



J A D E TM

XML in JADE

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 © 2006 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	3
Introduction	3
XML Structure	3
Well-Formed XML	4
Valid XML	4
Why XML?	4
A Short History	4
XML versus Web Services	5
XML in JADE	7
Objectives	7
Using XML in JADE	7
Creation and Manipulation of XML	7
Interpretation of XML	15
Examples	17
File In/File Out	17
Scenario	17
Instructions	18
Explanation	19
Data Streaming	23
Scenario	23
Instructions	24
Explanation	26
Conclusion	34
Appendix A – Persisting XML	35
Parsing an XML Document into a Persistent Structure	35
Parsing an XML Document into Multiple Persistent Structures	36

XML in JADE

Introduction

Open communication; it's the key to any successful relationship. In the mid-90s, with the proliferation of new, enhanced, and increasingly divergent technologies, the question of how best to ensure ongoing interoperability between systems was crying out for an answer. That answer was found in 1997, and Extensible Markup Language (XML) was born.

Like HTML, XML is a markup language made up of sets of tags describing data that are in turn interpreted by applications. Unlike HTML, XML does not have a prescribed set of tags into which you must shoehorn your information requirements. Rather than prescribing tags, XML defines a standard by which you can, in effect, create your own markup languages. The result is that an infinite number of tags can be created to describe information and used as part of an XML document. Unlike HTML, interpretation of XML is not restricted to a specific set of applications.

All of this delivers flexibility, improved efficiency, and provides the mechanism by which to achieve that all-important interoperability.

The XML framework in JADE has been implemented to enable rapid development of XML applications and XML components. Its objective is to make XML construction, manipulation, and interpretation as easy and as intuitive as possible.

This paper illustrates just how the JADE XML framework achieves this objective, along with detailed examples.

XML Structure

Structurally, an XML document is an entity that "owns" a set of other XML nodes. It must have a root element but any of the following nodes can be optionally included.

- Declarations
- Processing Instructions
- Elements
- Character Data

Elements are the essential individual components of XML content, containing both the data and the markup that describes that data. The ability to group and nest user-defined elements within one another provides much of the extensibility of XML. The rest is provided by allowing you to define the rules by which the XML tree structure is to be interpreted.

All languages have rules to which users must adhere in order to be understood. In spoken languages, this means:

- Including subjects, objects, and verbs, as appropriate
- Using correct vocabulary
- Applying the correct grammatical rules.

Well-Formed XML

“But XML allows me to create my own language. Surely there are no rules”. Well, yes and no.

While XML is not restricted by a set of available tags and allows you to create your own markup, there are still overarching rules that you must follow. Adherence to these rules results in XML that is “well-formed”. By definition, a well-formed XML document is one that syntactically conforms to the W3C XML Specification.

Essentially, using the spoken language parallel, this means that the right components of a sentence must be included and spelt correctly. An example in XML is that a processing instruction must always begin with the `<?` characters.

Valid XML

Unfortunately, having a sentence that includes the correct components does not necessarily result in something that makes sense. The words need to be included in the correct order and conform to grammatical rules. In this, XML is no different. However, as users of XML are effectively creating their own markup language, these rules are theirs to define. This is done through the Document Type Definition (a type of declaration) or more recently, XML Schema.

An XML Schema definition is effectively a DTD written in XML, with all the accompanying benefits of extensibility. Another added advantage is its support for data types in describing the rules for an XML document’s content.

To be structured and interpreted correctly, XML documents must be both well-formed and valid.

Why XML?

The best way to answer this question is to outline the background to its emergence as a global standard for communications between heterogeneous systems.

A Short History

The development of XML is a result of the evolution of markup languages, designed to capitalize on the advantages, while overcoming the disadvantages of its two precursors; that is, SGML and HTML.

Like XML, SGML does not impose the use of specific tags on you and therein lies its power and flexibility. Unfortunately, there the similarities end. Simple, widely supported tools with a low cost of entry are not easy to come by.

HTML was originally designed as a simplified version of SGML to be accessible to a wider range of users of varying knowledge and abilities. Its simplicity, coupled with the fact it

was free and compatible with Web browsers, resulted in rapid widespread adoption. The downside is that in simplifying SGML, much of the associated power and flexibility was lost. With its finite set of tags, primarily concerned with information presentation and layout in browsers, use of HTML to structure any set of data for a range of uses is just not an option.

A clear need was identified for a markup language that delivered the flexibility, extensibility, and portability of SGML, combined with the simplicity of HTML. That it should also be free was obvious and that it be developed in a way that was easy for applications to develop functionality to support it was essential.

The World Wide Web Consortium (W3C) sponsored a group of developers to come up with the solution. XML was the result. In 1998, the *W3C XML Specification Version 1.0* was approved and is still the standard.

That this is the case is a strong testament to the fact that the work carried out by the original group of developers of XML was right on the money. However, developers the world over will be familiar with how quickly business needs evolve. No sooner has one dream been delivered and a new one has been formulated, the predecessor taken for granted. While the XML specification has well and truly earned the title of a standard, the associated technology that uses XML has rapidly evolved.

XML versus Web Services

As a standard for communication between systems, by nature XML has limitless application including:

- Populating data on Web pages
- Framing data used in complex enquiries and transactions
- Producing files for interpretation by a wide range of systems
- Streaming data from one application to another

XML can be understood and interpreted by all.

To explain the difference between XML and Web services, let's look at a situation that presents itself to most of us many times each day. You receive a phone call from a colleague or family member asking you to carry out a task and get back to them with the result. Implicitly, all of the following decisions and abilities are in place.

1. You have both decided to use a common language (for example, English)
2. You both conform to grammatical rules and acceptable vocabulary (for example, English grammar and words)
3. You both understand the meaning of that vocabulary (for example, please run the xyz report)
4. You both know what you can do for each other and how to get each other to do it (for example, that you can provide the other person with the data to run the xyz report and he or she can run it)
5. You are able to act upon the request received (for example, you are able to run the report and provide him or her with the information)
6. They are able to understand the information returned (for example, he or she can understand the report)

7. You have both agreed on a mechanism by which the exchange is facilitated (for example, you know each other's phone number)

All of the above can be broadly cast into three separate areas of understanding.

- Language
- Meaning and behavior
- How to establish a dialog

The situation is similar when two systems need to communicate. Use of XML is effectively an agreement to use a common language. However, both systems must be able to know all of the other details required to interact. To continue with the analogy, you and your colleagues speak the same language, understand each other's capabilities, how to make contact, make meaningful requests, and understand responses. You know this through experience with each other and training.

Use of Web services makes it possible for an application to publish language, behavior, and connection information about itself. This information can then be used by other systems that want to interact with it. The document that provides this description (also written in XML) is known as a WSDL and includes the following.

- Language
 - XML
 - Vocabulary and type definitions (for example, XML Schema)
- Meaning and behavior
 - Operations and messages
 - Exposed methods and properties
- How to establish a dialog
 - Connection types
 - Connection ports
 - Communication method (for example, SOAP)

Use of Web services is equivalent to you publishing who you are, what language you speak, what you can do for anyone, what you need to do it for them and how you will respond, where they can contact you and how they should communicate.

XML on its own is only the language component of the broader Web services functionality, but is significant in that it's the foundation on which it's all built. It can also be used effectively on its own and this white paper covers how you can do this in JADE.

JADE provides Web services functionality, and more information on that can be found in the *Web Services in JADE* white paper.

XML in JADE

The JADE XML framework has been developed to provide the ability to create, interpret, and manipulate XML using JADE code, and designed with the JADE developer in mind.

Objectives

The objectives of the framework are to:

- Enable rapid development of XML applications and XML components in JADE
- Make the most common XML development tasks easy and intuitive

Using XML in JADE

How do you actually go about working with XML in JADE? What's involved? Most importantly, how the is use of XML in JADE made easier?

A number of classes exist in the **RootSchema** that contain all of the methods, references and properties you require to create, manipulate, and interpret well-formed and valid XML using JADE code.

These classes can be separated into two groups with two distinct purposes. The first group is primarily concerned with the creation and manipulation of XML. The second relates to the processing or interpretation of XML.

Creation and Manipulation of XML

The XML framework in JADE allows you to create and manipulate XML through the creation of objects upon which you can carry out operations, and to which you can make property alterations, just like anything else you might do in JADE. As these objects are treated and behave in the same way as other JADE objects, this means there is very little new material for you to learn.

The inherent ease and flexibility of JADE, coupled with the extensibility of XML, makes for a powerful partnership. This section discusses how the XML framework in JADE has been implemented, its individual components, and how they are used to generate XML.

Overview

As discussed earlier, an XML document is comprised of a set of entities, with components that can include:

- Declarations
- Processing instructions
- Elements
 - Attributes
- Character data

All of these components, and their variants, have corresponding classes provided in the JADE **RootSchema**.

It is with these classes that you are able to use JADE to model an XML tree structure either from an external XML document source or from your own JADE database. The

objects created when modeling XML actually mirror the visual structure of the real-life document, making use intuitive.

Including the document itself, all of the XML entities have properties in common with each other and common actions are taken upon them. All XML classes are grouped under one abstract superclass, **JadeXMLNode**.

