



XML in JADE

Version 2018

JADE Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of JADE Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2019 JADE Software Corporation Limited.

All rights reserved.

JADE is a trademark of JADE Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

Contents

| | |
|---|----------|
| XML in JADE | 4 |
| Introduction | 4 |
| Structure of XML | 4 |
| Creating XML | 5 |
| Exercise 1 – Creating an XML Document | 8 |
| Exercise 2 – Converting a JADE Database to XML | 9 |
| Loading XML | 10 |
| Handling JADE XML Exceptions | 11 |
| Searching JadeXMLDocuments | 14 |
| Exercise 3 – Handling XML Exceptions..... | 16 |
| Exercise 4 – Populating a Database from an XML File..... | 18 |
| Persistent XML | 22 |
| Exercise 5 – Creating Persistent JadeXMLNode Subclasses | 24 |
| Exercise 6 – Parsing an XML Document Persistently..... | 25 |

XML in JADE

Introduction

XML, or eXtensible Markup Language, is a human-readable, text-based language that consists of *nodes*, containing data, and *tags*, describing the type of the data between the tags.

Although XML can be used for a variety of applications, the most common is transferring data over the internet. JADE provides several RootSchema classes for the conversion of data between JADE database classes and XML documents.

Structure of XML

XML has a similar structure to other markup languages (for example, HTML) in that an XML document consists of data contained within tags, potentially nested. As with other markup languages, tags are of the form **<TagName> data </TagName>**. However, unlike many other markup languages, XML allows custom tags.

Despite the flexibility to markup data with whatever term best describes it, there are still rules as to how the document must be laid out. For example, an XML document must have a 1:1 pairing between opening and closing tags and nesting integrity must be maintained.

When all XML rules are followed, the document is deemed to be *well-formed*. Being well-formed means the XML document is syntactically correct according to the World Wide Web Consortium (W3C) XML Specification. It does not, however, guarantee that the data is correct.

The following XML document is well-formed.

```
1  <?xml version="1.0"?>
2  <!--This is a comment -->
3  <company>
4    <employee id="jtm314">
5      <name>Jack Mason</name>
6      <role>Abstraction Instantiator</role>
7      <salary>99550</salary>
8    </employee>
9
10   <customer id="CUST00349">
11     <name>Lee Sah</name>
12     <address>42 Fiction Ave</address>
13     <email>ls99@e.mail</email>
14   </customer>
15 </company>
```

The following XML document is *not* well-formed, because the **employee** opening tag does not have a closing tag.

```
1 <?xml version="1.0"?>
2 <!--This is a comment -->
3 <company>
4   <employee id="jtm314">
5     <name>Jack Mason</name>
6     <role>Abstraction Instantiator</role>
7     <salary>99550</salary>
8
9
10  <customer id="CUST00349">
11    <name>Lee Sah</name>
12    <address>42 Fiction Ave</address>
13    <email>ls99@e.mail</email>
14  </customer>
15 </company>
```

The following XML document is *also* not well-formed, because the **customer** opening tag is nested inside the **company** tag, but the closing tag lies outside it.

```
1 <?xml version="1.0"?>
2 <!--This is a comment -->
3 <company>
4   <employee id="jtm314">
5     <name>Jack Mason</name>
6     <role>Abstraction Instantiator</role>
7     <salary>99550</salary>
8   </employee>
9
10  <customer id="CUST00349">
11    <name>Lee Sah</name>
12    <address>42 Fiction Ave</address>
13    <email>ls99@e.mail</email>
14 </company>
15 </customer>
```

Note Although the indentation is irrelevant to whether the XML is well-formed, having good indentation practices makes XML documents easier to read by humans. When JADE generates XML documents, it handles the indentation for you.

Creating XML

JADE provides the **JadeXMLDocument** and **JadeXMLElement** classes for generating and presenting well-formed XML.

The **JadeXMLDocument** class represents an XML document as a tree of nodes and defines the root; that is, the owning object of all objects in the tree.

The **JadeXMLElement** class represents an XML element in a document tree and along with a unique name, can contain any of the following.

- Child nodes; that is, other **JadeXMLElement** objects
- Attributes; for example, `id="CUST00349"`

- Text; for example, **Lee Sah** in a **name** node

To add **JadeXMLElement** nodes to the main document (a **JadeXMLDocument** object) or another **JadeXMLElement** node, both classes provide the **addElement** method. This method takes a **tagName** parameter of type **String**, which represents the name of the tag.

To add attributes to a node, the **JadeXMLElement** class provides the **addAttribute** method, which takes two parameters of type **String**. The **name** parameter represents the name of the attribute (for example, **id**) and the **value** parameter represents the value of the attribute (for example, **rs9312**).

Text can be added to a node by using the **JadeXMLElement** class **setText** method, which takes a single **String** parameter for the text to add to the node.

The following example show the method used to create an XML document.

```
exampleXML();

vars
    xmlDoc      : JadeXMLDocument;
    xmlNode1    : JadeXMLElement;
    xmlNode2    : JadeXMLElement;
begin
    // xmlDoc represents the whole XML document,
    // and the root of the tree.
    create xmlDoc transient;

    // xmlNode1 represents a <Person> tag in the xml document,
    // and a node in the tree.
    xmlNode1 := xmlDoc.addElement("Person");

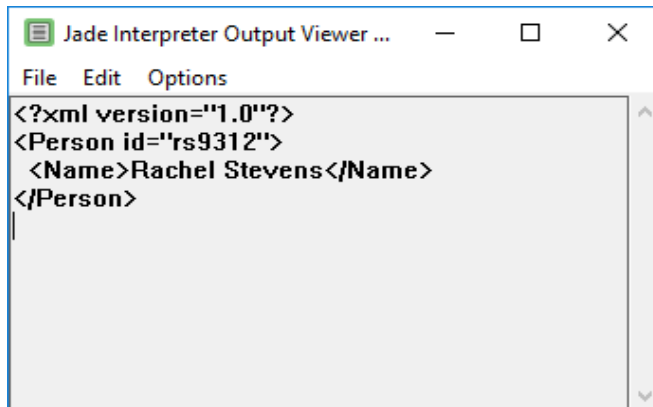
    // xmlNode1 has one attribute, which will
    // look like <Person id="rs9312"> in the xml.
    xmlNode1.addAttribute("id", "rs9312");

    // xmlNode2 represents a <Name> tag within the xml,
    // and is a child of <Person>
    xmlNode2 := xmlNode1.addElement("Name");

    // The text inside the <Name> tag will be
    // Rachel Stevens
    xmlNode2.setText("Rachel Stevens");

    // Write the XML document to the output window.
    // Can also save this to a file (usually more useful).
    write xmlDoc.writeToString();
epilog
    delete xmlDoc;
end;
```

The following example shows the output of the above method used to create an XML document.



