XML Serialization of Java Objects

**2008**

PEG - STRIDEHub

Headstrong

11/21/2008



Table of Contents

[**1.** **Long Term Persistence of JavaBeans Components: XML Schema** 1](#_Toc215061353)

[**Basic Elements** 2](#_Toc215061354)

[**Objects** 2](#_Toc215061355)

[**Abbreviations** 5](#_Toc215061356)

[**The Top Level** 9](#_Toc215061357)

[**DTD** 10](#_Toc215061358)

[**2.** **XML Serialization of Java Objects** 10](#_Toc215061359)

1. **Long Term Persistence of JavaBeans Components: XML Schema**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| <http://java.sun.com/products/jfc/tsc/articles/persistence3/index.html> |  |  |  |  |  |
| |  |  |  | | --- | --- | --- | |  |  |  |   The persistence scheme added in v 1.4 uses instances of the XMLEncoder class to write out files representing JavaBeans components (beans). Every file written by XMLEncoder uses the same XML schema, regardless of the beans the file contains. In this document we describe this schema so that implementations other than XMLEncoder and its corresponding reader, XMLDecoder, can be used to write and read compatible files.  This document presents the basic elements of each XML archive, followed by the tags necessary to represent objects. Next comes a section of abbreviations -- tags that aren't strictly necessary to write out an XML archive, but that make the archive shorter and easier to read. The final sections describe the top level of the XML archive, which can refer to properties of the decoder, and give a DTD for the XML schema.  You can find an example XML archive here: [Browse.xml](http://java.sun.com/products/jfc/tsc/articles/persistence3/Browse.xml). This example is an archive of a simple application that accepts a URL and, using a JEditorPane, displays the HTML from that URL. You can read the archive and run the application using the following code:   |  | | --- | | try {  XMLDecoder d = new XMLDecoder(  new BufferedInputStream(  new FileInputStream("Browse.xml")));  d.readObject();  d.close();  } catch (IOException e) {  ...*handle the exception*...  } |     For a ready-made program that reads XML archives, see [TestInput.java](http://java.sun.com/products/jfc/tsc/articles/persistence3/TestInput.java). Sample scripts for running it on Win32 and UNIX are in [xml.bat](http://java.sun.com/products/jfc/tsc/articles/persistence3/xml.bat) and [xml](http://java.sun.com/products/jfc/tsc/articles/persistence3/xml), respectively.  **Basic Elements**  Each document begins with the optional XML preamble, specifying the version of the XML specification and encoding it uses. The preamble is followed by a single element with the <java> tag.  <?xml version="1.0" encoding="UTF-8" ?>  <java version="1.4.0" class="java.beans.XMLDecoder">  *...objects go here...*  </java>  The <java> element contains two informational attributes (not currently used by XMLDecoder): the version attribute, which records the version of the Java platform that was used to write the archive, and the class attribute, which specifies the class of the decoder for which the document was written. The objects that the archive contains make up the body of this element and appear in the order they will be returned by calls to the decoder's readObject method.  **Objects**  Objects are represented by the sequence of method calls that will be used to create them. Each element in the XML document represents a method call that either creates an object (an *expression*) or has a side effect on an object (a *statement*). Strings are treated as special kinds of expressions. Identifiers name objects so they can be referred to after their creation.  **Strings**  Strings are the atomic expressions of the XML document. The characters in a string form the body of an element with the <string> tag. For example, the string "Hello, World" is represented by the following XML code:  <string>Hello, World</string>  The '<' and '&' characters are represented by the &lt; and &amp; escape sequences.  **Expressions and Statements**  Recall that expressions are method calls that return a value, and statements are method calls that don't. Each expression can be represented by an element with the <object> tag; each statement is represented by the <void> tag. For both <object> and <void> tags, the method attribute specifies the name of the method to be called when the document is read.  The class attribute can be used in <object> tags to specify a class as the target of a static method. Constructors are represented as static methods that have the name new.  When an expression or statement contains expressions, the contained expressions are used as arguments to the method represented by the outer expression or statement. For example, to create an instance of the JButton class we can write the following:  <object class="javax.swing.JButton" method="new">    <string>Press me</string>  </object>  Because the enclosed element represents a string, which is an expression, the value "Press me" is used as an argument to the constructor of the JButton class. The equivalent code written in the Java programming language ("Java code") would be:  new JButton("Press me");  The default method name is new. It can therefore be omitted, yielding the equivalent:  <object class="javax.swing.JButton">    <string>Press me</string>  </object>  When statements appear inside the body of an expression, the statements are applied to the enclosing object after it is created. To execute a method with a side effect on an object we place the method inside the object to be affected. For example:   |  | | --- | | <object class="javax.swing.JButton">    <void method="setText">      <string>Hello, world</string>    </void>  </object> |     This corresponds to the following code fragment:  JButton b = new JButton();  b.setText("Hello, world");  If an expression should not be used as an argument to the enclosing method, it should be represented with the <void> tag. The result of an expression in a <void> tag is still evaluated and used by any objects it encloses.  