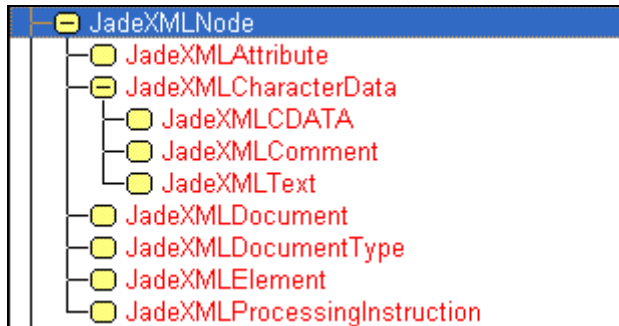


Fig 1.1: **JadeXMLNode** in the JADE Class Browser

A full class diagram is provided in the following section.

As useful as these system classes are for creating and manipulating XML, it was important to us to ensure that you have the flexibility to adapt and extend what has been provided to suit your own unique needs. The result is that you can add your own methods and user subclasses, allowing you to capitalize on existing functionality without being restricted in any way.

Before giving an example of JADE code creating an XML document, we will look at each of the **JadeXMLNode** subclasses and how they relate to XML components.

Example Reference

Below is an example XML document, complete with declarations, processing instructions, elements and attributes. This will be used as a reference in discussing each of the **JadeXMLNode** subclasses. Note that the numbers down the side of the page are not part of the XML document, rather they are used as a reference tool.

```

1. <?xml version="1.0"?>
2. <?xml:stylesheet type="text/xsl" href="peoplereults.xsl"?>
3. <!DOCTYPE PEOPLE_DETAILS [
   <!ELEMENT PEOPLE_DETAILS ANY>
   <!ELEMENT PERSON_DETAILS (PERSON_FIRST_NAME, PERSON_LAST_NAME,
     OCCUPATION, CONTACT_DETAILS)>
   <!ELEMENT PERSON_FIRST_NAME (#PCDATA)>
   <!ELEMENT PERSON_LAST_NAME (#PCDATA)>
   <!ELEMENT OCCUPATION (#PCDATA)>
   <!ELEMENT CONTACT_DETAILS (EMAIL+, PHONE+)>
   <!ELEMENT EMAIL (#PCDATA)>
   <!ELEMENT PHONE (#PCDATA)>
 ]>
4. <!--This document includes Person Details-->
5. <PEOPLE_DETAILS>
6.   <PERSON_DETAILS>
7.     <PERSON_FIRST_NAME>Joanne</PERSON_FIRST_NAME>
8.     <PERSON_LAST_NAME>Howard</PERSON_LAST_NAME>
9.     <OCCUPATION>Chief Executive Officer</OCCUPATION>

```

```
10.         <CONTACT_DETAILS>
11.             <EMAIL TYPE="Work">j.howard@company.com.au</EMAIL>
12.             <PHONE TYPE="Home">+61 2 9256 2222</PHONE>
13.             <PHONE TYPE="Work">+61 2 9246 4621</PHONE>
14.             <PHONE TYPE="Cell">+61 41 378 6787</PHONE>
             </CONTACT_DETAILS>
         </PERSON_DETAILS>
</PEOPLE_DETAILS>
```

Fig 1.2: Example XML

Note The use of capital letters in the XML tags above (and other examples throughout this white paper) is only a convention and not at all a requirement.

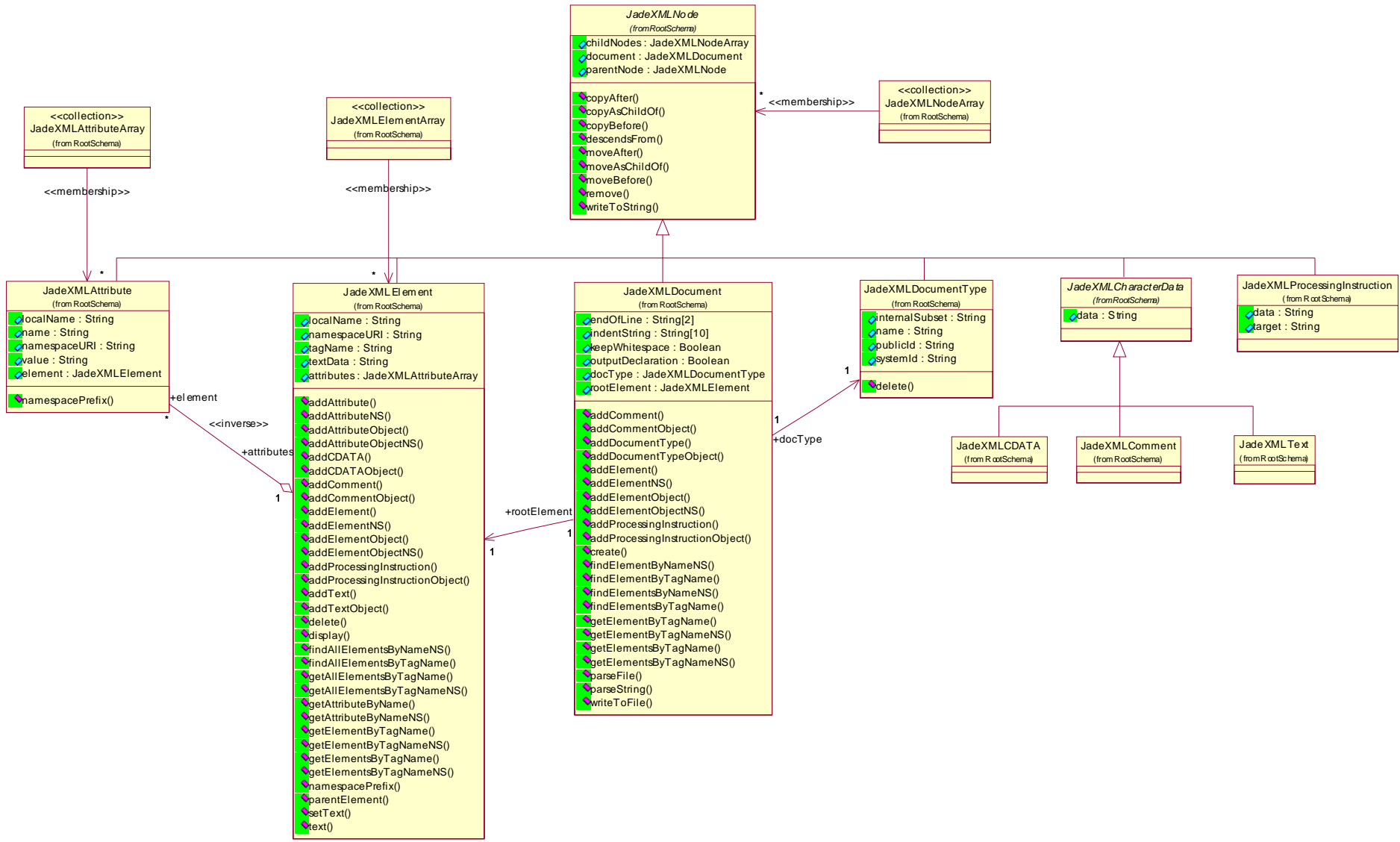


Fig 1.3 JadeXMLNode Class Diagram

JadeXMLNode Class

As stated earlier, all XML nodes share common properties and are subject to common operations. Perhaps the most demonstrable example of this is illustrated in the recursive structure of XML documents. All XML nodes can be both a parent and child of other nodes in an XML tree structure, and all belong to an owning document.

These relationships are specified in the **parentNode** and **childNodes** reference properties, while the document reference links the node with its owning **JadeXMLDocument**.

Every numbered item in Fig 1.2 represents an instance of a **JadeXMLNode** subclass.

Each node can be copied, moved or removed and can output its XML representation.

The abstract **JadeXMLNode** class provides you with the reference properties and methods that are inherited and available for use across its subclasses.

Note You can create your own persistent user subclasses of the **JadeXMLNode** subclasses. For more information, see “Appendix A – Persisting XML”, later in this document.

JadeXMLDocument Class

JADE allows you to represent an XML document as a tree structure, each node in the tree corresponding to an object. The purpose of the **JadeXMLDocument** class is to act as the owning object of those objects. In other words, it's the ultimate parent node.

Once instantiated, you can treat it as you would any other object in JADE. As management of the document and its contents is a key part of utilizing XML effectively, methods have been provided to assist you with adding, copying, moving, and getting nodes within the document structure.

Each of the elements, processing instructions, and declarations in Fig 1.2 belong to an instance of **JadeXMLDocument**.

JadeXMLElement Class

Equally important as the document itself, the elements it contains are responsible for describing and recording the data in your XML tree structure. These elements are represented by instances of the **JadeXMLElement** class and are associated with one another in a tree of objects, reflecting their physical layout in an XML document.

The importance of elements is reflected by the methods provided by JADE to support the actions you will most frequently take upon them as a developer, while allowing you to implement your own behaviors as well.