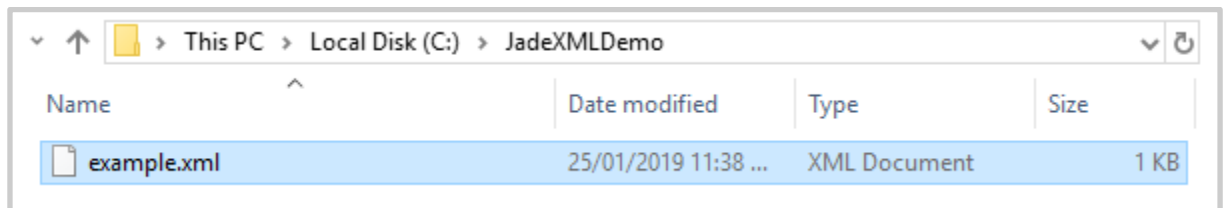
```
<?xml version="1.0"?>
<Person id="rs9312">
  <Name>Rachel Stevens</Name>
</Person>
```

The **JadeXMLDocument** class also provides the **writeToFile** method, which takes a **filePath** as a parameter of type **String**, and outputs the XML document to the specified file. Note that if there is no file at the **filePath** location, it creates one, and if the file exists, the contents of that file are overwritten following confirmation.

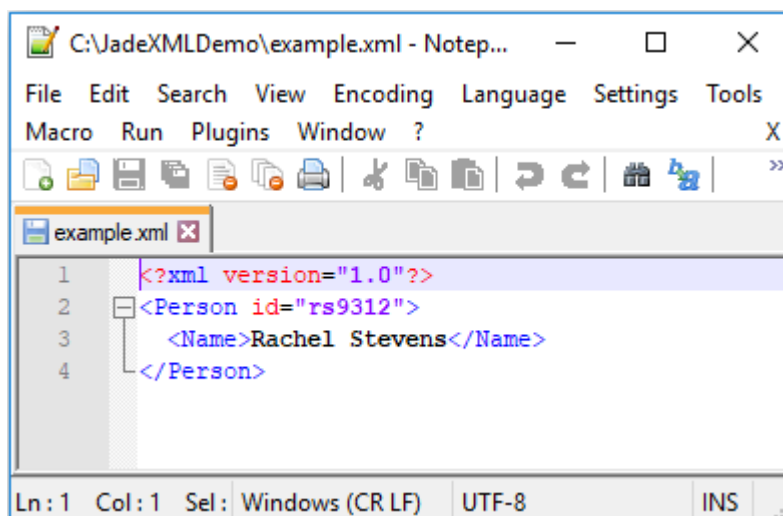
The following code fragment changes from **writeToString** to **writeToFile**.

```
// Writing the XML to a file instead...
xmlDoc.writeToFile("C:\\JadeXMLDemo\\example.xml");
```

The above change results in the creation of an XML file at the file path in the following example.



This file path location contains the XML document shown in the following example.



```
1 <?xml version="1.0"?>
2 <Person id="rs9312">
3   <Name>Rachel Stevens</Name>
4 </Person>
```

Exercise 1 – Creating an XML Document

In this exercise, you will create an XML document, add nodes to form a tree structure, and then write the XML document to a file.

1. Create a new schema called **XMLSchema**.
2. Create a new JadeScript method called **createXML** and code it as follows.

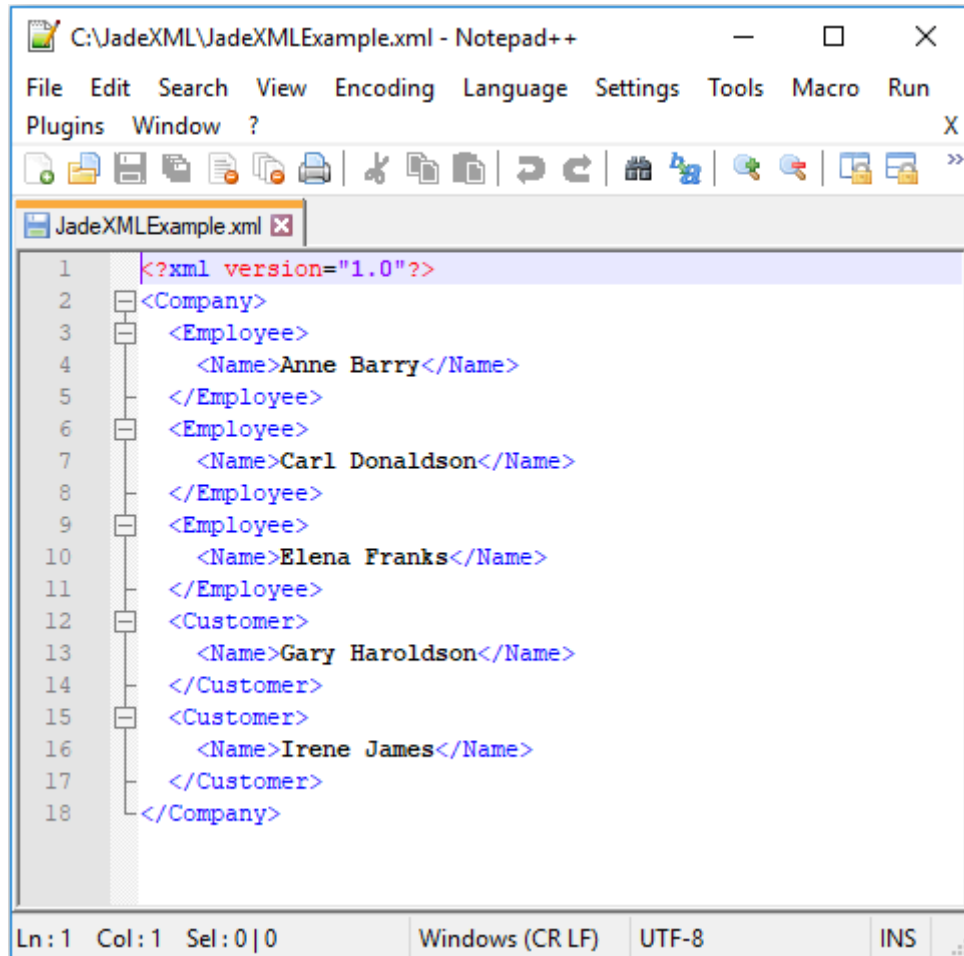
```
createXML();

vars
  xmlDoc      : JadeXMLDocument;
  xmlCompany  : JadeXMLElement;
  xmlEmployee : JadeXMLElement;
  xmlCustomer : JadeXMLElement;
  xmlData     : JadeXMLElement;
begin
  create xmlDoc transient;

  xmlCompany := xmlDoc.addElement("Company");
  xmlEmployee := xmlCompany.addElement("Employee");
  xmlData := xmlEmployee.addElement("Name");
  xmlData.setText("Anne Barry");
  xmlEmployee := xmlCompany.addElement("Employee");
  xmlData := xmlEmployee.addElement("Name");
  xmlData.setText("Carl Donaldson");
  xmlEmployee := xmlCompany.addElement("Employee");
  xmlData := xmlEmployee.addElement("Name");
  xmlData.setText("Elena Franks");
  xmlCustomer := xmlCompany.addElement("Customer");
  xmlData := xmlCustomer.addElement("Name");
  xmlData.setText("Gary Haroldson");
  xmlCustomer := xmlCompany.addElement("Customer");
  xmlData := xmlCustomer.addElement("Name");
  xmlData.setText("Irene James");
  xmlDoc.writeFile("C:\JadeXML\JadeXMLExample.xml");
epilog
  delete xmlDoc;
end;
```

