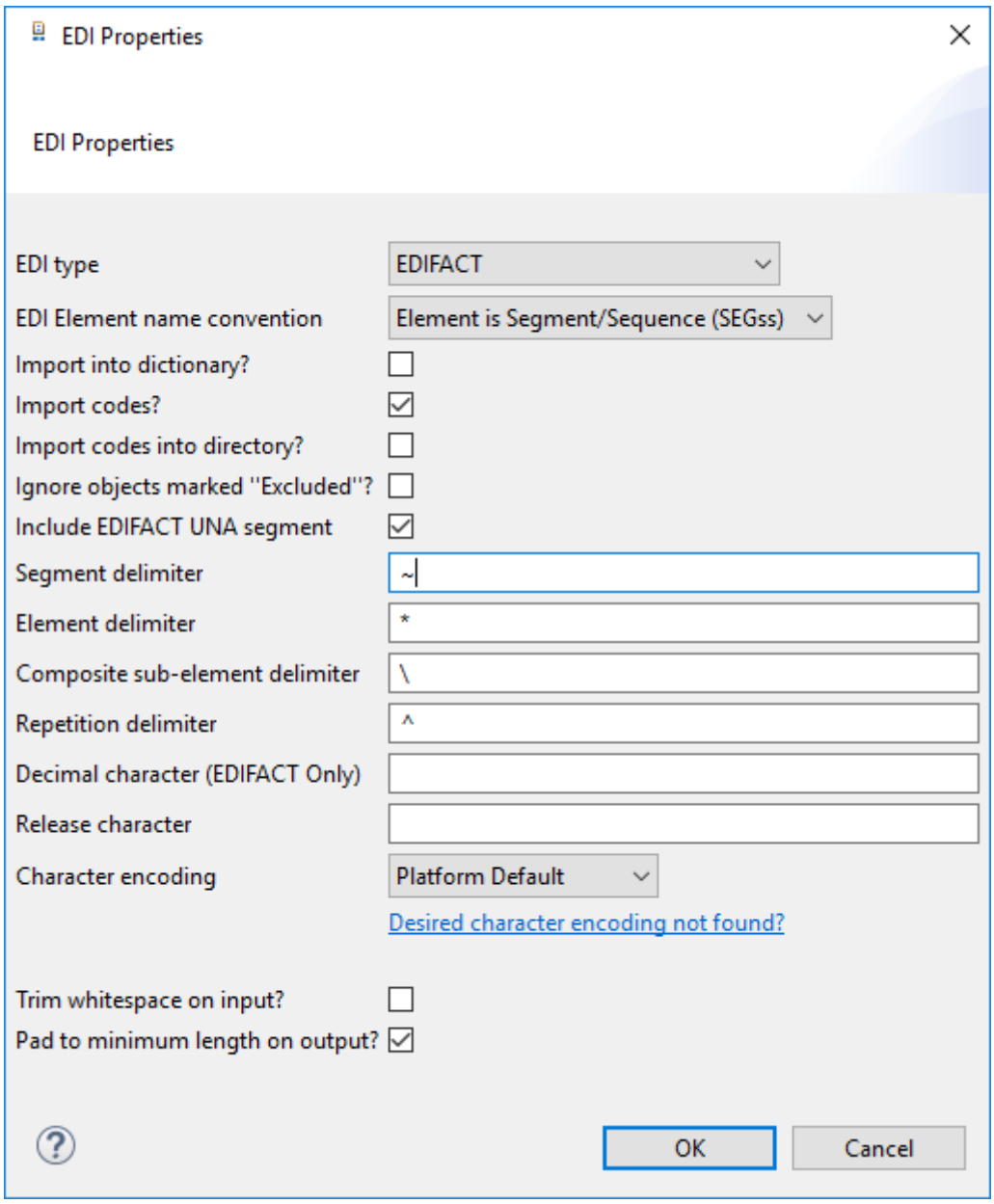


In the above figure we are joining from the departments to find each department's employees (a one to many join). This works exactly the same way, except that the join element (*EMPLOYEES_JOIN*) must be a loop (that is, have a occurs maximum times of *-1*).

Chapter 16. Mapping EDI

16.1. Overview

The EDI representation supports either X12 (including HIPAA) or EDIFACT EDI documents. When you use this representation, the names of the elements must follow the EDI naming convention in order to match with the EDI segments. An EDI representation is created when EDI definitions are imported.



See also the common properties associated with representations.

16.2. EDI Specifications

The *Talend Data Mapper* provides projects that contain the EDI specifications. These definitions allow you to create maps that interact with the EDI transactions. These projects are installed as read-only projects and can be deployed to the *Data Mapper Runtime* using the normal deployment mechanism for projects.

16.2.1. Installation

You can install the EDI specifications either when you launch the Studio for the first time, or later from the **Help** menu. They are installed using the same mechanism as other optional packages, such as certain third-party libraries or language packs. Currently, supported EDI specifications are *X12 HIPAA version 4010* and *X12 HIPAA version 5010,*.

1. When the Studio is launched for the first time, the **[Additional Talend Packages]** wizard opens. Note that you can also access this wizard later by clicking **Install Additional Packages...** from the **Help** menu of your Studio.

2. In the **Available features** area, select the check box that corresponds to the specifications you want to install, and then click **Next**.

3. Select the update site from which the package(s) will be downloaded. The update site can be remote or local.

4. Select the **Do not show this again** check box if you do not want to display the wizard at Studio start-up, and then click **Finish** to complete the installation.

Note that selecting the **Do not show this again** check box at the end of the process means you will not be informed of any new versions of the specifications that may be made available after you perform this initial download. However, you can still check manually if new versions are available by clicking **Install Additional Packages...** from the **Help** menu of your Studio.

16.2.2. Usage

You should use these definitions by inheriting from them (creating a structure that customizes another structure using the new structure wizard in your own project), even if you don't intend to actually customize the transaction set. The main reason for this is to allow flexibility for a migration to a later version. To migrate to the later version you would need to change only one place which is where your custom structure inherits from the standard structures defined in the specification project. Another reason for this is these specification definitions are immutable in a read-only project, so you won't be able to associate your own sample documents directly with the structures in this project.

16.2.3. Contents

This section describes the contents of the EDI specification project.

16.2.3.1. Project Name

The name of the project is the EDI standard name, followed by the version name, like *X12_4010*. If there is any additional qualification to the version it appears after the version name, like *X12_4010_HIPAA*.

16.2.3.2. Transactions

The transactions are contained in a folder called **Transactions**.

16.2.3.3. Envelopes (Interchange, Functional Group)

The **Envelopes** folder contains a structure that defines the interchange. This is used if the interchange segments (*ISA*, *GS*, *GE*, *IEA*) are required. To use this, inherit from it in your own project (create a structure that customizes it). Then, in your custom structure, expand it until you see the *Transaction* element. You can then drag your custom transaction structure (like the *270-A1* for example) to the *Transaction* element which will cause it to inherit at that point. Then the custom interchange structure is ready to use.

16.2.3.4. Control Number Generation and Mapping

For X12 EDI, the control numbers (*ISA13/IEA02*, *GS06/GE02*, *ST02/SE02*) are automatically generated if they are not otherwise mapped. The automatic generation is handled for each element separately. So, for example, you can map the interchange control numbers (*ISA/IEA*) and use the automatic generation for the functional group (*GS/GE*) and transaction set (*ST/SE*) control numbers.

The interchange control number (*ISA13/IEA02*) is generated using the same logic as the **GetSequenceFromLocalFile** function and using the path *${user.home}/odtEdiControl.txt* which is used to store the next sequence number in the local file system. If you wish, you can map the control number to another function, like **DatabaseLookupAndUpdate** to have more control over its generation.

The functional group control number (*GS06/GE02*) is generated by prepending the interchange control number multiplied by 100 and adding the sequence number of the group, starting with zero. So if the interchange control number is *17*, the first group would have a control number of *1700*. This is done because the X12 functional acknowledgment transaction (*997*) requires acknowledgment by uniquely identifying the functional group.

The transaction set control number is generated starting with one for the first set and increasing by one for each set in the group.

16.2.3.5. Automatic Mapping of the Sub Element (Composite) and Repetition Separator

For X12 EDI, if the *ISA16* element is not mapped, the sub element separator will be automatically emitted. Also the repetition separator is also automatically emitted for the *ISA11* element if not otherwise mapped. If you are using a version prior to 4030 where the repetition separator is not used for ISA11, then map it to its proper value.

16.2.3.6. Acknowledgment (997, 999, TA1)

There are three types of acknowledgment that you may need to process when using HIPAA transactions. The Functional Acknowledgment (*997*) transaction is used to report problems at the functional group or transaction set level. This is located in the *Transactions/997*. The Implementation Acknowledgment (*999*) is used to provide additional implementation specification validation reporting. This is located in the *Transactions/999-A1*. Both the *997* and *999* are provided with and without the X12 envelope segments. For problems with the interchange or functional group (*ISA/GS*), the *TA1* segment is used. This is found at *Segments/TA1*.

16.2.3.7. Other

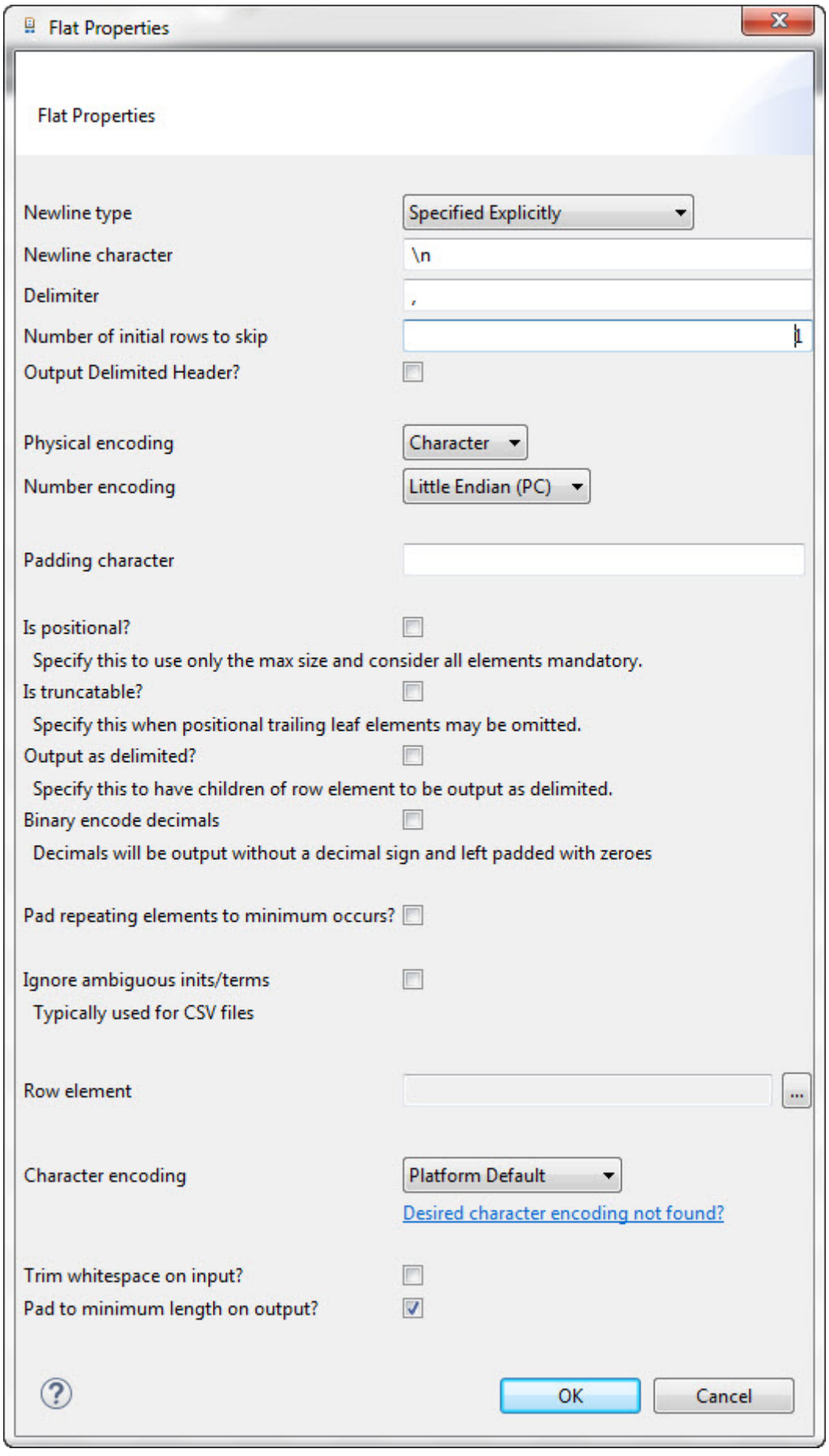
There are other folders (*Segments*, *Composites*, *Elements*) that contain structure definitions for those EDI components. You should never need to use these directly as everything required is present in the transaction

structure. They are provided because the specification metadata is created exactly it is specified by the EDI standard. You may also note that in the transaction structure some elements show the change indication icon (a small gold triangle in the icon). This represents the cases where the definition in the transaction specification overrides or augments that of the underlying segment/composite/element. Again, no action on your part is required, this is just for information.

Chapter 17. Mapping Flat (Delimited or Positional) Objects

17.1. Overview

The flat representation allows delimited or positional flat documents to be processed with the structure. The delimited/positional support handles any kind of character or binary data. For this representation to work correctly, you must set up the flat element properties. The most common delimited representation is CSV (comma-separated values). You can import a CSV sample document, which will automatically set up the elements for CSV support. COBOL records are also supported using the flat representation, as a more flexible alternative to the specific COBOL representation.


Flat Properties
Flat Properties
Newline type
Specified Explicitly
Newline character
\n
Delimiter
,
Number of initial rows to skip
1
Output Delimited Header?
Physical encoding
Character
Number encoding
Little Endian (PC)
Padding character
Is positional?
Specify this to use only the max size and consider all elements mandatory.
Is truncatable?
Specify this when positional trailing leaf elements may be omitted.
Output as delimited?
Specify this to have children of row element to be output as delimited.
Binary encode decimals
Decimals will be output without a decimal sign and left padded with zeroes
Pad repeating elements to minimum occurs?
Ignore ambiguous inits/terms
Typically used for CSV files
Row element
...
Character encoding
Platform Default
Desired character encoding not found?
Trim whitespace on input?
Pad to minimum length on output?
OK
Cancel

The flat representation works by interpreting the data using the following settings from both the element properties and the flat representation properties. Here are the element properties used for flat:

- *Element Initiator/Terminator* - For delimited use, this specifies characters that must appear either before or after an element. Depending on the setting of *Include Element Initiator/Terminator*, these characters may be considered part of the element. These values can be specified for elements at any level.
- *Include Element Initiator/Terminator* - For delimited use, this specifies whether the initiator/terminator characters are considered part of the element. If this is true, the characters will be put in the element's value on input, and the delimited initiator/terminator will not be written on output as they are assumed to be part of the element's value. This option is meaningful only for elements with a group type of none (that is, non container elements).
- *Element Release Character* - Allows a character to be specified that's not considered part of the initiator/terminator characters.
- *Element Occurrence* - Defines the minimum and maximum number of times the element is allowed to occur.
- *Element Size* - The minimum and maximum size of the element. A zero-sized element is sometimes handy to allow you to specify only an initiator or terminator.
- *Element Group Type* - Allows elements to be organized as a sequence or choice of elements. A sequence requires each of the present child elements to occur in order, whereas a choice allows only one of the elements to occur. The initiator and terminator are respected to help determine which elements are present within choices and sequences.
- *Element Column* - Used to indicate that the data starts at a certain column. The column is the number of characters after the last newline character.
- *Element Start Offset* - Used to indicate the gap between the end of the previous element and the beginning of this element. Useful for positional data.
- *Element Quote Handling* - Provides automatic handling of quotes around the element. You can specify that quotes are always required, or that they are optional. This is useful for CSV data.
- *Consume Expression* - Use these to specify the conditions for when the element is to be read. For example, a certain element might not be present depending on the value of a preceding element, so you can specify this with an **IfThen** function in the *Consume* expression associated with the element.

Here are the Flat representation properties.

- **Newline Character** - In the structure editor, the newline character is specified as a *\n*, which is an *abstract* newline. In the Flat Representation Properties, you can specify the actual character sequence that the newline represents, allowing you to use the same structure specification for different actual newline sequences.
- **Physical Encoding** - Binary data is fully supported. When using binary encoding, numeric values are formatted using their binary formats, for example a data type *Integer (32)* is formatted as 4 bytes. The data format allows you to override this for each element. For example, if a file is generally encoded as character data, but there are a few binary elements in it, you can use the data format to specify the elements that are binary.
- **Delimiter** - Enter the character used to specify the boundary between separate, independent regions in the input data. If this is left blank, the delimiter is assumed to be a comma (**,**).
- **Number of initial rows to skip** - In the case of CSV files, this property lets you specify the number of rows from the beginning of the file to skip.
- **Output Delimited Header?** - In the case of CSV files, select this checkbox to add a delimited header to the output file generated.
- **Physical encoding** - Choose between *Character* or *Binary*

- **Number encoding** - Either big endian or little endian may be specified for all binary numeric values.
- **Padding character** - Enter the character to be used when padding.
- **Is positional?** - Treats the size of all elements as their maximum size (ignores the minimum size) and treats them all as mandatory (minimum occurrence of 1). This option is used to conveniently specify a positional usage.
- **Is truncatable?** - Select this checkbox if it is acceptable for positional trailing leaf elements to be omitted.
- **Output as delimited?** - Specify this to have children of row element to be output as delimited.
- **Binary encode decimals** - By default, decimals have a decimal sign and are left padded with spaces when the positional option is set. Select this checkbox if you want decimals to be output without a decimal sign and left padded with zeroes.
- **Pad repeating elements to minimum occurs** - Automatically add empty (padded with the appropriate padding character) elements to any elements that occur where the actual number of iterations is less than the minimum required number. This is set true by default for structures created by the COBOL importer.
- **Ignore ambiguous initiator/terminators** - This makes the assumption that optional elements are handled sequentially in the order they appear. That is, if there is ambiguity about the next possible optional element to process, it is assumed to be the next optional element. Other optional elements are not considered. This is useful for CSV data when you want the trailing fields and their delimiters to be optional, and you want to end the record (line) early without specifying all of the delimiters (commas). This is the default when generating a structure from a CSV instance.
- **Row element**: This lets you specify which element Talend Data Mapper should use to identify where a new row begins.
- **Character Encoding** - The overall character encoding for the transformation is specified in the Flat Representation Properties.
- **Trim whitespace on input?** - When you select this checkbox, any leading or trailing whitespace is removed automatically from the input data for all elements. The non-leading or trailing whitespace is not affected.
- **Generate Default Header?** - When you select this checkbox, the option generates default column names (*Col_1*, *Col_2*, and so on) for a CSV file. You must only use this option on a file that does not have a header, as the importer does not check if the file has a header already.

See also the common properties associated with representations.

17.2. Comma Separated Values (CSV)

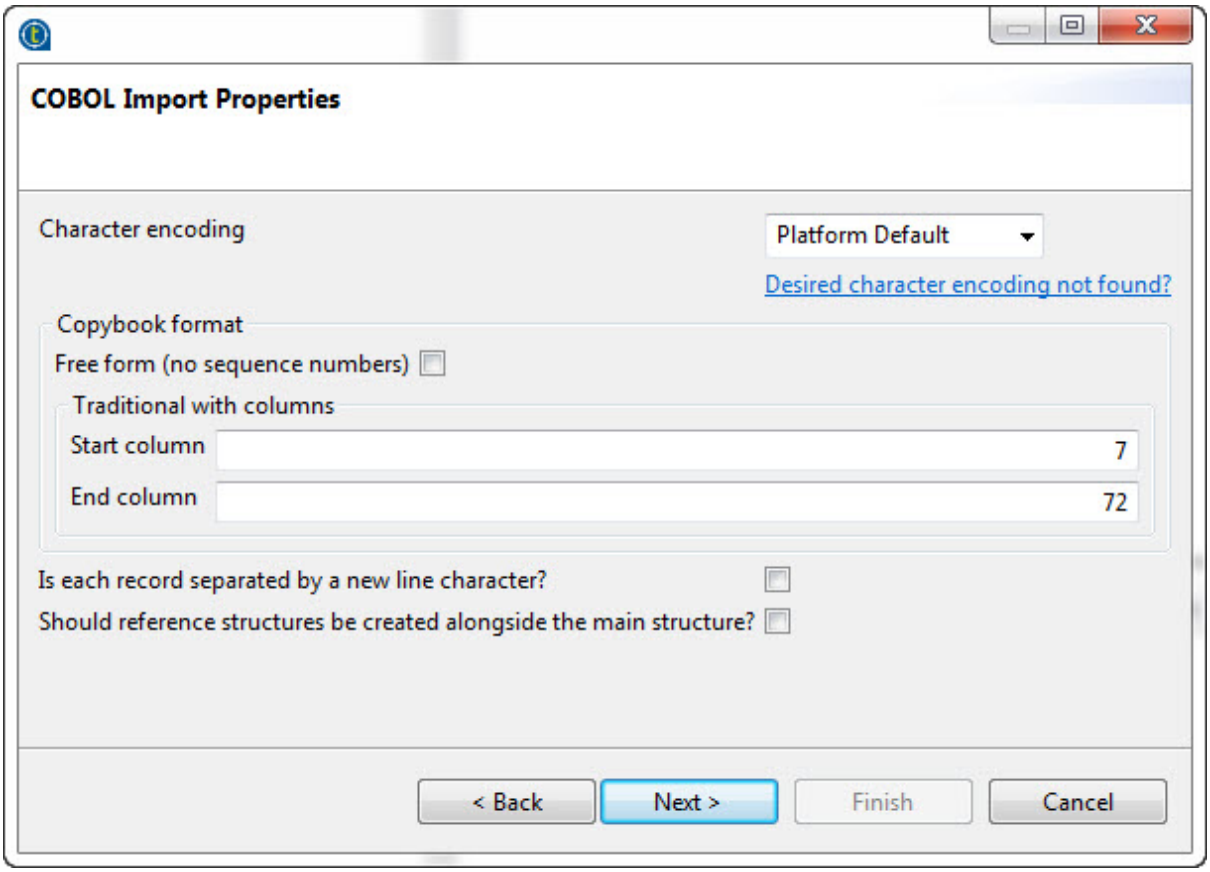
CSV files are widely used because of their simplicity and the fact that spreadsheet programs can export them or directly manipulate them. The *Data Mapper* can import a CSV file and create a structure definition based on the file. This requires that the first row in the file be the names of the columns which is typical.

The imported structure will have a Flat representation and specify the number of rows to skip as *1* (skipping the header information). In addition, it will generate an *__osdtTerminator* element at the end of the row. This is necessary to be flexible in handling a terminating newline. In some files, the last row is terminated by a newline and in others it's not. Having the optional generated *__osdtTerminator* element handles both of these situations. There is nothing special about the *__osdtTerminator* name, it's just made up so as not to conflict with the names of the columns specified in the file.