When looking at Fig 1.2, every opening tag pair and including **PEOPLE_DETAILS** is an element. **PEOPLE_DETAILS** is known as the root element of the document. In JADE, defining the data or textual content for an element is usually done through updating the **textData** property of the **JadeXMLElement** object. Using this approach results in reduced processing time and improves parsing performance. However, there is another option available that is discussed in the “JadeXMLText Class” section, later in this document.

The following code shows the creation of an XML document and the construction of a tree of elements within it.

```

vars
  xmlDoc : JadeXMLDocument;
  elmnt  : JadeXMLElement;
begin
  create xmlDoc transient;
  elmnt := xmlDoc.addElement("PEOPLE_DETAILS");
  elmnt := elmnt.addElement("PERSON_DETAILS");
  elmnt := elmnt.addElement("PERSON_FIRST_NAME");
  elmnt.setText("John");
  write xmlDoc.writeToString;
epilog
  delete xmlDoc;
end;

```

Fig 1.4 Creating an XML String

Note Unless otherwise specified, XML objects are created as transients.

The above code example results in a tree of objects being created, as illustrated in Fig 1.6.

The **addElement** method is used repeatedly and is available on both the **JadeXMLDocument** and **JadeXMLElement** classes. Essentially, the two methods are the same. The only difference is that on **JadeXMLDocument**, the method also sets the **rootElement** reference to the element created. A document can have only one root element.

The method creates a new instance of **JadeXMLElement** as a child node of the object for which it is called, but it also returns the newly created element, making it available to be assigned to a local variable. This allows you to work with it further. The above code therefore creates **PEOPLE_DETAILS** as a child node (and the root element) of the XML Document and then assigns that element to the *elmnt* variable.

The **addElement** method is used in the same way throughout the examples in this document. The format is shown below. *element-to-work-with* is always of type **JadeXMLElement** and *parent-node-you-are-adding-to* is of type **JadeXMLDocument** or **JadeXMLElement**.

```

element-to-work-with :=
  parent-node-you-are-adding-to.addElement("element-to-work-with-tag-
  name");
element-to-work-with.setText("element-to-work-with-data");

```

Fig 1.5 Explaining **addElement**

All methods that begin with **add**, but don't end in **Object**, behave in the same way as **addElement**. Those methods that do end in **Object** are similar but are generally used in the context of user subclasses. This is discussed further in "Appendix A – Persisting XML", later in this document.

The **writeToString** method, when called for the **xmlDoc** object, produces the following XML tree structure.

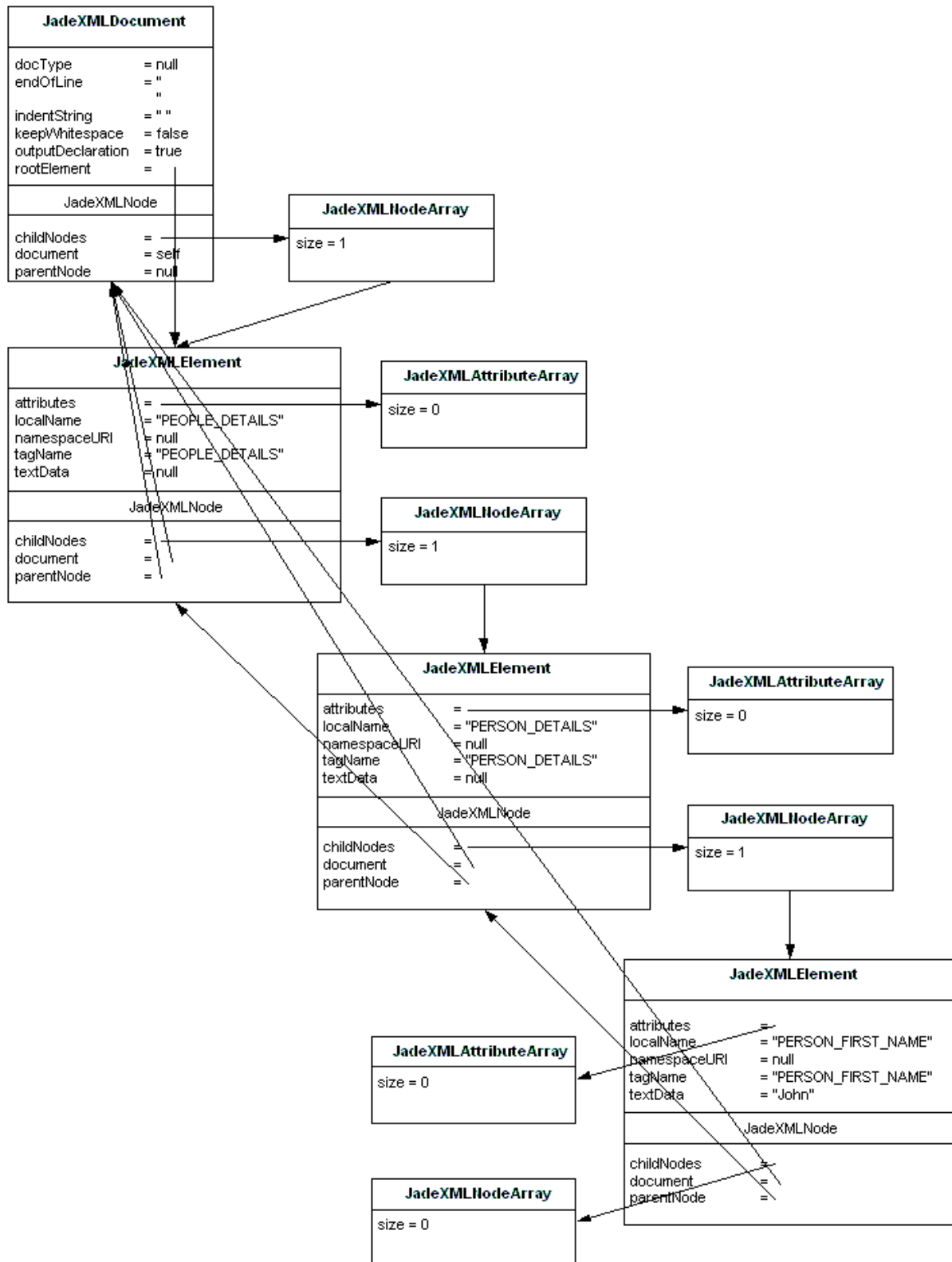


Fig 1.6 **JadeXMLNodes** created by the Fig 1.4 code example

```
<?xml version="1.0"?>
<PEOPLE_DETAILS>
  <PERSON_DETAILS>
    <PERSON_FIRST_NAME>John</PERSON_FIRST_NAME>
  </PERSON_DETAILS>
</PEOPLE_DETAILS>
```

Fig 1.7 XML String created by the Fig 1.4 code example

JadeXMLDocumentType Class

XML Document Type Definitions, as mentioned in the “Valid XML” section earlier in this document, determine the grammar and vocabulary of the markup used in the XML document. It is the mechanism by which rules are established to validate XML structure, inclusions, exclusions, and so on.

In Fig 1.2 above, item 3 covers the Document Type Definition. It states that the **PERSON_DETAILS** element must have children elements of **PERSON_FIRST_NAME**, **PERSON_LAST_NAME**, **OCCUPATION**, and **CONTACT_DETAILS**. An additional rule is specified regarding the **CONTACT_DETAILS** element, stating that it must have one or more **EMAIL** child elements, followed by one or more **PHONE** child elements. The XML tree structure must conform to these rules in order to be considered valid; however the JADE parser is non-validating.

The **JadeXMLDocumentType** class represents the document type declaration in an XML document tree. A reference to the document type (if the document has one) is stored in the **docType** property of the **JadeXMLDocument** class.

JadeXMLProcessingInstruction Class

This class represents a processing instruction in an XML document tree. Processing instructions are application-specific instructions on how to handle an XML document after the document has been parsed.

In Fig 1.2, item 2 represents a processing instruction. In this case, it is providing style sheet information that can be applied by an application such as an Internet browser. This means that XML can be sent to the browser and it can present the included information in the intended format.

Creating XML processing instructions is a simple matter of calling the **addProcessingInstruction** method for the **JadeXMLDocument** or **JadeXMLElement** class to which you want to add the instruction. The target application and instructions are then passed into the method as parameters.

JadeXMLAttribute Class

As with HTML, XML elements can have attributes. Each of the elements with a **parentNode** of **CONTACT_DETAILS** has a **TYPE** attribute. This attribute is used to describe the type of contact detail that the element contains, whether it is a *Home* or *Work* contact, and so on.

The **JadeXMLAttribute** class is used to represent an attribute of an XML element in your XML document tree. It will have a name, an optional namespace, and a value. It is added by calling the **addAttribute** method for the element to which you want to add the attribute.

JadeXMLCharacterData Class

XML defines Character Data to be the text that falls within XML start and end tags, and it can be any Unicode character with the exception of `<`, as this is reserved to denote the beginning of a tag. This sounds fairly simple, but why have a class, or indeed a set of classes to cover it?

The **JadeXMLCharacterData** class is the abstract superclass of character-based nodes in an XML document tree. These nodes include the text, CDATA, and comment nodes.