3. On your file system, create a new folder on your **C:** drive called **JadeXML**.
4. Execute (run) the JadeScript **createXML** method. The file **JadeXMLExample.xml** should be generated in your **JadeXML** folder.
5. Inspect the created file (using Notepad or Notepad++).

The following XML should be displayed.



```
1 <?xml version="1.0"?>
2 <Company>
3   <Employee>
4     <Name>Anne Barry</Name>
5   </Employee>
6   <Employee>
7     <Name>Carl Donaldson</Name>
8   </Employee>
9   <Employee>
10    <Name>Elena Franks</Name>
11  </Employee>
12  <Customer>
13    <Name>Gary Haroldson</Name>
14  </Customer>
15  <Customer>
16    <Name>Irene James</Name>
17  </Customer>
18 </Company>
```

Exercise 2 – Converting a JADE Database to XML

In this exercise, you will convert JADE objects into XML nodes for a small database.

1. Select the **Load** command from the Schema menu and then load the provided **XMLExampleSchema.scm**.
2. Copy the provided **customers.txt** and **employees.txt** files to the **C:\JadeXML** folder you created in the previous exercise in this module.
3. From **XMLExampleSchema**, run the JadeScript **loadAll** method, which populates the database with several employees and customers.

4. Create a new JadeScript **buildXML** method and code it as follows.

```
buildXML();

vars
  xmlDocument : JadeXMLDocument;
  xmlCompany  : JadeXMLElement;
  xmlCustomer : JadeXMLElement;
  xmlEmployee : JadeXMLElement;
  xmlProperty : JadeXMLElement;
  company     : Company;
  customer    : Customer;
  employee    : Employee;
begin
  create xmlDocument transient;
  company := Company.firstInstance();

  xmlCompany := xmlDocument.addElement("Company");

  foreach customer in company.allMyCustomers do
    xmlCustomer := xmlCompany.addElement("Customer");
    xmlCustomer.addAttribute("id", customer.id);
    xmlProperty := xmlCustomer.addElement("name");
    xmlProperty.setText(customer.name);
  endforeach;

  foreach employee in company.allMyEmployees do
    xmlEmployee := xmlCompany.addElement("Employee");
    xmlEmployee.addAttribute("id", employee.id);
    xmlProperty := xmlEmployee.addElement("name");
    xmlProperty.setText(employee.name);
    xmlProperty := xmlEmployee.addElement("salary");
    xmlProperty.setText(employee.salary.String);
  endforeach;

  xmlDocument.writeFile("C:\JadeXML\JadeCompanyXML.xml");

epilog
  delete xmlDocument;
end;
```

5. Press F9 to execute the method and then inspect the **C:\JadeXML\JadeCompanyXML.xml** file to view the contents of the database in XML format.

Loading XML

In addition to generating XML documents, the **JadeXMLDocument** can also load existing XML documents, creating a tree structure suitable for traversing and extracting data.

The **JadeXMLDocument** class provides the **parseFile** and **parseString** methods, which essentially mirror the **writeToFile** and **writeToString** methods. The difference between the **writeToFile** and **parseFile** methods is that **parseFile** creates a new **JadeXMLDocument** based on an existing XML file, while **writeToFile** takes an existing **JadeXMLDocument** and creates a new XML file.

To use either method, you need only create the transient **JadeXMLDocument** and call the appropriate parse method; for example, to read an XML file using the **parseFile** method needs only the following.

```
parseXML();

vars
    xmlDocument : JadeXMLDocument;
begin
    create xmlDocument transient;
    xmlDocument.parseFile("C:\JadeXML\JadeCompanyXML.xml");
epilog
    delete xmlDocument;
end;
```

Handling JADE XML Exceptions

Any XML document generated using JADE is always be well-formed XML. However, when parsing XML from external sources, there is a chance that the XML being parsed may not be well-formed.

If the XML being parsed is not well-formed, the **parseFile** method raises a **JadeXMLException**.

```
parseXML();

vars
    xmlDocument : JadeXMLDocument;
begin
    create xmlDocument transient;
    xmlDocument.parseFile("C:\JadeXML\DodgyXML.xml");
epilog
    delete xmlDocument;
end;
```

The following is an example of the exception raised by the above method.

J Unhandled Exception on 2019/01/28 11:56:11 by [187.72] pid...

Description

| | |
|--------------|------------------|
| Application: | XMLExampleSchema |
| Schema: | XMLExampleSchema |
| Type: | JadeXMLException |
| Error Code: | 8901 |
| Continuable: | No |
| Error Item: | Customer> |

XML parser error
mismatched tag

Caused By

| | | |
|----------------|----------------------------|---------|
| Receiver Type: | JadeXMLDocumentParser | Inspect |
| Receiver OID: | 719.1 (transient) | |
| Ext Method: | JadeXMLParser::parseString | |

'jadexpatParseString' in 'jadexpat' @ line# 582

Reported By

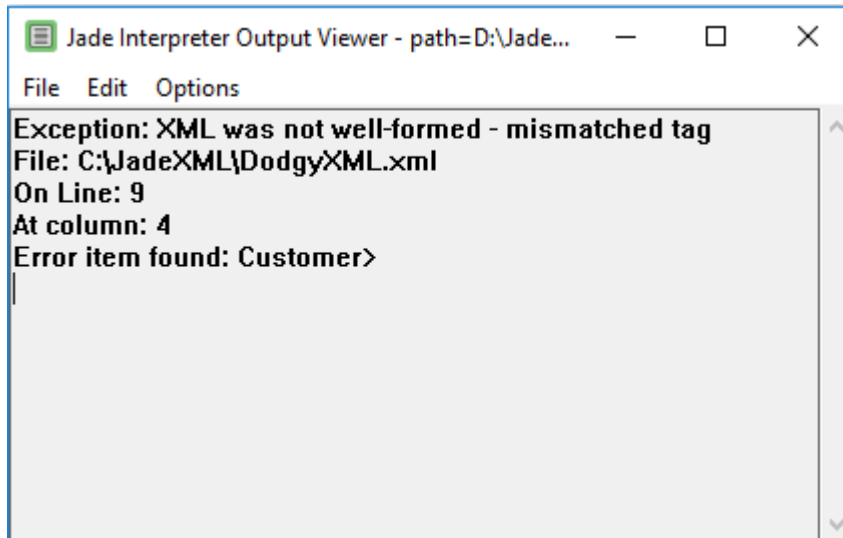
| | | |
|--|--|---------|
| | | Inspect |
| | | |
| | | |

