Inside WebObjects

XML Serialization

Preliminary

November 2002

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Listings, and Tables

About This Document

This document explains how to you can use XML serialization in your applications. Binary serialization is a simple and efficient way of serializing data. To see this appealing format, invoke the `cat` command on an executable file. Binary is a wonderful format for computers to use, but it's not easy for people to understand. XML (Extensible Markup Language) is a specification that defines a format that can be used to represent data in a way that is more understandable to human beings than binary.

The Java language has an excellent API for binary serialization. WebObjects extends that API to provide you with XML serialization.

Encoding objects and data into XML documents allows you to easily view and modify objects and data in their serialized form. It also lets you share information between applications, systems, and even organizations using a standard format. In addition, when receiving streams of serialized data over the Internet, you may want to make sure that the document is valid before deserializing it. XML serialization provides you with facilities to accomplish this.

What You Should Already Know

This document assumes that you are familiar with XML, binary serialization in Java, and Sun's security manager. If you plan on using the XSLT processor included with the product or one of your own, you should have enough knowledge of Extensible Stylesheet Language Transformations (XSLT) to develop XSLT stylesheets. "Additional Resources" (page 13) provides a list of resources that get you started in Java binary serialization and XSLT.

You should also have experience developing WebObjects applications. In particular, you need to know how to create applications using Project Builder (the project-management tool of WebObjects). See "Related Documentation" (page 13) for a list of documents that address this and other essential subjects.

Contents

The document contains the following chapters and appendixes:

- Chapter 2, "Introduction to XML Serialization" (page 15), provides you with an overview of XML, XML Schema files, document type definition (DTD) files, XML namespaces, and XSLT.
- Chapter 3, "XML Serialization Essentials" (page 23), explains XML serialization in WebObjects. In particular, you learn about the API used to serialize and deserialize objects and data, security, and versioning.
- Chapter 4, "Serializing Objects and Data" (page 37), walks you through the creation of a project that implements both binary and XML serialization.
- Chapter 5, "Transformation of XML Documents" (page 67), explains the process of transforming XML documents using an XSLT processor. It also contains details on how you can use your favorite XML parser and transformer in WebObjects applications and some performance issues to keep in mind when serializing and transforming data.
- Chapter 6, "Transforming XML Documents" (page 75), expands the project introduced in "Serializing Objects and Data" (page 37) by adding transformation of serialized data.
- Appendix A, "XML Schema and DTD Files" (page 85), contains listings of the XML Schema and DTD files that define the format of XML documents that represent serialized data.
- Appendix B, "Code Listings" (page 109), contains listings of example classes and the XSLT script introduced in "Transforming XML Documents" (page 75).

This document also contains a glossary of terms and an index.

Chapter 4, "Serializing Objects and Data" (page 37), and Chapter 6, "Transforming XML Documents" (page 75), walk you through developing applications that use binary and XML serialization. The projects created in those chapters are included in `/Developer/Documentation/WebObjects/XML_Serialization/Projects`. As a companion to the document, there is a compressed version of the projects at http://developer.apple.com/techpubs/webobjects/webobjects.html.

Related Documentation

If you need to learn the basics about developing WebObjects applications, you can find that information in the following documents:

- *Inside WebObjects: WebObjects Overview* provides you with a survey of WebObjects technologies and capabilities.
- *Inside WebObjects: Developing Web Applications* shows you how to develop HTML-based applications with WebObjects.
- *Inside WebObjects: Java Desktop Applications* explains how to develop Swing-based applications with WebObjects.

For additional WebObjects documentation and links to other resources, visit http://developer.apple.com/webobjects.

Additional Resources

In addition to WebObjects development experience, you also need to be acquainted with the Java binary serialization API and XML.

The following resources provide information on serialization and XML:

- *Java Object Serialization Specification* (http://java.sun.com/products/jdk/1.3/docs/guide/serialization/spec/serialTOC.doc.html)
- "Advanced Object Serialization" (http://developer.java.sun.com/developer/technicalArticles/ALT/index.html)

- *Java and XML* (published by O'Reilly)
- *XSLT* (published by O'Reilly)
- *XSLT Programmer's Reference* (published by Wrox Press Ltd.)

Other related resources:

- *JAXP Tutorial* (http://java.sun.com/xml/jaxp/docs.html)
- http://xml.apache.org contains information on the Apache Xerces XML parser and the Apache Xalan XSLT processor.
- *XSL Transformations (XSLT) Version 1.0* (http://www.w3.org/TR/xslt)
- *Working With XML* (http://java.sun.com/xml/tutorial_intro.html)
- *XML From the Inside Out* (http://xml.com) is a great resource of XML-related information.
- *XML Schema* (http://www.w3.org/XML/Schema)
- Mulberry Technologies, Inc. (http://www.mulberrytech.com)
- *Security in Java 2 SDK 1.2* (http://java.sun.com/docs/books/tutorial/index.html)

CHAPTER 2

Introduction to XML Serialization

If you plan to have your applications exchange data with other applications over the Internet, you will most probably use **Extensible Markup Language (XML)**, either because of its flexibility or because of its widespread use in Internet applications. XML is a text-based markup language based on **Standard Generalized Markup Language (SGML)** and it's used mostly to represent structured data. XML is similar to HTML, but has stricter rules regarding the form and validity of documents.

This chapter contains the following sections:

- "XML Documents" (page 15) describes two essential concepts related to XML documents: being well formed and being valid.
- "XML Namespaces" (page 18) explains how XML namespaces help to differentiate elements that have identical names but are in different contexts.
- "Benefits of XML Serialization" (page 21) lists a few benefits that XML serialization provides to your applications.
- "Transforming XML Documents" (page 22) explains what it means to transform an XML document and why you would want to do it.

XML Documents

To be usable, XML documents must be *well formed*. Well-formed documents have open and close tags for all their elements (in the correct sequence) and contain one root element. In addition, XML documents must have at least one XML declaration, an element that provides XML parsers with essential information needed to process a document.

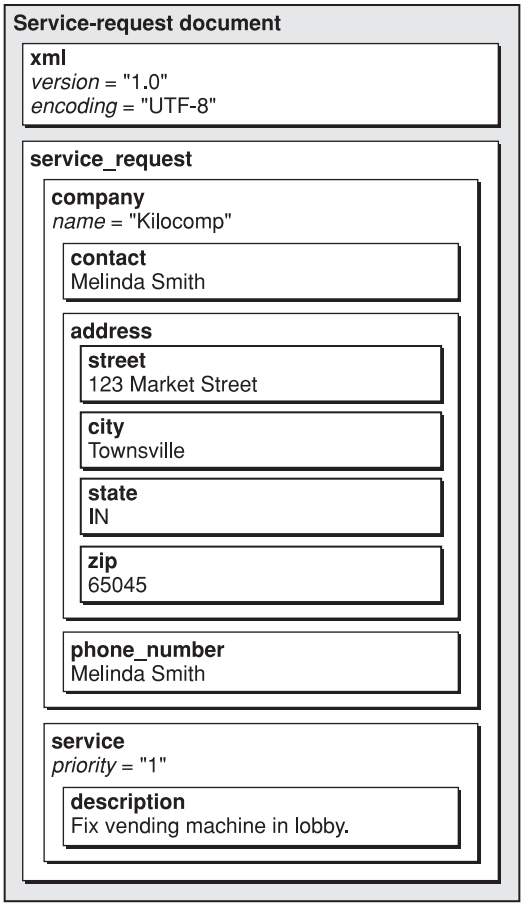
Listing 2-1 shows an example of an XML document.

Listing 2-1 Service-request document

```
<?xml version="1.0" encoding="UTF-8"?>
<service_request>
    <company name="Kilocomp">
        <contact>Melinda Smith</contact>
        <address>
            <street>123 Market Street</street>
            <city>Townsville</city>
            <state>IN</state>
            <zip>65045</zip>
        </address>
        <phone_number>345-555-1234</phone_number>
    </company>
    <service priority="1">
        <description>Fix vending machine in lobby.</description>
    </service>
</service_request>
```

The XML declaration of the service-request document indicates that the document is written using the XML 1.0 standard and the UTF-8 character encoding. The `service_request` element is the root element; it encloses the data the document contains. `service_request` contains two elements: `company` and `service`. The `company` element contains one attribute, `name`, and three elements: `contact`, `address`, and `phone_number`. The service element has one attribute, `priority`, and one element, `description`. The address element contains four elements, `street`, `city`, `state`, and `zip`; it has no attributes. Figure 2-1 provides a graphical representation of the service-request document.