JadeXMLCDATA Class

There are times when you may want to use certain characters inside XML tags that are reserved for specific markup purposes. An example of this might be quotation marks. These are used in markup to surround the attribute values of XML Elements. If, however, you want to use actual quotation marks within an attribute value (that is, within another set of markup quotation marks), you would need to either replace it with an entity reference or create a CDATA string with those quotation marks included. Instances of the **JadeXMLCDATA** class automatically escape blocks of text for correct representation in XML.

It is a subclass of **JadeXMLCharacterData**, but does not add any additional properties or methods.

JadeXMLComment Class

In JADE, single-line comments are preceded by `//`, while multiple line comments are surrounded by `/*` and `*/`. XML also provides the facility for including comments in the body of XML documents. Comment tags begin with `<!--` and end with `-->`. In order to create an XML comment in JADE, an instance of the **JadeXMLComment** class is created and the **data** property set to the String value of the comment. In Fig 1.2, item 4 is an XML comment. In JADE, the **data** property would be equal to *"This document includes Person Details"*.

JadeXMLComment is a subclass of **JadeXMLCharacterData**, but does not add any additional properties or methods.

JadeXMLText Class

This class represents the textual content within an XML document tree. Normally, in order to set the value for a **JadeXMLElement**, you would simply set the **textData** property of that **JadeXMLElement**. Using this approach reduces the size of the XML document tree and improves parsing performance.

However, an alternative has been provided, allowing you to create instances of **JadeXMLText** as child nodes of the element for which they represent a value. Where multiple instances exist, the values of their respective **data** properties are concatenated and returned within the start and end tags of their parent node element.

Interpretation of XML

In most cases, you will find that simply creating and manipulating XML isn't enough to meet your requirements. Conversations are much more rewarding if they are two-way. This means that you need to be able to interpret XML as well.

With an in-built XML document parser and facilities in place that will enable you to build your own, the JADE XML framework makes this easy for you.

The following class diagram shows the **JadeXMLParser** class with its **JadeXMLDocument** subclass.

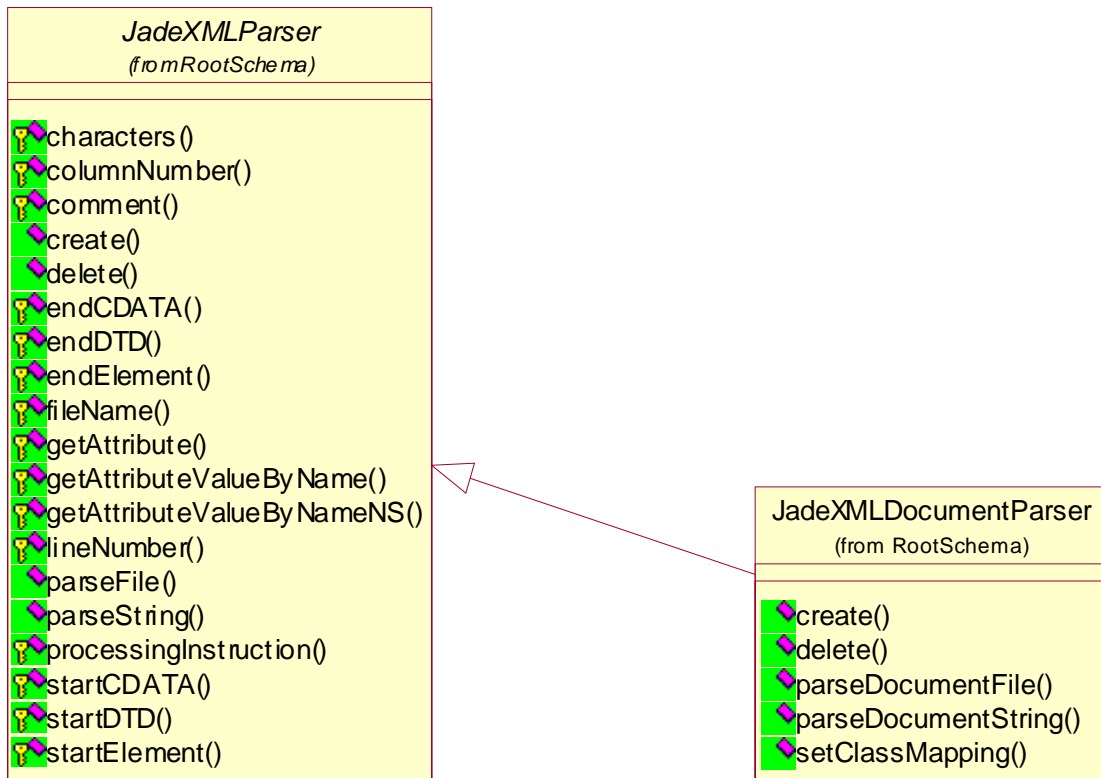


Fig 2.1 **JadeXMLParser** Class Diagram

JadeXMLParser Class

The purpose of this abstract class is to provide a base on which you can create your own parser subclasses that can be used as the interface for parsing XML documents. The fundamental properties and methods required to parse XML documents are provided with this class.

JadeXMLDocumentParser Class

One parser implementation is already provided. The **JadeXMLDocumentParser** class is a transient-only class that provides the interface for parsing XML documents into a tree of objects. The parser reads an XML file and creates a tree of object nodes that are instances of **JadeXMLNode** subclasses or your own subclasses.

This class enables you to take an XML file or String and create a tree of XML nodes, like that shown in Fig 1.6.

```
vars
    xmlDoc : JadeXMLDocument;
    parser : JadeXMLDocumentParser;
begin
    create xmlDoc transient;
    create parser transient;
    parser.parseDocumentFile(xmlDoc, "c:\XMLDocument.xml");
    xmlDoc.inspectModal;
epilog
    delete xmlDoc;
    delete parser;
end;
```

Fig 2.2 XML Document Parsing

Executing this code takes the file **XMLDocument.xml**, parses it, and creates an XML tree structure with **xmlDoc** as the owning document object. These objects can then be used and manipulated in the same way as any other JADE objects. The **inspectModal** method invokes the Schema Inspector, allowing you to review the tree of objects created.

Examples

In this section, we will explore two different examples of XML creation, manipulation, and interpretation using JADE code.

There will be times when you will want to create an XML tree structure using information in your database, and write it to file. While very useful, it will also be imperative to import an XML file from which you can build an XML tree structure.

Other situations will call for a greater level of direct interaction. You will have two applications that will need to communicate on a real-time basis, with one retrieving information from the other's database and sending instructions for specified updates.

File In/File Out

This section demonstrates how the first of these situations can be supported in JADE.

Scenario

You have a database that stores details of people and you would like to export that information as an XML document. You would also like the ability to import an XML document structured in the same way as the one you have output and create person records according to the information it contains.

In your database, each instance of the **Person** class contains the following information.

- First name
- Last name
- Occupation
- Address
- Home phone number

- Work phone number
- Cell phone number
- E-mail address

The following class diagram shows how the information is structured in the database.

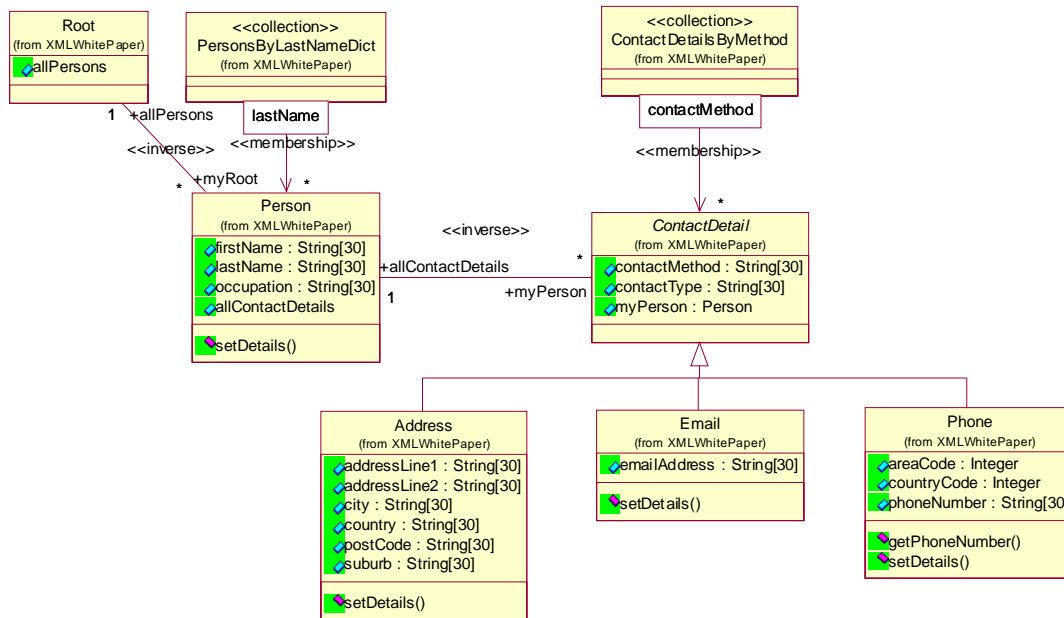


Fig 3.1 Example Scenario Class Diagram

Instructions

This section contains the instructions that result in the export and import of an XML document using JADE. General set-up instructions are covered first, followed by an export and import scenario. The “Explanation” section that follows explains what happens programmatically in each of the scenarios.