The imported structure will also have the quote handling set to optional so that any element may be quoted or not.

17.3. COBOL Considerations

While it is possible to work with COBOL files using the specific COBOL representation, they can also be handled through the flat representation. This section describes some factors to take into account when working with COBOL files using the flat representation.



The properties for the COBOL importer are:

- Character Encoding - Specify the character encoding for the data to be processed. For IBM COBOL this will typically be some EBCDIC encoding, such as CP037 or IBM037. For other environments, it's likely to be the more standard ASCII (UTF-8) encoding. See Character Encodings for more information.
- Copybook format - Specifies the format of the copybook source. If there are no sequence numbers, use the Free form option. If there are sequence numbers, use the column specifications to indicate where the actual source starts (past the sequence number and continuation character). The standard column values are the default.
- Is each record separated by a newline character? - Specify this in the (relatively rare case) where each record is separated by a newline that is not specified in the COBOL definition. That is the data is a combination of a bunch of positional records, each separated by a newline character. If you use this, the *Record* element in the generated structure will have a newline as the element terminator. This can be changed to another sequence of characters if desired.
- Should reference structures be created alongside the main structure? - If you select this checkbox, additional Structures are created called reference structures. The main structure inherits from the reference structure. There is one reference structure for the entire copybook and other reference structures for the REDEFINES (choices). In the REDEFINES, each alternative becomes an individual reference structure.

The COBOL importer generates structures for each top-level record definition in the copybook being imported.

The following table shows how elementary COBOL data items are mapped to Data Types based on their *USAGE* and *PICTURE* clauses. For simplicity, this table uses the following abbreviations:

- *BINARY* - An item with *COBOL USAGE BINARY*, *COMP*, *COMP-4* or *COMP-5*.
- *PACKED-DECIMAL* - An item with *COBOL USAGE COMP-3* or *PACKED-DECIMAL*.
- *ZONED-DECIMAL* - An item with *COBOL USAGE DISPLAY*, whose *PICTURE* clause only contains the *9*, *V*, *S* or *P* symbols.

	Data Type	**Data Format**
BINARY, signed, totalDigits < 5	*Short (16)*	
BINARY, unsigned, totalDigits < 5	*Unsigned Short (16)*	
BINARY, signed, totalDigits < 10	*Integer (32)*	
BINARY, unsigned, totalDigits < 10	*Unsigned Integer (32)*	
BINARY, totalDigits < 19	*Long (64)*	
PACKED-DECIMAL, signed	*Decimal*	DF_DEC_PACKED_SIGNED
PACKED-DECIMAL, unsigned	*Decimal*	DF_DEC_PACKED
COMP-1	*Float (32)*	
COMP-2	*Double (64)*	
ZONED-DECIMAL, signed, *SIGN LEADING SEPARATE*	*Decimal*	DF_DEC_ZONED_LEADING_SEP
ZONED-DECIMAL, signed, *SIGN LEADING*	*Decimal*	DF_DEC_ZONED_LEADING
ZONED-DECIMAL, signed, *SIGN TRAILING SEPARATE*	*Decimal*	DF_DEC_ZONED_TRAILING_SEP
ZONED-DECIMAL, signed, *SIGN TRAILING*	*Decimal*	DF_DEC_ZONED
ZONED-DECIMAL, unsigned	*Decimal*	
DISPLAY, BLANK WHEN ZERO	*Decimal*	DF_DEC_BWZ
DISPLAY, other *PICTURE* symbols	*String*	

The COBOL importer supports the following COBOL features:

- *Numeric Scaling* - The implied decimal character *V* in the *PICTURE* clause is implemented using the Decimal Places property of the element.
- *Level 88* - *Level 88* clauses are supported using an Element Type of *Value* for the element (like for code values). The name of the Level 88 clause is included as the description of the value element.
- *OCCURS DEPENDING ON* - This will set the element's occurs minimum and maximum times based on the range of occurrences specified. To implement the *DEPENDING ON* portion, use a **FixedLoop** function taking the value of the element that the occurs depends on.
- *REDEFINES* - The *REDEFINES* clause is implemented using the Group Type of *Choice*. The Group Type of parent element of the element containing the *REDEFINES* clause is set to *Choice*. Each element with a *REDEFINES* clause is a branch of the choice. See below for restrictions on this.

  You can use the IsPresent expression to define the condition used to determine which of *REDEFINES* (member of the *Choice*) is to be available when reading the input. By default, a Constant **IsPresent** expression is generated for each *Choice* member, the first member getting the value *true* and the rest *false*. This makes it easy to change if you want to unconditionally select a different member.
- *REDEFINES Record Types* - When records are redefined using *REDEFINES*, for each member of the *Choice* (see above), if the first field has a single 88 level constant, a delimited initiator will be generated for the field and the length of the field will be reduced by the size of the 88 level constant (typically to zero). This allows the flat reader and writer to automatically consume or generate these providing the correct record.
- *Binary vs. Character* - *Newline* - If all of the encoding for the data is determined to be character, the detection of a newline that separates the records is optionally (and by default) added to the structure during the import. Be

sure that the newline character in the representation properties is correct for the type of data you are importing. By default the newline character will be set to a line feed character.

Here are some issues and limitations on the COBOL copybook import:

- *Alignment or SYNCHRONIZED* - No adjustment is made when *SYNCHRONIZED* values are encountered. Also no attempt is made to align binary data as may be required for certain architectures. These may need to be adjusted to be aligned to the nearest 32-bit boundary in the record. However, this depends on the compiler and platform that was used. If an adjustment is required, you must do it manually by adding the appropriate filler.
- *Level 88 for Non-leaf Elements* - The level 88 values are not supported for non-leaf elements. You will get a warning and they will be ignored.
- *REDEFINES Padding* - No additional padding is added to any of the element subtrees that are involved with a *REDEFINES*. If you are dealing with positional fixed length records with no line delimiters, you will need to make sure the appropriate padding is provided (usually using *FILLER* declarations) to make things line up.

17.4. Character Encodings

There are several places where you can select a character encoding from a list of the available character sets. The list of available character sets is determined by the Java Runtime Environment (JRE). Most of the time the JRE will have the character set you need. However, if you are using EBCDIC, the default character sets that come with the JRE do not include the EBCDIC character sets. There is no single character set for EBCDIC. Rather, there are EBCDIC character sets for different locales. For example, the English EBCDIC encoding is called IBM037 or CP037. When referencing the links below that describe the character sets, the EBCDIC character sets generally included in those are identified as IBM, but there are many IBM character sets on the list that are not actually EBCDIC.

If your character set is not present, it's likely part of the extended characters sets that are not automatically installed into your Java Runtime Environment (JRE). These links list the supported character sets for JRE 5 or JRE 6. To install the extended character set, get the *charsets.jar* file, which is an option in the Java installation, and place it in the lib directory of your JRE. See your system administrator if you need help with this.

If the character set is not present in any of the lists, then it is invalid and needs to changed to a value that is on the list.

17.5. Example of mapping a multiple-record-type flat file

The following example shows how to create a map that writes from a simple multiple-record-type flat file to another multiple-record-type flat file. It shows how to set up the structures manually and uses features like terminator and initiator in the mapping.

Create the input structure

1. In the **Mapping** perspective, in the **Data Mapper** view, expand the **Hierarchical Mapper** node, right-click **Structures**, click **New** and then click **Structure**.
2. In the **[New Structure]** dialog box that opens, select **Create a new structure where you manually enter elements**, and then click **Next**.
3. Name the structure *multiple_record_input*, and then click **Next**.
4. Select the **Flat Files (including csv)** representation, and then click **Finish**.

Create the elements of A_Record

1. Right-click in the **multiple_record_input** structure you just created, and then click **New Element** to create a new element and name it *Root*.
2. Right-click the **Root** element, and then click **New Element**.

   Enter *A_Record* in the **Name** field and *A* in the **Initiator** field. Do not select the **Include Initiator** check box.
3. Right-click the **A_Record** element, and then click **New Element**.

   Name this new element *Text* and set the values of the **Size Min/Max** fields to *1* and *10* respectively.
4. Right-click the **A_Record** element, and then click **New Element**.

   Name this new element *Number*, set the values of the **Size Min/Max** fields to *1* and *5* respectively, and then enter *\n* in the **Terminator** field.

Create the elements of B_Record

1. Right-click the **A_Record** element, and then click **New Element**.

   Enter *B_Record* in the **Name** field and *B* in the **Initiator** field, and then enter *-1* for **Max** in the **Occurs Min/Max** fields to make it loop.
2. Right-click the **B_Record** element, and then click **New Element**.

   Name this new element *Text* and set the values of the **Size Min/Max** fields to *1* and *15* respectively.
3. Right-click the **B_Record** element, and then click **New Element**.

   Name this new element *Code*, set the values of the **Size Min/Max** fields to *1* and *2* respectively, and then enter *\n* in the **Terminator** field.

Create the elements of C_Record

1. Right-click the **B_Record** element, and then click **New Element**.

   Enter *C_Record* in the **Name** field and *C* in the **Initiator** field, and then enter *-1* for **Max** in the **Occurs Min/Max** fields to make it loop.
2. Right-click the **C_Record** element, and then click **New Element**.

   Name this new element *Quantity*, set the values of the **Size Min/Max** fields to *1* and *5* respectively, and then select **Integer (32)** for the **Data Type** field.
3. Right-click the **C_Record** element, and then click **New Element**.

   Name this new element *Filler*, and then enter *\n* in the **Terminator** field.

Create the elements of D_Record

1. Right-click the **D_Record** element, and then click **New Element**.

   Enter *D_Record* in the **Name** field and *D* in the **Initiator** field, and then enter *-1* for **Max** in the **Occurs Min/Max** fields to make it loop.
2. Right-click the **D_Record** element, and then click **New Element**.

   Name this new element *Unit_Price*, set the values of the **Size Min/Max** fields to *1* and *5* respectively, and then select **Decimal** for the **Data Type** field.
3. Right-click the **D_Record** element, and then click **New Element**.

Name this new element *Filler*, and then enter \n in the **Terminator** field.

Create the elements of Z_Record

1. Right-click the **Root** element, and then click **New Element**.

   Enter *Z_Record* in the **Name** field and *Z* in the **Initiator** field.

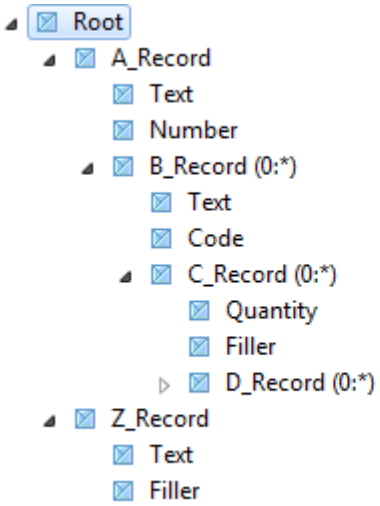
2. Right-click the **Z_Record** element, and then click **New Element**.

   Name this new element *Text*, and then set the values of the **Size Min/Max** fields to *1* and *12* respectively.

3. Right-click **Z_Record**, and then click **New Element**.

   Name this new element *Filler*, and then enter \n in the **Terminator** field.

4. Press **CTRL+S** to save your structure.



Select a sample document

1. Copy the following into a text file and save it under the name *mutiple_record_type.txt*.

```
ATexthdr   00001
BTextdetail1    01
C00010
D00999
BTextdetail2    02
C00001
D00100
ZTextFooter
```

2. Click the arrow next to the **Show Document** button, and then click **Select Sample Document** > **Import Document from File**.

3. In the **[Select Input Document Instance]** dialog box, click **Browse** and then browse to the directory where you saved the file *multiple_record_type.txt*.

4. Select the *multiple_record_type.txt* file in the dialog box, and then click **Finish**.

Create the output structure

1. In the **Mapping** perspective, in the **Data Mapper** view, expand the **Hierarchical Mapper** node, right-click **Structures**, click **New** and then click **Structure**.

2. In the **[New Structure]** dialog box that opens, select **Create a new structure where you manually enter elements**, and then click **Next**.
3. Name the structure *multiple_record_outut*, and then click **Next**.
4. Select the **Flat Files (including csv)** representation, and then click **Finish**.

Create the elements of AB_Record

1. Right-click in the **multiple_record_output** structure you just created, and then click **New Element**.

   Name this new element *Root*.
2. Right-click the **Root** element, and then click **New Element**.

   Enter *AB_Record* in the **Name** field.
3. Right-click the **AB_Record** element, and then click **New Element**.

   Name this new element *Type* and set the values of each of the **Size Min/Max** fields to *2*.
4. Right-click the **AB_Record** element, and then click **New Element**.

   Name this new element *Text*, set the values of the **Size Min/Max** fields to *1* and *10* respectively, and enter *\n* in the **Terminator** field.

Create the elements of AC_Record

1. Right-click the **AB_Record** element, and then click **New Element**.

   Name this new element *AC_Record*, and then enter *-1* for **Max** in the **Occurs Min/Max** fields to make it loop.
2. Right-click the **AC_Record** element, and then click **New Element**.

   Name this new element *Type* and set the values of each of the **Size Min/Max** fields to *2*.
3. Right-click the **AC_Record** element, and then click **New Element**.

   Name this new element *Text*, set the values of the **Size Min/Max** fields to *1* and *15* respectively.
4. Right-click the **AC_Record** element, and then click **New_Element**.

   Name this new element *Quantity*, set the values of each of the **Size Min/Max** fields to *5*, and then select **Integer (32)** for the **Data Type** field.
5. Right-click the **AC_Record** element, and then click **New Element**.

   Name this new element *Total_Price*, set the values of each of the **Size Min/Max** fields to *5*, and then select **Integer (32)** for the **Data Type** field.
6. Right-click the **AC_Record** element, and then click **New Element**.

   Name this new element *Code_Text*, set the values of the **Size Min/Max** fields to *1* and *20* respectively, and enter *\n* in the **Terminator** field.

Create the elements of AZ_Record

1. Right-click the **Root** element, and then click **New Element**.

   Name this new element *AZ_Record*.
2. Right-click the **AZ_Record** element, and then click **New Element**.

Name this new element *Type* and set the values of each of the **Size Min/Max** fields to *2*.

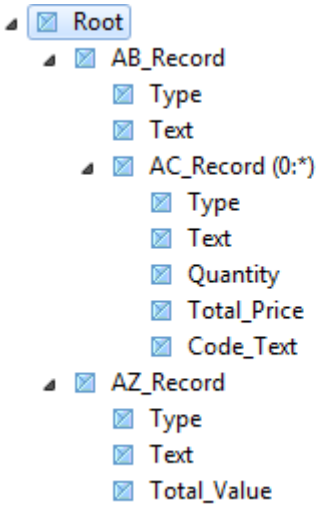
3. Right-click the **AZ_Record** element, and then click **New Element**.

   Name this new element *Text*, and then set the values of each of the **Size Min/Max** fields to *12*.

4. Right-click the **AZ_Record** element, and then click **New Element**.

   Name this new element *Total_Value*, set the values of each of the **Size Min/Max** fields to *7*, select **Decimal** for the **Data Type** field, and enter *\n* in the **Terminator** field.

5. Press **CTRL+S** to save your structure.



Create the map

1. In the **Data Mapper** view, expand the **Hierarchical Mapper** node, right-click **Maps**, click **New** and then click **Map**.

2. Select **Standard Map - Maps instances of an input structure to an output structure**, and then click **Next**.

3. Name the map *multiple_record_type*, and then click **Finish**.

Build the mapping expressions

1. In the map that opens, drag the **multiple_record_input** structure from the **Data Mapper** view and drop it into the **Input** side of the map.

2. Drag the **multiple_record_output** structure and drop it into the **Output** side of the map.

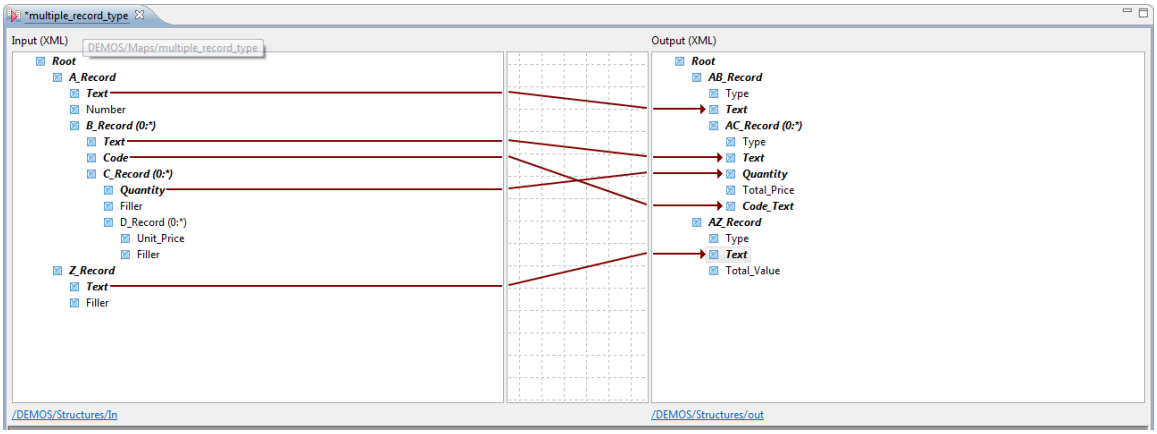
3. Under the **A_Record** element on the **Input** side, drag and drop the input element **Text** to the output element **Text** that is under the **AB_Record** element on the **Output** side.

4. Under the **B_Record** element on the **Input** side, drag and drop the input element **Text** to the output element **Text** that is under the **AC_Record** element on the **Output** side.

5. Under the **B_Record** element on the **Input** side, drag and drop the input element **Code** to the output element **Code_Text** that is under the **AC_Record** element on the **Output** side.

6. Under the **C_Record** element on the **Input** side, drag and drop the input element **Quantity** to the output element **Quantity** that is under the **AC_Record** element on the **Output** side, and then click **Yes** when asked if you want to change the output map element looping.

7. Under the **Z_Record** element on the **Input** side, drag and drop the input element **Text** to the output element **Text** that is under the **AZ_Record** element on the **Output** side.



Add functions

1. Click the **Total_Price** element under the **AC_Record** element on the **Output** side to select it.
2. In the **Functions** view on the left, expand the **Arithmetic** folder, and then drag the **Multiply** function and drop it onto the **Value** tab of the **Total_Price** element.
3. Drag the **Quantity** element from under the **C_Record** element on the **Input** side and drop it into **First Value - Drop/paste** in the **Multiply** function on the **Value** tab.
4. Drag the **Unit_Price** element from under the **D_Record** element on the **Input** side and drop it into **Second Value - Drop/paste** in the **Multiply** function on the **Value** tab, and then click **Yes** when asked if you want to change the output map element looping.
5. Click the **Type** element under the **AB_Record** element on the **Output** side to select it.
6. In the **Functions** view, expand the **General** folder, and then drag the **Constant** function and drop it onto the **Value** tab of the **Type** element.
7. Double-click the **Constant** function in the **Value** tab, enter **AB** in the **Value** field of the **[Expression Constant Properties]** dialog box that opens, and then click **OK**.
8. Click the **Type** element under the **AC_Record** element on the **Output** side to select it.
9. In the **Functions** view, expand the **General** folder, and then drag the **Constant** function and drop it onto the **Value** tab of the **Type** element.
10. Double-click the **Constant** function in the **Value** tab, enter **AC** in the **Value** field of the **[Expression Constant Properties]** dialog box that opens, and then click **OK**.
11. Click the **Type** element under the **AZ_Record** element on the **Output** side to select it.
12. In the **Functions** view, expand the **General** folder, and then drag the **Constant** function and drop it onto the **Value** tab of the **Type** element.
13. Double-click the **Constant** function in the **Value** tab, enter **AZ** in the **Value** field of the **[Expression Constant Properties]** dialog box that opens, and then click **OK**.
14. Click the **Total_Value** element under the **AZ_Record** element on the **Output** side to select it.
15. In the **Functions** view, expand the **Aggregate** folder, and then drag the **AgSum** function and drop it onto the **Value** tab of the **Total_Value** element.
16. Expand the **Arithmetic** folder, and then drag the **Multiply** function and drop it on the **AgSum** function.

17. Click the **AC_Record** element on the **Output** side to select it.

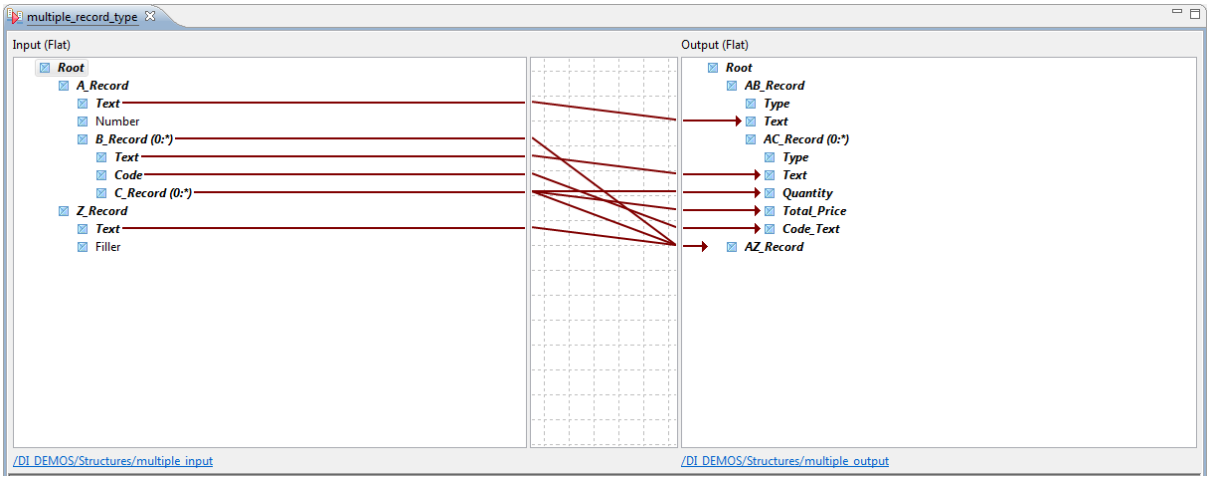
18. On the **Loop** tab, right click **SimpleLoop** and then click **Copy**.

19. Return to the **Total_Value** element under the **AZ_Record** on the **Output** side and, on the **Value** tab, under the **Multiply** function, right-click **Loop Expression - Drop/paste here**, and then click **Paste**.

20. Drag the **Quantity** element under the **C_Record** element on the **Input** side and drop it onto **First Value - Drop/paste here**, under the **Multiply** function on the **Value** tab.

21. Drag the **Unit_Price** element under the **D_Record** element on the **Input** side and drop it onto **Second Value - Drop/paste here**, under the **Multiply** function on the **Value** tab.