When an expression contains <void> tags (whether they denote expressions or statements) without class attributes, those <void> tags must follow all other tags in the expression. Each non-<void> expression is evaluated and the enclosing method is called with the results as arguments. The <void>-tagged statements and expressions are then applied, in order, to the result.  For example, consider the following expression:   |  | | --- | | <object class="javax.swing.JButton">    <string>Press me</string>    <void method="setName">      <string>Greeting</string>    </void>  </object> |     It could be written as follows:  JButton button1 = new JButton("Press me");  button1.setName("Greeting");  The ability to nest expressions and statements greatly reduces the number of identifiers that are needed to represent a given graph.  **Identifiers**  When a graph contains cycles or multiple references to the same object, a name (an *identifier*) must be given to the object so that it can be referred to later. Identifiers are created using the id attribute, which binds a name to the expression value. The identifier has global scope extending from the last argument of the expression to the end of the file. In a streamed environment, the scope extends until the stream is flushed.  The following expression creates an identifier button1, bound to an instance of the JButton class:  <object id="button1" class="javax.swing.JButton"/>  Reference is made to named instances by using an idref attribute in an element with the <object> tag. The expression defining the identifier must precede any reference made to that identifier. The following expression makes reference to a previously defined instance button1:  <object idref="button1"/>  The expression   |  | | --- | | <object class="javax.swing.JPanel">    <void method="add">      <object id="button1" class="javax.swing.JButton"/>    </void>    <void method="add">      <object class="javax.swing.JLabel">        <void method="setLabelFor">          <object idref="button1"/>        </void>      </object>    </void>  </object> |     corresponds to the following Java code:   |  | | --- | | JPanel panel1 = new JPanel();  JButton button1 = new JButton();  JLabel label1 = new JLabel();  panel1.add(button1);  panel1.add(label1);  label1.setLabelFor(button1); |     The id attribute can be used with a <void> tag when the tag denotes an expression, rather than a statement. This allows you to make a reference to the result of an expression without contributing to the arguments of the enclosing method.  For example, consider the following fragment:  <object class="java.util.Date">    <void id="now" method="getTime"/>  </object>  It allows an expression to be evaluated in the context of the enclosing instance, in this case defining the variable now as the value of the expression. It corresponds to the following Java code:  long now = new Date().getTime();  **Abbreviations**  The preceding information is all you need to be able to write XML archives readable by XMLDecoder. To read all archives produced by XMLEncoder, however, you need to know about the abbreviations for primitives, null, Class objects, static constants, properties, indexes, and arrays.  **Primitives**  The eight primitive data types of the Java platform can be used interchangeably with their corresponding wrapper classes in archives. Abbreviations exist for all the wrapper classes, as elements whose tag name is the name of the primitive type and whose body is the string representation of the value, as specified by the toString method of the wrapper class. Except for char, the body of the expression is supplied to the constructor of the corresponding wrapper that takes a single, String argument. The char type is handled as a special case since no string constructor exists in its wrapper class, Character.  The following tags represent both the primitive types and their corresponding wrapper classes:   * <boolean> * <byte> * <char> * <short> * <int> * <long> * <float> * <double>   For example, the expression  <object class="java.lang.Integer">    <string>123</string>  </object>  is shortened to  <int>123</int>  which might represent either of the following Java code fragments:  new Integer("123")  123  **Null**  To specify null, use the <null> tag. A null element has no attributes and contains no other tags. For example:  <null/>  **Class Objects**  The <class> tag can be used to represent an instance of Class. For example,  <object class="java.lang.Class method="forName">     <string>java.awt.event.ActionListener</string>  </object>  is shortened to  <class>java.awt.event.ActionListener</class>  which is equivalent to ActionListener.class.  **Static Constants  (only in releases after 1.4.0 beta)**  As of the release following 1.4.0 beta, the values of static constants may be written using the class and field attributes to specify the declaring class and field name of the constant, respectively. Thus   |  | | --- | | <void class="javax.swing.JTable" method="getField">  <string>AUTO\_RESIZE\_OFF</string>  <void id="Integer0" method="get">  <null/>  </void>  </void>  <object idref="Integer0"/> |     is shortened to  <object class="javax.swing.JTable" field="AUTO\_RESIZE\_OFF"/>  which represents JTable.AUTO\_RESIZE\_OFF.  **Properties**  Method names beginning with get and set can be written using the property attribute instead of the method attribute.  For expressions with methods whose names begin with "get", the property name is the method name with "get" removed and the next letter made lowercase. Thus  <void method="getText"/>  is shortened to:  <void property="text"/>  For statements with methods whose names begin with "set", the property name is derived in a similar way. Thus  <void method="setText">    <string>Hello, world</string>  </void>  is shortened to:  <void property="text">    <string>Hello, world</string>  </void>  **Indexes**  Methods named exactly get or set, as defined in the [java.util.List](http://java.sun.com/javase/6/docs/api/java/util/List.html) interface, can be written using the index attribute instead of the method attribute.  For expressions with the method name get, the value of the index attribute is used as the argument. Thus  <void method="get">    <int>3</int>  <void>  is shortened to  <void index="3"/>  which corresponds to the following Java code:  Object o = aList.get(3);  For statements with the method name set, the value of the index attribute is prepended to the arguments of the enclosed body. Thus  <void index="3">    <string>Hello, world</string>  </void>  is equivalent to  <void method="set">    <int>3</int>    <string>Hello, world</string>  </void>  which corresponds to the following Java code:  aList.set(3, "Hello, world")  **Arrays**  The <array> tag can be used to represent arrays. The class and length attributes denote the subtype and the length of the array respectively. You can use an id attribute to name the array. Here is an example of using an <array> tag:  <array class="java.awt.Component" length="3"/>  It corresponds to the following Java code:  Component[] a = new Component[3];  The set and get methods, as defined in the java.util.List interface, can be used as if they could be applied to array instances. The index attribute can thus be used with arrays.  Thus the expression  <array class="java.lang.String" length="3">    <void index="1">      <string>Hello, world</string>    </void>  </array>  is equivalent to the following:  String[] s = new String[3];  s[1] = "Hello, world";  ***After the 1.4.0 beta release***, you can omit the length attribute from an <array> tag and specify the values of entries directly, without using void tags. The length of the array is equal to the number of values specified. For example,  <array class="int">  <int>123</int>  <int>456</int>  </array>  represents the following Java code fragment:  int[] intArray = {123, 456};  **The Top Level**  Each element that appears in the body of the outermost (<java>) element is evaluated in the context of the decoder itself. Typically this outer context is used to retrieve the *owner* of the decoder, which can be set before reading the archive. (See the API documentation of [XMLDecoder](http://java.sun.com/javase/6/docs/api/java/beans/XMLDecoder.html) and [XMLEncoder](http://java.sun.com/javase/6/docs/api/java/beans/XMLEncoder.html) for details of the setOwner and getOwner methods.) The owner is a property of the decoder and can be accessed in the usual way:   |  | | --- | | <?xml version="1.0" encoding="UTF-8" ?>  <java version="1.4.0" class="java.beans.XMLDecoder">    <void id="myController" property="owner"/>  *...objects go here...*  </java> |     The myController identifier can then by used throughout the body of the document to refer to the owner of the decoder. The following XML code creates a button that calls a no-argument doIt method on the owner when the button is pressed:   |  | | --- | | <?xml version="1.0" encoding="UTF-8" ?>  <java version="1.4.0" class="java.beans.XMLDecoder">    <void id="myController" property="owner"/>    <object class="javax.swing.JButton">      <void method="addActionListener">        <object class="java.beans.EventHandler" method="create">          <class>java.awt.event.ActionListener</class>          <object idref="myController"/>          <string>doIt</string>        </object>      </void>    </object>  </java> |     It is also possible to use the top-level environment to produce side effects on the owner. Typically this is used to set property values on the owner to supply it with references to parts of a user interface so that the owner can manipulate the UI programatically. The following XML code creates a button and assigns it to the quitButton property of the owner by calling the setQuitButton method on the owner when the file is read.   |  | | --- | | <?xml version="1.0" encoding="UTF-8" ?>  <java version="1.4.0" class="java.beans.XMLDecoder">    <void property="owner">      <void property="quitButton">         <object class="javax.swing.JButton"/>      </void>    </void>  </java> |     **DTD**  This document described both the syntax and semantics of the XML schema used by XMLEncoder for saving archives of beans. A DTD that describes the syntax of the schema is in the file [javabeans.dtd](http://java.sun.com/products/jfc/tsc/articles/persistence3/javabeans.dtd). | | | | | | |