Figure 2-1 Graphical representation of the service-request document



Note: As a general rule, XML elements describe proper content data (for example, a phone number or an inventory item), whereas XML attributes describe metadata, such as priority or ID number.

To be usable in a particular context, an XML document must be well formed and *valid*. A valid document is one that follows the structure specified by a schema file, which can be either a **document type definition (DTD)** file or an **XML Schema** file. The schema determines the layout of an XML document's elements, the attributes and subelements that each can have, and the constraints that the attribute and

element data must adhere to. XML Schema filenames usually have the `.xsd` extension, while DTD filenames usually have the `.dtd` extension. You can think of a schema as a Java class and an XML document as an instance of the schema. For more information on document schemas, see *XML Schema* at http://www.w3.org/XML/Schema.

Document type definition (DTD) files can also be used to validate XML documents. However, because DTD files are not written in XML and are not as powerful as XML Schema files, XML Schema files are increasingly taking their place.

XML Namespaces

With the interoperability of XML documents comes the problem of differentiating between the element names you use in your documents and the names used in documents from other sources. Take a look at the document in Listing 2-2.

Listing 2-2 Service-response document

```
<?xml version="1.0" encoding="UTF-8"?>
<service_response>
    <service_request>
        <company name="Kilocomp">                                          //1
            <contact>Melinda Smith</contact>
            <address>
                <street>123 Market Street</street>
                <city>Townsville</city>
                <state>IN</state>
                <zip>65045</zip>
            </address>
            <phone_number>345-555-1234</phone_number>
        </company>
        <service priority="1">
            <description>Fix vending machine in lobby.</description>
        </service>
    </service_request>
    <appointment>
```

```
        <company>We Fix It</company>                                    //2
        <contact name="Nancy Garcia" phone="345-555-2334"
                 pager="345-555-1112" />
        <date>2002-05-02</date>
        <time>1500</time>
    </appointment>
</service_response>
```

Unless you add information about the element hierarchy of the document to your logic, it's difficult to differentiate between the `company` element of the `service_request` element (the line numbered 1) and the `company` element of the `appointment` element (2). This is where **XML namespaces** provide a great deal of assistance.

A namespace is like a Java package: It's a way of grouping related elements. Listing 2-3 shows a version of the service-response document that uses namespaces. Observe that the document has two distinct elements that enclose information about a company: `client:company` and `provider:company`. The prefixes tell you the category of each element.

To avoid having to put prefixes on all element names and to reduce the size of XML documents, you can define a default namespace for the document. By not including a prefix in the namespace definition, the line numbered 1 of Listing 2-4 defines a default namespace for the `service_response` element and the subelements of `service_response` that do not themselves define a namespace, such as `appointment` starting at the line numbered 2. You can find more information on XML namespaces in *Namespaces in XML*, located at http://www.w3.org/TR/REC-xml-names.

Listing 2-3 Service-response document using namespaces

```
<?xml version="1.0" encoding="UTF-8"?>
<provider:service_response xmlns:provider="http://provider.com/b_to_b">
    <client:service_request xmlns:client="http://client.com/svcs">
        <client:company name="Kilocomp">                                //1
            <client:contact>Melinda Smith</client:contact>
            <client:address>
                <client:street>123 Market Street</client:street>
                <client:city>Townsville</client:city>
                <client:state>IN</client:state>
                <client:zip>65045</client:zip>
```

```
            </client:address>
            <client:phone_number>345-555-1234</client:phone_number>
        </client:company>
        <client:service priority="1">
            <client:description>Fix vending machine in lobby.</client:description>
        </client:service>
    </client:service_request>
    <provider:appointment>
        <provider:company>We Fix It</provider:company>                              //2
        <provider:contact name="Nancy Garcia" phone="345-555-2334" pager="345-555-1112"
/>
        <provider:date>2002-05-02</provider:date>
        <provider:time>1500</provider:time>
    </provider:appointment>
</provider:service_response>
```

Listing 2-4 Service-response document using a default namespace for the provider entity

```
<?xml version="1.0" encoding="UTF-8"?>
<service_response xmlns="http://provider.com/b_to_b">                               //1
    <client:service_request xmlns:client="http://client.com/svcs">
        <client:company name="Kilocomp">
            <client:contact>Melinda Smith</client:contact>
            <client:address>
                <client:street>123 Market Street</client:street>
                <client:city>Townsville</client:city>
                <client:state>IN</client:state>
                <client:zip>65045</client:zip>
            </client:address>
            <client:phone_number>345-555-1234</client:phone_number>
        </client:company>
        <client:service priority="1">
            <client:description>Fix vending machine in lobby.</client:description>
        </client:service>
    </client:service_request>
    <appointment>                                                                   //2
        <company>We Fix It</company>
        <contact name="Nancy Garcia" phone="345-555-2334"
                 pager="345-555-1112" />
```

```
        <date>2002-05-02</date>
        <time>1500</time>
    </appointment>
</service_response>
```

Benefits of XML Serialization

There are many benefits of using XML to encode data, including the ability to read and modify serialized or archived information easily. Java provides a great binary serialization API. WebObjects XML serialization leverages this well-known API to allow you to easily serialize your objects and data into XML documents. For more information on XML, visit http://www.w3.org/XML.

Serializing data into XML documents provides you with several benefits:

- **Long-term persistence:** By storing XML-encoded data in a database, you can perform searches on columns that would be unsearchable otherwise.
- **Transparent protocol for component communication:** Useful in communication among components of disparate applications, which could be running on separate computers.
- **Debugging aid:** Serializing objects into XML documents can help you debug complex class hierarchies because the serialized versions of the objects are easy to read. Although the output produced by the WebObjects XML serialization process is verbose, you can transform it into a succinct document. See "Transforming an Array of Movies" (page 82) for an example.
- **Configuration files:** Serializing the values of configuration settings into an XML document can help streamline the management of the configuration options of your applications. From your application's perspective, writing and reading an entire configuration can be as simple as serializing and deserializing a single object.
- **Human-modifiable files:** Once an object is serialized, you can change the values of its fields using a text editor.

Transforming XML Documents

You may need to transform the XML documents generated by NSXMLOutputStream to a format that your customers or service providers are more familiar with. (NSXMLOutputStream is the WebObjects class that serializes objects and data into XML documents, while NSXMLInputStream is the class that deserializes XML documents into objects.) This can help expedite the creation of data-exchange systems. In other words, you'll be able to seamlessly transfer information to and from your business partners. Keep in mind, however, that, unless the data transfer is one-way, you'll have to create transformation scripts that convert your data to the format your partners need and data from your partners to the format that your applications require. In addition, you can deserialize data (using NSXMLInputStream) only from untransformed NSXMLOutputStream output.

XSL Transformations, or XSLT, is a specification that allows you to convert an XML document into another XML document or into any other type of document. An XSLT stylesheet or script contains instructions that tell a transformer how to process an input document (the product of XML serialization) to produce an output document. For more information on XSLT, see *XSL Transformations (XSLT) Version 1.0*, located at http://www.w3.org/TR/xslt.

CHAPTER 3

XML Serialization Essentials

XML serialization is a great way for applications to maintain state, read and write configuration files, and transfer data between processes, applications, and enterprises over a network, including the Internet. Because XML documents are text-based, you can view and modify serialized data with a text editor.

Java's binary serialization API (whose major classes are ObjectOutputStream and ObjectInputStream) provides an infrastructure that supports data serialization into binary form. Binary data, however, is not easily read by people nor appropriate for communication across disparate applications or systems.

WebObjects allows you serialize objects and data into XML documents using the API defined for binary serialization. The classes NSXMLOutputStream and NSXMLInputStream extend ObjectOutputStream and ObjectInputStream, respectively. These classes use the **Java API for XML Processing (JAXP)** to communicate with the XML parser. See "XML Parsers and XSLT Processors" (page 71) for more information.

As in binary serialization, an NSXMLOutputStream object writes enough data to a stream for an NSXMLInputStream object to be able to reconstruct the object graph and data that the stream represents. This includes fully qualified class names, field names, and data types. This level of verbosity is adequate for serialization and deserialization by similar systems, but may not be appropriate for data transmission between companies, for example. "Transforming an Array of Movies" (page 82) shows you how to transform the output of NSXMLOutputStream into a simpler XML document suitable for communication among business partners.