Abort Ignore Debug Help

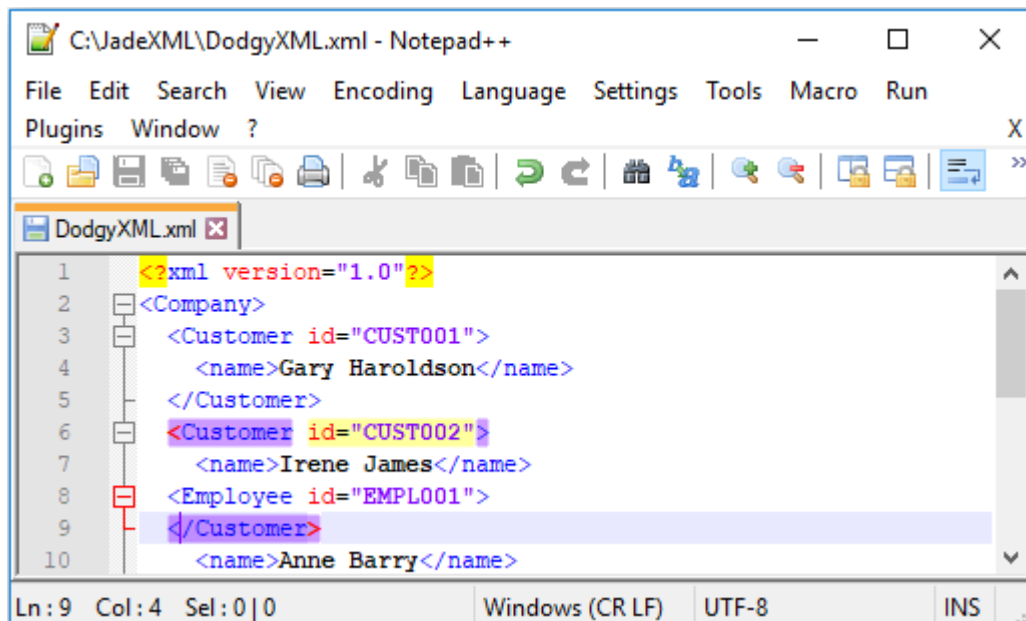
The generated **JadeXMLException** contains the line and column number of the error, as well as the type of error and the item in error. As such, an exception handler can display the following.

```
xmlExceptionHandler(e : JadeXMLException) : Integer;
begin
  write "Exception: XML was not well-formed - " & e.extendedErrorText;
  write "File: " & e.fileName;
  write "On Line: " & e.lineNumber.String;
  write "At column: " & e.columnNumber.String;
  write "Error item found: " & e.errorItem;
  return Ex_Abort_Action;
end;
```

The above method writes the following to the Jade Interpreter Output Viewer.



This allows for the quick and easy identification of where exactly, and what exactly, is the problem in the XML source. In this example, the **Customer** tag is mismatched, as the **Employee** tag was opened but not closed when the **Customer** tag was closed.



Searching JadeXMLDocuments

Once an XML document is successfully loaded and a **JadeXMLDocument** created, the following methods can be used to locate specific elements in the tree.

| Method | Purpose |
|------------------------------------|--|
| <code>getElementByTagName</code> | Takes a tagName parameter of type String and returns the first JadeXMLElement that has a matching tagName property. |
| <code>findElementByTagName</code> | Takes a tagName parameter of type String and returns the first JadeXMLElement that has a matching tagName property. Has a faster performance than getElementByTagName but may not return the document's first instance when there is more than one element of that tagName . |
| <code>getElementsByTagName</code> | Takes a tagName parameter of type String and an elements parameter of type JadeXMLElementArray and populates the specified elements array (in document order) with all JadeXMLElements that have a matching tagName property. |
| <code>findElementsByTagName</code> | Takes a tagName parameter of type String and an elements parameter of type JadeXMLElementArray and populates the specified elements array with all JadeXMLElements that have a matching tagName property. Has a faster performance than the getElementsByTagName method but the elements are not guaranteed to be sorted by document order. |

The **findElementByTagName** and **getElementByTagName** methods are most useful for locating elements for which you know there will be no other elements that share a tag; for example, a root object such as **Company**.

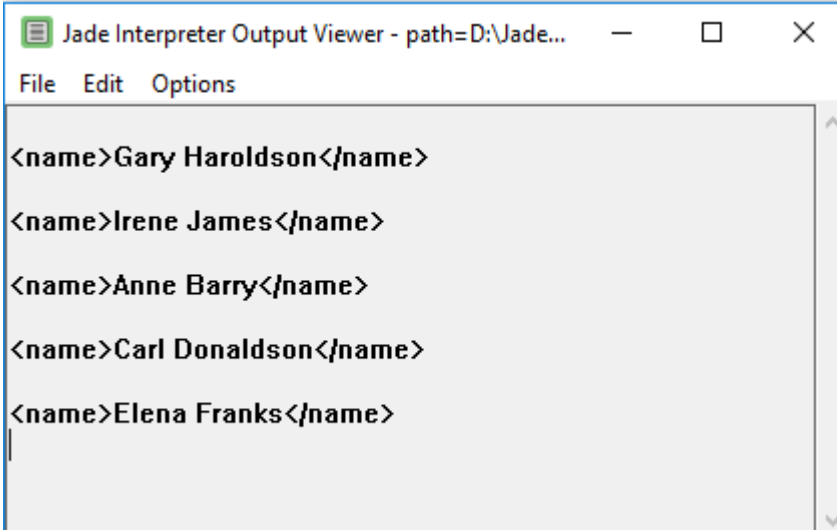
For finding elements of which there is expected to be many, the **getElementsByTagName** and **findElementsByTagName** methods are more appropriate, as the populated arrays can be iterated to display or use the data contained in all elements.