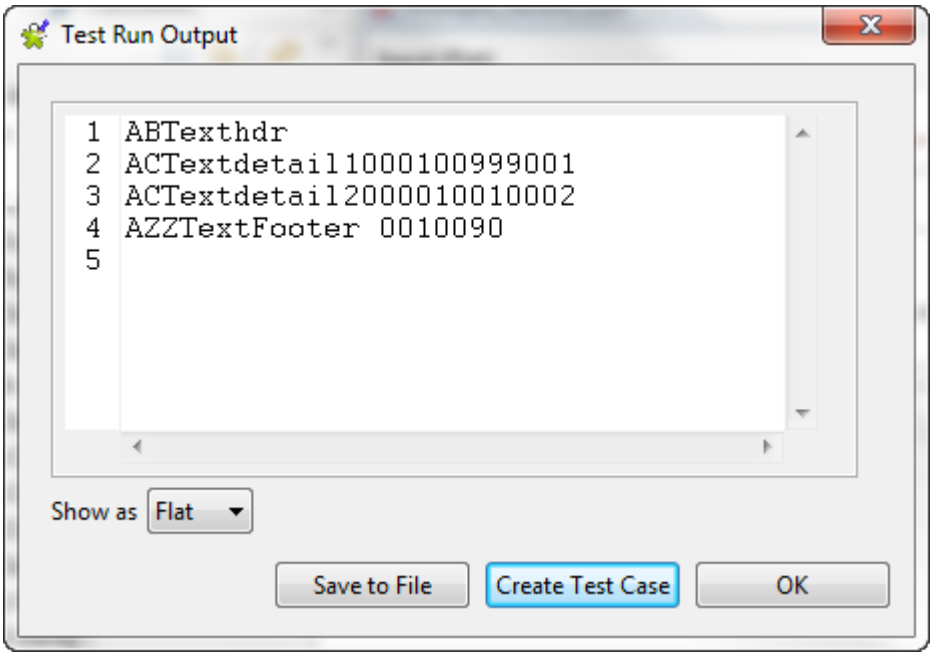
22. Press **CTRL+S** to save your changes.



Test run

1. Right-click the **Root** element on the **Output** side and click **Test Run**. This runs the map using the sample document you created earlier as the input.

2. Check the result compares to the screen shot below.



Chapter 18. Mapping IDocs

18.1. Overview

The IDocs representation allows you to work with SAP IDoc documents as input and output formats.

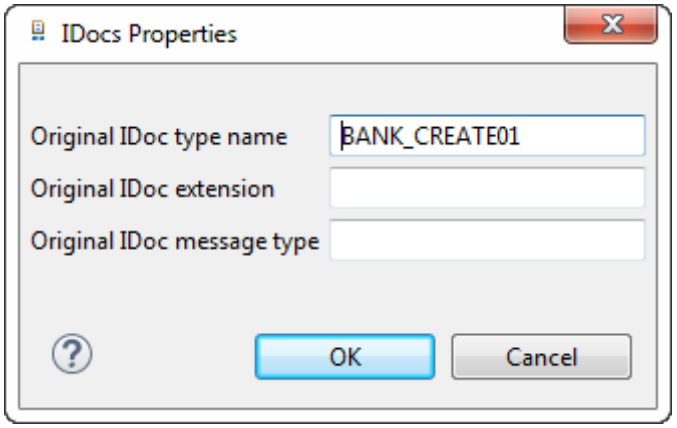
IDoc files imported from an SAP system automatically have the IDocs representation. You can also switch existing IDocs from the Flat representation to the IDocs representation in order to take advantage of the features described below.

18.2. Properties

Properties of the IDocs representation:

- Original IDoc type name: The original IDoc type name is taken by default when the original IDoc file is imported, and it can be different from the Structure name in TDM.
- Original IDoc extension (optional): If the source IDoc has an extension, this extension is appended to the Structure name to form a unique name.
- Original IDoc message type (optional): If you specify the IDoc message type in this field, this type is used as the default type when an IDoc instance is written.

The "Original IDoc type name" and "Original IDoc extension" (if one exists) properties are both taken automatically from the source IDoc.



18.3. Specifying the Segments Release for IDocs

IDoc segments evolve over time, from release to release. For example, the segment layout for ORDERS02 was different when SAP initially released it in version 30D, and it has kept evolving since then. ORDERS02 has a different segment layout in 45B, and this was the last time the segment layout for ORDERS02 changed.

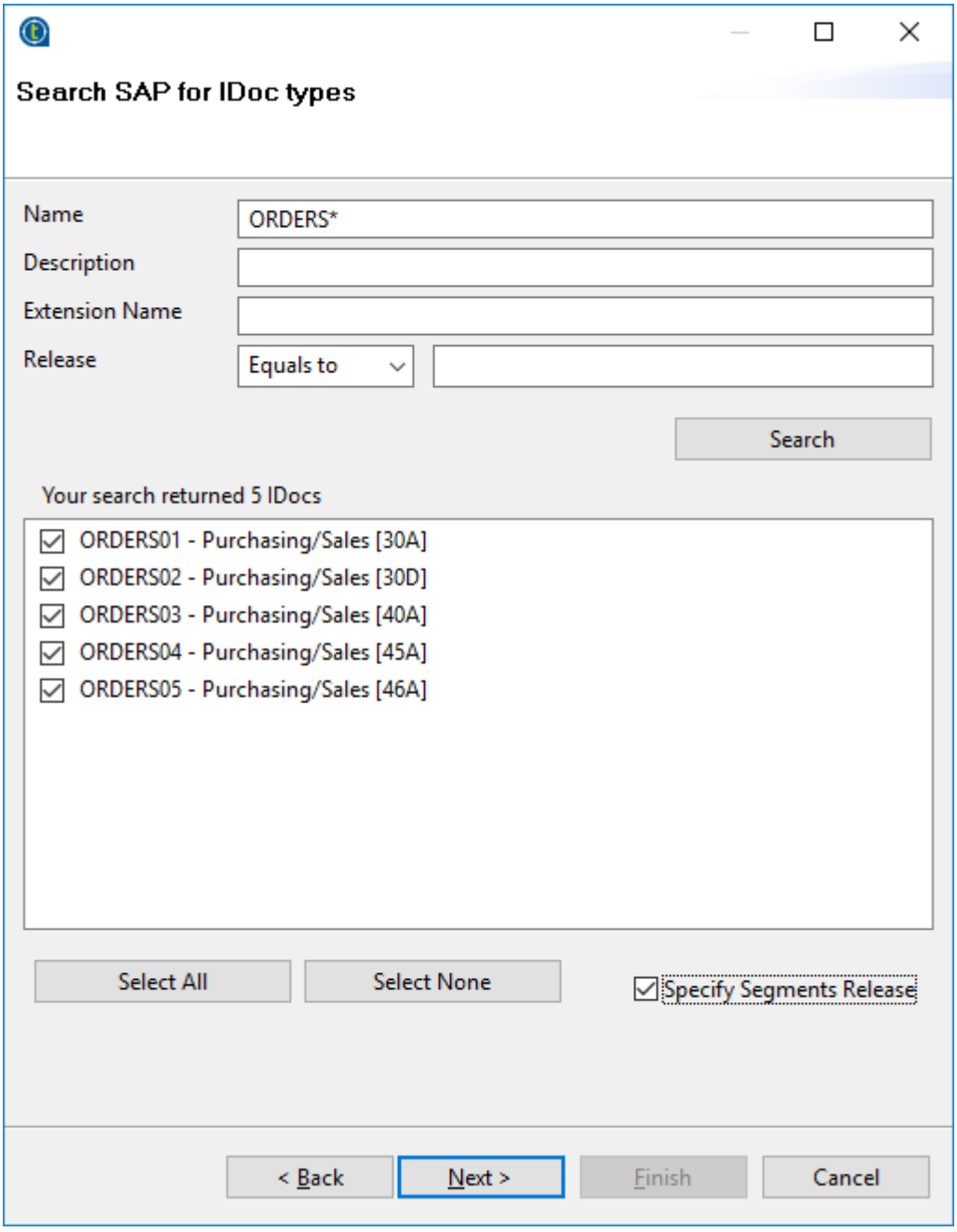
For this reason, if you want to get ORDERS02 with the segment layout that it had in 30D, you need to specify 30D as the Segment Release. If you want to get the segment layout as it was in 45A, you specify 45A and so on. If you specify 700, for example, you will get the layout as it was in 700, but this will be no different from 45B since no changes have occurred since 45B.

If you do not specify a Segements Release, you get the layout as it is in the current release of the target SAP system.

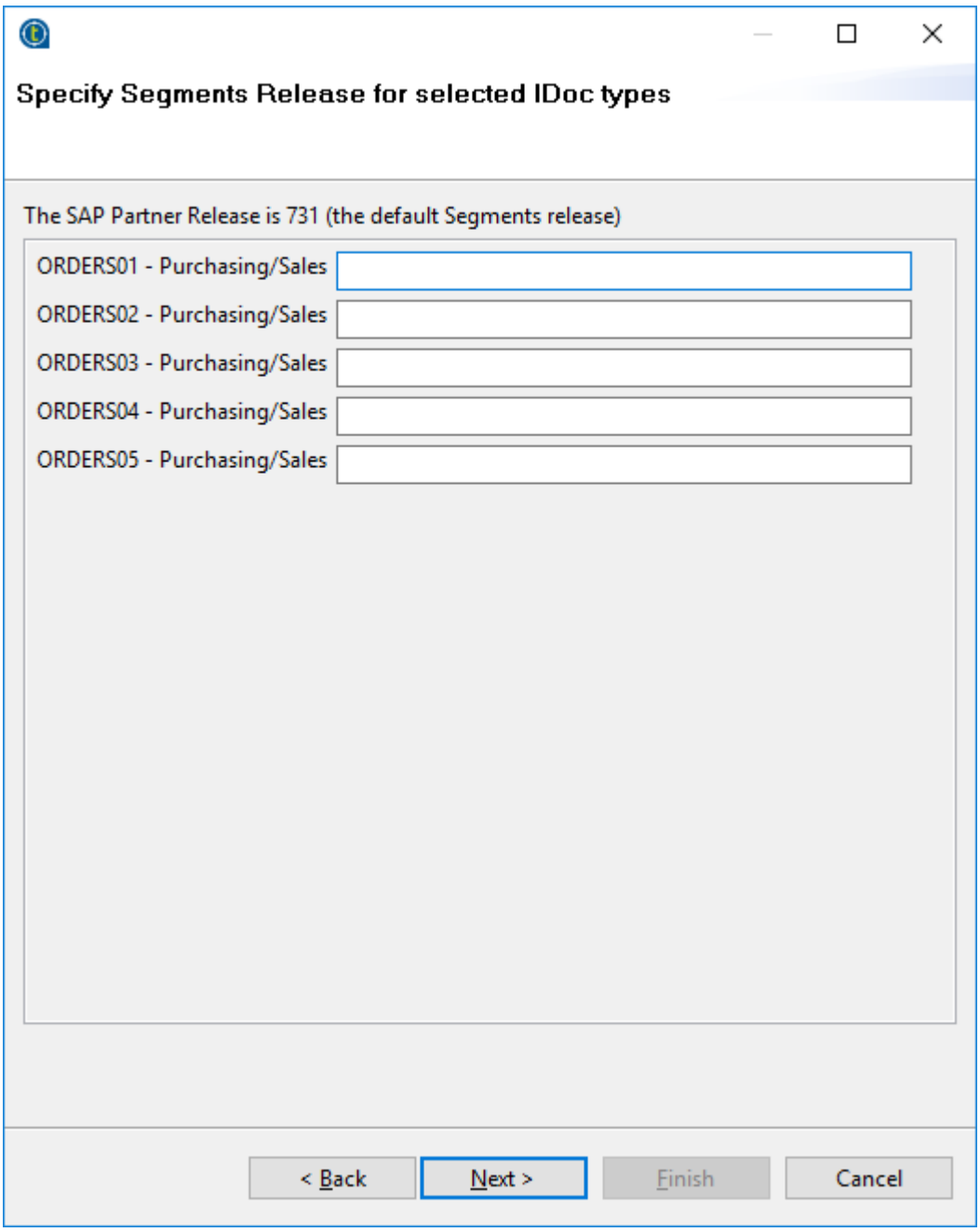
To specify a Segments Release for a new IDoc structure, do the following:

1. In the Structures menu, click **New** > **Structure**.
2. Select **Import Structure Defintion**, and then, on the next screen in the wizard, choose **SAP IDocs**.
3. If you are not already connected to an SAP Server, you must provide information about the SAP Server Connection.

   For details of how to connect to an SAP Server, see the information on setting up an SAP connection in the *Studio User Guide*.
4. In the **[Search SAP for IDoc types]** screen in the wizard, search for the IDocs file you want to use as the basis for your structure.



5. Select the **Specify Segments Release** checkbox to download a specific segment release from the SAP server, and then click **Next**.
6. In the **[Search SAP for IDoc types]** screen in the wizard, click **Next** to continue importing your Structure as usual.



18.4. Mandatory field values for IDoc files

When IDoc files are sent to an SAP system, they must be well-formed and they must include a number of mandatory field values.

You can either provide these values explicitly using mapping *value expressions*, or you can let *Talend Data Mapper* generate the values for you when it outputs an IDoc file.

The following tables show the fields that *Talend Data Mapper* can populate for you if you do not specify a value expression yourself, and provide rules showing how the value is obtained. In most cases the values come from context parameters set in the Studio or from the IDocs representation properties.

For information on how to set context parameters, see *Talend Studio User Guide*.

18.4.1. EDI_DC40 segment

Field	Rule	Example
TABNAM	Constant value	EDI_DC40
MANDT	Context parameter "sap_client"	800
DOCNUM	Sequence number starting at 1. Number is unique for each map execution.	0000000000000001
DOCREL	Context parameter "sap_release"	731
DIRECT	Constant value giving the direction (inbound from an SAP standpoint)	2
IDOCTYP	Stored in the IDocs representation properties (Original IDoc type name)	BANK_CREATE01
CIMTYP	Stored in the IDocs representation properties (Original IDoc extension)	
MESTYP	Stored in the IDocs representation properties (Original IDoc message type)	BANK_CREATE
SNDPOR	Context parameter "sap_sender_port"	SAPCEI
SNDPRT	Context parameter "sap_sender_partner_type"	LS
SNDPRN	Context parameter "sap_sender_partner_number"	CEICLNT800
RCVPOR	Context parameter "sap_receiver_port"	TALENDF
RCVPRT	Context parameter "sap_receiver_partner_type"	LS
RCVPRN	Context parameter "sap_receiver_partner_number"	EEICLNT800
CREDAT	Creation date	
CRETIM	Creation time	

18.4.2. All data segments

Field	Rule	Example
SEGNAM	Constant value - Stored in the Initiator	E2BANK_CREATE001
MANDT	Same as EDI_DC40/MANDT	800
DOCNUM	Same as EDI_DC40/DOCNUM	0000000000814490
SEGNUM	Sequential number starting at 1 for this IDoc instance	
PSGNUM	Parent SEGNUM for nested segments, 0 for top level segments	
HLEVEL	Depth of the segment starting at 2 (1 for non looping segements)	

Chapter 19. Mapping HL7v2

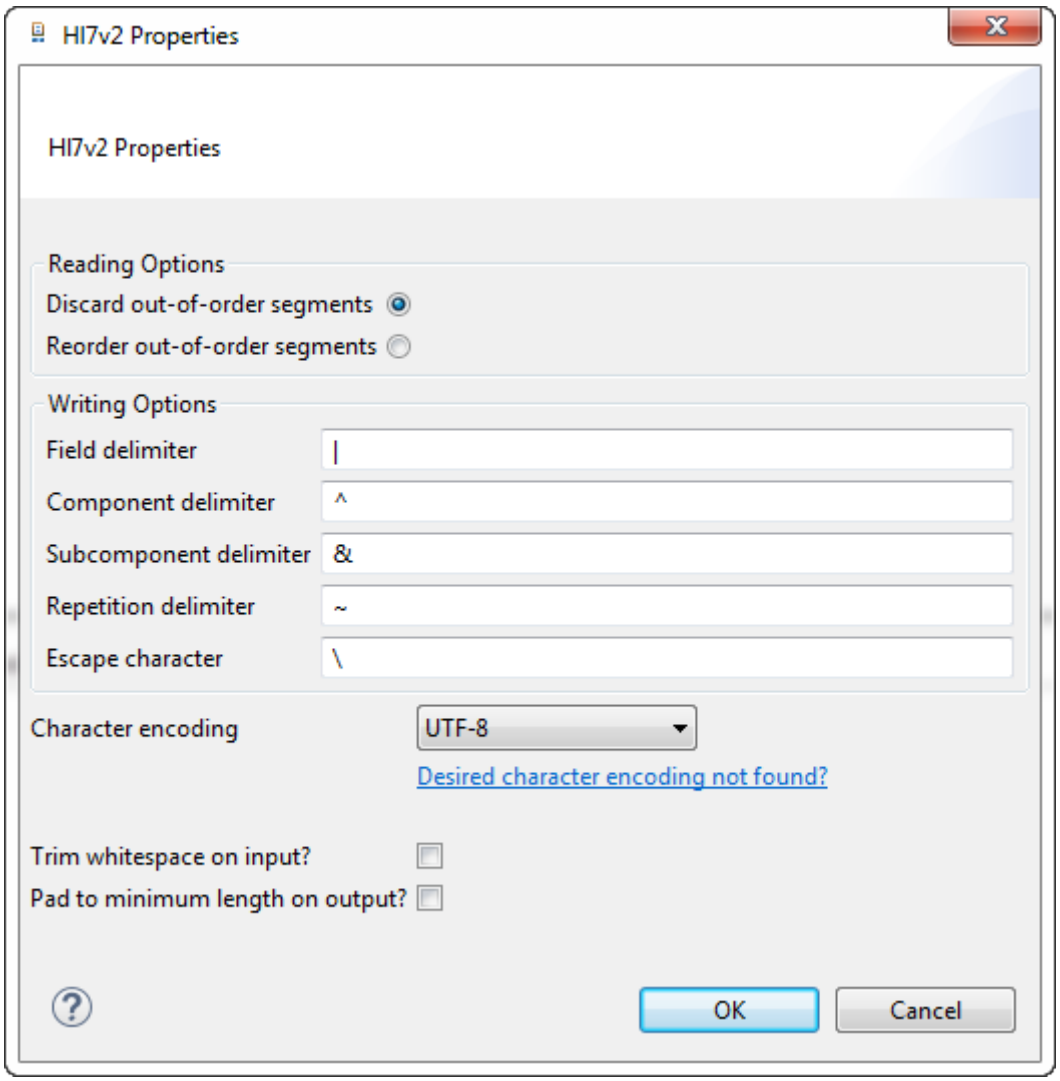
19.1. Overview

The HL7v2 representation allows you to read and write HL7v2 documents.

19.2. Properties

Properties of the HL7v2 representation:

- Discard out-of-order segments: If you select this option, the reader will not include segements that do not follow the order defined by the structure.
- Reorder out-of-order segments: If you select this option, the reader will include segments that do not follow the order defined by the structure, and will reorder these segments to match the structure.
- Field delimiter: Define the character to use to separate fields in a segment.
- Component delimiter: Define the character to use to separate components in a field.
- Subcomponent delimiter: Define the character to use to separate subcomponents in a component.
- Repetition delimiter: Define the character to use to separate repeated fields in a segment.
- Escape character: Define the character used to distinguish special characters.



19.3. HL7v2 Specifications

The *Talend Data Mapper* provides projects that contain the HL7v2 specifications. These definitions allow you to create maps that interact with the HL7 transactions. These projects are installed as read-only projects and can be deployed to the *Data Mapper Runtime* using the normal deployment mechanism for projects.

19.3.1. Installation

You can install the specifications either when you launch the Studio for the first time, or later from the **Help** menu. They are installed using the same mechanism as other optional packages, such as certain third-party libraries or language packs. Currently, supported HL7 specifications are *HL7v2.1*, *HL7v2.2*, *HL7v2.3*, *HL7v2.3.1*, *HL7v2.4*, *HL7v2.5*, *HL7v2.5.1*, *HL7v2.6*, *HL7v2.7*, *HL7v2.7.1*, *HL7v2.8* and *HL7v2.8.1*.

1. When the Studio is launched for the first time, the **[Additional Talend Packages]** wizard opens. Note that you can also access this wizard later by clicking **Install Additional Packages...** from the **Help** menu of your Studio.

2. In the **Available features** area, select the check box that corresponds to the specifications you want to install, and then click **Next**.

3. Select the update site from which the package(s) will be downloaded. The update site can be remote or local.

4. Select the **Do not show this again** check box if you do not want to display the wizard at Studio start-up, and then click **Finish** to complete the installation.

Note that selecting the **Do not show this again** check box at the end of the process means you will not be informed of any new versions of the specifications that may be made available after you perform this initial download. However, you can still check manually if new versions are available by clicking **Install Additional Packages...** from the **Help** menu of your Studio.

19.3.2. Usage

You should use these definitions by inheriting from them (creating a structure that customizes another structure using the new structure wizard in your own project), even if you don't intend to actually customize the transaction set. The main reason for this is to allow flexibility for a migration to a later version. To migrate to the later version you would need to change only one place which is where your custom structure inherits from the standard structures defined in the specification project. Another reason for this is these specification definitions are immutable in a read-only project, so you won't be able to associate your own sample documents directly with the structures in this project.

19.3.3. Extending HL7 messages with new segments

If you want to extend the HL7 messages contained in the Specifications provided out-of-the-box, for instance to add new *Zxx* segments, you can do so in the Structure editor.

For example, to add the *ZPI* element, do the following.