1. **XML Serialization of Java Objects**

<http://java.sys-con.com/node/37550>

XML Serialization of Java Objects

By: [Joe Winchester](http://java.sys-con.com/author/1111); [Philip Milne](http://java.sys-con.com/author/1112)

Jun. 1, 2003 12:00 AM

*Java serialization was initially used to support remote method invocation (RMI), allowing argument objects to be passed between two virtual machines.*

RMI works best when the two VMs contain compatible versions of the class being transmitted, and can reliably transmit a binary representation of the object based on its internal state. When an object is serialized, it must also serialize the objects to which its fields refer - resulting in what is commonly called an object graph of connected components. Although the transient keyword can be used to control the extent to which the serialization process penetrates the object graph, this level of control is seldom enough.

Many have tried to use Java's serialization to achieve the so-called "long-term persistence" of data - where the serialized form of a Java data structure is written to a file for later use. One such area is the development tools domain, in which designs must be saved for later use. Because the logic that saves and restores serialized objects is based on the internal structure of the constituent classes, any changes to those classes between the time that the object was saved and when it was retrieved may cause the deserialization process to fail outright; for example, a field was added or removed, existing fields were renamed or reordered, or the class's superclass or package was altered. Such changes are to be expected during the development process, and any mechanism that relies on the internal structure of all classes being identical between versions to work has the odds stacked against it. Over the last few years the "versioning issues" associated with Java's serialization mechanism have indeed proved to be insurmountable and have led to widespread abandonment of Java's serialization as a viable long-term persistence strategy in the development tools space.

To tackle Java serialization problems, a Java Specification Request (JSR 57) was created, titled "Long-Term Persistence for JavaBeans." JSR 57 is included in JRE 1.4 and is part of the "java.beans" package. This article describes the mechanism with which the JSR solved the problems of long-term persistence, and how you can take control of the way that the XMLEncoder generates archives to represent the data in your application.