For example, to display all name elements (whether **Customer** or **Employee**) in the **JadeCompanyXML.xml** file, the following JadeScript method can be used.

```
displayAllNamesExample();

vars
  xmlDocument : JadeXMLDocument;
  allNames    : JadeXMLElementArray;
  name        : JadeXMLElement;
begin
  create xmlDocument transient;
  create allNames transient;
  on JadeXMLException do xmlExceptionHandler(exception);
  xmlDocument.parseFile("C:\JadeXML\JadeCompanyXML.xml");
  xmlDocument.getElementsByTagName("name", allNames);
  foreach name in allNames do
    write name.writeToString();
  endforeach;
epilog
  delete xmlDocument;
  delete allNames;
end;
```

This method produces the following output.



The screenshot shows a window titled "Jade Interpreter Output Viewer - path=D:\Jade...". The window contains a text area with the following XML output:

```
<name>Gary Haroldson</name>
<name>Irene James</name>
<name>Anne Barry</name>
<name>Carl Donaldson</name>
<name>Elena Franks</name>
```

For a **JadeXMLElement** object, the following methods are available to extract data from the element.

| Method | Purpose |
|--------------------------|---|
| getAttributeByName | Takes a name parameter of type String and returns the JadeXMLAttribute of the element with that name. |
| getElementByTagName | Takes a tagName parameter of type String and returns the first immediate child JadeXMLElement with that tagName . |
| getElementsByTagName | Takes a tagName parameter of type String and an elements parameter of type JadeXMLElementArray and populates the specified elements array with all immediate child elements with a matching tagName . |
| findAllElementsByTagName | Similar to the getElementsByTagName method except that it populates the specified elements parameter with <i>all</i> child elements that match the tagName ; not just the immediate children. |

Exercise 3 – Handling XML Exceptions

In this exercise, you will load an XML document that is not well-formed and use the **JadeXMLException** object to find and fix the error.

1. Ensure that the **JadeCompanyMalformed.xml** file has been copied to the **C:\JadeXML** folder.
2. Add a JadeScript method called **xmlExceptionHandler**, coded as follows, to the **XMLExampleSchema**.

```
xmlExceptionHandler(e : JadeXMLException) : Integer;

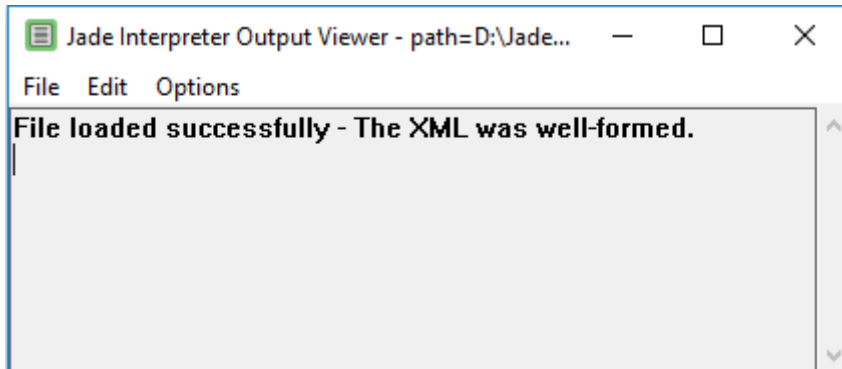
begin
    write "The XML was malformed on line "
        & e.lineNumber.String
        & " at position "
        & e.columnNumber.String;
    write "Error item found: " & e.errorItem;
    return Ex_Abort_Action;
end;
```

3. Add a JadeScript method called **loadMalformedXML**, coded as follows.

```
loadMalformedXML();

vars
    xmlDocument : JadeXMLDocument;
    foundElement : JadeXMLElement;
begin
    create xmlDocument transient;
    on JadeXMLException do xmlExceptionHandler(exception);
    xmlDocument.parseFile("C:\JadeXML\JadeCompanyMalformed.xml");
    write "File loaded successfully - The XML was well-formed.";
epilog
    delete xmlDocument;
end;
```


4. Run the method. Note the line and position of the error, output to the Jade Interpreter Output Viewer. The error item is also displayed.
5. Open the **JadeCompanyMalformed.xml** file in a text editor perform one of the following actions.
 - Navigate to the line and position of the error and correct the typographical error in the tag name.
 - Search for the misspelled tag name shown in the error item and then correct it.
6. Rerun the **loadMalformedXML** method. This time it should load without errors and the following should be displayed in the Jade Interpreter Output Viewer.



Exercise 4 – Populating a Database from an XML File

In this exercise, you will populate the database with the **Customers** and **Employees** in the now well-formed **loadMalformedXML.xml** file.

1. Add a JadeScript method called **createCustomers**, coded as follows, to **XMLExampleSchema**.

```
createCustomers(company : Company input; xmlDoc : JadeXMLDocument);

vars
  allCustomerElements : JadeXMLElementArray;
  foundElement : JadeXMLElement;
  nameElement : JadeXMLElement;
  idAttribute : JadeXMLAttribute;
  customer : Customer;
begin
  create allCustomerElements transient;
  xmlDoc.findElementsByTagName("Customer", allCustomerElements);
  foreach foundElement in allCustomerElements do
    nameElement := foundElement.getElementByTagName("name");
    idAttribute := foundElement.getAttributeByName("id");
    beginTransaction;
    create customer persistent;
    customer.id := idAttribute.value;
    customer.name := nameElement.textData;
    company.allMyCustomers.add(customer);
    commitTransaction;
  endforeach;
epilog
  delete allCustomerElements;
end;
```

2. Add a new JadeScript method called **createEmployees**, coded as follows.

```
createEmployees(company : Company input; xmlDoc : JadeXMLDocument);

vars
  allEmployeeElements : JadeXMLElementArray;
  foundElement : JadeXMLElement;
  nameElement : JadeXMLElement;
  idAttribute : JadeXMLAttribute;
  salaryElement : JadeXMLElement;
  employee : Employee;
begin
  create allEmployeeElements transient;
  xmlDoc.findElementsByTagName("Employee", allEmployeeElements);
  foreach foundElement in allEmployeeElements do
    nameElement := foundElement.getElementByTagName("name");
    idAttribute := foundElement.getAttributeByName("id");
    salaryElement := foundElement.getElementByTagName("salary");
    beginTransaction;
    create employee persistent;
    employee.id := idAttribute.value;
    employee.name := nameElement.textData;
    employee.salary := salaryElement.textData.Integer;
    company.allMyEmployees.add(employee);
    commitTransaction;
  endforeach;
epilog
  delete allEmployeeElements;
end;
```