1. In the Structure editor, add a new Element to the message you want to edit.

2. Name this new element *ZPI*. It is important that the name matches.

3. Set the **EDI Elem Type** as **Segment**.

4. Add any children elements to ZPI as needed.

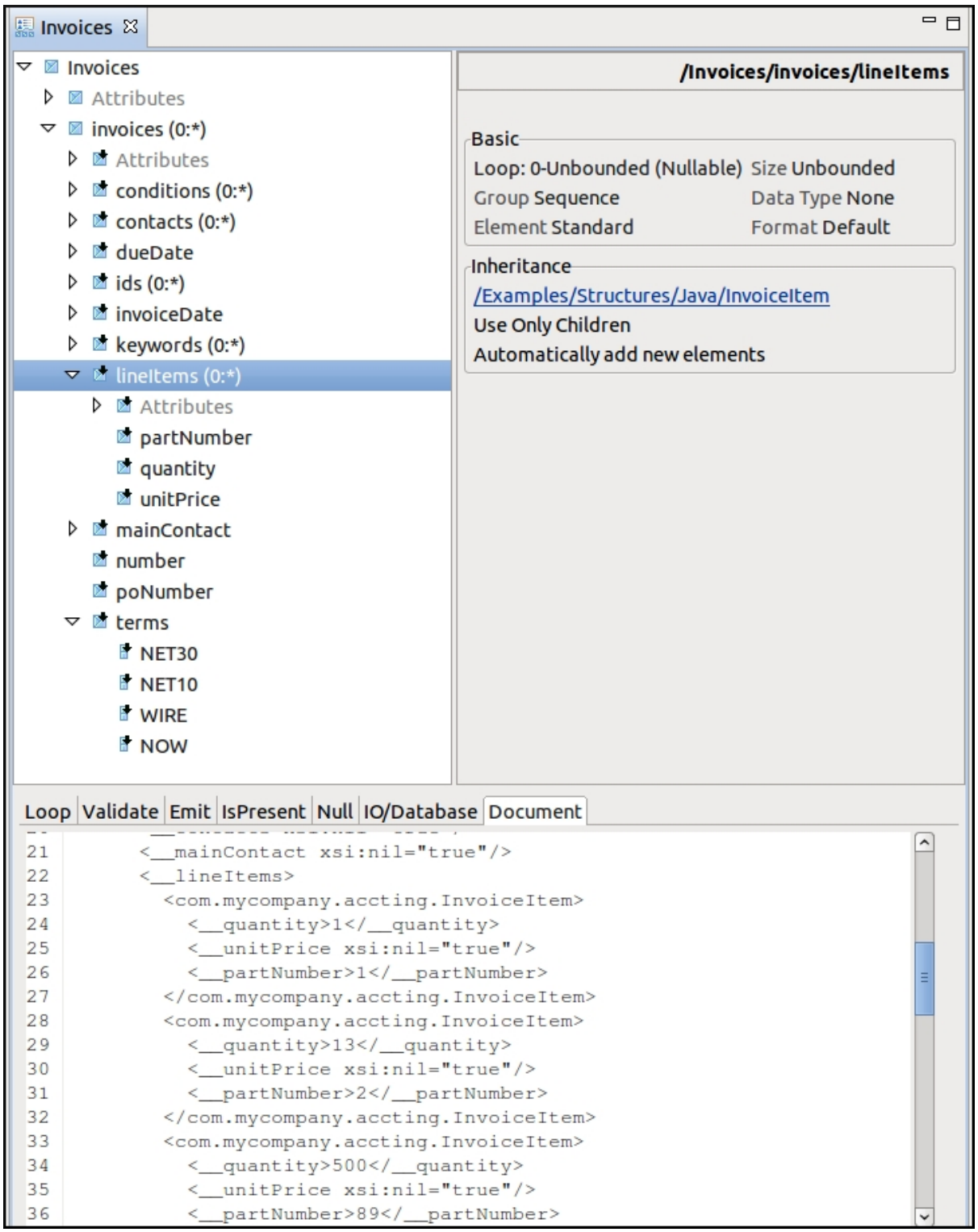
Chapter 20. Mapping Java Objects

20.1. Overview

The Java representation allows you to transform Java object instances. You can import Java classes individually, from a folder, or in a JAR file, and the Java importer will create structure definitions from each class. At runtime you can provide Java object(s) as the source(s) of a transformation or accept them as the result(s).

20.2. Java Class Considerations

The Java importer generates a structure containing all of the fields or beans associated with your Java classes. One structure is created for each class, but they are connected using the inheritance mechanism, so they refer to each other just as they would when using Java pointers. The figure below shows a structure (from the examples) that has an Invoices class as the outer object, which refers to other classes such as *Invoice*, *LineItem*, and *Contact*.



When creating the structure definitions for your Java classes, you can choose to have the structure elements created for the Java fields or for the Java Bean properties. This is specified when you import the Java classes and cannot be changed without reimporting.

- Lists and Maps must include their types - A field or bean property that is just a *java.util.List* or *java.util.Map* will not work. Use *java.util.List<ClassName>* or *java.util.Map<ClassName, ClassName>*.

In the case where a field or bean refers to a class that has subclasses, an element with a group type of choice can be generated which has each known subclass as a member. In this way, it's easy to map the desired concrete subclass. The simple class name (that is, the name without the package) is used for each member.

Note that the Java Bean properties are included even when only the getter or the setter is present, but not both.

20.3. Java Object Mappings

In a Java object graph (a series of objects connected by pointers), you can have multiple pointers to the same object. For example, you can have a *Department* object with a list of *Person* objects and have a separate property for the manager that is also included in the list of *Person* objects. The manager's *Person* object appears in two different places, but it is a single object. When using Java objects as input, all Java objects are fully expanded in the structure definition even if they appear multiple times. This is to allow you to conveniently map from the object wherever it is. You can determine the unique identity of an object by looking at the *id* attribute of the root of the object's fields.

The only time an object is not expanded is if it would create a recursive loop. In the above example, suppose the *Person* object contains a pointer to its *Department* object. In this case, the *Department* object would not be expanded because it is an enclosing object of the *Person* object. It will appear without fields and with its *id* attribute referencing the enclosing *Department* object.

20.4. Handling types of Object

Only primitive values (like numbers, dates and strings) can be directly manipulated and mapped to elements using *Talend Data Mapper*. This is because it needs uniformly support mapping data to or from any source. Because of this there is no notion of mapping the entire contents of a Java object directly. That is to say, if you are doing a Java to Java mapping, and you have a property that's a type *Object* on the input, you cannot map the object that is contained in that property to some property on the output, because objects cannot be directly manipulated.

This is rarely an issue because you would typically define your property as a type more specific than *Object* and then use the subclass handling mechanisms to determine which type is to be emitted in the output, and then populate that type with the desired values.

If a property or field is encountered with a type of *Object*, the generated element in the structure will have a group type of *Sequence* to prevent the element from being mapped.

20.5. Importing Java Classes

The Java representation allows you to transform Java object instances. You can import Java classes individually, from a folder, or in a JAR file, and the Java importer will create structure definitions from each class. At runtime you can provide Java object(s) as the source(s) of a transformation or accept them as the result(s).

- Select - Here you select the classes to import. Each class is read and any other classes that this class depends on is also processed (for example references classes or superclasses). These dependent classes are resolved using the classpath (specified at the bottom of the properties). Initially, the classpath is setup to be identical to the entries selected for import, but there are some cases where you will not want this. For example, if you are selecting folders containing classes to import, you may only want certain folders, but in order to resolve any classes correctly your classpath needs to point to the enclosing top-level folder (like the *bin* folder).

  If you are using the Java Subclass Generation (see below, and selected by default) to generate the choices for each subclass, then you must select all desired subclasses here.

  Use the buttons on the side of the select area to add either resources (folders or files), external Jar or class files, or external folders. When a folder (resource or external) is specified, any JAR or class files contained it

or any descendent folders are also examined (descendent folders are not examined when you turn off **Include Subpackages** below).

Typically you would specify the *bin* folder of the project you are interested in to get all classes in the project. However, you can specify any package level to get only the classes in that package.

Whatever you specify here automatically becomes the Classpath (see below) if you don't specify anything else for the classpath.

By default, all classes in a selected folder are imported. If you wish to import only classes at the level that you select and no subpackages, select a folder or folders that contains classes and then turn off the **Include Subpackages** option.

- Create Elements From - You can specify fields, which creates the structure elements from the fields of your classes, or specify Bean Properties to create the structure elements from the JavaBean properties.
- Include Deprecated Fields/Properties - By default deprecated fields or properties are includes. Turn this off if you don't want them to be.
- Create Structures As - You can control how the structures are named based on the class of the objects. Note that for small numbers of objects, using the simple class names can make things more manageable.
- Handling References to Classes Having Subclasses - See Java Subclass Generation below.
- Classpath - When processing the Java classes for import, dependent classes must be resolved against a classpath. This is where you can add JAR files or folders (which can be either external or resources) to the classpath. If you don't specify anything, the selected file/resource/folders you are importing becomes the classpath.

See also the common properties associated with representations.

20.6. Java Sample Data Objects

See Java Sample Data Objects for instructions on creating Java sample objects to be used for testing.

20.7. Java Subclass Generation

Often in a Java class, a field or property will refer to a (possibly abstract) class that has one or more subclasses. When mapping this field/property, it's important to be able to determine which concrete class was specified in the input, and in the output to specify the desired concrete class. It's important to note that the subclass handling applies to all subclasses known to the import (contained in the Jar files or folders being imported). There may be subclasses that exist but are not known to the import because they have not been specified.

There are two different strategies for handling subclassing:

1. Emitting all Subclasses in a Sequence - This is convenient (and necessary) if you are using map inheritance combined with Java subclasses. Using this mechanism you can map elements in a superclass in a parent map, and map elements in a subclass in a child (inheriting) map and the superclass mappings will be properly inherited. The downside of this is that it's a little less automatic declaring the class you want to emit for the output. The declaring of the output class is done by modifying a **Choice** function in the value of the *class* element associated with the superclass. This is not done automatically. This is the default option.

   When this option is used a structure for the root of the inheritance tree with a name ending in *_Composite* is generated (in addition to all of the structures generated for each Java class). This composite structure contains

all of the inherited structures in the inheritance tree for the field/property and can therefore be used as a structure in an inherited map.

2. Emitting a Choice for Each Subclass - Each possible subclass is emitted as a member of a choice element with all of the properties associated with the subclass (and all superclasses). This means the superclass properties will be duplicated among members of the choice. If you are not using map inheritance, then this is probably fine and easy. It's also a bit easier because when you map something to an output subclass, it should automatically generate the correct Emit expression to indicate which subclass (member) is to be output.

The following options are provided for subclass generation:

- Generate a sequence of elements for all possible subclasses (Default) - Implements the first strategy described above for all possible direct and descendent subclasses.
- Generate only the elements for the specified class - Generates only the class specified by the field/property. Use this option if you don't use the subclasses in your mappings.
- Generate a choice and a child for each direct and descendent subclass - Implements the second strategy above by generating a choice among the class of the field/property and all descendant subclasses. Use this option if the mappings can include any descendant subclass.
- Generate a choice and a child for each direct subclass - Implements the second strategy above by generating a choice among the class of the field/property and the only direct subclasses of that class. Use this option if you only use the direct subclasses in your mappings.

20.8. Java Known Class Mappings

This section gives the correspondence between known Java classes (such as *String*, *Date*) and the element data type.

The *String* and primitive numeric types correspond exactly to the element data types with the same name.

The *java.util.Calendar* class is a *Date/Time* and carries the timezone information. The only implementation class supported for this is *java.util.GregorianCalendar*. The *java.util.Date* class also is a *Date/Time* (but does not have the timezone information).

The *java.sql.Date*, *java.sql.Time*, and *java.sql.Timestamp* classes are manifested as a *Date*, *Time* and *Date/Time* respectively.

The *java.util.List* interface is generated as a loop of whatever type is specified for the list. The interface is implemented by *java.util.ArrayList* when emitting a Java object.

The *java.util.Map* interface is generated as a loop which contains two elements: a *key* element which is the key of the map entry, and a *value* element which supplies the value. These elements can be any support subclass of Java object. The *java.util.Map* interface is implemented by *java.util.HashMap* when emitting a Java object.

20.9. Limitations

The following Java constructs are currently not supported:

- Multi-Dimensional Arrays (*MyType[][] myField*)
- Collections other than *java.util.List*, *java.util.Map* and *java.util.Set*.

Chapter 21. Mapping JSON

21.1. Overview

The JSON representation is used with the JavaScript Object Notation (JSON) documents. You can import the JSON definitions based on a JSON sample document, and you can map JSON documents as input or output.

There are no JSON-specific properties associated with the JSON representation. See the common properties associated with representations.

21.2. Importing/Creating JSON Definitions

Since JSON does not have a standard means of specifying a schema (like XML Schema), the usual way of automatically creating the JSON structure is to import a sample JSON document. It is important that this sample document have all of the possible objects and non-object values that can occur in any JSON instance documents. If this is not the case, then it's possible to get runtime errors when processing the JSON instance documents since required elements will not be present in the structure.

It is possible to manually create a JSON structure or manually modify the imported JSON structure to add the missing elements, so long as you follow the conventions in this section.

21.2.1. Numeric Types

JSON does not specify a mechanism to determine the scope of numbers, so these have to be derived by looking at the actual values when importing the JSON. The importer will use either the *Decimal* or *Double* data type when encountering a number, depending on the presence or absence of the *e* (exponent) value. Decimal is assumed rather than integer because the size of the number could be larger than an *Integer* (or even *64-bit Long*). In addition, the number could actually have a fractional component in the actual data. *Decimal* will thus always work in these situations. *Double* is assumed for floating point numbers (having the exponent value).

21.2.2. Arrays

An array in JSON appears as a loop in the structure. All of the other elements are defined as you would expect, in a hierarchy based on the JSON objects. Arrays in JSON have the following situations, depending on the possible members:

- All same typed non-objects - Every member of the array is a non-object (*decimal*, *double*, *string* or *boolean*) and they are all the same type (not *decimals* mixed with *doubles* for example). In this case, the element of the array will loop, have the data type corresponding to the detected non-object type, and will not be a container.
- All objects - Every member of the array is an object. This is handled like any other object nesting.
- Mixture of objects and non-objects - The "mixed" case where some members of the array are objects and others are non-objects. Alternatively all members could be non-objects, but their types are not the same (a mix of decimal numbers and strings for example).

  This is not treated as a loop, but rather a specific collection of the objects and non-objects as specified in the array and a separate element is created for each. The usual member element is created for each object. For the non-objects, an anonymous member with the name *_anonDecimal* (replacing *Decimal* with *String*, *Double*, or *Boolean*) is created for each type of non-object that is present. Each element name in this "mixed" array will be suffixed with the position of the element in the array.

21.2.3. Dates

JSON does not provide a means to specify that a non-object value is a date type. However you can do this simply by changing the data type to *Date*, *Date/Time*, or *Time* in the structure editor for the element that contains the date. You can also select the applicable data format if it's not the standard ISO 8601 date format. If you do this, then the dates will be properly recognized so they can be automatically converted when being mapped.

21.3. Mapping JSON

There is nothing special about using JSON in maps, as it's little different than XML. One thing to note is there is a *Root* element that's required to be at the top level, because JSON allows the top-level object to contain multiple fields, and the structure definition requires a single root element. This root element does not actually appear in the JSON data.

21.3.1. Dynamic Name support

In certain cases, JSON names are actually a dynamic value corresponding to some data in the transformation. However in the structure definition, a static element name is required. To handle this, we use the *jsonName* attribute of the element in question. If the *jsonName* attribute is specified, on input, it indicates the element in that position may have a different name, and it will correctly read the element's value. On output, it is used to set the name of the element in JSON to whatever value is required.

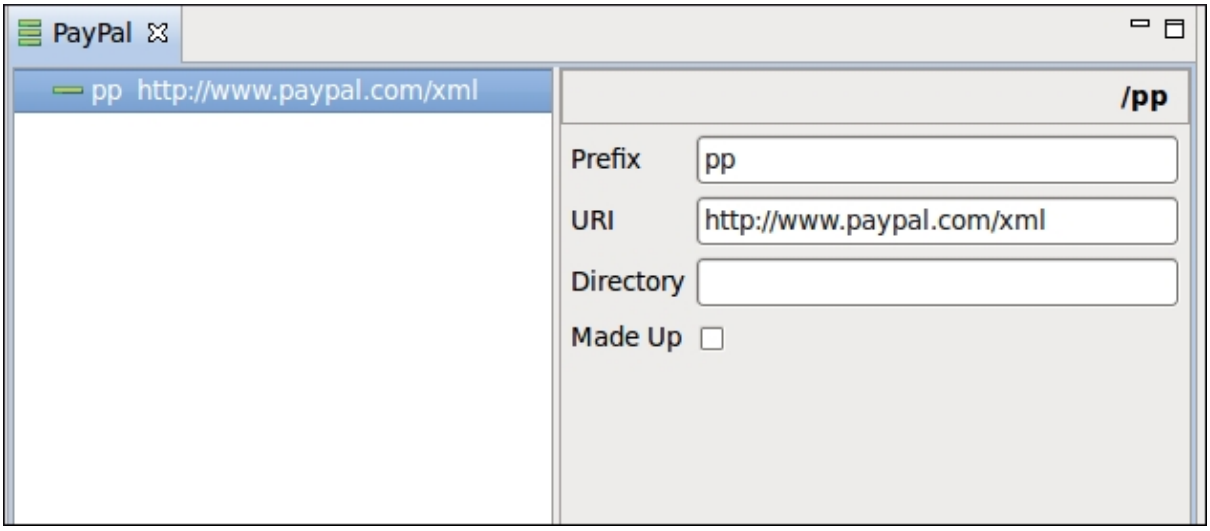
Chapter 22. XML Namespaces and Namespace Containers

Namespace containers are used to define a collection of XML namespaces, along with their preferred prefix values. Any number of structures can be associated with a given namespace container.

An XML namespace is simply a URI used to provide a context for the name of an XML element. This allows large documents to contain sets of elements defined separately without fear of name collision. In the XML document, the namespace is represented in each element using a namespace prefix, which is just a pointer to the full namespace URI. The namespace prefix itself has no significance. When comparing the names of elements, only the full namespace URI is used.

The namespace container simply provides a list of namespace URIs with their prefixes. When viewing a structure, the namespace prefixes are shown as part of the element name. If you prefer a different namespace prefix for a given namespace URI, you can just change the namespace prefix value in the namespace container. The namespace prefixes used in the structure editor are defined by the current values in the associated namespace container. If you change the prefix in the namespace container, this change is automatically shown in the elements shown in the structure editor.

If you are only using the XML Schema Instance namespace *http://www.w3.org/2001/XMLSchema-instance*, then you don't need to create a namespace container, you can just select the property on the XML representation.


PayPal
pp http://www.paypal.com/xml
/pp
Prefix
pp
URI
http://www.paypal.com/xml
Directory
Made Up

22.1. Namespaces and XML Schema Import

The XML Schema import automatically creates a namespace container to hold the namespaces found in the imported XML schemas. The name of the namespace container is the concatenation of the directory names into which the schemas are imported. If a namespace container exists with that name, it is used/updated as required.

The XML schema import process adds an entry (if it does not exist) to the namespace container for each namespace found in the XML schemas. If it finds that a namespace prefix has been defined for the namespace, it uses that. If multiple prefixes are defined, it uses only the first one it finds. If there is no prefix defined, it creates a unique prefix, and indicates this using the "made up" property.

The XML schema import has a number of options to allow the imported structures to be organized by namespace, if desired. One of the options is to allow the directory name corresponding to the namespace to be specified by the user. This is done by specifying the directory property in the namespace entry.

22.2. Default Namespace

A namespace container may be associated with a default namespace. This is specified in the properties of the namespace container. To access these properties, in the **Repository navigator** select the namespace container and then right click and select **Properties**.

The default namespace is only used on the output side of the map and causes the emitted XML to use an *xmlns* attribute with no namespace prefix and a value of the specified default namespace URL for the root element of the output document. This sets the namespace URI for all enclosing elements to be that of the default namespace URI without need for a namespace prefix on each element. In the event that there are multiple namespace containers used in the output structure of the map, only the first specified default namespace is used.

22.3. Troubleshooting Namespace Issues

A frequent problem when using namespaces is the XML element(s) are not read because their namespace does not match the expected namespace. This can result in no output when you are running a map, especially if this is true of the root element of the XML instance document. Generally if you are using structures created by importing XML Schema/DTD/WSDL you should not have troubles if the XML instance documents have the correct corresponding namespaces.

If you add elements manually or create a structure manually it is important to carefully check that your namespace definitions in the structure and namespace container are correct.

Here are some things to check:

- Default Namespace - Often an XML document will use a default namespace, so the elements are not marked with a prefix, but there is an *xmlns* attribute that defines the namespace. If this is the case make sure the elements in the structure definition are using the same namespace. To do this, check the following points.
- XML Representation (Namespace Container) - Does the XML representation of the structure pointed to by the map input contain a reference to a namespace container? This can be determined by finding the XML representation of the structure and viewing its properties (double clicking on it). If not, create a namespace container if necessary and point the representation to it.
- Namespace Container - Is the default (or other) namespace defined in the namespace container associated with the map's input structure? If not, add it with the appropriate prefix. If it's the default namespace then you can just use the prefix *default*. Also, for the default namespace you should change the properties of the namespace

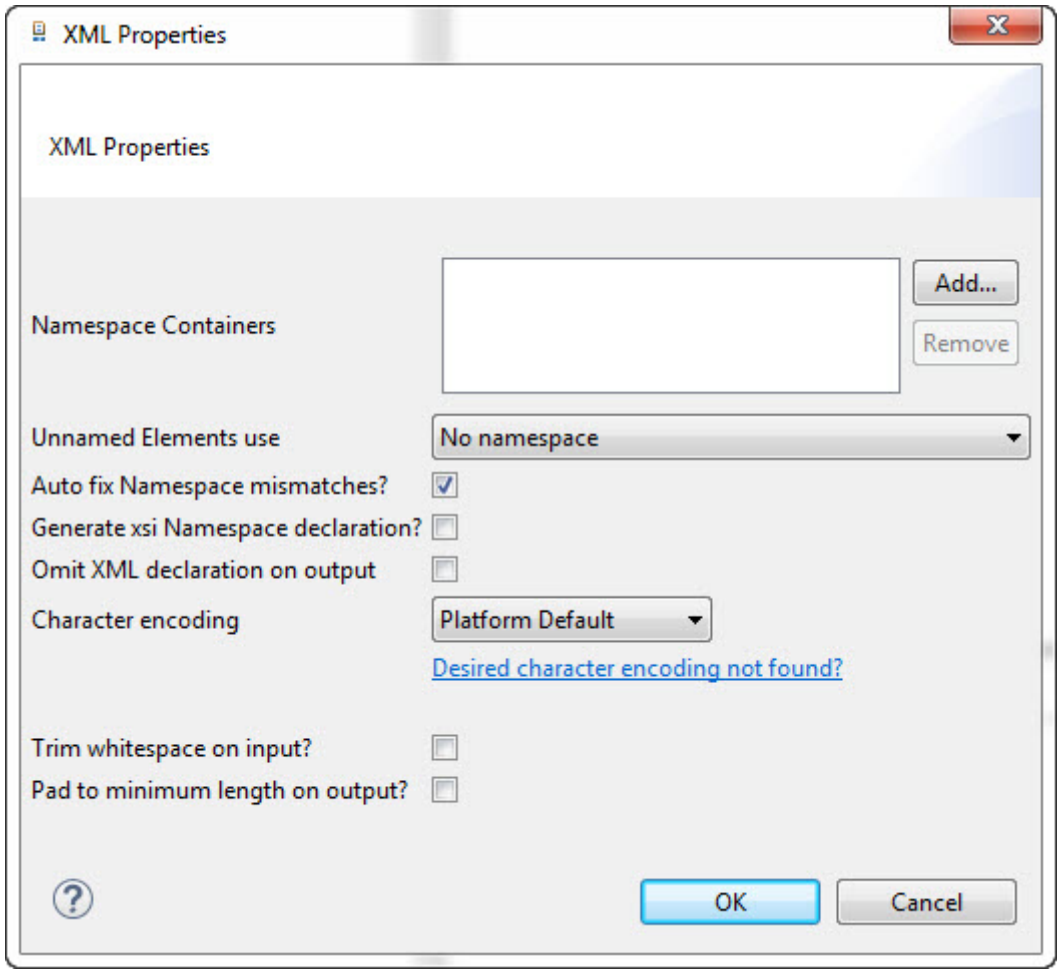
container (right click -> **Properties**) to specify the default namespace URI. When you specify a namespace URI as the default URI, the prefixes associated with that namespace do not appear in the structure or map editors.