We'll start our section by dispelling two popular myths that have grown up around XML serialization: that it can only be used for JavaBeans and that all JavaBeans are GUI widgets. In fact, the XMLEncoder can support any public Java class; these classes don't have to be JavaBeans and they certainly don't have to be GUI widgets. The only constraint that the encoder places on the classes it can archive is that there must be a means to create and configure each instance through public method calls. If the class implements the getter/setter paradigm of the JavaBeans specification, the encoder can acheive its goal automatically - even for a class it knows nothing about. On top of this default behavior, the XMLEncoder comes with a small but very powerful API that allows it to be "taught" how to save instances of any class - even if they don't use any of the JavaBeans design patterns. In fact, most of the Swing classes deviate from the JavaBeans specification in some way and yet the XMLEncoder handles them via a set of rules with which it comes preconfigured. The XMLEcoder is currently spec'ed to provide automatic support for all subclasses of Component in the SDK and all of their property types (recursively). This means that as well as being able to serialize all of AWT and Swing GUI widgets, the XMLEncoder can also serialize: primitive values (int, double, etc.), strings, dates, arrays, lists, hashtables (including all Collection classes), and many other classes that you might not think of as having anything to do with JavaBeans. The support for all these classes is not "hard-wired" into the XMLEncoder; instead it is provided to the Encoder through the API that it exposes for general use. The variety in the APIs among even the small subset of classes mentioned earlier should give some idea of the generality and scope of the persistence techniques we will cover in the next sections.

**Background**  
When problems are encountered with an object stream, they're hard to correct because the format is binary. An XML document is human readable, and therefore easier for a user to examine and manipulate when problems arise. To serialize objects to an XML document, use the class java.beans.XMLEncoder; to read objects, use the class java.beans.XMLDecoder.

One reason object streams are brittle is that they rely on the internal shape of the class remaining unchanged between encoding and decoding. The XMLEncoder takes a completely different approach here: instead of storing a bit-wise representation of the field values that make up an object's state, the XMLEncoder stores the steps necessary to create the object through its public API. There are two key factors that make XML files written this way remarkably robust when compared with their serialized counterparts.

First, many changes to a class's internal implementation can be made while preserving backward compatibility in its public APIs. In public libraries, this is often a requirement of new releases - as breaking a committed public API would break all the third-party code that had used the library in its older form. As a result of this, many software vendors have internal policies that prevent its developers from knowingly "breaking" any of the public APIs in new releases. While exceptions inevitably arise, they are on a much, much smaller scale than the internal changes that are made to the private implementations of the classes within the library. In this way, the XMLDecoder derives much of its resilience to versioning by aligning its requirements with those of developers who program against APIs directly.

The second reason for the stability of the decoding process as implemented by the XMLDecoder is just as important. If you were to take an instance of any class, choose an arbitrary member variable, and set it to null - the behavior of that instance would be completely undefined in all subsequent operations - and a bug-free implementation would be entitled to fail catastrophically under these circumstances. This is exactly what happens when a field is added to a new version of a class and this causes people to cross their fingers when trying to deserialize an instance of a class that was written out with an older version. The XMLEncoder, by contrast, doesn't store a list of private fields but a program that represents the object's state. Here's an XML file representing a window with the title "Test":

<?xml version="1.0" encoding="UTF-8"?>  
<java version="1.4.1" class="java.beans.XMLDecoder">  
<object class="javax.swing.JFrame">  
<void property="title">  
<string>Test</string>  
</void>  
<void property="visible">  
<boolean>true<boolean/>  
</void>  
</object>  
</java>

XML archives, written by XMLEncoder, have exactly the same information as a Java program - they're just written using an XML encoding rather than a Java one. Here's what the above program would look like in Java:

JFrame f = new JFrame();  
f.setTitle("Test");  
f.setVisible(true);

When a backward compatibility issue arises in one of the classes in the archive, it may cause one of the earlier statements to fail. A new version of the class might, for example, choose not to define the "setTitle()" method. When this happens, the XMLDecoder detects that this method is now missing from the class and doesn't try to call it. Instead, it issues a warning, ignores the offending statement, and continues with the other statements in the file. The critical point is that not calling the "setTitle()" method does not violate the contract of the implementation (as deleting an instance variable would), and the resulting instance should be a valid and fully functional Java object. If the resulting Java object fails in any way, an ordinary Java program could be written against its API to demonstrate a genuine bug in its implementation.

The vendors of popular Java libraries tend to devote significant resources toward programs to manage demonstrable bugs of this kind and enlist the support of the development community to work toward their eradication - Sun's "BugParade" is a well-known example. As a result of these kinds of programs, bugs that can be demonstrated by simple "setup code" tend to be rare in mature libraries. Once again, the XMLDecoder benefits here as it's able to ride on the coattails of the Java developer by using the public APIs of the classes instead of relying on special privileges to circumvent them.