Most of this chapter is based on Sun's *Java Object Serialization Specification*, which you can find at http://java.sun.com/products/jdk/1.3/docs. If you are familiar with that document, you can just skim through the chapter. You should, however,

read "Application Security" (page 34), as it contains information on how to set up the security manager to allow WebObjects's serialization classes to work unrestricted.

This chapter contains the following sections:

- "Serialization Process" (page 24) lists the steps you perform to serialize data.
- "Deserialization Process" (page 26) lists the steps you perform to deserialize data.
- "Secure Serialization" (page 28) explains how to exclude fields from the serialization process.
- "Validation of Deserialized Data" (page 30) briefly explains how to validate an object after it's deserialized.
- "Multiple Class-Version Support" (page 32) lists issues to consider when you update a Serializable class to maintain compatibility with previous versions.
- "Serialization With Keys" (page 34) provides an overview of key-based serialization.
- "Application Security" (page 34) explains how to set up Sun's security manager to grant WebObjects classes permissions to allow them to perform XML serialization.

Serialization Process

To serialize objects and data you perform the following steps:

1. Open an output stream of type `java.io.OutputStream` or a subclass of it.
2. Initialize an NSXMLOutputStream with the output stream.
3. Invoke the `writeObject` method to serialize objects or the appropriate `write` method to serialize primitive-type data (see the API documentation for the `java.io.DataOutput` interface for a list of primitive-data serialization methods).
4. Close the OutputStream and the NSXMLOutputStream.

Listing 3-1 shows an example of a method that serializes an object and an integer value.

Listing 3-1 Example of a serialization method

```
/**
 * Serializes an object and an integer.
 */
public void serialize() {
    // Filename of the output file.
    String filename = "/tmp/example.xml";

    try {
        // Create a stream to the output file.
        FileOutputStream output_stream = new FileOutputStream(filename);

        // Create an XML-output stream.
        NSXMLOutputStream xml_stream = new NSXMLOutputStream(output_stream);

        // Write the data.
        xml_stream.writeObject("Hello, World!");
        xml_stream.writeInt(5);

        // Close the streams.
        xml_stream.flush(); // not really needed, but doesn't hurt
        xml_stream.close();
        output_stream.close();
    }

    catch (IOException e) {
        e.printStackTrace();
    }
}
```

When an object is serialized, all the objects it refers to are also serialized. But this brings up the issue of cyclic references or multiple references to the same object. The problem is addressed by uniquely identifying each object as it is serialized. As each object is written to the output stream, its `id` attribute is set to a number that is unique within the XML document being generated. References to previously serialized objects use those objects' identification numbers instead of writing additional

copies of them. This method is also used by object instances when referring to their class descriptions. See "Serializing Custom Serializable Objects to an XML Document" (page 61) for an example.

Deserialization Process

To deserialize data from an untransformed XML stream encoded with NSXMLOutputStream, you perform the following steps:

1. Open an input stream of type `java.lang.InputStream` or a subclass of it.
2. Initialize an NSXMLInputStream with the input stream.
3. Invoke the `readObject` method to deserialize objects or the appropriate `read` method to deserialize primitive-type data (see the API documentation for the `java.io.DataInput` interface for a list of primitive-data serialization methods).
4. Close the InputStream and the NSXMLInputStream.

Listing 3-2 shows an example of a method that deserializes an object and an integer value.

Listing 3-2 Example of a deserialization method

```
/**
 * Deserializes an object and an integer.
 */
public void deserialize() {
    // Filename of the input file.
    String filename = "/tmp/example.xml";

    try {
        // Create a stream from the input file.
        NSXMLInputStream input_stream = new FileInputStream(filename);

        // Create an XML-input stream.
        NSXMLInputStream xml_stream = new NSXMLInputStream(input_stream);
```

```
        // Read the data.
        String theString = xml_stream.readObject();
        int theInt = xml_stream.readInt();

        // Close the streams.
        xml_stream.close();
        output_stream.close();
    }

    catch (IOException e) {
        e.printStackTrace();
    }

    catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

When you deserialize an object, the original object graph is recreated by restoring the values of nontransient and nonstatic fields. Objects referred to in the original object graph are restored recursively. After deserializing an object with transient or static fields, you must set those fields to the appropriate values. See "Validation of Deserialized Data" (page 30) and "Secure Serialization" (page 28) for more information.

You may want to have the parser validate source documents before deserializing objects; this is helpful in debugging and when transferring data across a network, such as an intranet or the Internet. However, you incur a performance penalty when the parser validates the documents it processes. To turn on parser validation, set the `NSXMLValidation` system property to `true`. As a general rule, you should turn on validation during application development and turn it off in deployed applications.

Secure Serialization

When you deserialize an object, its private state is restored. To protect sensitive data you may have to remove certain fields from the serialization and deserialization processes. You can accomplish this in two ways:

- Define fields whose data you want to protect as `private transient` or `static`.
- Implement `writeObject` and `readObject` in the class you want to protect and serialize nonsensitive fields only.

To prevent serialization, a class must not implement the `java.io.Serializable` or `java.io.Externalizable` interfaces. In subclasses of classes that implement those interfaces, you can throw a NotSerializableException. Listing 3-3 shows an example of a class with a transient field.

Listing 3-3 Example of a secure class

```
/**
 * Encapsulates secret data.
 */
public class Secret extends Object implements Serializable {
    private transient String details;    // do not serialize
    private int id;

    /*
     * Creates a Secret object.
     *
     * @param id            identification
     * @param details       sensitive information
     */
    Secret(int id, String details) {
        super();

        this.id = id;
        this.details = details;
```

```
    }

    /*
     * Gets this secret's id.
     *
     * @return secret id.
     */
    public int id() {
        return this.id;
    }

    /*
     * Gets this secret's details.
     *
     * @return secret details.
     */
    public String details() {
        return this.details;
    }
}
```

Note: You should also define as transient fields that contain objects whose class does not implement Serializable or Externalizable.

Listing 3-4 shows a class that extends a serializable class, but inhibits instances from being serialized or deserialized.

Listing 3-4 Example of a class that disallows serialization and deserialization by throwing NotSerializableException

```
/**
 * This class must inhibit serialization and deserialization
 * of its instances.
 */
public class SuperSecret extends GeneralInfo {
    ...

    /**
     * Prevents deserialization.
     */
```

```
    private void readObject(ObjectInputStream stream) throws IOException,
                    ClassNotFoundException {
        throws new java.io.NotSerializableException("SuperSecret");
    }

    /**
     * Prevents serialization.
     */
    private void writeObject(ObjectOutputStream stream) throws IOException {
        throws new java.io.NotSerializableException("SuperSecret");
    }
}
```

Validation of Deserialized Data

Sometimes, especially when deserializing objects with transient or static fields, you may want to validate an object before it is returned to the method that invoked `readObject`. To do that, you invoke the `registerValidation` method to tell the ObjectInputStream which object to notify when the deserialized object graph has been restored, but before `readObject` returns. The callback method is named `validateObject`. If the object's data is invalid, `validateObject` throws an InvalidObjectException. For more information, see the API documentation on `java.io.ObjectInputStream`, `java.io.ObjectInputValidation`, `java.io.InvalidObjectException`, and `com.webobjects.foundation.xml.NSXMLObjectInputStream`.

Listing 3-5 shows an example of a class that validates the data of an object using `validateObject`. In this case, the validation code is contained in the class of the object being deserialized, but this need not be the case. You may instead choose to have a validation class that contains all XML-document validation logic.

Listing 3-5 Example of a class that validates deserialized data

```
import java.io.InvalidObjectException;
import java.io.IOException;
import java.io.ObjectInputStream;
```

```
import java.io.ObjectInputValidation;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.sql.Timestamp;

/**
 * Manages movie information.
 */
public class ValidMovie extends Object implements ObjectInputValidation,
Serializable {
    ...