- Element's Namespace - Each element can have a namespace associated with it. This is done exactly like an XML document using a namespace prefix. So for example if your default namespace URI is *urn:default* and the prefix is *default* edit the element (called *elementName* for example) in the stucture editor and change it's name to *default:elementName*. You will see in the element's properties (the right side of the structure editor) that the element now has a namespace URI (use the read only view of the element's properties to see this).

Chapter 23. Mapping XML

23.1. Overview

The XML representation allows XML documents to be processed with the structure. All structures will work correctly with an XML representation, and if there are no representations defined for the structure, XML is assumed.



23.2. Properties

Properties of the XML representation:

- Generate *xsi* namespace? - Used for emitting the standard definition of the *xsi* namespace prefix which is associated with the *http://www.w3.org/2001/XMLSchema-instance* namespace. This is useful if you are using the Null Value Support in XML documents and don't wish to have a Namespace Container just to define the *xsi* namespace.

- Omit XML declaration on output - When you select this checkbox, the XML output of a map will not start with the usual XML declaration *<?xml version ="1.0"?>*. This is useful when you want to append multiple XML documents in a single file.

See also the common properties associated with representations.

23.3. XSD Issues

The XML support includes import capabilities for XML Schema (XSD), WSDL, and DTDs. Generally in order to map XML documents you will not need to understand that details of how XSD works as they XSD schemas are translated into a structure which clearly reflects the schema and is consistent with the way other structures are handled.

Here is a discussion of some of the XSD features and how they are translated into structures during the import process.

- Abstract Types - An abstract type is represented by a generated non-visible enclosing choice element. The members of the choice are all of the possible concrete types that are possible under the given abstract type. In addition, an *xsi:type* attribute is generated and an emit expression is generated for each branch of the choice check the value of the *xsi:type* element.

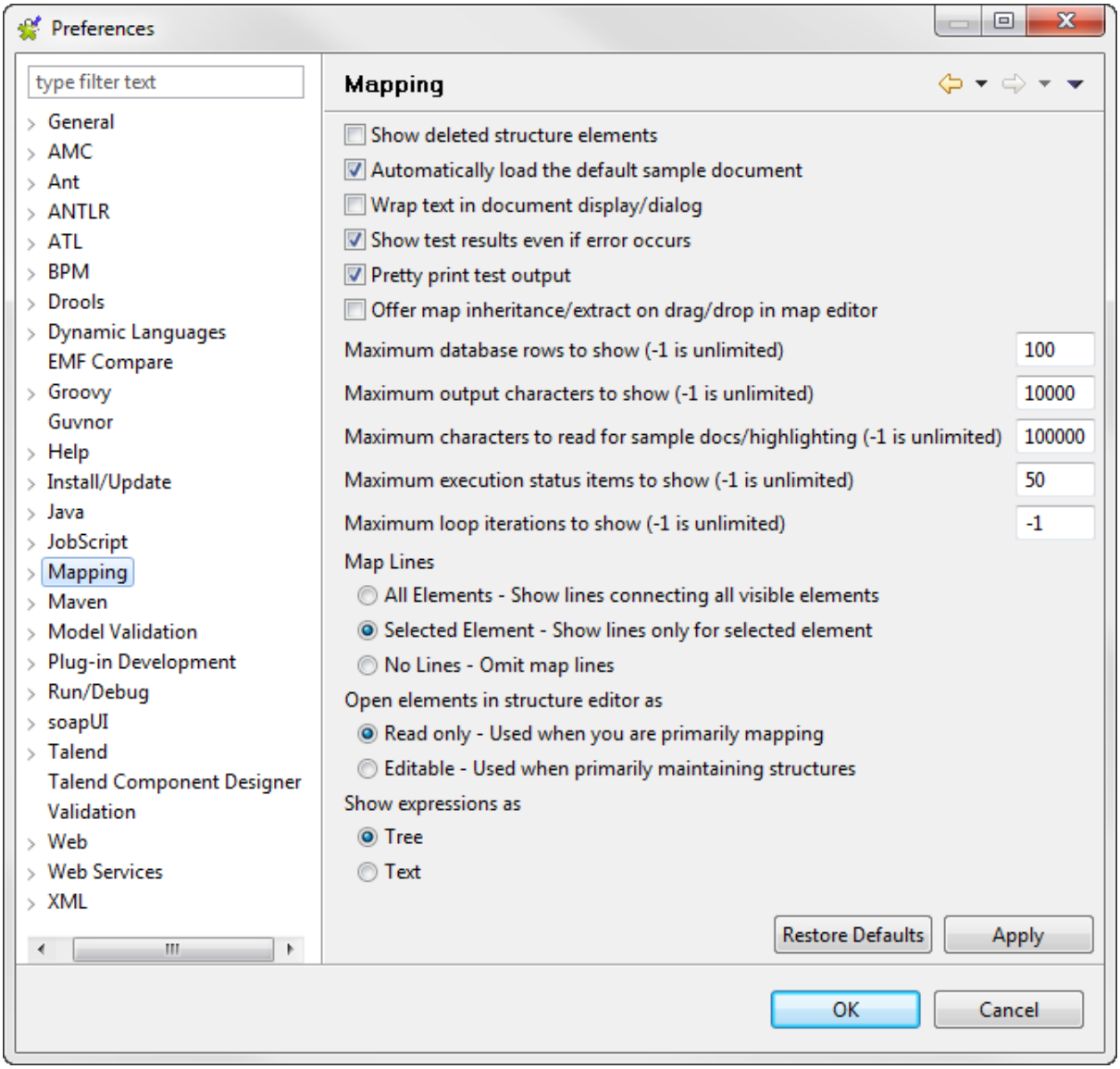
  Generally the members of the choice will be mapped to some members of a choice in the output, so the emit expressions will just be copied from the input and you don't have to do anything special. For more exotic mappings you can simply use the same logic as what is generated in the emit expressions to provide the conditions for selecting members of the choice.

  For the output, a value expression is generated to provide the default value for the *xsi:type* element to make sure it's correctly populated.

Chapter 24. Preferences

The preferences for the studio can be accessed using the through the standard Eclipse preferences.

24.1. Data Mapper Preferences



- **Show deleted structure elements** - This is used only when you are working with elements that inherit from another structure. Normally when you delete a structure element, it is not shown. When you turn this preference on, it will show the element with the indication *(deleted)*. By default, deleted structure elements are not shown.

- **Automatically load the default test document** - If there is one or more sample input documents associated with a structure, and this preference is selected, the first sample input document is loaded into the structure editor when you open the structure. If this preference is turned off, the document is not automatically loaded. By default, the document is automatically loaded.

- **Wrap text in document display/dialog** - If this is specified, the instance document text is wrapped so that horizontal scrolling is not necessary. This can be convenient for instance documents that have long lines so you can see the entire line at once. If this is not selected, the lines are shown at their natural length and scrolling is used. By default, instance documents are not wrapped.

- **Show test results even if an error occurs** - When testing a map, this will show the output of a map even if there were errors in running or validating the map (or validating the instance data). In some cases the error is severe enough to halt the execution of the map (these are indicated as *Fatal*). If this is not specified, when running a map with errors, the output will be suppressed. By default, this preference is on.

- **Pretty print test output** - If this is true, XML documents are formatted for easy viewing by indenting the elements and adding line breaks. Also, the XML namespace declarations are suppressed, because they can cause quite a bit of clutter when viewing the output. By default, this preference is on.
- **Show startup dialog** - The startup dialog contains links to tutorials and actions that help you get started.
- **Maximum number of database rows to show** - This is the limit of database rows per table that are selected when testing a map. This is used so that testing can be done against large databases without unnecessary delay.
- **Maximum number of output characters to show** - This is the maximum size of the test output documents.
- **Maximum characters to read for structure highlighting** - The maximum number of characters to read when automatically showing a sample document in the structure editor. This is used to avoid consuming too much time and memory when calculating the highlighting for very large sample documents.
- **Maximum execution status items to show** - These are the items in the **Problems** view or dialog indicating errors or warnings in the validation or execution of maps or structures. With very large maps, hundreds of errors can occur, and it is sometimes very slow to display all of them. This is why they are limited by default.
- **Maximum loop iterations to show** - Sometimes when testing maps, you want to limit the number of loops processed so you can test with a manageable amount of data. This restricts the maximum number of iterations on all loops for testing purposes.
- **Map Lines** - Lines can be shown that connect the input map elements to output map elements. By default, lines between only the selected element and its related elements on the other side of the map are shown. You can also show map lines between all elements, or turn them off entirely.
- **Open elements in the structure editor as** - There are two different displays for the element properties in the structure editor. The read-only display shows the properties in a very compact format that is suitable for viewing. The editable display requires much more space but allows you to modify the properties. If you are not modifying the properties, select read only (which is the default).
- **Show expressions as** - Expressions can be shown (and created) as a graphical tree (the default) or as text where you type in the expression using the text format. The default is to show the graphical tree.

Chapter 25. Deploying to the Runtime

This chapter shows how to deploy projects to use them at runtime.

25.1. Deploying a Project

The unit of deployment to the Runtime is a project. You must deploy all necessary projects to the runtime. You can do this using either Maven or the standard Eclipse export mechanism.

To deploy a project, export it to a Zip file. We refer to this as a project archive. The location of the project archive file at runtime depends on which type of runtime you are using. All runtime types allow referencing a project archive using a URI that can refer to a file, any schemes in a standard Java URL handler, e.g. HTTP, the classpath, or an OSGi bundle (for the OSGi-based runtime types).

For further details on how to access the project archive, refer to the documentation for the runtime type you are using.

25.1.1. Using Maven

When a *Data Transformation* project is created, a *pom.xml* file is also created. You need to edit the POM to set the group Id and version, and then you can use it to create and install the deployment artifact using the normal Maven targets (package and install).

25.1.2. Using Eclipse Export

You can export a project by right-clicking the project and selecting **Export** (or selecting **Export** from the **File** menu), and then in the wizard dialog under **General**, select **Archive File**.

Chapter 26. Runtime Overview

This lists the methods you can use to execute a map at runtime and outlines the deployment process.

This document covers all of the runtime products. Each runtime product is installed separately so these capabilities depend on which runtime products you have installed.

26.1. Runtime Execution Methods

The *Talend Data Mapper* allows you to execute maps defined in the *Data Mapper*. You can execute maps using any of the following methods:

- **Java API** - You can embed the translation process in your application by calling the Java API.
- **Blueprint/OSGi** - Using the Blueprint Container Specification with OSGi. Special support is also provided for Apache Karaf.
- **Apache Camel** - As a Component or Processor in Apache Camel.

26.2. Installation

To install the a runtime product, follow the specific instructions in the section for the project.

26.2.1. Automatic Installation in Local File System

When the runtime executes it depends on various files to be present in the local file system. To do this, it automatically creates these files, installing the product from the content in the runtime Jar file if they are not already present. Once these files are created they are only read; nothing related to execution of maps is stored in this area and it can be freely shared by anyone on the same machine. This installation directory is expected to be a local directory named by the *ODT_INSTALL_DIR* Java System property. This directory is automatically created if it does not exist on the first execution of the runtime. You can use the same *ODT_INSTALL_DIR* property for all installations, since the installation is installed into a subdirectory that is qualified by product identification (across all Talend transformation product variants and brandings), user name (to prevent permissions problems), and the full version string.

If the *ODT_INSTALL_DIR* Java System property is not set, the installation will take place into a temporary directory.

Whether or not the *ODT_INSTALL_DIR* property is specified, these files may be freely deleted at any time (when the product is not running) and they will be automatically and silently recreated. If you are interested in the details about exactly where these files are created and seeing when the are created, specify the *log="all"* attribute in your transformer or service configuration.

Chapter 27. Java API Runtime

27.1. Overview

The *tdm-api-(version).jar* file implements the runtime Java API, which allows you to execute a map. Like the executable, you can specify an input and/or output document when executing the map. The runtime API is fully multi-threaded.

The runtime Java API is described in the Javadoc, which is located in the *javadocs* directory of the runtime installation directory. The *tdm-api-(version).jar* file contains all of the code on which the *Data Mapper Runtime* is dependent and is all you need for using the API.

If you are using the database support (that is you are accessing any databases in your maps), then you must also include the JDBC driver files in your classpath for the following databases: Oracle, MySQL, and DB2. (Deprecated)

When you execute the runtime in the Java API you will specify the location of the projects to include using API methods.

In addition to executing a map, you can explicitly validate the map using the API. Validation is done automatically when a map is executed, so the use of the explicit validation method is optional.

The map validation and execution results are returned in the form of an *ExecutionStatus* object. This object contains both warnings and errors associated with the map validation as well as any issues encountered in the processing of the input or output documents, depending on the level of validation selected. The *ExecutionStatus* object is designed to be programmatically analyzed.

27.2. Installation

To install, unpack (unzip) the installation archive anywhere you like. You will see directories for examples and the documentation. If you are using Maven, you can then install the Jar files into your local Maven repository by doing.

```
mvn install
```

and you can the API Jar by adding this dependency:

```
<dependency>
 <groupId>org.talend.transform</groupId>
 <artifactId>tdm-api</artifactId>
 <version>(desired TDM version)</version>
</dependency>
```

27.3. Logging

Logging in the *Data Mapper Runtime* is done using the Log4j instance that is found in the classpath of the caller of the API (using the thread context class loader). If Log4j is not present in the classpath, no logging is possible. Configuration of the logging is done using the usual mechanism for Log4j.

There are other methods related to logging in the runtime API, but these are just convenience methods that turn on certain types of logging using Log4j as described above.

27.4. Examples

A simple example is located in the *examples* directory of the runtime installation kit. The instructions are in the *README.txt* file and you can use Maven to build and execute it.

27.5. Runtime API Reference

See *org.talend.transform.runtime.api* in your local Javadoc.

Chapter 28. OSGi/Blueprint Runtime

28.1. Overview

You can access the *Data Mapper Runtime* Runtime API in an OSGi container that supports the Equinox OSGi implementation. You can do this by means of the Blueprint Container Specification. Also if you are using Apache Karaf, including the *Talend ESB Runtime*, a features.xml file is provided that gives a convenient means of loading all of the required OSGi bundles.

The OSGi runtime supports execution with only the Equinox OSGi implementation. The reason for this is that it uses parts of the Eclipse runtime which are hardwired to work with Equinox.

This chapter explains how to install the Blueprint *Data Mapper Runtime*, and how to run the examples. It will use the Apache Karaf OSGi container running the Equinox OSGi implementation.

28.2. Installation

To install, unpack (unzip) the installation archive anywhere you like. You will see directories for examples and the documentation.

Then, do a:

```
mvn install
```

to install all of the OSGi bundles into your local repository. This allows the *features.xml* or *features.talend-esb.xml* file that is provided to load the bundles using the *mvn* URI protocol when working with Apache Karaf.

There are two versions of the *features.xml* file for different purposes

- Talend ESB Runtime - this is called *features.talend-esb.xml* and contains the necessary bundles to be installed in the *Talend ESB Runtime* and a reference to the features required by the *Data Mapper Runtime*.
- Apache Karaf - this is called *features.xml* and contains the necessary bundles to be installed in an unmodified Apache Karaf distribution. This includes bundles that the *Data Mapper Runtime* is dependent on.

28.3. Example

The *Data Mapper Runtime* provides an example in a directory beginning *examples*, which executes a simple transformation using the Examples project. You can familiarize yourself with the Examples project using the *Data Mapper*.

28.3.1. Running the Examples

The example uses Maven to build. To build and run the example, see the *README.txt* file in the *examples* directory for the example you want.

28.4. Using Blueprint/OSGi Support

OSGi Blueprint is a simple and powerful means of finding an executing code that is declared in an OSGi bundle. The *RuntimeEngine* object is registered as a Blueprint service that implements the

org.talend.transform.runtime.api.RuntimeEngine interface. The *RuntimeEngine* allows you to add project archives and find and execute maps. See the javadoc for how to use the *RuntimeEngine* interface.

When working with Blueprint you have your client in some OSGi bundle. We recommend including your project archives as resources in that bundle so that everything is in the same place. You can load the archives into the runtime using *RuntimeEngine.addProjectUri(URI)* and use the *platform* URI scheme to refer to the project archive. For example, considering the bundle *org.talend.transform.examples.blueprint* you would use *platform:/plugin/org.talend.transform.examples.blueprint/Examples.zip* to get the *Examples.zip* project archive.

28.5. Deployment Process

To deploy, export the project(s) that contain the transformer object (maps, structures, etc.) to Zip file(s) using the Export wizard in the *Data Mapper* (you will use **Export** -> **General** and then select **Archive File**). Then either bundle the zip file into your client OSGi bundle or locate it somewhere where you can access it via the file system or a URL.

28.6. Runtime API Reference

See *org.talend.transform.runtime.api* in your local Javadoc.

Chapter 29. Apache Camel Runtime

29.1. Overview

You can execute maps in Apache Camel using the **tdm** Component or you can directly call the *TdmProcessor* class.

In addition, you can generate instances of the sample document that is associated with a map for testing purposes. This is using the **tdmsample** Component which simply emits the requested document. For performance testing, this can be configured to emit the sample document for a specified number of times.

This chapter explains how to install the *Mapper Runtime for Camel*, how to run the examples, and how to configure the Camel component or processor to be used with your maps.

The *Mapper Runtime for Camel* has been certified with Apache Camel version 2.8.2. It is likely to work on most earlier editions as well because it uses only a few well defined interfaces provided. If you wish certification on a prior version, please contact support.

29.2. cMap Mediation Component

The **cMap** mediation component allows you to perform data transformation using a map as part of a mediation route. For details on the **cMap** component, see the *Talend Mediation Components Reference Guide*.

29.3. tdm Component

The **tdm** Component creates an endpoint that transforms a message by running a map.

The map's execution status is a Java object that contains detailed information on any issues with execution of the map. The javadoc found in the runtime kit describes the *ExecutionStatus* class. If the map's execution status has any warnings or more severe items it is associated with the Exchange as a property using the key *com.oaklandsw.transform.executionstatus*.

29.3.1. URI Format

```
tdm://pathToMap?options
```

The *pathToMap* is the fully qualified path to the map to invoke, where the first component is the project name. For example: *//Examples/Maps/Java/AcmoPOToJava*.

Table 29.1. tdm Component Options

Name	Values	Required	Default	Description
exceptionThreshold	*WARNING, ERROR, FATAL*	no	*FATAL*	The value of the map execution status severity that will result in an exception being thrown.
outputType	*Default, String, Byte Array, InputStream*	no	*Default*	The following output types are available: • **Default**: The default output is the same as the input, or Java if the map outputs Java. • **String**: Use this option if the data in

Name	Values	Required	Default	Description
				the output column is to be a `String`. • **Byte Array**: Use this option if the data in the output column is to be a `Byte array`. • **InputStream**: Use this option if you are working with *Talend Data Mapper* metadata and the input is a stream.
projectArchives	List of String separated by commas	no	None	A list of URLs for the project archives that are required to execute the map. This includes the project that contains the map and any other required project (due to map or structure inheritance). The URL can have either a *file* scheme, or *classpath* . For the *classpath* scheme, the path of the URL is the project archive on the classpath.
projects	List of String separated by commas	no	None	A list of the names of the projects that are required to execute the map. This is used when the *Mapper Runtime for Camel* is being executed inside of Eclipse and you want to use the projects that are in the Eclipse workspace. If this is specified, the *projectArchives* option is not used.
log	string	no		Used to define runtime logging. This value is passed to the runtime to define the desired content for logging. Use *infrequent* to log only warnings or higher as well as startup/shutdown events. Use *frequent* to log a message about each transformation executed. Use *all* to get full debug tracing, which can be helpful in diagnosing problems.

29.4. TdmProcessor Processor

The **TdmProcessor** is an implementation of the *Processor* interface that executes a map. This is what is called by the **tdm** Component.

TdmProcessor is a Java Bean that has properties corresponding to the URI options described above. In addition, it has a *mapPath* property that specifies the path to the map to execute as described above.

29.5. tdmsample Component

The **tdmsample** Component creates a consume endpoint that emits instances of the sample document that is currently associated with a given map.

29.5.1. URI Format

```
tdmsample://pathToMap?options
```

The `pathToMap` is the fully qualified path of the map to find the sample document, where the first component is the project name. For example: `//Examples/Maps/Java/AcmoPOToJava.`

Table 29.2. tdmsample Component Options

Name	Values	Required	Default	Description
numberTimes	an integer	no	*1*	The number of times to emit the sample document per thread.
projectArchives	List of String separated by commas	no	None	A list of URLs for the project archives that are required to execute the map. This includes the project that contains the map and any other required project (due to map or structure inheritance). The URL can have either a *file* scheme, or *classpath*. For the *classpath* scheme, the path of the URL is the project archive on the classpath.
projects	List of String separated by commas	no	None	A list of the names of the projects that are required to execute the map. This is used when the *Mapper Runtime for Camel* is being executed inside of Eclipse and you want to use the projects that are in the Eclipse workspace. If this is specified, the *projectArchives* option is not required.
log	string	no		Used to define runtime logging. This value is passed to the runtime to define the desired content for logging. Use *infrequent* to log only

Name	Values	Required	Default	Description
				warnings or higher as well as startup/shutdown events. Use *frequent* to log a message about each transformation executed. Use *all* to get full debug tracing, which can be helpful in diagnosing problems.

29.6. Installation

To install, unpack (unzip) the installation archive anywhere you like. You will see directories for examples and the documentation. Also present is the *tdm-camel-(version).jar* (used with standard Java applications) and (used with OSGi applications).

If you are using the database support (that is you are accessing any databases in your maps), then you must also include the JDBC driver files in your classpath for the following databases: Oracle, MySQL, and DB2. (Deprecated)

You can install the Jar files into your local Maven repository by doing.

```
mvn install
```

and you can reference them by adding this dependency:

```
<dependency>
 <groupId>org.talend.transform</groupId>
 <artifactId>tdm-camel</artifactId>
 <version>(desired TDM version)</version>
</dependency>
```

Note that nothing in the *Mapper Runtime for Camel* depends on the *tdm-camel-(version).jar* file being in a particular location, so long as they are in your classpath.

29.7. Usage with Standard Java

Simply include the *tdm-camel-(version).jar* in your classpath, along with Apache Camel. This will automatically add the necessary TDM components to Camel.

29.8. Usage with OSGi/Blueprint

To use TDM support for Camel in a OSGi/Blueprint environment, you will need to install and start the TDM Blueprint runtime as this includes the Camel OSGi bundle. See the instructions in the TDM runtime kit.

After this point, you can start any bundles you have that require TDM and Camel. Have a look at the examples provided, they will all run in the OSGi/Blueprint environment and the *README* files will tell you how to do it.

29.9. Examples

The *Mapper Runtime for Camel* provides several examples each in a directory beginning *examples*, which execute different transformations based on the maps and sample documents included in the standard Examples project. You can familiarize yourself with the *Examples* project using the *Data Mapper*.

Most of the examples make use of the tdmsample Component which can be used in testing to provide the sample document currently associated with a map.

