

# XML import / export filter for jOdra

## *User's Guide*

Krzysztof Kaczmarek

23. November 2006

## 1. Introduction

This document describes general behavior for XML import and export package for jOdra. The latest stable version of the package `odra.filters.XML` is always included in the project's repository.

The package implements more abstract interfaces used by jOdra and placed in `odra.filters` package.

## 2. Integration with CLI and IDE

jOdra is currently prepared to use plugins which are registered in `odra.system.Config` class. XML importing class is registered as `XMLImporter`.

jOdra's text console (CLI) automatically converts query output to XML form. This behavior may be changed by CLI's variable setting:

- `set output default` – sets output to internal jOdra format
- `set output xml` – sets output to XML format

However, jOdra IDE works differently. It presents query results graphically (in tree-like form) and optionally may convert them to XML format. Please, refer to IDE documentation.

### 2.1 Importing XML from Cli command line

General structure of plugin execution command is as follows:

```
load "resource" using plugin-name ("params")
```

or

```
load "resource" using plugin-name
```

where

*resource* is a path to a file

*plugin-name* is the name of a registered plugin

*params* is a list of parameters recognized by the plugin, separated by: “[space] , ; \n \t \r \f”.

For example:

```
load "res/xml/bookstore.xml" using XMLImporter("M0")
```

```
load "res/xml/bookstore.xml" using XMLImporter("M0, noGuessType")
```

## 2.2 XMLImporter parameters

Currently the list of recognized parameters is not very long:

`M0` – do not use annotated object during import procedure. See import procedure description for details.

`noGuessType` – do not perform automatic type guessing. See automatic type guessing for details

## 3. XML import procedure

When an XML document is imported into jOdra object store all information found in XML are converted to appropriate SBA objects using the following algorithm:

1. Tagged element is converted to a complex object. Tag name is used as object's tag.
2. Text inside tagged element is stored in simple type object named `_VALUE`. Type may be guessed, see point 4.
3. Element's attributes are stored in subobjects:
  1. in case of simple import procedure, subobject is a simple type object. Its type may be guessed automatically. Its name is preceded by '@' sign to distinguish objects created from attributes (parameter `M0` in `XMLImporter`). Note that using this method attribute objects are distinguished from normal objects;

Example:

XML	jOdra
<pre>&lt;Text font="Arial"&gt;   Foo. &lt;/Text&gt; &lt;Text&gt;   Boo. &lt;/Text&gt;</pre>	<pre>Text{   @font="Arial"   _VALUE="Foo." } Text{   _VALUE="Boo." }</pre>

2. in case of import using annotated objects, subobject is a complex object containing single simple type object named `_VALUE`. The subobject's name is equal to attribute's name but an appropriate annotation is created (`attribute=true`). In this way attributes are treated in the same way as all other objects. The annotation is the only way to distinguish non attribute objects.

Example:

XML	jOdra
<pre>&lt;Text font="Arial"&gt;   Foo. &lt;/Text&gt; &lt;Text&gt;   Boo. &lt;/Text&gt;</pre>	<pre>Text{   font&lt;attribute="true"&gt;{     _VALUE="Arial"   }   _VALUE="Foo." } Text{   _VALUE="Boo." }</pre>

4. Type guessing. For some purposes (comparing values or selecting minimal value) XML importing procedure tries to guess the type of imported simple type value. If it is a parseable

integer then an integer object is produced. If it is a parseable double then a double object is produced. Otherwise string is produced. Please note that this option does not use any kind of schema. Type guessing may be switched off by “noGuessType” plugin option.

5. Links between elements:

1. if an element Y contains attribute `idref="X"` it is interpreted as a pointer to another element;
2. if appropriate element Z with `id="X"` attribute is found, then element Y is imported as a reference object pointing to object Z. If more than one Z is found, then only the first one is connected (it is generally impossible since id attributes have to be unique);
3. if appropriate element identified by X is not found then Y is created as a complex object containing string object named `idref` with value X.

6. Namespaces:

1. in case of a simple import procedure all namespace declarations and prefixes are omitted;
2. in case of annotated objects import namespaces are converted to annotations objects:
  1. namespace definition is converted to an annotation object:  
`namespaceDef( prefix:String, uri:String )`
  2. a single object may have many `namespaceDef` annotations;
  3. namespace assignment creates a reference annotation `namespaceRef` pointing to an appropriate `namespaceDef` object;
  4. an object may contain only one `namespaceRef` annotation;
  5. if an object is assigned to a namespace it must contain `namespaceRef` annotation, even if it points to its own `namespaceDef`;
  6. attributes may contain only single `namespaceRef` annotation.

## 4. XML export procedure

Object export is currently unavailable from Cli command line, however it may be accessed via programmer's interface using `ObjectExporter` from package `odra.filters.XML`. It requires a destination character stream and object OID. The object is exported with all its content.

There is one general rule for object exporting:

Export must produce XML equivalent to the source XML before import.

Please note that:

1. Export procedure must be chosen according to import procedure. It may use annotations or not. If one will use M0 mode during import then also `M0DefaultExporter` must be used. If annotations are involved then `M0AnnotatedExporter` should be used.
2. Any object may be exported to XML, not necessarily created by XML import procedure.
3. If a reference object is exported, an XML element with `idref` attribute is generated. However, `idref` will point to an object using its `jOdra` OID. Presentation of OID as id attributes for all objects must be turned on to get completely functional export (a parameter in `M0DefaultExporter` constructor).

## 5. XMLResultPrinter

This class included in `odra.filters.XML` package is intended to generate XML upon query results. For more information on `Result` class and its subclasses please go to [jOdra documentation](#).

Results are generally converted to XML in the same way as objects. XML tagged elements are produced by binders. Values are converted to texts. Special object names assumed by import procedures are correctly interpreted:

1. objects with name beginning with '@' or objects annotated with annotation `<attribute=true>` are interpreted as attributes – they must be simple types only. Please note, that so far we have no possibility to create annotation from SBQL, so the only way to create XML attributes is to create objects with @ at the beginning of the name;
2. attribute value is assumed to be of string value;
3. attributes are omitted if an object has no name and no surrounding XML tag may be found. Please, remember that an element in XML may have only one attribute of given name. Results will be unpredicted if you try to add many same attributes to an object;
4. objects with names `_VALUE` are converted to XML nameless text elements;
5. objects' annotations describing namespaces and namespace prefixes are converted to appropriate constructs in XML;
6. the XML output will contain `RESULT` tag as a root for the document, if there is no single named object, which could be document's root.

Examples:

SBQL query in jOdra	Results in XML
<code>(10, 20 as r, 30) as k;</code>	<pre>&lt;k&gt;10   &lt;r&gt;20&lt;/r&gt;30 &lt;/k&gt;</pre>
<code>("10" as @myattr, "text") as res;</code>	<pre>&lt;res myattr="10"&gt;text&lt;/res&gt;</pre>
<code>("12" as @k, "13", "14") as X;</code>	<pre>&lt;X k="12"&gt;13   14&lt;/X&gt;</pre>
<code>("12" as @k, 13, 14);</code>	<pre>&lt;RESULT&gt;13   14&lt;/RESULT&gt;</pre>
<code>("VAL1" as @KK, "VAL2" as @KK) as FOO;</code>	<pre>&lt;FOO KK="VAL2"/&gt;</pre>
<code>("") as EMPTY;</code>	<pre>&lt;/EMPTY&gt;</pre>