General Setup

1. Extract the **JADE_WP_XML_Examples.zip** file associated with this white paper to your **c:** drive. It can be downloaded from www.jadeworld.com.

A new directory is created, so it is recommended that you don't change the extract path to anything other than **c:**. This is because instructions and methods throughout this “Examples” section rely on the path remaining the same.

2. Navigate to **c:\JadeXMLWhitePaper**. This new directory contains:

- **People to Import 1.xml**
- **People to Import 2.xml**
- **Persistent XML Structure.xml**
- **XMLWhitePaper.scm**
- **XMLWhitePaper.ddb**
- **XMLWeb (directory)**

- **XMLWeb_jadehttp (directory)**
- 3. Load the **XMLWhitePaper** schema and forms definition files provided with this white paper into your JADE development environment.
- 4. Navigate to the **JadeScript** class, select the **setupAllData** method and execute the script by pressing the F9 key.

Export

1. Select the **listPeople** method and execute the script by pressing the F9 key.
This method outputs a list of the person objects from the database. The list is output to a JADE Interpreter window. The XML document created in the next step contains details of all these objects, including contact information. Use the output to check that all objects are represented in the XML document produced.
2. Select the **createPeopleXMLDocument** method.
This method iterates through all instances of the **Person** class and creates an XML file. The file is called **People Exported.xml** in your **c:\JadeXMLWhitePaper** directory.
3. Execute the script by pressing the F9 key.
4. Review the XML file produced.

Import

1. Review the **c:\JadeXMLWhitePaper\People to Import 1.xml** file.
This file contains the details of people that will be imported.
2. Select the **importPeople** method on the **JadeScript** class, and press the F9 key to execute the script.
This method processes the specified **People to Import.xml** file and creates instances of the **Person** class, populating them with the details contained in the file.
3. Select the **listPeople** method on the **JadeScript** class.
The output of this script shows the details of all people, including those added to the database by the **importPeople** method.

Explanation

This section covers what actually happened when you ran the above scripts, illustrating how easily the XML framework enables production and interpretation of XML.

Export

JadeScript::createPeopleXMLDocument Method

The above method begins by creating a transient instance of the **JadeXMLDocument** class, adding a comment and a document type declaration. The next step is to add a root element to the document. As the document will be containing details of person objects, the element will be named **PEOPLE_DETAILS**.

```
rootElement := xmlDoc.addElement("PEOPLE_DETAILS");
```

The **addElement** method is available on both the **JadeXMLDocument** and **JadeXMLElement** classes and is explained further in the “JadeXMLElement Class” section, earlier in the document.

The next step is to move through each instance of **Person** and add the details under the root element in a logical structure.

```
foreach person in root.allPersons do
  pers := rootElement.addElement("PERSON_DETAILS");
  elmnt := pers.addElement("PERSON_FIRST_NAME");
  elmnt.setText(person.firstName);
  elmnt := pers.addElement("PERSON_LAST_NAME");
  elmnt.setText(person.lastName);
```

In the above code fragment, **pers** is used to denote the element that frames the information associated with an individual person. For every **Person** instance, an element of **PERSON_DETAILS** is therefore added to the root element (**PEOPLE_DETAILS**) of the document.

Details directly associated with the person are then added as child nodes of the **PERSON_DETAILS** element. In the above code fragment, **PERSON_FIRST_NAME** is added as a child node of **PERSON_DETAILS**. The created element is then assigned to the **elmnt** variable. The text for **elmnt** is then set to the database value of **firstName** for the **Person** instance. The **elmnt** variable is then changed with the addition of the **PERSON_LAST_NAME** element as a child node of the **PERSON_DETAILS** element, and the process is repeated.

We next step through each of the contact details for the person record and group those details together. Consideration needs to be given to the different types of contact details.

```
contact := pers.addElement("CONTACT_DETAILS");
foreach contDet in person.allContactDetails do
  if contDet.isKindOf(Address) then
    address := contDet.Address;
    addr := contact.addElement("ADDRESS_DETAILS");
    addr.addAttribute("TYPE", address.contactType);
    elmnt := addr.addElement("ADDRESS_LINE_1");
    elmnt.setText(address.addressLine1);
    elmnt := addr.addElement("ADDRESS_LINE_2");
    elmnt.setText(address.addressLine2);
```

The above code fragment creates an element of **CONTACT_DETAILS** under the **PERSON_DETAILS** element within which to group address, phone, and e-mail information.

Once the method has iterated through all instances of the **Person** class, the XML document tree that has been created is written to file by the **JadeXMLDocument::writeToFile** method. In this example, it is written to **People Exported.xml** in the **c:\JadeXMLWhitePaper** directory.

Import

JadeScript::importPeople Method

This method parses the XML document and creates an XML tree structure with transient JADE objects.

Once this is done, instances of **Person** are created and updated based on the details of those JADE objects.

```
vars
  xmlDoc : JadeXMLDocument;
  parser : JadeXMLDocumentParser;
begin
  create xmlDoc transient;
  create parser transient;
  parser.parseDocumentFile(xmlDoc, "c:\JadeXMLWhitePaper\People to
    Import 1.xml");
  createPeopleRecords(xmlDoc);
epilog
  delete xmlDoc;
  delete parser;
end;
```

JadeXMLDocumentParser::parseDocumentFile Method

This method on the **JadeXMLDocumentParser** class parses the specified file and builds a tree structure of XML node objects in JADE based on the information the file contains. The resulting instance of the **JadeXMLDocument** class is then returned for further use.

JadeScript::createPeopleRecords Method

This method begins by assigning a value to the **root** variable and then creates transient instances of the **JadeXMLElementArray** class.

The next step is to evaluate the **JadeXMLDocument** provided in the method's parameter.

```
pXMLDocument.getElementsByTagName("PERSON_DETAILS", people);
```

All of the elements in the XML document with a **tagName** of **PERSON_DETAILS** are found and added into the **JadeXMLElementArray** object represented by the **people** variable.

Because each **PERSON_DETAILS** element represents details associated with an individual person, a persistent instance of the **Person** class is created for each one.

```
foreach elmnt in people do
  create person persistent;
```

The method then assigns values to the properties of the **Person** instance based on the **childNodes** of the **PERSON_DETAILS** element.

```
firstName := elmnt.getElementByTagName("PERSON_FIRST_NAME").textData;
lastName := elmnt.getElementByTagName("PERSON_LAST_NAME").textData;
occupation := elmnt.getElementByTagName("OCCUPATION").textData;
person.setDetails(firstName, lastName, occupation);
```

The contact details of the person are then added one by one. First the parent contact details node is found and assigned to the **contDet** variable.

```
contDet := elmnt.getElementByTagName("CONTACT_DETAILS");
```

Of the contact details, the **address** is processed first.

```
eladdr := contDet.getElementByTagName("ADDRESS_DETAILS");
if eladdr <> null then
  create address persistent;
  contactType := eladdr.getAttributeByName("TYPE").value;
  addressLine1 :=
    eladdr.getElementByTagName("ADDRESS_LINE_1").textData;
  addressLine2 :=
    eladdr.getElementByTagName("ADDRESS_LINE_2").textData;
  suburb := eladdr.getElementByTagName("SUBURB").textData;
  city := eladdr.getElementByTagName("CITY").textData;
  country := eladdr.getElementByTagName("COUNTRY").textData;
  postCode := eladdr.getElementByTagName("POSTCODE").textData;
  address.setDetails(contactType, addressLine1, addressLine2,
    suburb, city, country, postCode);
```

The values of the **textData** properties of the various address elements are assigned to the properties of the **Address** instance created.

Once the address details have been updated, it is added to the **allContactDetails** collection on the **Person**.

```
person.allContactDetails.add(address);
```

All of the other contact details are processed in a similar manner.

Data Streaming

While the preceding file-in/file-out example goes some way to demonstrating how to create and interpret XML files using JADE code, it is important for you to be able to interact on a real-time basis.

Note This example is designed only to work on a Windows operating system and to operate on the local host computer, rather than across a network.

Scenario

Using the same details as the previous example, you require the ability to search over the **Person** objects in your database over the Internet. A Web browser will be used as the application with which to interact with your database.

Following good design practice, you want to decouple the database and presentation by returning XML from the server and using XSL to generate the HTML.

You will search by first name and last name for the **Person** objects you want. An option will be included that will allow you to produce an XML file based on the results of their search.

An import facility will allow you to import an XML file, from which the application will create **Person** objects.

Note In order for this example to work, Microsoft IIS must be installed on your machine.

Instructions

General Set Up

If you haven't already done so, complete the general set up steps outlined below.

1. Extract the **JADE_WP_XML_Examples.zip** file associated with this white paper to your **c:** drive. It can be downloaded from www.jadeworld.com.

As a new directory is created, it is recommended that you don't change the extract path to anything other than **c:**. This is because instructions and methods throughout this "Examples" section rely on the path remaining the same.