29.9.1. Running the Examples

The examples use Maven to build and execute. To run the examples, see the *README.txt* file in the *examples* directory in the expanded kit. Also see the *README.txt* file in each example directory for more specific information for each example.

When building the examples, if it complains about not finding the *tdm-camel-(version).jar* be sure you have followed installation instructions to install the Jar into your local Maven repository.

29.10. Deployment Process

To deploy, export the project(s) that contain the transformer object (maps, structures, etc.) to Zip file(s) using the Export wizard in the Data Mapper (you will use **Export** -> **General** and then select **Archive File**). Refer to the location of these Zip files using the *projectArchives* option of the URI when invoking the **tdm** or **tdmsample** Component.

29.11. Runtime API Reference

See Talend Data Mapper API Overview in you local Javadoc.

29.11.1. org.talend.transform.runtime.api

See *org.talend.transform.runtime.api* in your local Javadoc.

29.11.2. org.talend.transform.camel

See *org.talend.transform.camel* in your local Javadoc.

Chapter 30. Function Reference

This chapter describes each of the built-in functions available for mapping.

30.1. Function Description

Each function description contains:

- *Loop Type* - Specified only if the function supports Aggregate looping.
- *Return Type* - The data type that is returned by the function.
- *Argument* - The name and description of a function argument. This argument appears by name in the expression panel when the function is specified.
- *Argument Type* - The data type of the argument.
- *Variable Arguments* - Indicates that this function uses a variable number of arguments. The arguments are specified just below the function expression in the expression panel.
- *Variable Argument Type* - The data type of each of the arguments for functions that support variable arguments.
- *Property* - The name and description of a function property. A function property is specified by opening the properties dialog associated with the function expression.

30.2. Properties and Arguments at the Same Time

Some functions (like **SingleIndex**) have both arguments and properties for the same purpose. This is only for convenience. Sometimes it is more convenient to specify the value of an something as a property (since it is a part of the specification of the map and will never change). Other times, the value is best specified with an argument because it requires a dynamic value. If you specify both, the argument value is used.

30.3. Argument and Return Types

In addition to the Data Types described previously, some other types are used in the descriptions below:

- *Simple* - Any of the simple data types: string, boolean, any number, date, date/time, or time.
- *Generic Number* - Any numeric data type.
- *[Map] Element* - Only a (map) element may be specified as the argument, this is used mainly for the looping functions where you refer to a looping element. It is also used for aggregate functions.

30.4. Functions

This section lists all the available functions in Talend Data Mapper.

30.4.1. Add

Name	**Add**
Purpose	Adds two numbers.

Description	Adds the first value and second value returning the sum.	
Return Type	Generic Number	
Argument	*First Value (Generic Number)*	The first value to be added.
Argument	*Second Value (Generic Number)*	The second value to be added.

30.4.2. AddToDateTime

Name	**AddToDateTime**	
Purpose	Adds a value to a date/time returning a date/time.	
Description	Returns a date/time value which is the input value plus some amount of years, weeks, days, etc. The date/time returned is the actual time given the addition operation. For example, adding *10* months to *1937-08-07* results in *1938-06-07*.	
Return Type	Date/Time	
Property	*What*	What to add. Select years, months, weeks, days, hours, minutes, seconds, or milliseconds.
Argument	*Amount (Integer)*	The amount to add to the date time of the units specified in the properties.
Argument	*DateTime Value (Date/Time)*	The date/time value.

30.4.3. AgAverage

Name	**AgAverage**
Purpose	Compute the average of a set of values.
Description	Returns the average value over all of the specified *[map]* elements. Since this is an aggregate function, each argument is subject to different looping than that of the enclosing expression. See Aggregate Looping for details on how this looping works. If you wish to perform additional computation in conjunction with this aggregate function (a sum for example), then do that in an enclosing expression that uses this aggregate function as an argument.
Loop Type	Aggregate
Return Type	Decimal
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.4. AgConcat

Name	**AgConcat**
Purpose	Concatenate values, with aggregation.
Description	Returns the concatenation of each of the argument values. To separate each element in the concatenation with a delimiter, double-click the **AgConcat** function in the **Value** tab and specify the delimiter to use. Since this is an aggregate function, each argument is subject to different looping than that of the enclosing expression. See Aggregate Looping for details on how this looping works.

	If you wish to perform additional computation in conjunction with this aggregate function (a sum for example), then do that in an enclosing expression that uses this aggregate function as an argument.
Loop Type	Aggregate
Return Type	String
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.5. AgConcatFirstPresentValue

Name	**AgConcatFirstPresentValue**
Purpose	Copies the first argument that has a value (aggregate).
Description	Evaluates each argument in order and returns only the first value that is present and non-blank. If none of the values is present, nothing is returned. If first present value is a loop, only the value of the element in the first iteration of the loop is returned. Since this is an aggregate function, each argument is subject to different looping than that of the enclosing expression. See Aggregate Looping for details on how this looping works. If you wish to perform additional computation in conjunction with this aggregate function (a sum for example), then do that in an enclosing expression that uses this aggregate function as an argument.
Loop Type	Aggregate
Return Type	String
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.6. AgCount

Name	**AgCount**
Purpose	Count a set of iterations.
Description	Returns the number of iterations each of the *[map]* elements in the argument. Since this is an aggregate function, each argument is subject to different looping than that of the enclosing expression. See Aggregate Looping for details on how this looping works. If you wish to perform additional computation in conjunction with this aggregate function (a sum for example), then do that in an enclosing expression that uses this aggregate function as an argument.
Loop Type	Aggregate
Return Type	Integer (32)
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.7. AgMaximum

Name	**AgMaximum**
Purpose	Computes the maximum of a set of values.

Description	Returns the maximum value over all of the specified *[map]* elements. Since this is an aggregate function, each argument is subject to different looping than that of the enclosing expression. See Aggregate Looping for details on how this looping works. If you wish to perform additional computation in conjunction with this aggregate function (a sum for example), then do that in an enclosing expression that uses this aggregate function as an argument.
Loop Type	Aggregate
Return Type	Decimal
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.8. AgMinimum

Name	**AgMinimum**
Purpose	Computes the minimum of a set of values.
Description	Returns the minimum value over all of the specified *[map]* elements. Since this is an aggregate function, each argument is subject to different looping than that of the enclosing expression. See Aggregate Looping for details on how this looping works. If you wish to perform additional computation in conjunction with this aggregate function (a sum for example), then do that in an enclosing expression that uses this aggregate function as an argument.
Return Type	Decimal
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.9. AgSum

Name	**AgSum**
Purpose	Computes the sum of a set of values across looping boundaries.
Description	Returns the sum of all specified *[map]* elements. Since this is an aggregate function, each argument is subject to different looping than that of the enclosing expression. See Aggregate Looping for details on how this looping works. If you wish to perform additional computation in conjunction with this aggregate function (a sum for example), then do that in an enclosing expression that uses this aggregate function as an argument.
Loop Type	Aggregate
Return Type	Decimal
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.10. And

Name	**And**

Purpose	Boolean *and*.
Description	Performs a *Boolean And* operation on all of the arguments. If all of arguments are true, returns true, otherwise returns false.
Loop Type	Aggregate
Return Type	Boolean
Variable Arguments	
Variable Argument Type	Boolean

30.4.11. AnyConcat

Name	**AnyConcat**
Purpose	Concatenate values from XML *Any* type elements.
Description	This function is used only for elements with an element type of *Any* and is used to copy the entire contents of one element to another element which is also an *Any* type. This effectively copies entire contents of the element that's a parent to the *Any* element where the function is used. An *Any* element is a placeholder for a tree of arbitrary XML elements. The root of the tree can be named anything.
Return Type	*Any*
Variable Arguments	
Variable Argument Type	*Any*

30.4.12. AscendingSort

Name	**AscendingSort**
Purpose	Specifies a map element to be sorted ascending in a loop.
Description	This function can be used only in the *Sort Keys* argument of a loop expression and specifies the *[map]* elements that comprise an ascending sort key.
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.13. Choice

Name	**Choice**
Purpose	Chooses a value from a set of values based on conditions.
Description	This function is used to provide a value according to a condition from a list of conditions. It's essentially the same as a collection of nested **IfThenElse** functions but more compact. Any number of **ChoiceValue** functions are provided as arguments. Each **ChoiceValue** function provides the a condition and value pair; the value is emitted when the first condition is satisfied. One of the **ChoiceValue** functions may be marked as a default value and that value is emitted if no other condition is satisfied.
Return Type	Depends on **ChoiceValue** functions
Variable Arguments	
Variable Argument Type	**ChoiceValue** function

30.4.14. ChoiceValue

Name	**ChoiceValue**	
Purpose	Specifies the condition and value for use with the **Choice** function.	
Description	This function is used only as an argument to the **Choice** function to provide the condition and value pair. See the **Choice** function for more details.	
Return Type	Simple	
Argument	*Condition (Boolean)*	The condition that selects this choice value. If the condition is omitted, this value is never selected (unless it is the default and satisfies the default value condition).
Argument Property	*Default Value*	True if this value is returned if no other **ChoiceValue** condition is true.

30.4.15. Compare

Name	**Compare**	
Purpose	Compares two values.	
Description	Used to compare two values. For example if the *First Value* argument evaluates to *23* and the *Second Value* evaluates to *47*, and the *Greater Comparison* property is selected, this expression returns *false*, as *23* is not greater than *47*.	
Return Type	Boolean	
Argument	*First Value (Simple)*	The first value of the comparison.
Argument	*Second Value (Simple)*	The second value of the comparison.
Property	*Comparison*	Specify one of *Equal*, *Not Equal*, *Greater/Equal*, *Lesser/Equal*, *Lesser*, or *Greater*.

30.4.16. CondValidateReport

Name	**CondValidateReport**	
Purpose	Conditionally report on a validation result.	
Description	This function is used to report some kind of validation problem, for example if a value is not within certain limits. It can be called in any type of expression. The report is done to the results of the execution of the map, which is shown through the GUI or can be accessed using the runtime API. This function is typically used as the root of a validation expression to provide richer reporting of a validation problem. A validation expression accepts a Boolean value, if true, the validation succeeds and nothing is reported, if false, the validation fails.	
Return Type	Boolean	
Argument	*Condition (Boolean)*	A condition, if *true* causes the function to do nothing. If *false*, cause the function to report a validation result. The value of this condition is also returned as the value of this function.
Argument	*Data (Simple (variable))*	Any number of expressions that data to be associated with the validation report. These are reported as name/value pairs in the validation report. If an expression specified here is a *[map]* element reference, the name is taken to be the name of the *[map]* element and the value is the element's value. Use the **Property** function with other types of expressions to provide the name label in the validation report.

Property	*Severity*	The severity of the validation issue which is either informational, warning, or error.
Property	*Message*	An optional number that identifies the validation issue.
Property	*Error Number*	An optional number that identifies the validation issue.

30.4.17. Constant

Name	**Constant**	
Purpose	Returns a constant value.	
Description	Returns the specified constant value.	
Return Type	String	
Property	*Value*	The constant value. A newline is specified as *\n*; a tab is *\t*, and a backslash is \\.
Property	*DataType*	An optional data type may be associated with the value. This is used only for special cases. One of these cases is for the *QName* datatype when used to process an *xsi:type* attribute for example. When using the *QName* datatype, if you wish to omit the URI portion of the *QName*, specify the colon as the first character, like this: *:localName*

30.4.18. ConstIfBlank

Name	**ConstIfBlank**	
Purpose	Returns the specified value if it is non-empty, otherwise returns a constant.	
Description	Returns either the specified input value, or the specified constant if the specified input value is blank or not present.	
Return Type	String	
Argument	*Input Value (Simple)*	The value to check. If the value is present and non empty, it is the result of the function. Otherwise, the constant value (specified below) is the result of the function.
Property	*Value*	Constant value to use
Property	DataType	

30.4.19. Concat

Name	**Concat**	
Purpose	Concatenate values.	
Description	The **Concat** function converts each of its arguments to a string and concatenates them together to form the result.	
Return Type	String or type of the first input parameter if only one parameter	
Variable Arguments		
Variable Argument Type	Simple	
Property	*Join String*	Specify the non-breaking space character or characters to be used when concatenating a series of values. For example, ", " would put a comma and a whitespace between each value, or " " would simply use a space.

		If any value is null, the non-breaking space character or characters are omitted for that value.

30.4.20. ConcatFirstPresentValue

Name	**ConcatFirstPresentValue**
Purpose	Copies the first argument that has a value.
Description	Evaluates each argument in order and returns only the first value that is present and non-blank. If none of the values is present, nothing is returned.
Return Type	String
Variable Arguments	
Variable Argument Type	Simple

30.4.21. ConvertFromBinary

Name	**ConvertFromBinary**	
Purpose	Converts the specified bytes from binary into a value with a simple type.	
Description	This is used to convert the binary value into a simple type, like an integer or string according to the specified data format and encoding. It can be used with the **ExtractBytes** function to read arbitrary data from binary data.	
Return Type	As specified by the Data Type property	
Property	*Data Type*	The data type this function is to return.
Property	*Data Format*	The data format to be used in the type conversion.
Property	*Encoding*	The character encoding used to convert the value. Used only for the String data type.
Property	*Input (Binary)*	The binary value to be converted.

30.4.22. DatabaseColumn

Name	**DatabaseColumn**	
Purpose	Specifies a database column value when using another database function.	
Description	The **DatabaseColumn** function is as an argument to other database functions like **DatabaseLookup** to specify the column name as part of the SQL expression. This function may be used only within the argument expression of the database function.	
Return Type	Depends on the database column specified	
Property	*Column*	The fully qualified column in the database of the form: *<path to database>.<table>.<column>* for example: */Default Project/Databases/MyDatabase.MYTABLE.MYCOLUMN*. Note that when using the *studio* a graphical interface specifies this, so you don't need to be concerned with this format.

30.4.23. DatabaseFunction

Name	**DatabaseFunction**
Purpose	Calls a SQL function.

Description	The **DatabaseFunction** function allows you to call any SQL function to be evaluated as part of a condition for other database functions like **DatabaseLookup** . This function may be used only within the argument expression of the database function.	
Return Type	Depends on the database function specified	
Property	*Function*	The name of the SQL function to be called for example: **concat**.

30.4.24. DatabaseInsert

Name	**DatabaseInsert**	
Purpose	Specifies the table to insert rows into a database.	
Description	The **DatabaseInsert** function may be used only as an I/O expression. **DatabaseInsert** specifies the table into which the row elements are inserted. Specify this on the I/O expression of the element that inherits from the database table, which is the parent of the *Row* element. This function is optional. If an output structure has a database representation, inserting into the database is assumed. The name of the table is assumed to be the name of the single row table structure inherited by the *Row* element.	
Return Type	N/A (this function may be used only as an I/O expression)	
Variable Arguments		
Property	*Table*	The table to insert into. If not specified, assumed to be the table specified by the single row table structure inherited by the child *Row* element.

30.4.25. DatabaseJoin

Name	**DatabaseJoin**	
Purpose	Specifies the conditions on a database join between two tables.	
Description	The **DatabaseJoin** function may be used only as an *I/O* expression. Specify it as the I/O expression of the element that contains the data to be written. This function specifies which columns are to be joined between the parent and child tables. It also specifies any selection criteria for the join.	
Return Type	N/A (this function may be used only as an I/O expression)	
Property	*Where String*	Enter a SQL condition as specified in a *Where* clause without the use of the word "Where".
Property	*Join Type*	Specify *Inner* to have the join exclude rows from the parent table with no corresponding rows in the child table. Specify *outer* to have all rows in the parent table included whether or not they are in the child table.
Argument	*Parent Elements ([Map] Elements)*	Specify the map elements corresponding to the columns in the parent table to be joined. These must be in the same order as the child elements.
Argument	*Child Elements ([Map] Elements)*	Specify the map elements corresponding to the columns in the child table to be joined. These must be in the same order as the corresponding parent elements.

30.4.26. DatabaseLookup

Name	**DatabaseLookup**

Purpose	Looks up a value in a database.	
Description	The **DatabaseLookup** function returns a single value of the specified column that satisfies the specified condition. Only certain functions are translated into SQL for evaluation by the database and can be used with **DatabaseColumn** and **DatabaseFunction** functions. These are: • *Add* • *And* • *Concat* • *Constant* • *Divide* • *Multiply* • *Or* • *Subtract* • *Not* That is to say that the **DatabaseColumn** and **DatabaseFunctions** must only be children of one of the above functions. You can also use *any* function as part of the expression tree to reference elements from your map. You just can't have the **DatabaseColumn** or **DatabaseFunction** be a child of functions other than on the list above.	
Return Type	Depends on the database column specified	
Property	*Output Column*	The fully qualified column whose value is to be returned by this function. The specification is of the form: *<path to database>.<table>.<column>* for example: */Default Project/Databases/MyDatabase.MYTABLE.MYCOLUMN*. Note that when using the *studio* a graphical interface specifies this, so you don't need to be concerned with this format.
Property	*Cache Lookups?*	Specifies if the values returned by this function are cached. Possible values are: • *DEFAULT* - cache according to the caching policy of the runtime • *ALWAYS* - always cache regardless of the runtime setting • *NEVER* - never cache values regardless of the runtime setting The runtime also has a cache timeout value which invalidates cached items after the expiration of the timeout value.
Argument	*Condition (Boolean)*	The "where" condition for this lookup. This is an expression that includes **DatabaseColumn** (and maybe **DatabaseFunction**) functions to specify to the database the selection criteria for the lookup. Use the normal functions for specifying conditions (e.g. **Equal**, **And**, **Add**). These functions will be translated into the corresponding SQL for evaluation by the database. You may use any other type of function in the Condition as well to refer to elements in the map for example.
Argument	*Output Column (String)*	The output column to select. This is mainly used when you want to use a function with the output column. For example, selecting the maximum value of a column. Use a **DatabaseFunction** to specify the function and

		a single **DatabaseColumn** within that to specify the column.

30.4.27. DatabaseLookupAndUpdate

Name	**DatabaseLookupAndUpdate**	
Purpose	Looks up and updates a value in a database.	
Description	The **DatabaseLookupAndUpdate** function returns a single value of the specified column that satisfies the specified condition and updates the value according to the expression in the *Update* parameter. This is useful for example when reading rows from a database to process and at the same time indicating they have been processes so they won't be processed again. The update occurs inside of the current database transaction (along with all other database activity during the execution of the map). Only certain functions are translated into SQL for evaluation by the database and can be used with **DatabaseColumn** and **DatabaseFunction** functions. These are: • *Add* • *And* • *Concat* • *Constant* • *Divide* • *Multiply* • *Or* • *Subtract* • *Not* That is to say that the **DatabaseColumn** and **DatabaseFunctions** must only be children of one of the above functions. You can also use *any* function as part of the expression tree to reference elements from your map. You just can't have the **DatabaseColumn** or **DatabaseFunction** be a child of functions other than on the list above.	
Return Type	Depends on the database column specified	
Property	*Output Column*	The fully qualified column whose value is to be returned by this function. The specification is of the form: *<path to database>.<table>.<column>* for example: */Default Project/Databases/MyDatabase.MYTABLE.MYCOLUMN*. Note that when using the *studio* a graphical interface specifies this, so you don't need to be concerned with this format.
Property	*Cache Lookups?*	Specifies if the values returned by this function are cached. Possible values are: • *DEFAULT* - cache according to the caching policy of the runtime • *ALWAYS* - always cache regardless of the runtime setting • *NEVER* - never cache values regardless of the runtime setting The runtime also has a cache timeout value which invalidates cached items after the expiration of the timeout value.

Argument	*Condition (Boolean)*	The "where" condition for this lookup. This is an expression that includes **DatabaseColumn** (and maybe **DatabaseFunction**) functions to specify to the database the selection criteria for the lookup. Use the normal functions for specifying conditions (e.g. **Equal**, **And**, **Add**). These functions will be translated into the corresponding SQL for evaluation by the database. You may use any other type of function in the Condition as well to refer to elements in the map for example.
Argument	*Update (Simple)*	An expression that specifies the value of the output column after the lookup. This is an expression that includes **DatabaseColumn** (and maybe **DatabaseFunction**) functions to specify the value of the output column. Use the normal functions for specifying conditions (e.g. **Equal**, **And**, **Add**). These functions will be translated into the corresponding SQL for evaluation by the database. You may use any other type of function in the Update as well to refer to elements in the map for example.
Argument	*Output Column (String)*	The output column to select. This is mainly used when you want to use a function with the output column. For example, selecting the maximum value of a column. Use a **DatabaseFunction** to specify the function and a single **DatabaseColumn** within that to specify the column.

30.4.28. DatabaseSelect

Name	**DatabaseSelect**	
Purpose	Specifies the conditions on a database select.	
Description	The **DatabaseSelect** function may be used only as an I/O expression. Specify it as the I/O expression of the element that contains the data to be read. Database select is used to define the selection criteria on the rows that are to be processed. You can provide only the "where" portion of the selection criteria, or provider the full contents of the SQL *Select* statement. In order to parametrize the *select* statement, you can access the map execution properties from within the *Where String* or *Query String* properties. Do this by specifying the name of the property as *${propertyName}*. For example *DepartmentCode = ${departmentCode}*.	
Return Type	N/A (this function may be used only as an I/O expression)	
Property	*Where String*	Enter a SQL condition as specified in a Where clause without the use of the word "Where". Specify only this or the *Query String* but not both. See notes below about how to parametrize the query.
Property	*Query String*	Specify the entire HQL (HQL is similar to SQL) query string to be used for the select. This allows all of the parts of the select statement. Specify only this or the *Where String* but not both. See notes below about how to parametrize the query.

30.4.29. DatabaseUpdate

Name	**DatabaseUpdate**
Purpose	Specifies the table to update rows into a database.

Description	The **DatabaseUpdate** function may be used only as an I/O expression. **DatabaseUpdate** specifies the table for which the row elements are updates. Specify this on the I/O expression of the element that inherits from the database table, which is the parent of the *Row* element.	
Return Type	N/A (this function may be used only as an I/O expression)	
Property	*Table*	The table to insert update. If not specified, assumed to be the table specified by the single row table structure inherited by the child *Row* element.

30.4.30. DescendingSort

Name	**DescendingSort**
Purpose	Specifies a map element to be sorted descending in a loop.
Description	This function can be used only in the *Sort Keys* argument of a loop expression and specifies the *[map]* elements that comprise a descending sort key.
Variable Arguments	
Variable Argument Type	*[Map]* Element

30.4.31. Divide

Name	**Divide**	
Purpose	Divides two numbers.	
Description	Divides the divisor into the dividend returning the quotient.	
Return Type	Generic Number	
Argument	*Dividend (Generic Number)*	The value to be divided by the divisor.
Argument	*Divisor (Generic Number)*	The value divided into the dividend.