    /**
     * Serializes this object.
     *
     * @param stream    object stream to serialize this object to
     */
    private void writeObject(ObjectOutputStream stream) throws IOException {
        ...
    }

    /**
     * Deserializes this object.
     *
     * @param stream   object stream from which the serialized data
     *                 is obtained
     */
    private void readObject(ObjectInputStream stream) throws IOException,
                    ClassNotFoundException {
        ...
    }

    /**
     * Validates a deserialized ValidMovie object.
     *
     * @throws InvalidObjectException when the deserialized ValidMovie
     *         is not valid.
     */
    public void validateObject() throws InvalidObjectException {
        // Determine validity of this object.
        boolean valid = someValidationMethod();
```

```
        if (!valid) {
           throw new InvalidObjectException("Deserialized ValidMovie object
contains invalid data.");
        }
    }
}
```

Multiple Class-Version Support

Both binary serialization in Java and XML serialization in WebObjects allow you to support more than one version of the same class for serialization and deserialization.

When dealing with multiple versions of a class, you must keep the class's identity in mind. Classes are identified by their name and API. For versioning to succeed, you must ensure that the changes you make when creating a new version of a class are compatible with the previous version. In other words, the new class's API must be a superset of the API defined in the previous version.

You can address versioning by implementing and maintaining `writeObject` and `readObject` in a class. However, binary serialization and, by extension, XML serialization provide facilities for the automatic management of multiple versions of an evolving, serializable class. In particular, binary and XML serialization provide support for bidirectional communication between class versions. This means that a class can read data serialized by a newer version. It also allows a class to write a stream from which an instance of a previous version can be successfully created.

When a later version of a class adds fields to the class, you need to initialize only the added fields when deserializing data from a stream created with the previous version of the class. However, when the new version changes field usage and you need to map fields of the new version to fields of the old version or perform conversions on existing fields, you can take advantage of the ObjectStreamField class. See "Advanced Object Serialization," located at http://developer.java.sun.com/developer/technicalArticles/ALT/index.html for details.

Table 3-1 lists compatible and incompatible changes for new class versions. It summarizes the information provided in Sun's *Java Object Serialization Specification*.

Table 3-1 Compatible and incompatible changes for new class versions

Change	Compatible	Incompatible
Adding fields, or changing a field from transient to nontransient or static to nonstatic.	x	
Adding fields, or changing a field from transient to nontransient or static to nonstatic.	x	
Adding classes or implementing `java.io.Serializable`.	x	
Removing classes or removing `extends Serializable` from a class declaration.	x	
Adding `writeObject` and `readObject` methods.	x	
Removing `writeObject` and `readObject` methods.	x	
Changing a field's access modifier.	x	
Deleting fields.		x
Modifying the class hierarchy.		x
Changing a field from nontransient to transient or nonstatic to static.		x
Changing the type of a field.		x
Changing `writeObject` so that it no longer writes default field data.		x
Changing `readObject` so that it reads default field data when the previous version does not write default field data.		x
Changing a class from Serializable to Externalizable or from Externalizable to Serializable.		x

Serialization With Keys

WebObjects XML serialization provides a useful feature: the ability to serialize objects and data with keys. To use this feature you add an additional argument to `writeObject` and `write` method invocations: the key, which is a String object.

Adding keys to your XML documents can help you perform useful transformations; that is, you can use the keys in the source document to create the elements in the target document. For example, the element `<int>32</int>` (created by executing `writeInt(32)`) provides no information about the integer 32. However, if you use `writeInt(32, "age")` to serialize the value, a transformation script can use the additional information about the datum to create the element `<age>32</age>`. See "Transforming Primitive-Type Values Using Keys" (page 80) for details.

Application Security

Generally, security-minded environments run Sun's security manager to protect their systems from potentially damaging activities by malicious applications. The security manager is disabled by default. You activate the security manager by adding

```
-Djava.security.manager
```

to the command line when launching the application manually or to the application project's `Properties` file, located in the Resources group. For more information on the security manager, see *Security in Java 2 SDK 1.2*, located at http://java.sun.com/docs/books/tutorial/index.html.

If you use the security manager, you must add the policy shown in Listing 3-6 for Mac OS X systems or Listing 3-7 for Windows systems to the policy file for XML serialization to work correctly in WebObjects applications.

Listing 3-6 Security-manager policies required for XML serialization in WebObjects for Mac OS X

```
grant codeBase "file:/System/Library/Frameworks/JavaFoundation.framework/Resources/
Java/javafoundation.jar"
{
permission java.io.SerializablePermission "enableSubclassImplementation";
permission java.lang.RuntimePermission "accessDeclaredMembers";
permission java.lang.RuntimePermission "XMLSerializationAccess";
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
permission java.io.FilePermission "<<ALL FILES>>", "read";
permission java.util.PropertyPermission "*", "read, write";
};
grant codeBase "file:/System/Library/Frameworks/JavaXML.framework/Resources/Java/
javaxml.jar"
{
permission java.io.FilePermission "<<ALL FILES>>", "read, write";
permission java.util.PropertyPermission "user.dir", "read";
};
```

Listing 3-7 Security-manager policies required for XML serialization in WebObjects for Windows

```
grant codeBase "C:/Apple/Library/Frameworks/JavaFoundation.framework/Resources/Java/
javafoundation.jar"
{
permission java.io.SerializablePermission "enableSubclassImplementation";
permission java.lang.RuntimePermission "accessDeclaredMembers";
permission java.lang.RuntimePermission "XMLSerializationAccess";
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
permission java.io.FilePermission "<<ALL FILES>>", "read";
permission java.util.PropertyPermission "*", "read, write";
};
grant codeBase "C:/Apple/Library/Frameworks/JavaXML.framework/Resources/Java/
javaxml.jar"
{
permission java.io.FilePermission "<<ALL FILES>>", "read, write";
permission java.util.PropertyPermission "user.dir", "read";
};
```

CHAPTER 3

XML Serialization Essentials

CHAPTER 4

Serializing Objects and Data

Chapter 3, "XML Serialization Essentials" (page 23), explained that serialization is a useful way to implement component-to-component or application-to-application communication. This chapter guides you through the creation of a simple WebObjects application that shows how serialization, both binary-based and XML-based, can be implemented.

The chapter is divided in two sections:

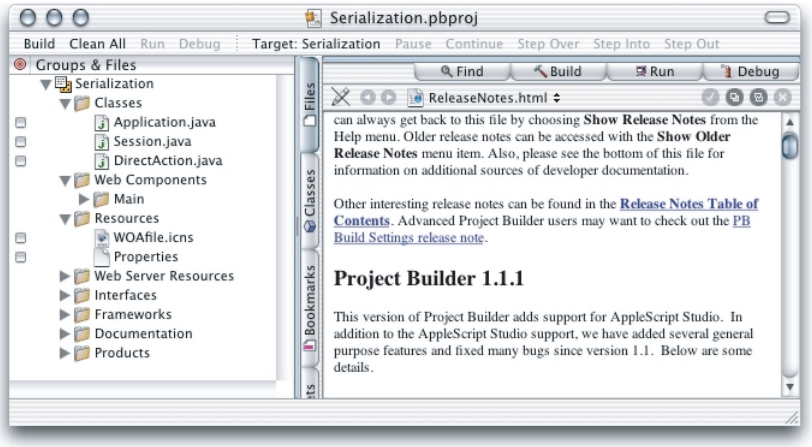
- "Binary Serialization Example" (page 37) walks you through the creation of the Serialization project, which includes an example utility class called BinarySerializer, used to serialize objects and data into binary files.
- "XML Serialization Example" (page 53) shows how to use WebObjects XML serialization with the example class named XMLSerializer to serialize objects and data. It demonstrates that serializing to XML documents is just as easy as serializing to binary files. In addition, it teaches you how to include keys in the XML documents that represent serialized data. These keys can make it easier to transform those documents into a format that other applications expect. Finally, it explains the use of NSXMLOutputFormat objects to set output-format properties for NSXMLOutputStream objects.

Binary Serialization Example

This section guides you through the creation of a straightforward application that serializes and deserializes objects and primitive values into and from binary form using Java's binary-serialization facilities.

Creating the Serialization Project

Using Project Builder, create a WebObjects application project named Serialization. You don't need to add any frameworks to the project, such as the Java JDBC Adaptor framework. (You can look at the finalized project in `projects/Serializing/Binary/Serialization`.)



Adding the BinarySerializer Class

The BinarySerializer class manages binary serialization and deserialization to and from files. To save yourself some typing, you can add the `BinarySerializer.java` file in `Projects/Serializing/Binary/Serialization` to your project's Application Server target. Otherwise, follow the steps below.

1. Select the Classes group in the Serializer project.
2. Choose File > New File.
3. In the New File pane of the Assistant, select Java Class under WebObjects and click Next.
4. In the New Java Class pane, name the class `BinarySerializer` and click Finish.
5. Replace the template code in `BinarySerializer.java` with the code in Listing B-1 (page 109).

BinarySerializer.java provides two main functions: serialization and deserialization of objects and creation and disposal of ObjectOutputStream and ObjectInputStream objects.

The serializeObject and deserializeObject methods serialize and deserialize objects to and from a file. You can use the openStream and closeStream methods to serialize primitive-type values or individual objects. See "Serializing Primitive-Type Values" (page 42) for an example.

Serializing an NSArray of Strings

Now that you have the BinarySerializer class as part of your project, you can use it to serialize objects.

The first step is to add three methods to the Application class: one that instantiates, populates, and serializes an NSArray of Strings; a second one that deserializes and displays the data; and a third one that invokes the other two. Listing 4-1 shows the methods.

Listing 4-1 The serializeArray, deserializeArray, and arraySerialization methods in Application.java