**Encoding of JavaBeans**  
To illustrate the XMLEncoder, this article shows serialization based on a number of scenarios using an example Person class. These range from simple JavaBeans encoding through nondefault construction and custom initialization.

In the simplest scenario, the class Person has String fields for firstName and lastName, together with get and set methods.

public class Person {  
private String firstName;  
private String lastName;  
public String getFirstName() { return firstName; }  
public String getLastName() { return lastName; }  
public void setFirstName(String str) { firstName = str; }  
public void setLastName(String str) { lastName = str; }  
}

The following code creates an encoder and serializes a Person.

FileOutputStream os = new FileOutputStream("C:/cust.xml");  
XMLEncoder encoder = new XMLEncoder(os);  
Person p = new Person();  
p.setFirstName("John");  
encoder.writeObject(p);  
encoder.close();

The XML file created shows that Person class has been encoded, and that its firstName property is the string "John".

<?xml version="1.0" encoding="UTF-8"?>  
<java version="1.4.1" class="java.beans.XMLDecoder">  
<object class="Person">  
<void property="firstName">  
<string>John</string>  
</void>  
</object>  
</java>

When the file is decoded with the XMLDecoder, the Person class will be instantiated with its default constructor, and the firstName property set by calling the method setFirstName("John").

FileInputStream os = new FileInputStream("C:/cust.xml");  
XMLDecoder decoder = new XMLDecoder(os);  
Person p = (Person)decoder.readObject();  
decoder.close();

To understand how to leverage the encoder and decoder for custom serialization requires an understanding of the JavaBeans component model. This describes a class's interface in terms of a set of properties, each of which can have a get and set method. To determine the set of operations required to re-create an object, the XMLEncoder creates a prototype instance using its default constructor and then compares the value of each property between this and the object being serialized. If any of the values don't match, the encoder adds it to the graph of objects to be serialized, and so on until it has a complete set of the objects and properties required to re-create the original object being serialized. When the encoder reaches objects that can't be broken down any further, such as Java's strings, ints, or doubles, it writes these values directly to the XML document as tag values. For a complete list of these primitive values and their associated tags, see [http://java.sun.com/products/jfc/tsc/ articles/persistence3/index.html](http://java.sun.com/products/jfc/tsc/articles/persistence3/index.html).

To serialize an object, XMLEncoder uses the Strategy pattern, and delegates the logic to an instance of java.beans.PersistenceDelegate. The persistence delegate is given the object being serialized and is responsible for determining which API methods can be used to re-create the same instance in the VM in which it will be decoded. The XMLEncoder then executes the API to create the prototype instance that it gives to the delegate, together with the original object being serialized, so the delegate can determine the API methods to re-create the nondefault state.

The method XMLEncoder.setPersistenceDelegate(Class objectClass, PersistenceDelegate delegate) is used to set a customized delegate for an object class. To illustrate this we'll change the original Person class so that it no longer conforms to the standard JavaBeans model, and show how persistence delegates can be used to teach the XMLEncoder to successfully serialize each instance.

**Constructor Arguments**  
One of the patterns that can be taught to the XMLEncoder is how to create an instance where there is no zero-argument constructor. The following is an example of this in which a Person must be constructed with its firstName and lastName as arguments.

public Person(String aFirstName, String aLastName){  
firstName = aFirstName;  
firstName = aLastName;  
}

In the absence of any customized delegate, the XMLEncoder uses the class java.beans.DefaultPersistenceDelegate. This expects the instance to conform to the JavaBeans component model with a zero-argument constructor and JavaBeans properties controlling its state. For the Person whose property values are supplied as constructor arguments, an instance of DefaultPersistenceDelegate can be created with the list of property names that represent the constructor arguments.