3. Modify the **loadMalformedXML** method as follows.

```
loadMalformedXML();

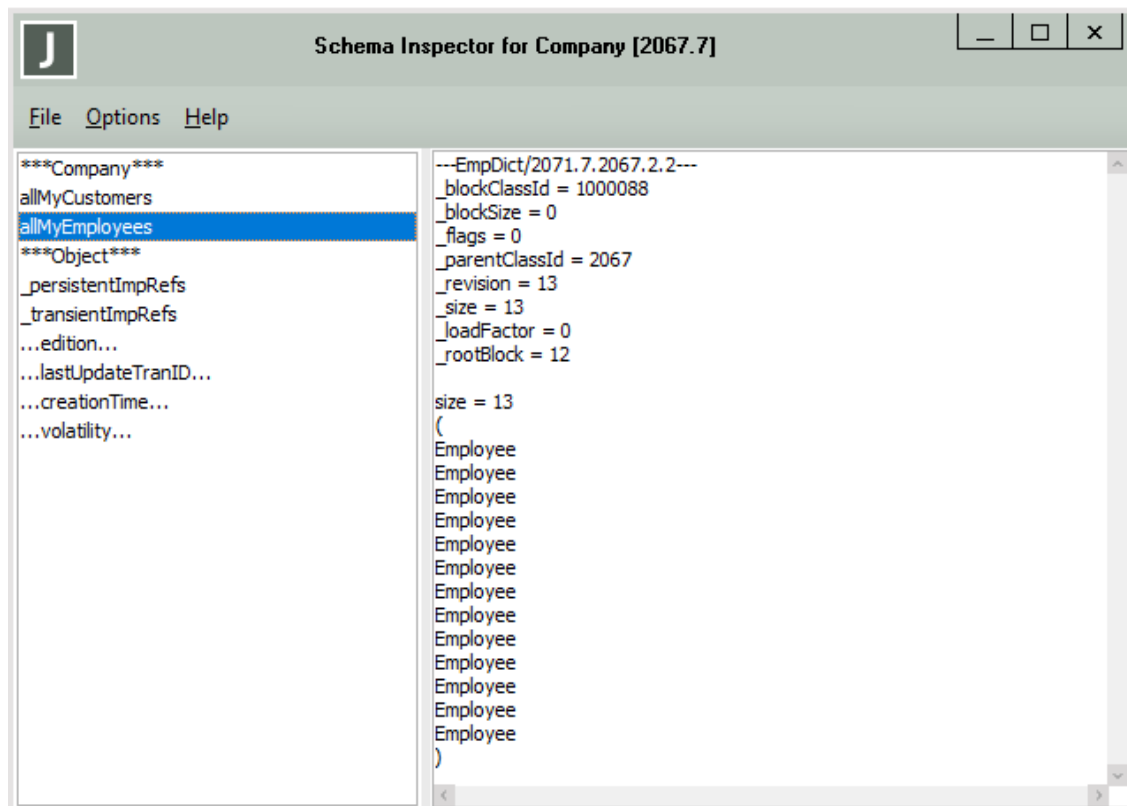
vars
    xmlDocument      : JadeXMLDocument;
    company          : Company;
begin
    create xmlDocument transient;
    on JadeXMLException do xmlExceptionHandler(exception);
    xmlDocument.parseFile("C:\JadeXML\JadeCompanyMalformed.xml");
    write "File loaded successfully - The XML was well-formed.";

    beginTransaction;
    Company.instances.purge();
    Customer.instances.purge();
    Employee.instances.purge();
    create company persistent;
    commitTransaction;

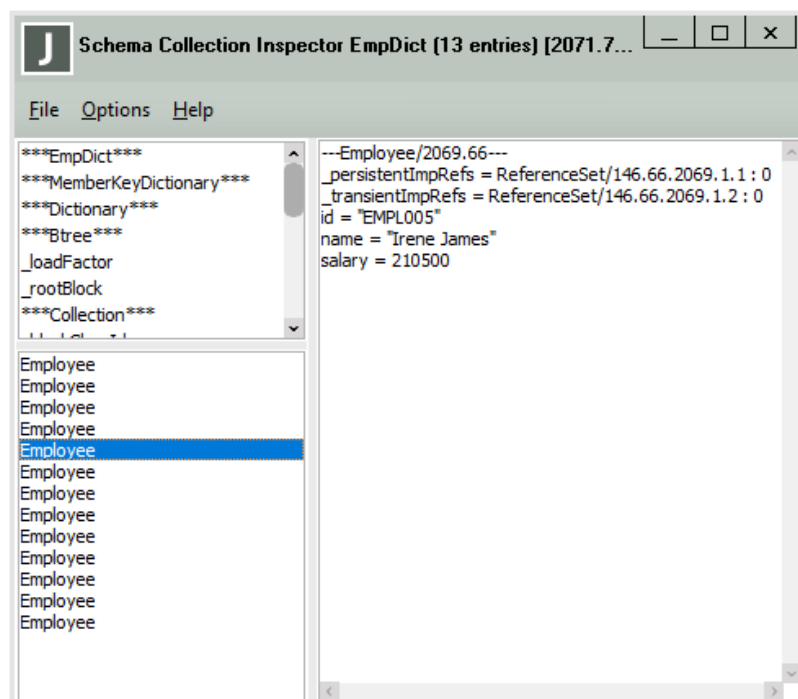
    createCustomers(company, xmlDocument);
    createEmployees(company, xmlDocument);
epilog
    delete xmlDocument;
end;
```

4. Run the **loadMalformedXML** method and then inspect the **Company** object by selecting **Company** in the Class Browser and using the CTRL+I shortcut keys.
5. Double-click the **Company** in the Schema Collection Inspector form to display the Schema Inspector for Company form and then inspect the **allMyCustomers** and **allMyEmployees** collections.

There should be 12 **Customers** and 13 **Employees**.



6. Double-click the **allMyEmployees** collection in the Schema Inspector form to display the Schema Collection Inspector for allMyEmployees form.
7. Click on each **Employee** to verify that the details match those in the **JadeCompanyMalformed.xml** file.



The following is the **JadeCompanyMalformed.xml** file.

```

47 <Employee id="EMPL003">
48   <name>Elena Franks</name>
49   <salary>56500</salary>
50 </Employee>
51 <Employee id="EMPL004">
52   <name>Greg Harriet</name>
53   <salary>12900</salary>
54 </Employee>
55 <Employee id="EMPL005">
56   <name>Irene James</name>
57   <salary>210500</salary>
58 </Employee>
59 <Employee id="EMPL006">
60   <name>Kid Leon</name>
61   <salary>88900</salary>
62 </Employee>
63 <Employee id="EMPL007">

```

- Do the same for each **Customer** in the **allMyCustomers** collection.

Persistent XML

When parsing an XML document in JADE, the **JadeXMLDocument** object and the nodes created with its **parseFile** and **parseString** methods are transient by default, which means that the XML must be parsed again each time it is to be used.