30.4.32. EnclosingContext

Name	**EnclosingContext**
Purpose	Provides a *[map]* element to be used as the context that encloses a loop.
Description	This function can be used only in the *Contexts* argument of a loop expression and specifies the output map element to enclose this loop. Normally (if this function is not specified), a loop in the output is enclosed by its nearest looping ancestor output map element. Use this function if you wish the enclosing loop to be a higher level output map element. This is useful when you want the loop context for input map elements inside of this loop to refer to the specific enclosing map element's loop instead of the nearest ancestor. This is used in the X12 EDI HL loop special processing. Though this function specifies variable arguments, it can have exactly one argument.
Variable Arguments	
Variable Argument Type	*[Map] Element*

30.4.33. Equal

Name	**Equal**	
Purpose	Equality test of two values.	
Description	Returns true if the two values are equal. Returns false otherwise.	
Return Type	Boolean	
Argument	*First Value (Simple)*	The first value to test.
Argument	*Second Value (Simple)*	The second value to test.

30.4.34. ExecuteMap

Name	**ExecuteMap**	
Purpose	Executes a map.	
Description	This function executes a map. Nothing is returned from the map execution, which is completely separate from the currently executing map. The map is executed entirely before the function returns, so it is possible for the current map (the map calling this function) to depend on the results of the called map execution. For convenience, this function allows you to specify the map path name as either a property or argument. If both are specified, the argument takes precedence.	
Return Type	None	
Property	*Map Path Name*	The full path name of the map to execute. For example, if you want to execute a map called *DepartmentMap* in the project *Personnel*, the name would be */Personnel/Maps/DepartmentMap*.
Argument	*Map Path Name (String)*	The full path name of the map to execute. For example, if you want to execute a map called *DepartmentMap* in the project *Personnel*, the name would be */Personnel/Maps/DepartmentMap*.
Argument	*Properties (Any number of Property functions)*	The properties to be passed to the map to execute.

30.4.35. ExtractBytes

Name	**ExtractBytes**	
Purpose	Returns the specified bytes extracted from the binary data.	
Description	Extracts the specified bytes from the binary value and returns them.	
Return Type	Binary	
Argument	*Input (Binary)*	The value to extract from.
Argument	*Offset (Integer)*	The offset into the binary data to start the extraction from. The first byte is *0*.
Argument	*Length (Integer)*	The number of bytes to extract.

30.4.36. ExtractFromDateTime

Name	ExtractFromDateTime
Purpose	Extracts a part of a date/time value.

Description	This extracts the specified type of value from a date/time value.	
Return Type	(depends on what's extracted)	
Property	What	What to extract. Select date, time, year, month, day of month, day of week, day of year, week, hour, minute, second, millisecond or timezone. The timezone is provided either as the timezone as specified in the ISO 8601 date format which is either "Z" or a string with the offset from UTC like "-04:30".
Argument	Value (Date/Time)	The date/time value.

30.4.37. FixedLoop

Name	**FixedLoop**	
Purpose	Loop a fixed number of times.	
Description	This loop function can be used only in the loop expression tab and specifies that the associated output map element is to loop a specified number of times. This function is typically used in the case where the minimum number of times the map element occurs is zero and a child map element has a default value or constant values that want to appear in the output. You can use the **LoopIndex** function in conjunction with this function to get the index number for each occurrence of the output loop.	
Property	*Count*	The number of times to loop. This is optional and if omitted 1 time is assumed. The *Number of Times* argument takes precedence over this.
Argument	*Number of Times (Integer)*	The number of times to loop. This is optional and if omitted, 1 time is assumed.

30.4.38. FlatToHierarchyLoop

Name	**FlatToHierarchyLoop**	
Purpose	Loop, emitting nested recursive elements based on their level in a flat set of input elements.	
Description	This loop function can be used only in the loop expression tab and allows you to create nested recursive elements in the output based on a level number in the input. To illustrate, if the level number is always one, this functions identically to the SimpleLoop function. When the level number is *2*, a nested element (that is the same as the enclosing output element) is emitted. For each increment of the level number a new level of nesting is created. It is assumed that the level number will increase and decrease monotonically over the input.	
Argument	*Input map element ([Map] Element)*	An input map element that loops.
Argument	*Filters (Boolean)*	Specify an expression that returns a boolean. This expression is evaluated for each instance of the loop. If this filter expression returns *true*, the instance is included in the loop. If it returns *false*, the instance is excluded from the loop.
Argument	*Sort Keys (Variable) (Either AscendingSortKey or DescendingSortKey functions)*	Any number of **AscendingSort** or **DescendingSort** functions that specify each sort key.
Argument	*Contexts (Either NestedContext or EnclosingContext functions)*	Specify either the **EnclosingContext** and/or **NestedContext** function. *EnclosingContext* specifies

		the output map element that encloses this loop. If not specified, the nearest looping ancestor map element is used. *NestedContext* allows another loop expression to be specified within this loop expression.

30.4.39. GetBytesFromURL

Name	**GetBytesFromURL**	
Purpose	Reads the specified URL and returns the contents as a binary.	
Description	This is used to read data from an arbitrary URL. It can be converted to any other data type (for further processing) using the **ExtractBytes** and **ConvertFromBinary** functions.	
Return Type	Binary	
Argument	*URL (String)*	The URL to read the bytes from.

30.4.40. GetCurrentDateTime

Name	**GetCurrentDateTime**
Purpose	Returns the current date and time in the local timezone.
Description	Returns the current date/time in the local timezone.
Return Type	Date/Time

30.4.41. GetElementProperty

Name	**GetElementProperty**	
Purpose	Return the selected property of the current element.	
Description	Returns the value of the specified property (for example, the name) of the current element. The current element is the element whose expression tree this function is in. This is useful for example to reuse the same set of expressions with different elements, where the expressions depend on the name of the element.	
Return Type	String	
Property	*Property*	The desired property of the element. • *Element name* - the name of the element. • *Level number* - the depth of the element. The root element has the value *1*. • *Byte offset* - the byte offset of the element from its enclosing element (usable in flat representations only).

30.4.42. GetMapProperty

Name	**GetMapProperty**
Purpose	Return the specified map execution property.

Description	When a map is executed, properties may be specified to pass small amounts of arbitrary data available during map execution. In addition, there are some built-in map properties available. When using the tHMap component, all of the context variables are set as map execution properties. When using Camel (including the cMap component), the exchange's inbound headers are set as map execution properties. To use this function, double-click it and then enter the name of the map execution property, without quotes, in the **Property Name** field of the **[Expression GetMapProperty Properties]** dialog box that opens.	
Return Type	String	
Property	*Property*	The name of the desired map execution property.

30.4.43. GetSequenceFromLocalFile

Name	**GetSequenceFromLocalFile**	
Purpose	Gets and updates sequence number stored in a local file.	
Description	This function is used to provide a monotonically increasing sequence number which can be used as a control number, for example in EDI. It does this by using a local file to store the current value of the sequence number. This file can be shared by any number of concurrent maps as it is protected using appropriate locking. If the file is not present, it is automatically created and the sequence number starts with *1*. The contents of the file is simply a ASCII encoded string with a numeric value, so it can be edited to use any desired sequence number.	
Return Type	Long	
Property	*FilePath*	The path name of the local file. You may use *${propName}* to substitute the value of a Java system property. For example, *${user.home}/control.txt* uses the *user.home* (home directory) Java system property.

30.4.44. GetVariable

Name	**GetVariable**	
Purpose	Return the value of the specified variable.	
Description	A map supports named variables which may be set by the **SetVariable** function. This function gets the value of a previously set variable. The data type of the variables is a string; like any other string numeric operations are possible so long as the value is numeric.	
Return Type	String	
Property	*Variable Name*	The name of the variable.

30.4.45. Greater

Name	**Greater**	
Purpose	Greater than test of two values.	
Description	Returns true if the first value is greater than the second value. Returns false otherwise.	
Return Type	Boolean	
Argument	*First Value (Simple)*	The first value to test.
Argument	*Second Value (Simple)*	The second value to test.

30.4.46. GreaterOrEqual

Name	**GreaterOrEqual**	
Purpose	Greater than or equal test of two values.	
Description	Returns *true* if the first value is greater than or equal to the second value. Returns *false* otherwise.	
Return Type	Boolean	
Argument	*First Value (Simple)*	The first value to test.
Argument	*Second Value (Simple)*	The second value to test.

30.4.47. HasValue

Name	**HasValue**	
Purpose	Returns *true* if the argument has a non-blank value.	
Description	Returns *true* if the value of *Input Value* argument has a non-blank value, that is a value whose length is greater than zero. It will return *false* if the element is not present or is present but has a blank value. This is different than the **IsPresent** function in that it requires the element to have text, not merely be present.	
Return Type	Boolean	
Argument	*Input Value (Simple)*	The value that is checked.

30.4.48. IfThen

Name	**IfThen**	
Purpose	Returns a value conditionally.	
Description	The **IfThen** Function evaluates the *Condition* argument and returns the value of the *Then* argument if the *Condition* is *true*. Nothing is returned if the value of the *Condition* is *false*. You may use the **Compare**, **And**, **Or**, or **Not** functions to specify the value of the *Condition* argument.	
Return Type	Simple	
Argument	*Condition (Boolean)*	The expression to evaluate.
Argument	*Then (Simple)*	The value to return if the condition is *true*.

30.4.49. IfThenElse

Name	**IfThenElse**	
Purpose	Returns a value conditionally.	
Description	The **IfThen** Function evaluates the *Condition* argument and returns the value of the *Then* argument if the *Condition* is *true*. It returns the value of the *Else* argument if the *Condition* is *false*. You may use the **Compare**, **And**, **Or**, or **Not** functions to specify the value of the *Condition* argument.	
Return Type	Simple	
Argument	*Condition (Boolean)*	The Expression to evaluate.

Argument	*Then (Simple)*	The value to return if the Condition is *true*.
Argument	*ElseThen (Simple)*	The value to return if the Condition is *false*.

30.4.50. IndexOf

Name	**IndexOf**	
Purpose	Return the index value of a substring in a string.	
Description	Return the one-based index of the *Compare String* within the specified *String*. For example, if the *String* is *abcd*, and the *Compare String* is *cd*, *3* is returns. If the *Compare String* is not found, then *-1* is returned.	
Return Type	Integer	
Argument	*String (String)*	The *Expression* to evaluate.
Argument	*Compare String (String)*	The value to return if the Condition is *true*.

30.4.51. IndexRangeLoop

Name	**IndexRangeLoop**	
Purpose	Loop extracting a range of input iterations.	
Description	This loop function can be used only in the loop expression tab and specifies that the output map element associated with the loop expression tab is to loop corresponding to the value of the input map element argument expression and selecting only iterations of the loop that fall between the start and end index values, and before the end condition is met. This is useful in the processing of something like the X12 EDI HL loop where it is desirable to process only a range of iterations from the input, and to allow this range to terminate with a certain condition. If no sort keys are specified, the order of the output is the same order as the input elements.	
Argument	*Input map element ([Map] Element)*	An input map element that loops.
Argument	*Start Index (Integer)*	The first index (inclusive) of the input map element's loop to consider. This is optional and if not specified it will take all iterations before the end index.
Argument	*End Index (Integer)*	The last index (inclusive) of the input map element's loop to consider. This is optional, if not specified, it will take all iterations after the start index.
Argument	*End Condition (Boolean)*	An expression that, when true, indicates the looping should stop. This is optional.
Argument	*Filters (Boolean)*	Specify an expression that returns a boolean. This expression is evaluated for each instance of the loop. If this filter expression returns *true*, the instance is included in the loop. If it returns *false*, the instance is excluded from the loop.
Argument	*Sort Keys (Variable) (Either AscendingSortKey or DescendingSortKey functions)*	Any number of **AscendingSort** or **DescendingSort** functions that specify each sort key.
Argument	*Contexts (Either NestedContext or EnclosingContext functions)*	Specify either the **EnclosingContext** and/or **NestedContext** function. *EnclosingContext* specifies the output map element that encloses this loop. If not specified, the nearest looping ancestor map element is used.

		NestedContext allows another loop expression to be specified within this loop expression.

30.4.52. IsNull

Name	**IsNull**	
Purpose	Returns *true* if the argument has a null value.	
Description	The behavior of this function is dependent on the representation. In XML, returns *true* if the value of the *Input Value* argument is an element with the *xsi:nil* attribute that indicates a null value. For Database, returns *true* if the value of the *Input Value* argument is a null value. For other representations *false* is always returned as they don't support the notion of null values.	
Return Type	Boolean	
Argument	*Input Value (Simple)*	The value that is checked.

30.4.53. IsPresent

Name	**IsPresent**	
Purpose	Returns *true* if the argument exists.	
Description	The behavior of this function is dependent on the representation. In XML and EDI, returns *true* if the *Input Value* argument is present in the document. That is if there is an element or attribute specified. This is different than the **HasValue** function in that this returns *true* if the element exists in the document but there is no text associated with the element. For flat representations, returns *true* if the value for the element appears, or if the element is mandatory. For database, always returns *true*, as all columns are present on the input. Like XML **HasValue** may or may not be true depending on if there is data present.	
Return Type	Boolean	
Argument	*Input Value (Simple)*	The value that is checked.

30.4.54. IsValid

Name	**IsValid**	
Purpose	Returns *true* if an element is valid according to validation constraints.	
Description	This function is used to select only valid elements in a loop expression. It may only be used when a validation group has been defined. More specifically, it may only be used within a loop expression that refers to an input element that has a **ValidateGroup** function specified in its validation expression.	
Argument	*Severity*	The validation severity for an element to be considered not valid. This is *FATAL*, *ERROR*, *WARNING*, or *INFO*. Any validation reports with a severity greater or equal to the specified severity mark the element as invalid for this filter (and it thus returns *false*). The default value is *ERROR*.

30.4.55. Java

Name	**Java**	
Purpose	Call a static Java method.	
Description	This calls a static Java method using the Java reflection mechanism and returns the value returned by the method. Use the properties to specify only the class name and method name. Use the arguments to specify the arguments to the function. For example, to get the *user.home* Java system property, use this expression: `Java (class: java.lang.System, method: getProperty)` `  Constant ("user.home")` Instance objects can also be called using this function. To do so, make the first argument of the method be an instance of the object to call. For example, to get a randon number, using this expression: `Java ([className="java.util.Random", methodName="nextInt"],` ` Java([classname="java.util.Random", methodName="new”))`	
Return Type	Simple	
Variable Arguments		
Variable Argument Type	Simple	
Property	*Class Name*	The fully qualified name of the Java class to call.
Property	*Method Name*	The name of the Java method to call within the class. The Java method may have any number of parameters; this number of parameters must match the number of arguments specified when this function is used.

30.4.56. Left

Name	**Left**	
Purpose	Return left-most characters.	
Description	Returns the specified number of left-most characters from the input argument.	
Return Type	String	
Argument	*Input value (String)*	The string to process.
Property	*Length*	The number of left-most characters to return.

30.4.57. Lesser

Name	**Lesser**	
Purpose	Less than test of two values.	
Description	Returns the specified number of left-most characters from the input argument.	
Return Type	Boolean	
Argument	*First Value (Simple)*	The first value to test.
Argument	*Second Value (Simple)*	The second value to test.

30.4.58. LesserOrEqual

Name	**LesserOrEqual**

Purpose	Less than or equal test of two values.	
Description	Returns *true* if the first value is less than or equal to the second value. Returns *false* otherwise.	
Return Type	Boolean	
Argument	*First Value (Simple)*	The first value to test.
Argument	*Second Value (Simple)*	The second value to test.

30.4.59. LoopCopy

Name	**LoopCopy**	
Purpose	Copy the loop expression from the specified output map element's loop expression.	
Description	This loop function can be used only in the loop expression tab or as part of an aggregate function. This specifies that the same looping instructions are to be used as the referenced output map element. This function is automatically generated when using an aggregate function to refer to the loop expression corresponding to the argument of the aggregate function. It is simply a shorthand for the loop expression to which is refers. The difference between this and the **LoopReference** function is that this may be used anywhere in the map, and it will execute newly the loop expression to which is refers rather than taking the values emitted by the loop expression as originally executed (which is what **LoopReference** does).	
Argument	*Map Element (Map Element)*	An output map element that has a loop expression.
Argument	*Contexts (Either NestedContext or EnclosingContext functions)*	Specify either the **EnclosingContext** and/or **NestedContext** function. *EnclosingContext* specifies the output map element that encloses this loop. If not specified, the nearest looping ancestor map element is used. *NestedContext* allows another loop expression to be specified within this loop expression.

30.4.60. LoopIndex

Name	**LoopIndex**	
Purpose	Returns the index value of the specified iterating output map element.	
Description	This is used if you want to use the current index value in some expression in an output loop. The value for the first iteration of a loop is one.	
Return Type	Integer (32)	
Argument	*Looping Output Element ([Map] Element)*	Specify the looping output map element for which you want the index value.

30.4.61. LoopReference

Name	**LoopReference**	
Purpose	Loop using values from a previous output map element loop.	
Description	This loop function can be used only in the loop expression tab and specifies that this output map element is to loop using the values that were provided in the loop expression from the specified map element. It may only be used for map elements that occur after the specified map element in the map output. This function is generally used in conjunction with references to output map elements.	

	The difference between this and the **LoopCopy** function is that this may be used only after the output map element to which it refers, and this will use the same values produced in the original loop without reexecuting the loop expression that created the loop.	
Argument	*Map Element (Map Element)*	An output map element that has a loop expression that occurs before this output map element in the map output.
Argument	*Contexts (Either NestedContext or EnclosingContext functions)*	Specify either the **EnclosingContext** and/or **NestedContext** function. *EnclosingContext* specifies the output map element that encloses this loop. If not specified, the nearest looping ancestor map element is used. *NestedContext* allows another loop expression to be specified within this loop expression.

30.4.62. LoopVariable

Name	**LoopVariable**	
Purpose	Returns the XQuery *for* loop variable associated with the specified looping output map element.	
Description	This is a low-level function used when it is necessary to directly access the generated XQuery code. Currently this is used with the functions that support the processing of the EDI HL loop.	
Return Type	String	
Argument	*Looping Output Element ([Map] Element)*	Specify the looping output map element for which you want the loop variable.

30.4.63. Property

Name	**Property**	
Purpose	Specifies a property, which is a name/value pair.	
Description	This function is used to specify a property and its value. It is used wherever a name/value pair property is required; this includes the following functions: **ExecuteMap**, **ValidateReport**, **CondValidateReport**, and **ValidateGroup**. For convenience, this function allows you to specify the property as either a property or argument. If both are specified, the argument takes precedence.	
Return Type	String	
Property	*Property Name*	The name of the property.
Property	*Property Value*	The value of the property.
Argument	*Property Name (String)*	The name of the property.
Argument	*Property Value (String)*	The value of the property.

30.4.64. MakeDateTime

Name	**MakeDateTime**
Purpose	Creates a DateTime from a Date and Time.
Description	This creates a value of type DateTime from a Date and a Time. Note that a value of type Date or Time is automatically converted to and from a DateTime type. Use this

	function only when you have both a Date and a Time and you wish to combine them to create a value of DateTime.	
Return Type	Date/Time	
Argument	*Date (Date)*	The date value.
Argument	*Time (Time)*	The time value.

30.4.65. MapValues

Name	**MapValues**	
Purpose	Specifies a mapping of a set of input values to output values.	
Description	This function is used map a set of input values to output values for a map element. It supports a variable number of **ValueMapping** functions each of which specifies the mapping of one or more input values to a single output value.	
Return Type	String	
Argument	*Source Map Element (String)*	The (typically input) map element that is the source of the input values.
Argument	*Values (ValueMapping)*	A mapping of a set of input values to an output value. Any number of **ValueMapping** functions may be specified here.

30.4.66. Multiply

Name	**Multiply**	
Purpose	Multiplies two numbers.	
Description	Multiplies the first value and second value returning the product.	
Return Type	Generic Number	
Argument	*First Value (Generic Number)*	The first value to multiply.
Argument	*Second Value (Generic Number)*	The second value to multiply.

30.4.67. NameValuePairLookup

Name	**NameValuePairLookup**
Purpose	Looks up a value in a series of adjacent element pairs that represent names and values.
Description	This is a special purpose Function designed to deal with pairs of adjacent elements that represent names and values. This occurs in some EDI documents, for example in the LIN Segment of X12 EDI. The best way to illustrate this Function is by example. The LIN Segment has the following 5 name/value pairs: `LIN01 - Product/Service ID Qualifier` `LIN02 - Product/Service ID` `LIN03 - Product/Service ID Qualifier` `LIN04 - Product/Service ID` `LIN05 - Product/Service ID Qualifier` `LIN06 - Product/Service ID` `LIN07 - Product/Service ID Qualifier` `LIN08 - Product/Service ID` `LIN09 - Product/Service ID Qualifier`

	`` LIN10 - Product/Service ID ``` The *Product/Service ID Qualifier* is the *Name*, and the *Product/Service ID* is the *Value*. If we wanted to look for the *Product/Service ID* with the qualifier of *UP*, we would use the following Expression tree: ``` NameValuePairLookup Lookup Value Constant 'UP' First Element LIN01 Last Element LIN09 ``` This will search through each of the *Name* elements *LIN01, 03, 05, 07, 09* and if what of those values is *UP* it will return the value of the *Value* element which is the following corresponding element *LIN02, 04, 06, 08, 10*.  Please note that the *Last Element* argument takes the last *Name* element (*LIN09* in this case), not the last *Value* element (*LIN10*).	
Return Type	Simple	
Argument	*Lookup Value (simple)*	The value to compare to the *Name* elements in the series.
Argument	*First Element (Sequence)*	The first *Name* element in the series of *Name/Value* elements.
Argument	*Last Element (Sequence)*	The last *Name* element in the series of *Name/Value* elements.

30.4.68. NestedContext

Name	**NestedContext**
Purpose	Allows a loop expression to be nested in this loop expression.
Description	This function can be used only in the *Contexts* argument of a loop expression and specifies a loop expression to be nested in this loop expression. When referencing input loops, every level of input looping that is an ancestor to the referenced loop must have a corresponding loop expression in the output. The nesting capability provides a way to refer to multiple input loops within a single loop expression. Though this function specifies variable arguments, it can have exactly one argument, which is a loop function.
Variable Arguments	
Variable Argument Type	Loop Expression

30.4.69. NormalizeSpace

Name	**NormalizeSpace**
Purpose	Removes excess white space.
Description	Removes excess leading or trailing whitespace. If there are multiple spaces between words in the value of the map element, they are reduced to one space. Any whitespace non blank characters are removed. This is identical to the XPath *fn:normalize-space* function. If you don't want to disturb the non-leading and trailing spaces, use the **Trim** function.
Return Type	String
Variable Arguments	
Variable Argument Type	String

30.4.70. Not

Name	**Not**	
Purpose	Boolean *not* (negation).	
Description	Returns *true* if the *Input Value* argument is *false*, and *false* if it is *true*.	
Return Type	Boolean	
Argument	*Input Value (Boolean)*	The input value to negate.

30.4.71. NotEqual

Name	**NotEqual**	
Purpose	Not equal test of two values.	
Description	Returns *true* if the first value is not equal to the second value. Returns *false* if they are equal.	
Return Type	Boolean	
Argument	*First Value (Simple)*	The first value to test.
Argument	*Second Value (Simple)*	The second value to test.