XMLEncoder e = new XMLEncoder(os);  
Person p = new Person("John","Smith");  
e.setPersistenceDelegate(Person.class,  
new DefaultPersistenceDelegate(  
new String[] { "firstName","lastName"}  
);  
e.writeObject(person);

When the XMLEncoder creates the XML for the Person object, it uses the supplied instance of the DefaultPersistenceDelegate, queries the values of the firstName and lastProperties, and creates the following XML document.

<?xml version="1.0" encoding="UTF-8"?>  
<java version="1.4.1" class="java.beans.XMLDecoder">  
<object class="Person">  
<string>John</string>  
<string>Smith</string>  
</object>  
</java>

The result is a record of the Object's state but written in such a way that the XMLDecoder can locate and call the public constructor of the Person object just as a Java program would. In the previous XML document where the Person was a standard JavaBeans component, the nondefault properties were specified with named <void property="propertyName"> tags that contained the argument values.

Although custom encoding rules can be supplied to the XMLEncoder, this is not true of the XMLDecoder. The XML document represents the API steps to re-create the serialized objects in a target VM. One advantage of not having custom decoder rules is that only the environment that serializes the objects requires customization, whereas the target environment just requires the classes with unchanged APIs. This makes it ideal for the following scenario - serialization of an object graph within a development tool that has access to design-time customization, where the XML document will be read in a runtime environment that does not have access to the persistence delegates used during encoding.

**Custom Instantiation**  
In addition to a class being constructed with property values as arguments, custom instantiation can include use of factory methods. An example of this would be if Person's constructor were package protected and instances of the Person class could only be created by calling a static createPerson() method defined in a PersonFactory class.

To write a persistence delegate requires a basic understanding of how the encoder creates its set of operations that will re-create the serialized objects when the stream is deserialized. The XMLEncoder uses the command pattern to record each of the required method calls as instances of the class java.beans.Statement. Each Statement represents an API call in which a method is sent to a target, together with any arguments. Commands that are responsible for the instantiation of objects are instances of java.beans.Expression. A subclass of Statement returns a value. Each object in the graph is represented by the Expression that creates it and a set of Statements that are used to initialize it.

For general control of instantiation, a subclass of the PersistenceDelegate class should be created with a specialized instantiate() method. The return value is the java.beans.Expression that indicates to the encoder which method or constructor should be used to create (or retrieve) the object. The returned Expression includes the object, the target (normally the class that defines the constructor), the method name (normally the fake name "new," which indicates a constructor call), and the argument values that the method or constructor takes.

The first argument of the instantiate() method is the instance of the Person object being serialized, and the second object is the encoder (see Listing 1).

When the XMLEncoder serializes the Person instance, instead of the DefaultPersistenceDelegate that uses standard JavaBeans rules for properties, it uses the anonymous inner class we registered as the persistence delegate of the Person.class. The resulting XML follows. In the <object> tag as well as the class name, the static method createPerson has also been included, and the arguments are specified as child tags.

<?xml version="1.0" encoding="UTF-8"?>  
<java version="1.4.1" class="java.beans.XMLDecoder">  
<object class="PersonFactory" method="createPerson">  
<string>Smith</string>  
<void property="firstName">  
<string>John</string>  
</void>  
</object>  
</java>

The inner class created for the Person persistence delegate subclasses from DefaultPersistenceDelegate, so the firstName property value of "John" is included in the XML document; however, no property tag is included for lastName. This is because the XMLEncoder compares the prototype instance of Person against the instance being serialized to determine which property values are not their default and need to be included in the XML document. The method that does this is protected void initialize(Class type, Object oldInstance, Object newInstance, Encoder out). The oldInstance argument is the object being serialized and the newInstance is the prototype. Because the prototype instance is created using the Expression returned by the persistence delegate's method protected Expression instantiate(Object oldInstance, Encoder encoder), the newInstance argument will already have the lastName set to be the same as the oldInstance so the encoder won't see their values as different and hence it does not serialize a property value for the lastName.