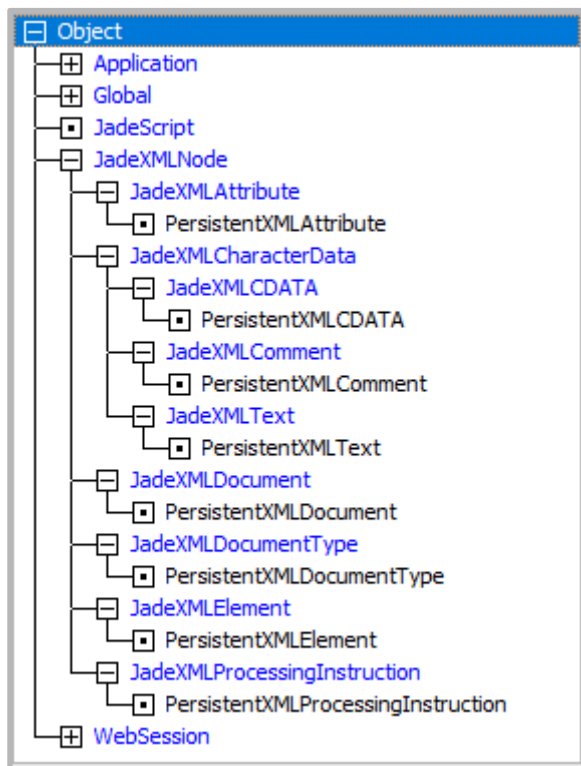
However, JADE provides the **JadeXMLDocumentParser** class that can create persistent implementations of the **JadeXMLDocument** tree structure, using its **parseDocumentString** and **parseDocumentFile** methods.

To create an XML tree structure in JADE, the following node classes are required, all of which have transient-only persistence.

| Class | Represents... |
|---------------------|---|
| JadeXMLAttribute | An attribute of an XML element. |
| JadeXMLCDATA | An XML escape character (for example, a < character that is not part of a tag). |
| JadeXMLComment | An XML comment. |
| JadeXMLDocument | The XML document itself. |
| JadeXMLDocumentType | The document type declaration in an XML document tree. |
| JadeXMLElement | An element in an XML document; for example, <name></name>. |

| Class | Represents... |
|------------------------------|---|
| JadeXMLProcessingInstruction | An XML processing instruction (that is, an application-specific instruction on how to handle an XML document after the document has been parsed). |
| JadeXMLText | The text contained within an XML element when that element contains both text and other elements. |

While the classes in the above table are transient only, persistent subclasses are allowed. To allow for the creation of persistent **JadeXMLDocuments**, you must first create persistent subclasses of each of these transient node classes. For example, the following class hierarchy shows a user subclass for each of the required classes.



When the persistent subclasses are established, the **JadeXMLDocumentParser** provides the **setClassMapping** method, which takes two parameters: a node class and the persistent user subclass to map to.

It also provides the **parseDocumentString** method, which takes a **JadeXMLDocument** (which can be transient, or a persistent subclass) and an XML string. The **parseDocumentFile** method is the same as the **parseDocumentString** method except for a file path string instead of an XML string.

For example, the following code will establish all required mappings, then create a persistent **JadeXMLDocument** using the **parseDocumentString** method of **JadeXMLDocumentParser**.

```
generatePersistentXMLExample ();

vars
    xmlParser    : JadeXMLDocumentParser;
    xmlDocument  : PersistentXMLDocument;
    xmlString    : String;
begin
    create xmlParser transient;
    xmlParser.setClassMapping(JadeXMLAttribute, PersistentXMLAttribute);
    xmlParser.setClassMapping(JadeXMLCDATA, PersistentXMLCDATA);
    xmlParser.setClassMapping(JadeXMLComment, PersistentXMLComment);
    xmlParser.setClassMapping(JadeXMLText, PersistentXMLText);
    xmlParser.setClassMapping(JadeXMLDocumentType, PersistentXMLDocumentType);
    xmlParser.setClassMapping(JadeXMLElement, PersistentXMLElement);
    xmlParser.setClassMapping(JadeXMLProcessingInstruction, PersistentXMLProcessingInstruction);

    xmlString := "<SomeTag>Some Data</SomeTag>";

    beginTransaction;
    create xmlDocument persistent;
    xmlParser.parseDocumentString(xmlDocument, xmlString);
    commitTransaction;
end;
```

Note All mappings must be set, regardless of whether they are needed for the specific XML being parsed. The **parseDocumentString** and **parseDocumentFile** methods will generate an 8909 exception (*XML class mapping is invalid*) if any are missing. However, the **JadeXMLDocument** subclass does not need to be mapped, as it is passed as a parameter to the method.

Exercise 5 – Creating Persistent JadeXMLNode Subclasses

In this exercise, you will create persistent subclasses for each of the required nodes in a persistent **JadeXMLDocument**.

1. Create a new schema called **PersistentXMLSchema**.
2. Open the **PersistentXMLSchema** in the Class Browser.
3. With focus on the **PersistentXMLSchema** class, press the F4 shortcut key to display the Find Type dialog.
4. Search for **JadeXMLAttribute** and then click the **Current Browser** button (or press Enter) to add it to the displayed classes in the Class Browser.
5. Add a subclass to **JadeXMLAttribute** called **PersistentXMLAttribute**.
6. Search for each of the following classes and then add the corresponding persistent subclass.

| Class | Subclass |
|---------------------|---------------------------|
| JadeXMLCDATA | PersistentXMLCDATA |
| JadeXMLComment | PersistentXMLComment |
| JadeXMLDocument | PersistentXMLDocument |
| JadeXMLDocumentType | PersistentXMLDocumentType |

| Class | Subclass |
|------------------------------|------------------------------------|
| JadeXMLElement | PersistentXMLElement |
| JadeXMLProcessingInstruction | PersistentXMLProcessingInstruction |
| JadeXMLText | PersistentXMLText |

Exercise 6 – Parsing an XML Document Persistently

In this exercise, you will use the **JadeXMLDocumentParser** to load an XML file and create a persistent XML tree and then inspect that XML tree using the Schema Inspector form.