```
/**
 * Creates and serializes an NSArray of Strings.
 * @param identifier      identifies the target file, without a
 *                        path or an extension
 */
public void serializeArray(String identifier) {
    // Instantiate object to serialize.
    NSArray book_titles =  new NSArray(new Object[] {"The Chestry Oak", "A Tree for
Peter", "The White Stag"});

    // Serialize the object.
    BinarySerializer.serializeObject(book_titles, identifier);
}

/**
 * Deserializes an NSArray and writes its contents to the console.
 * @param identifier      identifies the source file, without a
```

```
 *                        path or an extension
 */
public void deserializeArray(String identifier) {
    // Deserialize the data and assign it to an NSArray object.
    NSArray books = (NSArray)BinarySerializer.deserializeObject(identifier);

    // Display the contents of <code>books</code> on the console
    // (the Run pane in Project Builder).
    System.out.println("");
    System.out.println("** Deserialized NSArray **");
    System.out.println(books);
    System.out.println("");
}

/**
 * Invokes the <code>serializeArray</code> and
 * <code>deserializeArray</code> methods.
 */
public void arraySerialization() {
    String identifier = "BookTitles";

    // Serialize NSArray object.
    serializeArray(identifier);

    // Deserialize NSArray object.
    deserializeArray(identifier);
}
```

Finally, modify the Application class's constructor so that it looks like Listing 4-2.

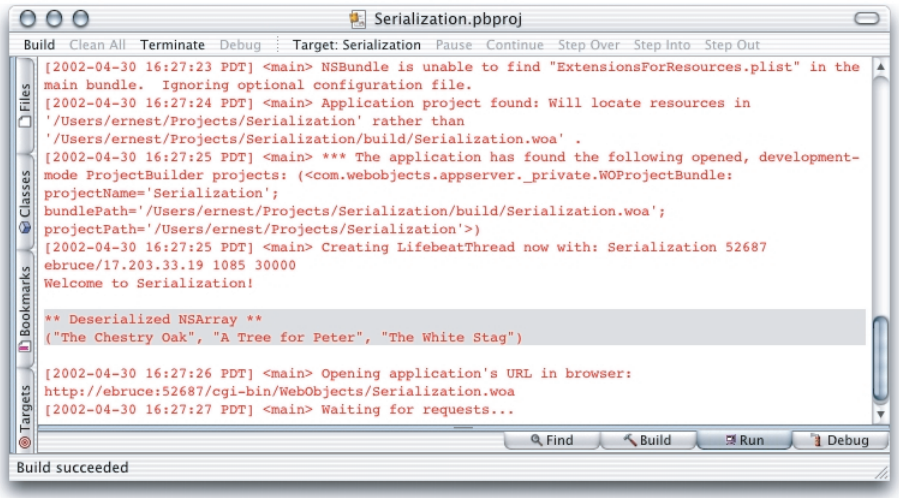
Listing 4-2 The constructor in `Application.java`

```
/**
 * Creates an Application object. Invoked once during application startup.
 */
public Application() {
    super();
    System.out.println("Welcome to " + this.name() + "!");

    // Test serialization of an array.
```

```
    arraySerialization();
}
```

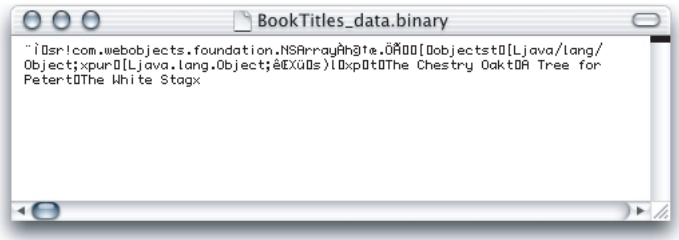
After you build and run the application, Project Builder's Run pane should look similar to Figure 4-1.

Figure 4-1 Project Builder's Run pane when running the array-serialization example



If you open `/tmp/BookTitles_data.binary` (generated by the serialization process) in a text editor, you should see something similar to Figure 4-2.

Figure 4-2 `BinaryTitles_data.binary` file viewed through a text editor



This hardly qualifies as human-readable. "XML Serialization Example" (page 53) shows you how to create files with serialized data that are easier for people to read and modify.

Serializing Primitive-Type Values

This section shows you how to serialize primitive-type values that are not encapsulated by objects. To accomplish this, you instantiate an ObjectOutputStream and invoke one or more of its `write` methods.

Add three methods to the Application class of the Serialization project: one that serializes values of type `int`, `boolean`, `char` and `double`; a second that deserializes the data; and a third one that calls the other two. Listing 4-3 gives you an example of such methods.

Listing 4-3 The `serializePrimitives`, `deserializePrimitives`, and `primitiveSerialization` methods in `Application.java`

```
/**
 * Serializes a set of primitive values.
 *
 * @param filename    identifies the target file, including its
 *                    path and extension
 *
 * @param an_int      value to serialize
 * @param a_boolean   value to serialize
 * @param a_char      value to serialize
 * @param a_double    value to serialize
 */
public void serializePrimitives(String filename, int an_int, boolean a_boolean, char
a_char, double a_double) {
    try {
        // Open an output stream.
        ObjectOutputStream stream = BinarySerializer.openOutputStream(filename);

        // Write values.
        stream.writeInt(an_int);
        stream.writeBoolean(a_boolean);
        stream.writeChar(a_char);
```

```
        stream.writeDouble(a_double);

        // Close the stream.
        BinarySerializer.closeStream(filename);
    }

    catch (IOException e) {
        e.printStackTrace();
    }
}

/**
 * Deserializes a set of primitive values.
 *
 * @param filename   identifies the source file, including its
 *                   path and extension
 */
public void deserializePrimitives(String filename) {
    try {
        // Open an input stream.
        ObjectInputStream stream = BinarySerializer.openInputStream(filename);

        // Read values.
        int the_int = stream.readInt();
        boolean the_boolean = stream.readBoolean();
        char the_char = stream.readChar();
        double the_double = stream.readDouble();

        BinarySerializer.closeStream(filename);

        // Write values to console (Run pane in Project Builder).
        System.out.println("");
        System.out.println("** Deserialized primitives **");
        System.out.println("int: " + the_int);
        System.out.println("boolean: " + the_boolean);
        System.out.println("char: " + the_char);
        System.out.println("double: " + the_double);
        System.out.println("");
    }

    catch (IOException e) {
```

```
        e.printStackTrace();
    }
}

/**
 * Invokes the <code>serializePrimitives</code> and
 * <code>deserializePrimitives</code> methods.
 */
public void primitiveSerialization() {
    String filename = "/tmp/PrimitiveValues_data.binary";

    // Serialize primitive values.
    serializePrimitives(filename, 5, true, 'u', 3.14);