2. Navigate to **c:\JadeXMLWhitePaper**. This new directory contains:
 - **People to Import 1.xml**
 - **People to Import 2.xml**
 - **Persistent XML Structure.xml**
 - **XMLWhitePaper.scm**
 - **XMLWhitePaper.ddb**
 - **XMLWeb (directory)**
 - **XMLWeb_jadehttp (directory)**
3. Load the **XMLWhitePaper** schema and forms definition files provided with this white paper into your JADE development environment.
4. Navigate to the **JadeScript** class, select the **setupAllData** method, and execute the script by pressing the F9 key.

IIS Set Up

1. Go to **Control Panel, Administrative Tools, Internet Information Services**.
2. Select **Default Web Site**, right-click, and select **New, Virtual Directory**.
3. Enter **xmlweb** as the alias.
4. Click **Next**.
5. Browse to the **XMLWeb** subdirectory of your **c:\JadeXMLWhitePaper** directory, click **OK**, and then click **Next**.
6. Ensure **Read, Run Scripts, Execute**, and **Browse** are selected.
7. Click **Next**, and then click **Finish**.

Application Introduction

1. From the JADE development environment, click the **Run Application** (green arrow) button and ensure that **XMLWhitePaper** is selected.
2. Click **OK**.
3. Navigate to the **XMLWeb** subdirectory of your **c:\JadeXMLWhitePaper** directory.

Authentication

1. Open **index.htm** in your Web browser.
2. Enter **test** in the **User Name** field, **xmlweb** in the **Password** field, and then click **Login**.

At this point, a page is presented that enables you to either search for people and optionally produce the results as an XML file created locally, or import people from an XML file.

Note If you experience any problems, copy the **jadehttp.dll** file from your **bin** directory into the **XMLWeb** subdirectory of your **c:\JadeXMLWhitePaper** directory.

Person Search

1. Enter **Dolce** into the **Last Name** field and then click **Person Search**.
2. Right-click in the results area of the page and then click **View Source**.
You will see that the source is an XML document despite its formatted presentation in the Web browser. How this is achieved is reviewed in the following sections.
3. Click the **Back** button in your browser.

File Export

1. Ensure the **First Name** and **Last Name** search criteria fields are blank.
2. Check the **Create File** check box and enter **c:\JadeXMLWhitePaper\Search Results.xml** in the **File Name** text box.
3. Click the **Person Search** button.
The details of the **Person** objects returned in the search have also been produced as an XML document called **Search Results.xml** in your **c:\JadeXMLWhitePaper** directory.
4. Click the **Back** button in your browser.

File Import

1. Navigate to your **c:\JadeXMLWhitePaper** directory.
2. Open the **People to Import 2.xml** file and review its contents.
3. Close the file.
4. In your Web browser, click the **Browse** button to select a file to import.
5. Browse to your **c:\JadeXMLWhitePaper** directory and select **People to Import 2.xml**.
6. Click the **Import** button.
A confirmation screen lists the people who have been imported.
7. Click the **Back** button in your browser.
8. Clear all of the search fields including the **Create File** and **File Name** fields.
9. Click the **Person Search** button.
The details of those people in the **People to Import 2.xml** file now appears among the search results.

Explanation

How exactly does all of the above happen? You have a Web page within a Web browser, and a JADE application. How are they communicating? How is it that raw XML can be passed back to the browser and be interpreted and displayed in a formatted way?

This section explores each of the actions carried out above, reviews the code, and explains how this communication takes place.

Common Behavior

While each of the actions results in a different logical process, there are some common behaviors across all actions. Broadly speaking, these common behaviors are encapsulated in three separate actions: the request, the response, and the formatting.

The Request

In this scenario, the request of the JADE application is being made through a Web browser. This is the case for authentication, searching for people, and for both exporting and importing files.

How does the browser place this request? If you are familiar with HTML, you will be familiar with the **<form>** tag. These typically take details entered by users and execute an action upon those details. The action is often defined by some kind of script, such as a CGI. In this example, all forms presented in the Web browser have the same *action* attribute value:

```
http://localhost/xmlweb/jadehttp.dll?XMLWhitePaper
```

This is effectively providing the destination, to which details entered in the various forms are sent once their submit buttons have been clicked.

A subclass of **JadeWebServiceProvider** has been created to act as the entry point for this information once it arrives. It is important to note that we are merely using the Web services framework to facilitate HTTP communications. No WSDL or SOAP functionality has been utilized.

Navigate to the **WPXMLWebServiceProvider** class.

WPXMLWebServiceProvider::processRequest Method

This method is the first to receive the **incomingMessage** and is a re-implementation of a superclass method. However, **inheritMethod** has been commented out, meaning that we have overridden the standard Web service provider processing. Our goal is to simply receive the message and carry out a series of actions based on that message.

```
vars
  message : String;
begin
  rawXML := true;
  message := incomingMessage;
  procMessage(message);
  //inheritMethod();
end;
```

This method takes the information received and passes it into the **procMessage** method.

WPXMLWebServiceProvider::procMessage Method

As you will see, this method takes the information received as a String and separates it out into its substrings that contain the separate components of the request.

The method then assigns the values of these components to properties of the global transient **Application** object for use later.

The Response

WPXMLWebServiceProvider::reply Method

A re-implementation of a superclass method, this method sends an XML string back to the Web browser. Before it is ready to do this, however, it must carry out the necessary processing based on the information received in the request phase, and build the response.

```
vars
  xml : String;
begin
  xml := buildXMLReply;
  return xml;
  //return inheritMethod();
end;
```

WPXMLWebServiceProvider::buildXMLReply Method

With details of the request stored in the **Application** object, the **buildXMLReply** method can:

- Authenticate users
- Perform person searches
- Export details of person records to an XML file
- Import details of person records from an XML file

We will look at each of the above actions in more detail, considering the HTML, XSL, and most importantly, the JADE code involved in providing the desired behavior.

The Format

If JADE is passing back raw XML to the browser, how does it present it in the browser in the way it does? This is the only part handled externally from JADE and is the means by which we decouple the server-side from the presentation.

For example, upon a successful login, the following is presented to the user.

However, the source of this page and the information passed back from JADE is simply:

```
<?xml version="1.0"?>
<?xml:stylesheet type="text/xsl" href="welcome.xsl"?>
<!--Welcome Details-->
<USER>
  <FIRSTNAME>Test</FIRSTNAME>
  <LASTNAME>User</LASTNAME>
</USER>
```

The key to the formatting is in the **href="welcome.xsl"** part of the XML processing instruction. Your browser recognizes this statement and applies the **welcome.xsl** stylesheet to the XML in the document. The result is the screen above. All responses sent to the Web browser in this example have an XSL stylesheet to apply. The separation of the presentation from the data results in increased flexibility for the customer and for you. This way, the responsibility of the system is to provide the data and the customers are free to format it in their own way, with their own branding, without requiring development extensions to the application.

Authentication

Having covered the common behaviors, this section looks at what the JADE code is doing in the **XMLWhitePaper** application to authenticate the user and how it is building the XML response.

JADE Code

WPXMLWebServiceProvicer::buildXMLReply Method

As stated in the "Common Behaviors" section earlier in the document, this method takes the values set on the global transient instance of **Application** and carries out actions as appropriate. In this case, we are looking at what happens when you log on.

When information is received from the search form, the **procMessage** method set the **xmlAction** value of **Application** to **AUTHENTICATE** (integer global constant of **1**), so this is where we begin the review.

```
if app.xmlAction = AUTHENTICATE then
  username := app.login_username;
  password := app.login_password;
  if username <> null and password <> null then
    xmlDoc := buildLoginResponse(username, password);
    xmlReply := xmlDoc.writeToString;
  endif;
  app.resetValues;
```

The applicable values from the **Application** instance are assigned to local variables. These are then used as parameters in calling the **buildLoginResponse** method.

WPXMLWebServiceProvider::buildLoginResponse Method

```
user := root.allUsers.getAtKey(pUsername);
create xmlDoc transient;
if user <> null then
  if user.password = pPassword then
    xmlDoc.addProcessingInstruction('xml:stylesheet
      type="text/xsl"', 'href="welcome.xsl"');
    xmlDoc.addComment('Welcome Details');
    rootElmnt := xmlDoc.addElement('USER');//the root element
    elmnt := rootElmnt.addElement('FIRSTNAME');
    elmnt.setText(user.firstname);
    elmnt := rootElmnt.addElement('LASTNAME');
    elmnt.setText(user.lastname);
```

This method begins by finding a **User** object with a **username** that matches that received as a parameter from the **buildXMLReply** method. It then creates a transient instance of **JadeXMLDocument**.

The method then carries out different actions depending on whether:

- There is a user with the provided username
- There is a user and the provided password matches the actual password

In all cases, information is added to the **JadeXMLDocument** object.

The above code fragment shows what happens when a matching user is found and the supplied password matches the value of the **password** property of the **User** object.

The first thing that happens is that a processing instruction is added to the document, followed by a comment. The next step is to add a root element to the document. To that root element, two further elements are added: **FIRSTNAME** and **LASTNAME**. The **User** details from the database are used to populate the **textData** properties of these elements.

The resulting **JadeXMLDocument** is then returned to the **buildXMLReply** method.