30.4.72. Or

Name	Or
Purpose	Boolean *or*.
Description	Performs a Boolean *Or* operation on all of the arguments. If any of the arguments are true, returns *true*, otherwise returns *false*.
Return Type	Boolean
Variable Arguments	
Variable Argument Type	Boolean

30.4.73. ParseDateTime

Name	**ParseDateTime**	
Purpose	Creates a DateTime from a value as specified by a pattern.	
Description	This creates a value of type DateTime from a string according to the specified pattern.	
Return Type	Date/Time	
Property	*Pattern*	The pattern to be used when parsing the value into a date/time. For example *YYYY-MM-dd* would parse *1960-09-11*. The following values are permitted: `Symbol Meaning Presentation Examples ------ ------- ------------ ------- G era text AD C century of era (>=0) number 20 Y year of era (>=0) year 1996`

		`` x weekyear year 1996 w week of weekyear number 27 e day of week number 2 E day of week text Tuesday; Tue y year year 1996 D day of year number 189 M month of year month July; Jul; 07 d day of month number 10 a halfday of day text PM K hour of halfday (0~11) number 0 h clockhour of halfday (1~12) number 12 H hour of day (0~23) number 0 k clockhour of day (1~24) number 24 m minute of hour number 30 s second of minute number 55 S fraction of second number 978 z time zone text Pacific Standard Time; PST Z time zone offset/id zone -0800; -08:00; America/ Los_Angeles ' escape for text delimiter '' single quote literal ' ``` The count of pattern letters determine the format. *Text* : If the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available. *Number* : The minimum number of digits. Shorter numbers are zero-padded to this amount. *Year* : Numeric presentation for year and weekyear fields are handled specially. For example, if the count of *y* is 2, the year will be displayed as the zero-based year of the century, which is two digits. *Month* : 3 or over, use text, otherwise use number. *Zone* : *Z* outputs offset without a colon, *ZZ* outputs the offset with a colon, *ZZZ* or more outputs the zone id. Any characters in the pattern that are not in the ranges of *['a'..'z']* and *['A'..'Z']* will be treated as quoted text. For instance, characters like ":", ".", " ", "#" and "?" will appear in the resulting time text even they are not embraced within single quotes.
Argument	*Value (String)*	The value to parse into a date/time.

30.4.74. ReadMapInput

Name	ReadMapInput	
Purpose	Reads the instance document specified by the input of the map into a map element.	
Description	The **ReadMapInput** function may be used only as an I/O expression. Specify it as the I/O expression of the element to contain the data to be read. The contents of the source URL for the map are read during the map execution populating the values of the element containing the I/O expression and any subordinate elements. You may not nest the use of this function. In other words, if a map element contains an I/O expression, no subordinate map element may contain an I/O expression. See the discussion on Input/Output for further details.	
Return Type	N/A (this function may be used only as an I/O expression)	
Property	*Representation*	The representation type to select when processing this URL. This is used if multiple representations are available in the structure controlling the I/O (the structure inheriting this element).
Property	*Offset*	The byte offset into the contents to start reading at. If not specified, *0* is assumed.
Property	*Length*	The number of bytes of content to read. If not specified all of the bytes are read.
Argument	*Offset (Integer)*	The byte offset into the contents to start reading at. If not specified, *0* is assumed.
Argument	*Length (Integer)*	The number of bytes of content to read. If not specified all of the bytes are read.

30.4.75. ReadMessage

Name	ReadMessage	
Purpose	Reads a message from an enclosing ESB into a map element.	
Description	The *ReadMessage* function may be used only as an I/O expression. Specify it as the I/O expression of the element to contain the data to be read. The contents of message are read during the map execution populating the values of the element containing the I/O expression and any subordinate elements. *Reading Message Properties* - if you want to access properties that are associated with the message, they can be read into map elements. To do this the first child map element must specify the **ReadMessageProperties** I/O function, inherit from the */Builtin/Structures/Properties* structure, and must be a loop. When the message is read, the property elements of this structure are populated with the property names and values. You may use this function in multiple map elements in order to read multiple messages. You may not however nest the use of this function. In other words, if a map element contains an I/O expression, no subordinate map element may contain an I/O expression. See the discussion on Input/Output for further details.	
Return Type	N/A (this function may be used only as an I/O expression)	
Property	*Representation*	The representation type to select when processing this message. This is used if multiple representations are available in the structure controlling the I/O (the structure inheriting this element).
Property	*Endpoint*	The endpoint name that is meaningful to the enclosing ESB to identify where to get the message.
Property	*Offset*	The byte offset into the contents to start reading at. If not specified, *0* is assumed.
Property	*Length*	The number of bytes of content to read. If not specified all of the bytes are read.

Argument	*Offset (Integer)*	The byte offset into the contents to start reading at. If not specified, *0* is assumed.
Argument	*Length (Integer)*	The number of bytes of content to read. If not specified all of the bytes are read.

30.4.76. ReadMessageProperties

Name	**ReadMessageProperties**
Purpose	Reads the properties from an ESB message into map elements.
Description	The **ReadMessageProperties** function may be used only as an I/O expression. It is used to read the properties from ESB message into the map elements. This may be used only in the first child element under the **ReadMessage** function. Also, the element on which this function is specified must have a occurs maximum times of *-1* and must inherit from */Builtin/Structures/Property*.
Return Type	N/A (this function may be used only as an I/O expression)

30.4.77. ReadNested

Name	**ReadNested**	
Purpose	Processes embedded data with the specified representation.	
Description	The **ReadNested** function may be used only as an I/O expression. Specify it as the I/O expression of the element to contain the data to be read. Specifying this function will cause the data normally read for this element to be processed using the representation of the structure from which this element inherits. Used this to handle mapping of embedded data of a different representation. You may use this function in multiple map elements in order to read multiple files. You may not however nest the use of this function. In other words, if a map element contains an I/O expression, no subordinate map element may contain an I/O expression. See the discussion on Input/Output for further details.	
Return Type	N/A (this function may be used only as an I/O expression)	
Property	*Representation*	The representation type to select when processing this URL. This is used if multiple representations are available in the structure controlling the I/O (the structure inheriting this element).

30.4.78. ReadURL

Name	**ReadURL**
Purpose	Reads the contents of a URL into a map element.
Description	The **ReadURL** function may be used only as an I/O expression. Specify it as the I/O expression of the element to contain the data to be read. The contents of the URL will be read during the map execution populating the values of the element containing the I/O expression and any subordinate elements. You may use this function in multiple map elements in order to read multiple files. You may not however nest the use of this function. In other words, if a map element contains an I/O expression, no subordinate map element may contain an I/O expression. See the discussion on Input/Output for further details. This function can be used to allow reading from multiple documents/objects in the map execution that are assigned dynamically by calling methods on the *MapExecutionContext*. In this case the URL is not required.

Return Type	N/A (this function may be used only as an I/O expression)	
Property	*URL*	The URL from which to read the data. If you want to read a file, specify a file URL which is: *file://* followed by the name of the file.
Property	*Representation*	The representation type to select when processing this URL. This is used if multiple representations are available in the structure controlling the I/O (the structure inheriting this element).
Property	*Offset*	The byte offset into the contents to start reading at. If not specified, *0* is assumed.
Property	*Length*	The number of bytes of content to read. If not specified all of the bytes are read.
Argument	*Offset (Integer)*	The byte offset into the contents to start reading at. If not specified, *0* is assumed.
Argument	*Length (Integer)*	The number of bytes of content to read. If not specified all of the bytes are read.

30.4.79. RecursiveLoop

Name	**RecursiveLoop**	
Purpose	Loop recursively according to some input map element.	
Description	This loop function can be used only in the loop expression tab and specifies that the output map element associated with the loop expression tab is to loop recursively corresponding to the value of the input map element argument expression. This is used in the case where both the enclosing output map element and the referenced input map element are recursive map elements. For each instance of the element in the input, regardless of the depth of recursion, an instance of the output is created. If no sort keys are specified, the order of the output is the same order as the input elements.	
Property	*Input map element ([Map] Element)*	An input map element that loops.
Property	*Filters (Boolean)*	Specify an expression that returns a boolean. This expression is evaluated for each instance of the loop. If this filter expression returns *true*, the instance is included in the loop. If it returns *false*, the instance is excluded from the loop.
Property	*Sort Keys (Variable) (Either AscendingSortKey or DescendingSortKey functions)*	Any number of **AscendingSort** or **DescendingSort** functions that specify each sort key.
Property	*Contexts (Either NestedContext or EnclosingContext functions)*	Specify either the **EnclosingContext** and/or **NestedContext** function. *EnclosingContext* specifies the output map element that encloses this loop. If not specified, the nearest looping ancestor map element is used. *NestedContext* allows another loop expression to be specified within this loop expression.

30.4.80. Right

Name	**Right**
Purpose	Return right-most characters.

Description	Returns the specified number of right-most characters from the input argument.	
Return Type	String	
Argument	*Input Value (String)*	The string to process.
Property	*Length*	The number of right-most characters to return.

30.4.81. SimpleLoop

Name	**SimpleLoop**	
Purpose	Loop according to some input map element.	
Description	This loop function can be used only in the loop expression tab and specifies that the output map element associated with the loop expression tab is to loop corresponding to the value of the input map element argument expression. When an input map element is mapped to an output map element that loops, the **SimpleLoop** function is automatically created at the nearest ancestor looping output element to define the loop. If no sort keys are specified, the order of the output is the same order as the input elements.	
Argument	*Input map element ([Map] Element)*	An input map element that loops.
Argument	*Filters (Boolean)*	Specify an expression that returns a boolean. This expression is evaluated for each instance of the loop. If this filter expression returns *true*, the instance is included in the loop. If it returns *false*, the instance is excluded from the loop.
Argument	*Sort Keys (Variable) (Either AscendingSortKey or DescendingSortKey functions)*	Any number of **AscendingSort** or **DescendingSort** functions that specify each sort key.
Argument	*Contexts (Either NestedContext or EnclosingContext functions)*	Specify either the **EnclosingContext** and/or **NestedContext** function. *EnclosingContext* specifies the output map element that encloses this loop. If not specified, the nearest looping ancestor map element is used. *NestedContext* allows another loop expression to be specified within this loop expression.
Property	*Emit Loop Instance If Empty*	Select this if you wish that a single instance of this loop be generated in the output if there are no instances of the input loop. This is useful to generate loop instance with a default value.
Property	*Distinct Child Element*	Provide the input map element to be used to select distinct values. Sadly, you have to type in the element name here. This name is a relative path of the element below the input map element specified as an argument. If there is a namespace involved include that. For example, if the input map element is a database row, then the distinct child element might be *Database:CustomerNumber*. If selected, only iterations of the loop that have distinct values for the selected element are considered. If this is not selected, all instances of the input map element are considered.
Property	*Stream Input*	Allows the input data for this loop to be broken into chunks and processed separately. Without this option, the entire input data is read into memory before the transformation is executed. When *Stream Input* is specified, since the transformation is processed in segments, input data of unlimited size can be processed. See the section on Streaming Execution for more information.

		 *When Stream Input is selected there are some significant restrictions on the transformation expressions. These restrictions are described in the section on Streaming Execution.*

30.4.82. SetElementProperty

Name	**SetElementProperty**	
Purpose	Sets the byte offset, the size of the element, and if the element loops, the actual number of occurrences.	
Description	Sets the value of the specified property of the current element. The current element is the element whose expression tree this function is in. It is used in the Util tab.  *This function is only used for flat representations.*	
Argument	*Byte offset (Integer)*	The byte offset of the element from its enclosing element.
	Number of bytes (Integer)	The size of the element in bytes.
	Loop Count (Integer)	If the element loops, the actual number of times it occurs.

30.4.83. SetEnclosingElement

Name	**SetEnclosingElement**	
Purpose	Sets the name of the enclosing element.	
Description	Sets the name of the current element. The current element is the element whose expression tree this function is in. This is useful for example to reuse the same set of expressions with different elements, where the expressions depend on the name of the element. With this function, you can specify an expression to calculate the name of the element (typically this would be a *Constant* function).  *This function may only be used as the root of the util expression tree for an element.*	
Argument	*Name (String)*	The name to be given to the enclosing element.

30.4.84. SetVariable

Name	**SetVariable**	
Purpose	Sets the value of the specified variable.	
Description	A map supports named variables which may be set by this function and retrieved using the **GetVariable** function. This function sets the value of a variable. The data type of the variables is a string; like any other string numeric operations are possible so long as the value is numeric.	
Return Type	String	
Argument	*Value (String)*	The value to which to set the variable.
Property	*Variable Name*	The name of the variable.

30.4.85. SingleIndex

Name	**SingleIndex**	
Purpose	Select a single input map element in a loop.	
Description	This function can be used only in the Filters argument of a loop expression and specifies the index value of the desired element.	
Argument	*Index (Integer)*	The index of the element to select, with the first being one. If the index property is also specified, this takes precedence.
Property	*Index*	The index of the element to select, with the first element being *1*.

30.4.86. StringLength

Name	**StringLength**
Purpose	Calculate the length, in characters, of a string.
Description	This function calculates the length of a string, in characters. It works in the same way as the XPath *fn:string-length* function.
Return Type	Integer
Variable Arguments	-
Variable Argument Type	String

30.4.87. Substring

Name	**Substring**	
Purpose	Return a substring of characters.	
Description	This function can be used only in the *Filters* argument of a loop expression and specifies the index value of the desired element.	
Return Type	String	
Argument	*Input Value (Sting)*	The string to process.
Property	*Start*	The one-based character that starts the returned substring. The first character is *1*.
Property	*Length*	The number of characters to return after (and including) the start character.

30.4.88. Subtract

Name	**Subtract**	
Purpose	Subtracts two numbers.	
Description	Subtracts the subtrahend from the first value minuend returning the difference.	
Return Type	Generic Number	
Argument	*Minuend (Generic Number)*	The value from which the subtrahend is subtracted.
Argument	*Subtrahend (Generic Number)*	The value to subtract from the minuend.

30.4.89. Trim

Name	**Trim**
Purpose	Removes excess leading and trailing white space.
Description	Removes excess leading or trailing whitespace. Whitespace that is not leading or trailing is unaffected, unlike the **NormalizeSpace** function.
Return Type	String
Variable Arguments	
Variable Argument Type	String

30.4.90. ValidateGroup

Name	**ValidateGroup**	
Purpose	Declares an element to be a validation group.	
Description	A validation group is used to identify an element where you want to take additional action (like filter it out of a loop) or provide additional reporting in the event of a validation failure within the group. The validation group is used with the **IsValid** loop filter function to filter valid elements. If a validation report happens to an element within the validation group (that is an element that is equal to or subordinate to the element associated with the validation group function), than a separate validation report is made according to this function, which is associated with the triggering validation report. This allows you to provide additional context for validation reports in a group. This function may be specified only in the validate expression tab of the desired element. This function also allows the **CondValidateReport** function to be specified whose value is passed through, so that you may also provide validation directly on the element that is a validation group.	
Return Type	Boolean	
Argument	*Condition (Boolean)*	Used to allow for a **CondValidateReport** to be used in this expression. Has no effect on the execution of this function, the *Condition* value is merely passed through as the return value of this function.
Argument	*Data (Simple (variable))*	Any number of expressions that data to be associated with the validation report. These are reported as name/value pairs in the validation report. If an expression specified here is a *[map]* element reference, the name is taken to be the name of the *[map]* element and the value is the element's value. Use the **Property** function with other types of expressions to provide the name label in the validation report.
Argument	*Severity*	The severity of the validation issue which is either informational, warning, or error.
Property	*Message*	Text describing the validation issue.
Property	*Error Number*	An optional number that identifies the validation issue.

30.4.91. ValidateReport

Name	**ValidateReport**
Purpose	Report on a validation result.

Description	This function is used to report some kind of validation problem, for example if a value is not within certain limits. It can be called in any type of expression. The report is done to the results of the execution of the map, which is shown through the GUI or can be accessed using the runtime API.	
Return Type	None	
Argument	*Data (Simple (variable))*	Any number of expressions that data to be associated with the validation report. These are reported as name/value pairs in the validation report. If an expression specified here is a *[map]* element reference, the name is taken to be the name of the *[map]* element and the value is the element's value. Use the **Property** function with other types of expressions to provide the name label in the validation report.
Property	*Severity*	The severity of the validation issue which is either informational, warning, or error.
Property	*Message*	Text describing the validation issue.
Property	*Error Number*	An optional number that identifies the validation issue.

30.4.92. ValueMapping

Name	**ValueMapping**	
Purpose	Specified a mapping of one or more input values to a single output value.	
Description	This function is used to specify a mapping of one or more input values to a single output value and is used in conjunction with the **MapValues** function. For convenience, this function allows you to specify the input or output values either a property or argument. If both are specified, the argument takes precedence.	
Return Type	String	
Property	*Output Value*	The output value mapped from the input values.
Property	*Input Value*	A single input value to be mapped to the output value.
Property	*Is Default Value*	If this is true, this becomes the default value for the element if no other **ValueMapping** functions are satisfied.
Argument	*Output Value (String)*	The output value mapped from the input values.
Argument	*Input Value (String)*	One or more input values to be mapped to the output value.

30.4.93. WriteMapOutput

Name	**WriteMapOutput**
Purpose	Writes this map element to the output result document specified with the map.
Description	The **WriteMapOutput** function may be used only as an I/O expression. Specify it as the I/O expression of the element that contains the data to be written. The values of the element containing the I/O expression and any subordinate elements are written to the result specified in the execution of the map. You may not however nest the use of this function. In other words, if a map element contains an I/O expression, no subordinate map element may contain an I/O expression. See the discussion on Input/Output for further details.
Return Type	N/A (this function may be used only as an I/O expression)

30.4.94. WriteMessage

Name	**WriteMessage**	
Purpose	Writes this map element to an ESB message.	
Description	The **WriteMessage** function may be used only as an I/O expression. Specify it as the I/O expression of the element that contains the data to be written. A message will be sent during the map execution from the values of the element containing the I/O expression and any subordinate elements. *Writing Message Properties* - if you want to write properties into the message, they can be provided from map elements. To do this the first child map element must specify the **WriteMessageProperties** I/O function, inherit from the */Builtin/Structures/Properties* structure, and must be a loop. When the message is written, the properties of the message are populated from the values in this structure. You may use this function in multiple map elements in order to write multiple messages. You may not however nest the use of this function. In other words, if a map element contains an I/O expression, no subordinate map element may contain an I/O expression. See the discussion on Input/Output for further details.	
Return Type	N/A (this function may be used only as an I/O expression)	
Property	*Properties*	The properties to be passed to the message.
Property	*Endpoint*	The ESB endpoint to which the message will be sent. With some ESBs the endpoint is a rather physical configuration dependent setting so the use of the message properties are preferred to allow the message to be routed by the settings in the ESB configuration.
Property	*Write Execution Context Properties*	Check this if you wish to have the map execution context properties added to the message. The default is they are not added. When a map is executed, the map execution context properties include the properties of the inbound ESB message that triggered the map.

30.4.95. WriteMessageProperties

Name	**WriteMessageProperties**
Purpose	Writes the properties in the map elements to an ESB message.
Description	The **WriteMessageProperties** function may be used only as an I/O expression. It is used to write the properties from the map element values into an ESB message. This may be used only in the first child element under the **WriteMessage** function. Also, the element on which this function is specified must have a occurs maximum times of *-1* and must inherit from */Builtin/Structures/Property*.
Return Type	N/A (this function may be used only as an I/O expression)

30.4.96. WriteURL

Name	**WriteURL**
Purpose	Writes a map element to the specified URL.
Description	The **WriteURL** function may be used only as an I/O expression. Specify it as the I/O expression of the element that contains the data to be written. The values of the element containing the I/O expression and any subordinate elements are written to the specified URL. You may use this function in multiple map elements in order to write to multiple URLs. You may not however nest the use of this function. In other words, if a map element

	contains an I/O expression, no subordinate map element may contain an I/O expression. See the discussion on Input/Output for further details. This function can be used to allow writing to multiple documents/objects in the map execution that are assigned dynamically by calling methods on the *MapExecutionContext*. In this case the URL is not required.	
Return Type	N/A (this function may be used only as an I/O expression)	
Property	*URL*	The URL to which to write the data. If you want to write a file, specify a file URL which is: *file:///* followed by the name of the file.

30.4.97. XPathFunction

Name	**XPathFunction**	
Purpose	Call an XPath function.	
Description	This calls an XPath function optionally passing it the each of the arguments of the Expression. To refer to all of the arguments of the expression in the parameter list, specify *%v* . For example, to call the *fn:concat* XPath function with all of the arguments, specify *fn:concat(%v)* in the *Function Name/Args* property. To get a single argument, you may also specify *%an* where n is the number of the argument starting with zero. For an example of this function in use, see the article about *Using a simple XPath function in Talend Data Mapper* on Talend Help Center (https://help.talend.com).	
Return Type	Simple	
Variable Arguments		
Variable Argument Type	Simple	
Property	*Function Name/Args*	The XPath function name to call, along with its arguments.

Appendix A. Compatibility

The runtime compatibility rules are specified here along with the release notes for the current and previous versions.

A.1. Compatibility

This section describes the rules of map or structure compatibility between different releases of *Talend Data Mapper*

A.1.1. Maps/Structures from Previous Versions

Any map or structure created in any previous release of *Talend Data Mapper* can be executed in the current version of *Talend Data Mapper*. Older maps or structures may have been created using different metadata rules, but since all meta-data is versioned any more current release of *Talend Data Mapper* will detect these differences and make the necessary adjustments. If you are using the *Data Mapper* to open an older map or structure, it may appear as unsaved when you open it (a star will appear in the editor's tab), this indicates the map or structure has been altered to bring it up to date with the current metadata format.

A.1.2. Designer and Runtime Compatibility

Care must be taken regarding map and structure compatibility between the designer and runtime since they may be different releases of *Talend Data Mapper*. It is possible to use a newer release of the designer than the runtime and therefore create a map that the runtime will not be able to process. If this happens, the runtime will give a clear error message indicating it could not open the map because of a version mismatch. At this point you will need to use a runtime of the same or later release than the designer that produced the map to execute the map.

As of release 3.1.0, designers and runtimes within the same major and minor version (e.g. 3.1.x) are guaranteed to be compatible. So for example, you could create a map in designer version 3.2.4 and execute it on a 3.2.0 runtime. However you would not necessarily be able to execute the map with a 3.1.4 runtime.

Appendix B. Acknowledgements

We gratefully acknowledge the use of the following open source projects:

- *Eclipse* - This product is built on the Eclipse platform (http://www.eclipse.org/).
- *Apache* - This product includes software developed by the Apache Software Foundation (http://www.apache.org/).
- *DOM4J* - http://www.dom4j.org
- *LegStar* - http://www.legstar.com/ - For COBOL import technology.
- *XPP* - This product includes software developed by the Indiana University Extreme! Lab (http://www.extreme.indiana.edu/) - Used for parsing the internal metadata.
- And especially *Saxon* - http://www.saxonica.com - The Saxon B product was used from the beginning in our product development and was essential for its high quality and rock solid implementation of XQuery.