    // Deserialize primitive values.
    deserializePrimitives(filename);
}
```

You also need to add the following to the Application class:

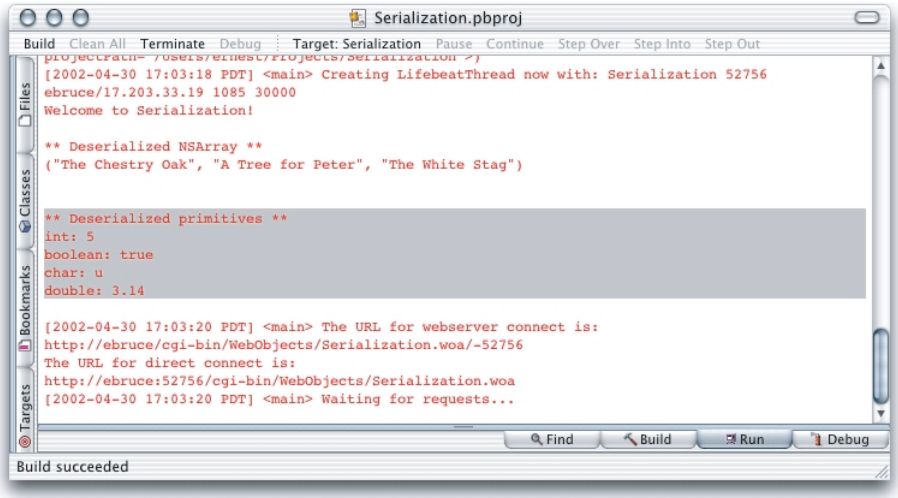
```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

Finally, add a call to the `primitiveSerialization` method in the Application class's constructor:

```
// Test serialization of primitive values.
primitiveSerialization();
```

Build and run the application. Project Builder's Run pane should look similar to Figure 4-3.

Figure 4-3 Project Builder's Run pane when running the primitive-values serialization example



Serializing Custom Serializable Objects

Now that you've mastered the art of serializing an instance of a WebObjects Serializable class and primitive-type values, you're ready to tackle the serialization of custom objects. For this, you create a Movie class that includes `writeObject` and `readObject` methods.

Add a class named Movie to the Serialization project and assign it to the Application Server target. Modify `Movie.java` so that it looks like Listing 4-4. (Alternatively, you can add the `Movie.java` file in `projects/Serializing/Binary/Serialization` to your project.)

Listing 4-4 `Movie.java` using binary serialization

```
import com.webobjects.appserver.*;
import com.webobjects.foundation.*;
import com.webobjects.foundation.xml.*;
import com.webobjects.eocontrol.*;
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.sql.Timestamp;

/**
 * Manages movie information.
 */
public class Movie extends Object implements Serializable {
    private String title;
    private String studio;
    private NSTimestamp releaseDate;

    /**
     * Creates a Movie object.
     *
     * @param name          movie title
     * @param studio        studio that released the movie
     * @param release_date  date the movie was released
     */
    Movie(String title, String studio, NSTimestamp releaseDate) {
        super();

        setTitle(title);
        setStudio(studio);
        setReleaseDate(releaseDate);
    }

    /**
     * Gets this movie's title.
     *
     * @return movie title.
     */
    public String title() {
        return this.title;
    }

    /**
     * Sets this movie's title.
     *
```

```
 * @param value            movie's title
 */
public void setTitle(String value) {
    this.title = value;
}

/**
 * Gets this movie's studio.
 *
 * @return movie studio.
 */
public String studio() {
    return this.studio;
}

/**
 * Sets this movie's studio.
 *
 * @param value           studio's name
 */
public void setStudio(String value) {
    this.studio = value;
}

/**
 * Gets this movie's release date.
 *
 * @return movie release date.
 */
public NSTimestamp releaseDate() {
    return this.releaseDate;
}

/**
 * Sets this movie's release date.
 *
 * @param value          release date
 */
public void setReleaseDate(NSTimestamp value) {
    this.releaseDate = value;
}
```

```
    /**
     * Gets the string representation of this movie.
     *
     * @return string representing this movie.
     */
    public String toString() {
        return "(Movie: (Title: " + title() + "), (Studio: " + studio() + "), (Release
Date: " + releaseDate().toString() +  "))";
    }

    /**
     * Serializes this object.
     *
     * @param stream   object stream to serialize this object to
     */
    private void writeObject(ObjectOutputStream stream) throws IOException {
        // Serialize the object's instance members.
        // (This is where you put special encoding logic,
        // such as the one used to encode the releaseDate field.)
        stream.writeObject(title());
        stream.writeObject(studio());
        stream.writeObject(releaseDate().toString());
    }

    /**
     * Deserializes this object.
     *
     * @param stream   object stream from which the serialized data
     *                 is obtained
     */
    private void readObject(ObjectInputStream stream) throws IOException,
ClassNotFoundException {
        // Deserializes the data a put it in the object's instance members.
        // (This is where you would put special de-encoding logic
        // such as the one used to decode the releaseDate field.)
        setTitle((String)stream.readObject());
        setStudio((String)stream.readObject());
        setReleaseDate(_timestampFromString((String)stream.readObject()));
    }
```

```
    /**
     * Converts a string into an NSTimestamp.
     *
     * @param timestampAsString     string to convert
     *
     * @return NSTimestamp object represented by timestampAsString.
     */
    private NSTimestamp _timestampFromString(String timestampAsString) {
        NSTimestampFormatter formatter = new NSTimestampFormatter();
        java.text.ParsePosition pp = new java.text.ParsePosition(0);

        return (NSTimestamp)formatter.parseObject(timestampAsString, pp);
    }
}
```

Now add the methods in Listing 4-5 to `Application.java`.

Listing 4-5 The `serializeMovie`, `deserializeMovie`, and `movieSerialization` methods in `Application.java`

```
/**
 * Serializes a Movie object.
 *
 * @param identifier   identifies the target file, without a
 *                     a path or an extension
 */
public void serializeMovie(String identifier) {
    // Set the local time zone.
    NSTimeZone timeZone = NSTimeZone.timeZoneWithName("America/Los_Angeles", true);

    Movie movie = new Movie("Alien", "20th Century Fox", new NSTimestamp(1979, 10, 25,
0, 0, 0, timeZone));

    BinarySerializer.serializeObject(movie, identifier);
}

/**
 * Deserializes Movie data into an object.
 *
 * @param identifier   identifies the source file, without a
```

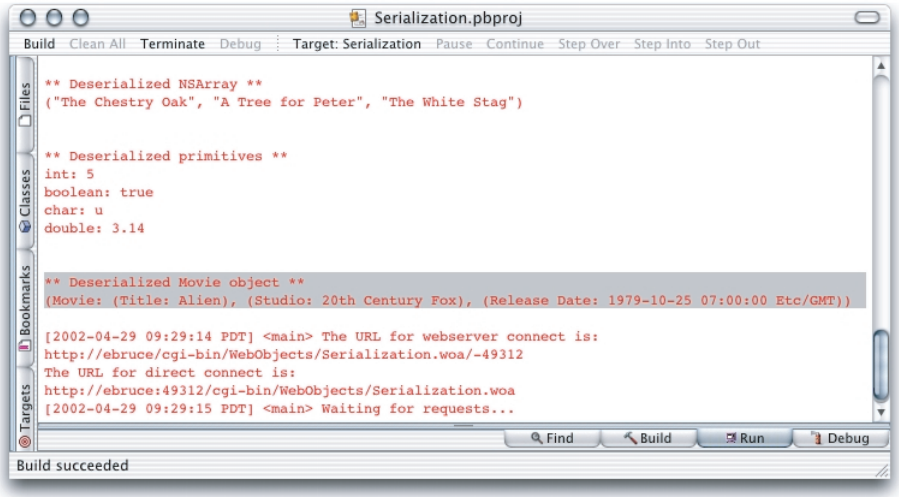
```
 *                     a path or an extension
 */
public void deserializeMovie(String identifier) {
    Movie movie = (Movie)BinarySerializer.deserializeObject(identifier);

    System.out.println("");
    System.out.println("** Deserialized Movie object **");
    System.out.println(movie.toString());
    System.out.println("");
}

/**
 * Invokes the <code>movieSerialization</code> and
 * <code>movieDeserialization</code> methods.
 */
public void movieSerialization() {
    String identifier = "Movie";

    serializeMovie(identifier);
    deserializeMovie(identifier);
}
```

Finally, modify the Application class's constructor by adding a call to the `movieSerialization` method:

```
// Serialize a Movie object.
movieSerialization();
```

After building and running the application, you should see something similar to Figure 4-4.

Figure 4-4 Project Builder's Run pane when running the Movie-object serialization example



The remainder of the section shows you how to serialize an NSMutableArray of Movies and deserialize the data into an NSArray.