WPXMLWebServiceProvider::buildXMLReply Method

```
xmlDoc := buildLoginResponse(username, password);
xmlReply := xmlDoc.writeToString;
```

The instance of **JadeXMLDocument** returned by the **buildLoginResponse** method is then assigned to the **xmlDoc** local variable. This is then written to a String and assigned to the **xmlReply** variable that is returned to the **reply** method.

As stated in the “Common Behaviors” section, the **reply** method then sends this string to the browser.

The browser receives the XML String, initiates the processing instruction (XSL style sheet) that runs through the XML document, and then appropriately presents the information it contains.

Person Search and Export

In addition to the common behaviors outlined above, this section looks at what the JADE code is doing in the **XMLWhitePaper** application to search for person records, export the details of those records to file, and how it is building the XML response.

JADE Code

WPXMLWebServiceProvider::buildXMLReply Method

As stated in the “Common Behaviors” section earlier in this document, this method takes the values set on the global transient instance of **Application** and carries out actions as appropriate. In this case, we are looking at what happens when you conduct a person search.

When information is received from the search form, the **procMessage** method set the **xmlAction** value of **Application** to **SEARCH** (integer global constant of **2**), so this is where we begin the review.

```

elseif app.xmlAction = SEARCH then
  firstname := app.srchFirstName;
  lastname := app.srchLastName;
  createFile := app.createFile;
  filename := app.xmlFilename;
  people := findPeople(firstname, lastname);
  xmldoc := buildPersonSearchResponse(people);
  xmlReply := xmldoc.writeToString;
  if createFile = EXPORT and filename <> null then
    createXMLFile(xmldoc, filename);
  endif;
  app.resetValues;

```

The applicable values of **Application** properties are assigned to local variables prior to calling the **findPeople** method.

WPXMLWebServiceProvider::findPeople Method

This method takes two of those properties (**firstname** and **lastname**) to search for people that match the parameters. Those that match are added to a transient instance of **PersonsByLastNameDict**.

WPXMLWebServiceProvider::buildXMLReply Method

```

people := findPeople(firstname, lastname);
xmldoc := buildPersonSearchResponse(people);

```

The **buildXMLReply** method then takes the **PersonsByLastNameDict** collection returned by the **findPeople** method, and assigns it to the **people** local variable. This collection of person objects is then passed into the **buildPersonSearchResponse** method.

WPXMLWebServiceProvider::buildPersonSearchResponse Method

This is the method that actually constructs the XML tree structure. It begins by adding processing instructions and declarations before adding the root element of **PEOPLE_DETAILS**. This tag encapsulates all data in the XML document.

It then iterates through the objects included in the **PersonsByLastNameDict** collection passed into it as a parameter from the **buildXMLReply** method. For every member of the collection, it creates a **PERSON_DETAILS** element under which it adds a tree structure of elements containing person information from the database.

The resulting **JadeXMLDocument** is then passed back to the **buildXMLReply** method.

WPXMLWebServiceProvider::buildXMLReply Method

This method then takes the **JadeXMLDocument** created in the **buildPersonSearchResponse** method, converts it to a String and assigns it to the **xmlReply** local variable.

```
xmlReply := xmlDoc.writeToString;
```

xmlReply is the String value returned to the **reply** method.

As stated in the "Common Behaviors" section, the **reply** method then returns this String to the browser.

The browser receives the XML string, initiates the processing instruction (XSL style sheet) that runs through the XML document, and then appropriately presents the information it contains.

There is an additional action taken in this section of the **buildXMLReply** method that is dependent upon a parameter entered by the user.

```
if createFile = EXPORT and filename <> null then
    createXMLFile(xmlDoc, filename);
endif;
```

If the user has checked the **Create File** check box and provided a file name, the **createXMLFile** method is called. The XML document created by the **buildPersonSearchResponse** method is then passed into the **createXMLFile** method, along with the specified **filename**.

WPXMLWebServiceProvider::createXMLFile Method

This method takes the **JadeXMLDocument** it receives as a parameter and removes the processing instruction that links the XML document to the XSL style sheet. This is done because the processing instruction is not applicable outside of a Web browser context.

The method then calls the **writeToFile** method for the **JadeXMLDocument**.

Import

In addition to the common behaviors outlined above, this section looks at what the JADE code is doing in the **XMLWhitePaper** application to import person details from an XML file, and how it is building the XML response.

WPXMLWebServiceProvider::buildXMLReply Method

As stated in the "Common Behaviors" section earlier in this document, this method takes the values set on the global transient instance of **Application** and carries out actions as appropriate. In this case, we are looking at what happens when you import person details from an XML file.

When information was received from the search form, the **procMessage** method set the **xmlAction** value of **Application** to **IMPORT** (integer global constant of **3**), so this is where we begin the review.

```
elseif app.xmlAction = IMPORT then
  filename := app.xmlFilename;
  if filename <> null then
    xmldoc := importPeople(filename);
  endif;
  xmlReply := xmldoc.writeToString;
  app.resetValues;
```

This method sets the **filename** local variable to that set on the **Application** earlier by the **procMessage** method.

Once set, the **importPeople** method is called and it is provided with the specified file name as a parameter.

WPXMLWebServiceProvider::importPeople Method

To begin with, this method needs to create three transient instances of two different classes: **JadeXMLDocument** and **JadeXMLDocumentParser**.

```
create xmlDoc transient;
create parser transient;
create response transient;
parser.parseDocumentFile(xmlDoc, pFilename);
persDict := createPeopleRecords(xmlDoc);
```

The next step is to create a JADE XML tree structure from the specified file. This is done by calling the **parseDocumentFile** method for the **JadeXMLDocumentParser**. The **parseDocumentFile** method reads the specified file and populates the instance of **JadeXMLDocument** passed into it as a parameter with nodes based on the information in that file.

Once we have the XML tree structure, the next step is to navigate through it, creating database information based on the data it contains. The **createPeopleRecords** method does this.

WPXMLWebServiceProvider::createPeopleRecords Method

This method takes the XML tree structure (**JadeXMLDocument**) as a parameter.

```
create persDict transient;
create people transient;
create contacts transient;
pXMLDocument.getElementsByTagName("PERSON_DETAILS", people);
beginTransaction;
foreach elmnt in people do
```

This method begins by creating transient instances of the **PersonsByLastNameDict** and **JadeXMLElementArray**. Next, all elements with a **tagName** of **PERSON_DETAILS** are collected and placed into the **JadeXMLElementArray**. In the above code fragment, this is represented by the **people** local variable.

As each **PERSON_DETAILS** element contains details for one person, the method then moves through each of these elements and processes its **childNodes**.

```
create person persistent;
firstName := elmnt.getElementByTagName("PERSON_FIRST_NAME").textData;
lastName := elmnt.getElementByTagName("PERSON_LAST_NAME").textData;
occupation := elmnt.getElementByTagName("OCCUPATION").textData;
person.setDetails(firstName, lastName, occupation);
```

To begin with, a persistent instance of **Person** is created. The **getElementByTagName** method is then used to retrieve the data associated with child node elements by tag name. This data is then assigned to the appropriate property of the **Person** instance.

```
contDet := elmnt.getElementByTagName("CONTACT_DETAILS");
eladdr := contDet.getElementByTagName("ADDRESS_DETAILS");
```

With the details of the **Person** set, the next step is to add contact information. The code then navigates its way to the **ADDRESS_DETAILS** element.

```
if eladdr <> null then
  create address persistent;
  contactType := eladdr.getAttributeByName("TYPE").value;
  addressLine1 :=
    eladdr.getElementByTagName("ADDRESS_LINE_1").textData;
  addressLine2 :=
    eladdr.getElementByTagName("ADDRESS_LINE_2").textData;
  suburb := eladdr.getElementByTagName("SUBURB").textData;
  city := eladdr.getElementByTagName("CITY").textData;
  country := eladdr.getElementByTagName("COUNTRY").textData;
  postCode := eladdr.getElementByTagName("POSTCODE").textData;
  address.setDetails(contactType, addressLine1, addressLine2,
    suburb, city, country, postCode);
  person.allContactDetails.add(address);
endif;
```

Where an **ADDRESS_DETAILS** element is found, a persistent instance of **Address** is created. The values of the **textData** properties of the child node elements are then assigned to the **Address** properties.

Once complete, the address is added to the **allContactDetails** collection against the **Person**.

The same process is followed for phone and e-mail details.

```
persDict.add(person);
```

At the conclusion of processing each **PERSON_DETAILS** element, they are added to **persDict**. Once all **PERSON_DETAILS** elements are processed, this instance of **PersonsByLastNameDict** is returned to the **importPeople** method.

WPXMLWebServiceProvider::importPeople Method

Once returned to the **importPeople** method, it is assigned to the **persDict** local variable. The **persDict** variable is then passed to the **buildImportResponse** method.