1. Search for the **JadeXMLDocumentParser** class using the Find Type dialog.
2. Add a method called **establishMappings** to the **JadeXMLDocumentParser** class and code it as follows.

```

establishMappings() updating;

begin
  self.setClassMapping(JadeXMLAttribute, PersistentXMLAttribute);
  self.setClassMapping(JadeXMLCDATA, PersistentXMLCDATA);
  self.setClassMapping(JadeXMLComment, PersistentXMLComment);
  self.setClassMapping(JadeXMLText, PersistentXMLText);
  self.setClassMapping(JadeXMLDocumentType, PersistentXMLDocumentType);
  self.setClassMapping(JadeXMLElement, PersistentXMLElement);
  self.setClassMapping(JadeXMLProcessingInstruction, PersistentXMLProcessingInstruction);
end;

```

3. Create a JadeScript method called **loadPersistentXML** and code it as follows.

```

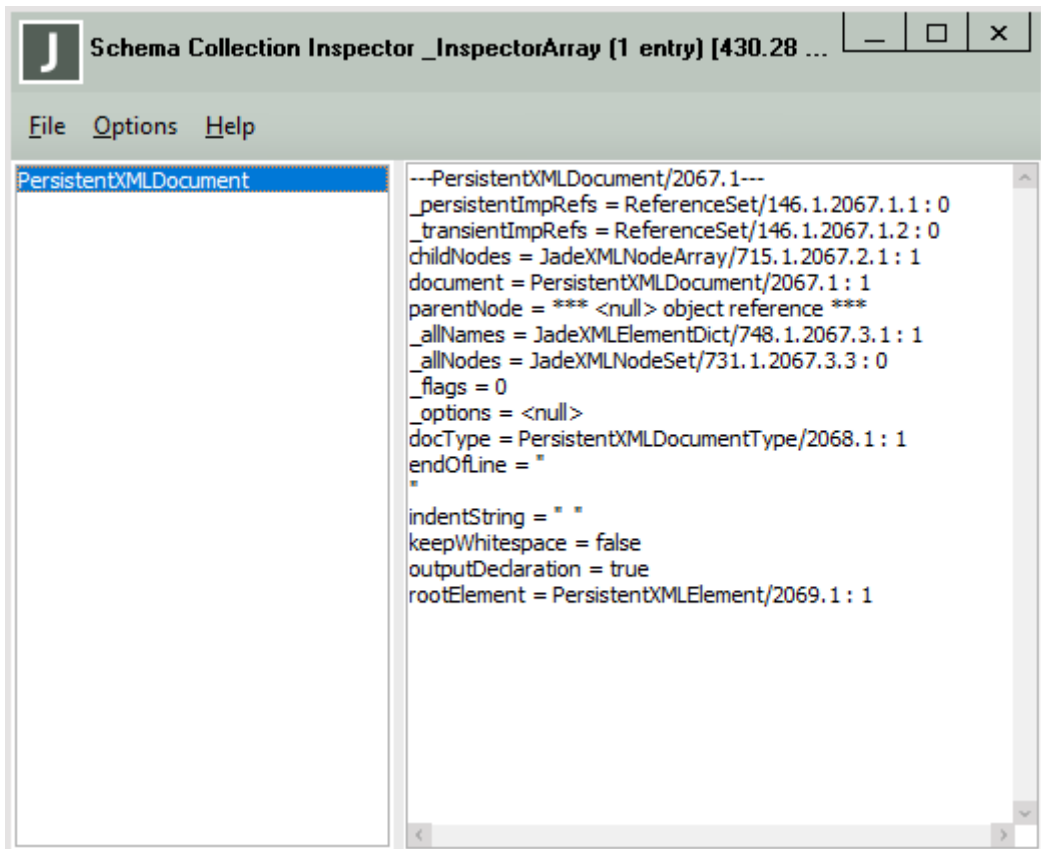
loadPersistentXML();

vars
  xmlParser : JadeXMLDocumentParser;
  xmlDocument : PersistentXMLDocument;
begin
  create xmlParser transient;
  xmlParser.establishMappings();

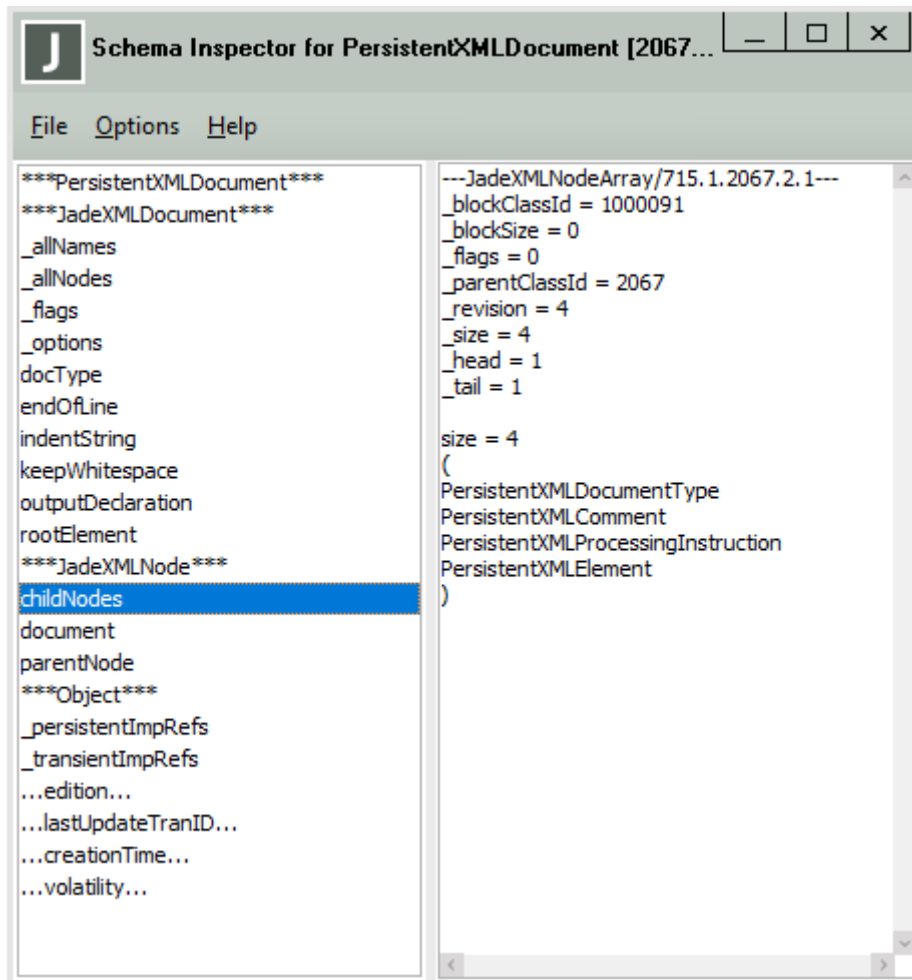
  beginTransaction;
  create xmlDocument persistent;
  xmlParser.parseDocumentFile(xmlDocument, "C:\JadeXMLWhitePaper\Persistent Example.xml");
  commitTransaction;
end;

```

4. Run the `loadPersistentXML` method, select the `PersistentXMLDocument` class in the Class Browser, and then press CTRL+I to open it in the Schema Inspector form.



5. Double-click the **PersistentXMLDocument** in the Schema Inspector form and ensure that the **ChildNodes** collection contains the correct child nodes, as follows.



6. Navigate around the JADE XML tree.
You should see that all the nodes from the **JadeXMLExample.xml** file are contained within the persistent tree in the appropriate structure.