Add the methods shown in Listing 4-6 to `Application.java`.

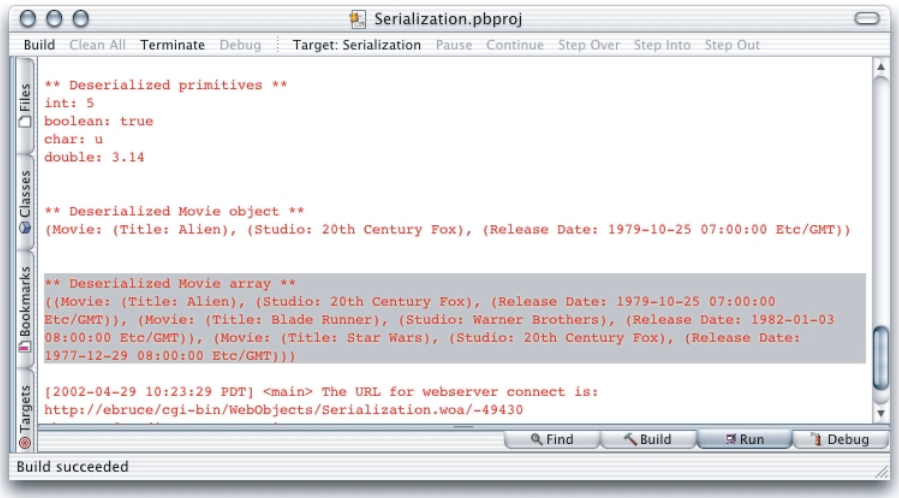
Listing 4-6 The `serializeMovieArray`, `deserializeMovieArray`, and `movieArraySerialization` methods of `Application.java`

```
/**
 * Serializes a Movie array.
 *
 * @param identifier   identifies the target file, without a
 *                     a path or an extension
 */
public void serializeMovieArray(String identifier) {
    // Set the local time zone.
    NSTimeZone timeZone = NSTimeZone.timeZoneWithName("America/Los_Angeles", true);

    // Initialize the array.
```

```
    NSMutableArray movies = new NSMutableArray();
    movies.addObject(new Movie("Alien", "20th Century Fox", new NSTimestamp(1979, 10, 25,
0, 0, 0, timeZone)));
    movies.addObject(new Movie("Blade Runner", "Warner Brothers", new NSTimestamp(1982,
1, 3, 0, 0, 0, timeZone)));
    movies.addObject(new Movie("Star Wars", "20th Century Fox", new NSTimestamp(1977, 12,
29, 0, 0, 0, timeZone)));

    // Serialize the array.
    BinarySerializer.serializeObject(movies, identifier);
}

/**
 * Deserializes Movie data into an NSArray.
 *
 * @param identifier   identifies the source file, without a
 *                     a path or an extension
 */
public void deserializeMovieArray(String identifier) {
    // Create an empty array.
    NSArray movies = new NSArray();

    // Deserialize data into movies.
    movies = (NSArray)BinarySerializer.deserializeObject(identifier);

    System.out.println("");
    System.out.println("** Deserialized Movie array **");
    System.out.println(movies.toString());
    System.out.println("");
}

/**
 * Invokes the <code>movieArraySerialization</code> and
 * <code>movieArrayDeserialization</code> methods.
 */
public void movieArraySerialization() {
    String identifier = "Movies";

    serializeMovieArray(identifier);
    deserializeMovieArray(identifier);
}
```

Add a call to the `movieArraySerialization` method in the Application class's constructor and build and run the application. The Project Builder Run pane should look like Figure 4-5.

Figure 4-5 Project Builder's Run pane when running the Movie-array serialization example



XML Serialization Example

As you learned in Chapter 3, "XML Serialization Essentials" (page 23), WebObjects XML serialization works essentially the same way Java binary serialization does. This section shows how to modify the Serialization project so that it uses a new class, XMLSerializer, to serialize and deserialize objects and data.

If you did not create the Serialization project in "Binary Serialization Example" (page 37), you can get it from this book's example projects in `projects/Serializing/Binary`.

Adding the XMLSerializer Class

Start by adding a new class called `XMLSerializer.java` to the project and assigning it to the Application Server target. Then modify the class so that it looks like Listing B-2 (page 115). (Alternatively, you can add the `XMLSerializer.java` file in `projects/Serializing/XML/Serialization` to your project.)

Serializing an NSArray of Objects to an XML Document

Now that the Serialization project contains the utility class XMLSerializer, modify the `serializeArray` and `deserializeArray` methods in `Application.java` so that they look like Listing 4-7. (All you need to do is change occurrences of `BinarySerializer` to `XMLSerializer` in the lines numbered 1 and 2.)

Listing 4-7 The `serializeArray` and `deserializeArray` methods in `Application.java` using XML serialization

```
/**
 * Creates and serializes an NSArray of Strings.
 * @param identifier          identifies the target file, without a
 *                            path or an extension
 */
public void serializeArray(String identifier) {
    // Instantiate object to serialize.
    NSArray book_titles =  new NSArray(new Object[] {"The Chestry Oak", "A Tree for
Peter", "The White Stag"});

    // Serialize the object.
    XMLSerializer.serializeObject(book_titles, identifier);                    //1
}

/**
 * Deserializes an NSArray and writes its contents to the console.
 * @param identifier          identifies the source file, without a
 *                            path or an extension
 */
public void deserializeArray(String identifier) {
    // Deserialize the data and assign it to an NSArray object.
```

```
    NSArray books = (NSArray)XMLSerializer.deserializeObject(identifier);           //2

    // Display the contents of <code>books</code> on the console
    // (the Run pane in Project Buider).
    System.out.println("");
    System.out.println("** Deserialized NSArray **");
    System.out.println(books);
    System.out.println("");
}
```

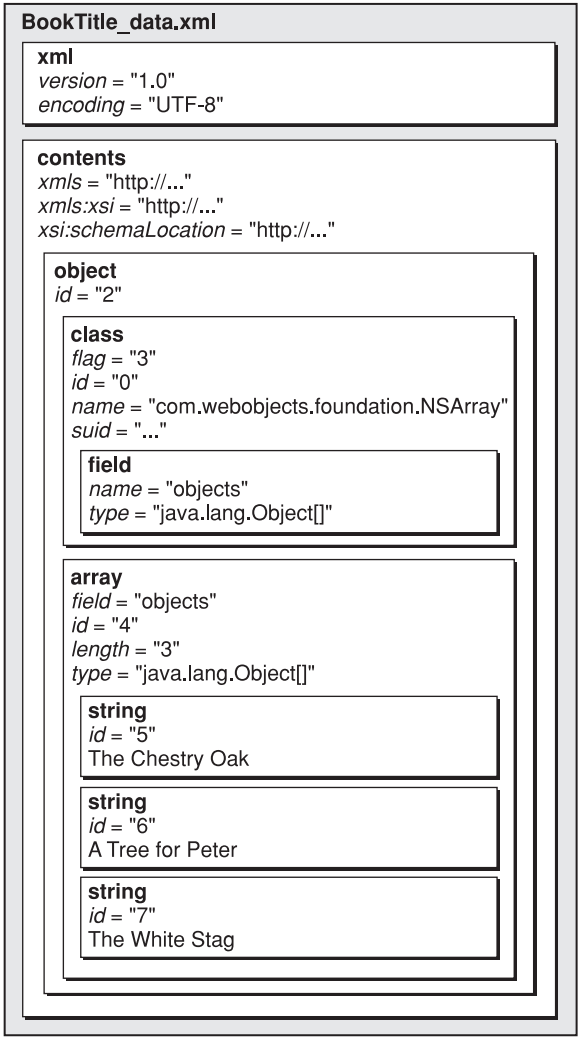
After building and running the application, you can find the `BookTitles_data.xml` file (shown in Listing 4-8) in `/tmp`.