WPXMLWebServiceProvider::buildImportResponse Method

```
create xmlDoc transient;
if pPeople.size > 0 then
  xmlDoc.outputDeclaration := true;
  xmlDoc.addProcessingInstruction('xml:stylesheet
    type="text/xsl"', 'href="confirmation.xsl"');
  rootElement := xmlDoc.addElement("PEOPLE_DETAILS");
```

```
    foreach person in pPeople do
      pers := rootElement.addElement("PERSON_DETAILS");
      elmnt := pers.addElement("PERSON_FIRST_NAME");
      elmnt.setText(person.firstName);
      elmnt := pers.addElement("PERSON_LAST_NAME");
      elmnt.setText(person.lastName);
      elmnt := pers.addElement("OCCUPATION");
      elmnt.setText(person.occupation);
    endforeach;
  endif;
  return xmlDoc;
```

The next thing this method does is traverse the dictionary of **Person** objects just created. The details of these **Person** objects are then used to create an XML document to confirm creation as shown above. The **href="confirmation.xsl"** statement indicates the XSL stylesheet that should be applied to the XML document created to ensure appropriate presentation in an Web browser. This XML document is then returned to the **importPeople** method and assigned to its **response** local variable.

WPXMLWebServiceProvider::importPeople

```
    response := buildImportResponse(persDict);
    return response;
```

This method takes the **JadeXMLDocument** represented by the **response** local variable and returns it to the **buildXMLReply** method.

WPXMLWebServiceProvider::buildXMLReply Method

This method takes the **JadeXMLDocument** and writes it to a String for the **reply** method to send back to the browser.

```
    xmlReply := xmlDoc.writeToString;
```

Conclusion

JADE enables the rapid development of XML applications with a framework that is intuitive in its use and powerful in its application. The XML framework in JADE takes advantage of all the benefits of XML, while exploiting the very best JADE has to offer. Using JADE, the creation, manipulation, and interpretation of XML just got easier.

More information on how to use the JADE XML framework can be found in the JADE product information library, available from www.jadeworld.com/jade.

Appendix A – Persisting XML

This appendix covers two examples of how a persistent structure of JADE objects is created through parsing an XML document. One of the advantages of this approach is that you do not need to store an XML document in its raw format and re-parse it every time you need to use its contents. Once created, you can use the created tree structure in the same way as you might any other set of JADE objects.

The two examples provided in this section demonstrate how an:

- Entire XML document can be parsed into a persistent structure
- XML document can be parsed, with JADE creating persistent structures based on sections of the raw XML document

Parsing an XML Document into a Persistent Structure

The file contains details of staff working within two departments. This exercise imports the document in its entirety.

JadeScript::persistentImport Method

This method parses the XML document and creates an XML tree structure with persistent JADE objects.

```
vars
  parser : JadeXMLDocumentParser;
  xmlDoc : WPXMLDocument;
  root   : Root;
begin
  root := Root.firstInstance;
  beginTransaction;
  create xmlDoc persistent;
  create parser transient;
```

We begin by creating a persistent instance of **WPXMLDocument** represented by the **xmlDoc** variable and a transient instance of **JadeXMLDocumentParser** represented by the **parser** variable.

```
parser.setClassMapping(JadeXMLAttribute, WPXMLAttribute);
parser.setClassMapping(JadeXMLCDATA, WPXMLCDATA);
parser.setClassMapping(JadeXMLComment, WPXMLComment);
parser.setClassMapping(JadeXMLText, WPXMLText);
parser.setClassMapping(JadeXMLDocumentType, WPXMLDocumentType);
parser.setClassMapping(JadeXMLElement, WPXMLElement);
parser.setClassMapping(JadeXMLProcessingInstruction,
  WPXMLProcessingInstruction);
```

Class mapping is a necessary part of parsing XML documents into a structure composed of instances of persistent user subclasses. As a system class such as **JadeXMLElement** can have multiple user subclasses, JADE needs to know which subclass should be used to create persistent objects when an XML element is found in the document.

For example, you may have two persistent user subclasses of **JadeXMLElement**, one to store information regarding cars and one related to car manufacturers. When parsing an XML document containing car information, the **setClassMapping** method is used to tell JADE to create instances of the car-related subclass of **JadeXMLElement**, rather than that related to car manufacturers.

In order to parse XML documents into a persistent structure, JADE requires you to specify the mapping for all real **JadeXMLNode** system subclasses regardless of whether there is an intent to use them all. This is done to prevent runtime exceptions if the file does in fact contain an XML node for which no mapping has been specified. Needless to say, parsing XML documents into a persistent structure requires you to have actually created user subclasses for all system subclasses first. The only exception is the **JadeXMLDocument** subclass, which is declared explicitly when the document is created prior to calling the **parseDocumentFile** method.

```

parser.parseDocumentFile(xmlDoc,
    "c:\JadeXMLWhitePaper\Persistent XML Structure.xml");
xmlDoc.setName("All People");
root.allWPXMLDocuments.add(xmlDoc);
commitTransaction;

epilog
    delete parser;
end;

```

Once the mapping is complete, the file is parsed and the **WPXMLDocument** is populated with an XML tree structure. As the document is stored persistently, it is added to a collection for easy retrieval at a later date.

Parsing an XML Document into Multiple Persistent Structures

This section covers an example of how an XML document is evaluated and multiple persistent structures created to contain separate sections of the document. This example takes the same file as used in the Example 1 above, and creates two persistent XML documents (instances of **WPXMLDocument**), into which the details of staff for each department are separated.

JadeScript::persistentSplitImport Method

This method parses the XML document and creates two XML tree structures with persistent JADE objects populated from two separate sections of the XML document.

```

root := Root.firstInstance;
create xmlDoc transient;
create elmntArray transient;
create parser transient;
parser.parseDocumentFile(xmlDoc,
    "c:\JadeXMLWhitePaper\Persistent XML Structure.xml");
xmlDoc.getElementsByTagName("DEPT_PEOPLE", elmntArray);
beginTransaction;
foreach elmnt in elmntArray do
    name := elmnt.getAttributeByName("NAME").value;
    create newXMLDoc persistent;

```

This section of the script creates the following transients.

1. **xmlDoc** – passed as a parameter into the **parseDocumentFile** method and represents the transient XML tree structure created by the method
2. **elmntArray** – used as a transient container for the elements that delimit the sections of the XML file in which we have an interest
3. **parser** – used to actually parse the XML file

The specified file (**Persistent XML Structure.xml**) is parsed as usual. The method then gets the delimiting elements from which we want to create our separate documents and adds them to the **elmntArray**. As we are creating a separate document for each element in this array, the method then steps through each element and creates a persistent XML document in JADE.

A name attribute has been added to our subclass of **JadeXMLDocument** (**WPXMLDocument**), to allow us to identify them for retrieval purposes. The name for each document is to be derived from the **NAME** attribute of the **DEPT_PEOPLE** element. We assign it to the **name** variable at this point for use later.

```
rootElmnt := elmnt.makePersistent(WPXMLElement, WPXMLAttribute,
    WPXMLComment, WPXMLCDATA, WPXMLText, WPXMLDocumentType,
    WPXMLProcessingInstruction, newXMLDoc).WPXMLElement;
```

This code fragment takes the selected element from the array and calls the **makePersistent** method to, oddly enough, make it persistent.

JadeXMLNode::makePersistent Method

With the exception of **newXMLDoc**, which has a type of **WPXMLDocument**, all parameters passed into this method are classes. In a similar way to class mapping covered in the previous example, it is necessary to specify what user subclasses are to be instantiated when creating a persistent copy of a transient class. It is when calling this method that those user subclasses are specified.

As we potentially need to create persistent instances of all of the different XML nodes, this method is placed on the **JadeXMLNode** class.

```
makePersistent(elmnt, attr, comment, cdata, text, docType, pi : Class;
    xmlDoc : JadeXMLDocument input) : JadeXMLNode updating;
vars
    newNode,
    newChildNode,
    childNode      : JadeXMLNode;
    attribute,
    newAttr        : JadeXMLAttribute;
begin
    if self.isKindOf(JadeXMLElement) then
        newNode := self.makePersistentIndividual(elmnt);
```

This method evaluates the transient object for which it is called and calls the **makePersistentIndividual** method, passing the user subclass that is used to create a persistent instance.

JadeXMLNode::makePersistentIndividual Method

This method actually creates the persistent JADE object based on the transient object created as a result of the parsing process. Once created, it is returned to the **makePersistent** method and assigned to the **newNode** variable.

```
xmlNode := copySelfAs(class, false).JadeXMLNode;
return xmlNode;
```

JadeXMLNode::makePersistent Method

Once the new persistent XML node has been created, it is necessary to continue traversing the transient structure to ensure the **childNodes** (and the **childNodes** of the **childNodes**, and so on) are processed.

```
foreach childNode in self.childNodes do
  newChildNode := childNode.makePersistent(elmnt, attr,
    comment, cdata, text, docType, pi, xmlDoc);
  newChildNode.parentNode := newNode;
endforeach;
foreach attribute in self.JadeXMLElement.attributes do
  newAttr := attribute.makePersistent(elmnt, attr, comment, cdata,
    text, docType, pi, xmlDoc).JadeXMLAttribute;
  newAttr.setElement(newNode.JadeXMLElement);
endforeach;
elseif self.isKindOf(JadeXMLAttribute) then
  newNode := self.makePersistentIndividual(attr);
```

It is also important to note that the **copySelfAs** method does not copy references and that collections are empty on the copy created. As a result, the **makePersistent** method then assigns values to the **parentNode**, **document**, **element**, and **docType** references, where appropriate.