**Custom State**  
The DefaultPersistenceDelegate assumes that the state of the oldInstance can be determined and restored by using the JavaBeans component model for properties. The list of properties for a class is retrieved using the method java.beans.Introspector.getBeanInfo(ClassaClass).getPropertyDescriptors(). Each property is an instance of java.beans.PropertyDescriptor and includes a get and set method. The Introspector uses a set of rules matching method name pairs to create properties, although these rules can be overridden by supplying a specific BeanInfo class. The BeanInfo class can use a different set of methods than those that the introspector would otherwise have determined as the property's get and set method. However, it can't deal with scenarios in which there is no get and set method, for example. For these the persistence delegate needs to be customized, and as an example we will have a property called nicknames that is multivalued.

private List nicknames = new ArrayList();  
public void addNickname(String name){nicknames.add(name); }  
public List getNicknames(){return nicknames; }

Nicknames are added to the class one at a time using the addNickname() method, and the complete list is retrieved using getNicknames(). The decoder needs to iterate through the nicknames and create an archive that uses the addNickname() method to re-create the Person.

The persistence delegate will subclass DefaultPersistenceDelegate that assumes construction of the class through a default Person, and will override the instantiate() method that's responsible for determining the expressions required to re-create the oldInstance (see Listing 2).

The persistence delegate iterates through the nicknames and for each one adds a statement to the encoder that specifies the API to re-create the nickname. For this the Statement includes the target of the method (the Person oldInstance), the method name (addNickname), and the arguments (the nickname) (see Listing 3).

**Specifying Delegates in BeanInfo Classes**  
In the examples used so far the custom persistence delegate was set directly onto the XMLEncoder by calling the method setPersistenceDelegate(Class,PersistenceDelegate). This works if you're the author of the code that's responsible for performing the serialization, but in some scenarios another piece of software such as an IDE tool is responsible for encoding the JavaBeans. In this situation you must teach the tool about the delegate that it should use for your class; this is done by specifying the delegate class name in the BeanDescriptor for a string key of "persistenceDelegate". For example, if the Person class is going to be introduced into an IDE together with PersonBeanInfo, the getBeanDescriptor() method would be specialized.

public class PersonBeanInfo extends SimpleBeanInfo {  
public BeanDescriptor getBeanDescriptor(){  
BeanDescriptor result = new BeanDescriptor(Person.class);  
result.setValue("persistenceDelegate", PersonPersistenceDelegate.class);  
return result;  
}  
}

If the PersonBeanInfo is not in the same package as the Person class, the search path of the Introspector in the tool will need to be updated to include the BeanInfo's package.

Another way in which BeanInfo classes can be used to leverage persistence is by marking properties as transient. When DefaultPersistenceDelegate is responsible for encoding the JavaBean, it looks at all the available read/write properties and compares the existing values on the object being serialized against the values on the prototype instance. To flag a property so that it will be ignored, the key "transient" should be set to the value Boolean.TRUE. For example, if the "firstName" property should be considered transient, the getPropertyDescriptors() method on PersonBeanInfo could be specialized as shown in Listing 4.

**Conclusion**  
This article explained how the design of the XMLEncoder avoids many of the fundamental pitfalls of binary serialization and makes the case that XML archives produced by the XMLEncoder can be trusted as a reliable means to store valuable data over the long term. Central to the design of the XMLEncoder is the java.beans.DefaultPersistenceDelegate class, which provides a default serialization strategy based on the idea of properties as laid out in a JavaBeans component model.

We show how custom delegates can be submitted to the encoder to teach it about idioms other than those of the JavaBeans component model, so classes that don't follow the JavaBeans conventions can be accommodated without changing their APIs. Because, in all cases, the decoder inflates object graphs using public API calls; deserialization is remarkably robust in the face of changes made to the classes referred to in the archives. If you need to save some critical data in your application to a file and are not interested in designing a new file format and coding the readers and writers for it - check out the XMLEncoder/XMLDecoder to see if they'll do it all for you.

**References**

 Using XML Encoder on the Swing Connection: [http://java.sun.com/products/jfc/tsc/ articles/persistence4/index.html](http://java.sun.com/products/jfc/tsc/articles/persistence4/index.html)

 JavaBeans: <http://java.sun.com/products/javabeans/>