Listing 4-8 `BookTitles_data.xml` (serialized array of Strings)

```
<?xml version="1.0" encoding="UTF-8"?>
<content xmlns="http://www.apple.com/webobjects/XMLSerialization" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.apple.com/
webobjects/XMLSerialization
http://www.apple.com/webobjects/5.2/schemas/woxml.xsd">
    <object id="2">
        <class flag="3" id="0" name="com.webobjects.foundation.NSArray" suid="-
3789592578296478260">
            <field name="objects" type="java.lang.Object[]"/>
        </class>
       <array field="objects" id="4" ignoreEDB=”1” length="3" type="java.lang.Object[]">
            <string id="5">The Chestry Oak</string>
            <string id="6">A Tree for Peter</string>
            <string id="7">The White Stag</string>
        </array>
    </object>
</content>
```

As you can see, the serialized version of the NSArray object is easier to read than `BookTitles_data.binary`, created in “Serializing an NSArray of Strings” (page 39). Figure 4-6 graphically depicts `BookTitles_data.xml`. However, the document is somewhat verbose. Chapter 6, “Transforming XML Documents” (page 75), shows you how to transform XML streams generated by NSXMLOutputStream into streamlined XML documents.

Figure 4-6 The element hierarchy of the `BoolTitles_data.xml` document



Serializing Primitive-Type Values to an XML Document

By now you've probably noticed how easy it is to serialize objects into XML documents. This section shows you how to serialize primitive-type values.

First, add the following code line to `Application.java`:

```
import com.webobjects.foundation.xml.*;
```

Now, modify the `serializePrimitives` and `deserializePrimitives` methods so that they match Listing 4-9 (You need to modify only the five numbered lines.)

Listing 4-9 The `serializePrimitives`, `deserializePrimitives` and `primitiveSerialization` methods in `Application.java` using XML serialization

```
/**
 * Serializes a set of primitive values.
 *
 * @param filename     identifies the target file, including its
 *                     path and extension
 * @param an_int       value to serialize
 * @param a_boolean    value to serialize
 * @param a_char       value to serialize
 * @param a_double     value to serialize
 */
public void serializePrimitives(String filename, int an_int, boolean a_boolean, char
a_char, double a_double) {
    try {
        // Open an output stream.
        NSXMLOutputStream stream = XMLSerializer.openOutputStream(filename, null);  //1

        // Write values.
        stream.writeInt(an_int);
        stream.writeBoolean(a_boolean);
        stream.writeChar(a_char);
        stream.writeDouble(a_double);

        // Close the stream.
```

```
        XMLSerializer.closeStream(filename);                                    //2
    }

    catch (IOException e) {
        e.printStackTrace();
    }
 }

/**
 * Deserializes a set of primitive values.
 *
 * @param filename     identifies the source file, including
 *                     its path and extension
 */
public void deserializePrimitives(String filename) {
    try {
        // Open an input stream.
        NSXMLInputStream stream = XMLSerializer.openInputStream(filename);      //3

        // Read values.
        int the_int = stream.readInt();
        boolean the_boolean = stream.readBoolean();
        char the_char = stream.readChar();
        double the_double = stream.readDouble();

        XMLSerializer.closeStream(filename);                                    //4

        // Write values to console (Run pane in Project Builder).
        System.out.println("");
        System.out.println("** Deserialized primitives **");
        System.out.println("int: " + the_int);
        System.out.println("boolean: " + the_boolean);
        System.out.println("char: " + the_char);
        System.out.println("double: " + the_double);
        System.out.println("");
    }

    catch (IOException e) {
        e.printStackTrace();
    }
 }
```

```
/**
 * Invokes the <code>serializePrimitives</code> and
 * <code>deserializePrimitives</code> methods.
 */
public void primitiveSerialization() {
    String filename = "/tmp/PrimitiveValues_data.xml";                          //5

    // Serialize primitive values.
    serializePrimitives(filename, 5, true, 'u', 3.14);

    // Deserialize primitive values.
    deserializePrimitives(filename);
}
```

Notice the signature of the `openOutputStream` method of the XMLSerializer class (Listing B-2 (page 115)):

```
openOutputStream(String filename, String transformation) throws IOException;
```

The invoking code specifies the type of transformation to perform on the serialized data through the *transformation* parameter. In this case, however, `serializePrimitives` does not perform a transformation; therefore, it invokes `openOutputStream` with *transformation* set to `null`. See Chapter 5, "Transformation of XML Documents" (page 67), for more information on transforming XML documents.

After building and running the application, your `/tmp` directory should contain the `PrimitiveValues_data.xml`. file. Listing 4-10 shows its contents.

Listing 4-10 The `PrimitiveValues_data.xml` file

```
<?xml version="1.0" encoding="UTF-8"?>
<content xmlns="http://www.apple.com/webobjects/XMLSerialization"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.apple.com/webobjects/XMLSerialization
http://www.apple.com/webobjects/5.2/schemas/woxml.xsd">
    <int>5</int>
    <boolean>true</boolean>
    <ch>u</ch>
    <double>3.14</double>
```

```
</content>
```

Serializing With Keys

Keys can help a great deal in describing what a data element represents. Adding keys to values as they are serialized is a simple process. Modify `serializePrimitives` so that it looks like Listing 4-11 (change the numbered code lines), and build and run the application.

Listing 4-11 The `serializePrimitives` method of `Application.java` using keys to identify elements in XML document

```
public void serializePrimitives(String filename, int an_int, boolean
a_boolean, char a_char, double a_double) {
    try {
        // Open an output stream.
        NSXMLOutputStream stream = XMLSerializer.openOutputStream(filename,
null);

        // Write values.
        stream.writeInt(an_int, "my_integer");                          //1
        stream.writeBoolean(a_boolean, "my_boolean");                   //2
        stream.writeChar(a_char, "my_char");                            //3
        stream.writeDouble(a_double, "my_double");                      //4

        // Close the stream.
        XMLSerializer.closeStream(filename);
    }

    catch (IOException e) {
        e.printStackTrace();
    }
}
```

Now, after building and running the project, the `PrimitiveValues_data.xml` file looks like Listing 4-12.

Listing 4-12 The `PrimitiveValues_data.xml` file with keys identifying each element

```
<?xml version="1.0" encoding="UTF-8"?>
<content xmlns="http://www.apple.com/webobjects/XMLSerialization"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.apple.com/webobjects/XMLSerialization
http://www.apple.com/webobjects/5.2/schemas/woxml.xsd">
    <int key="my_integer">5</int>
    <boolean key="my_boolean">true</boolean>
    <ch key="my_char">u</ch>
    <double key="my_double">3.14</double>
</content>
```

The code in "Transforming Primitive-Type Values Using Keys" (page 80) takes advantage of the `key` attribute on each element of the `content` element to transform the source XML document in Listing 4-12 to one that uses the values of those keys as the tag names of the elements that contain the data values in the target document, shown in Listing 6-6 (page 82).

Serializing Custom Serializable Objects to an XML Document

You can take advantage of keys in custom Serializable objects by invoking the `writeObject(Object, String)` method of NSXMLOutputStream in the `writeObject` method of your custom class. To accomplish this, however, you have to cast the ObjectOutputStream argument to NSXMLOutputStream before invoking the `writeObject(Object, String)` method.

Modify the `writeObject` method of `Movie.java` so that it looks like Listing 4-13.

Listing 4-13 The `writeObject` method in `Movie.java` using XML serialization with keys

```
private void writeObject(ObjectOutputStream stream) throws IOException {
    // Serialize the object's instance members.
    // (This is where you put special encoding logic;
    // this example doesn't perform any special encoding.)
```

```
    // Cast stream to NSXMLOutputStream to gain access to
    // <code>writeObject(Object, String)</code>.
    NSXMLOutputStream xml_stream = (NSXMLOutputStream)stream;            //1

    xml_stream.writeObject(title(), "title");                            //2
    xml_stream.writeObject(studio(), "studio");                          //3
    xml_stream.writeObject(releaseDate().toString(), "release_date");    //4
}
```

Now, modify the `serializeMovie`, `deserializeMovie`, `serializeMovieArray` and `deserializeMovieArray` methods in `Application.java` so that they use the XMLSerializer class's `serializeObject` and `deserializeObject` methods, respectively.

Build and run the application. If you open `/tmp/Movies_data.xml` in a text editor, you'll see something similar to the contents of Listing 4-14. Notice the `key` attribute included in the data elements of each `object` element corresponding to a Movie object. The `key` attribute is used by the transformation script in "Transforming an Array of Movies" (page 82) to generate the data elements of the target document.

You may also notice that the third Movie object in `Movies_data.xml` (starting at the line numbered 2) contains a reference to the studio defined in the first Movie object (the line numbered 1) instead of the name of the studio. This is how multiple references to the same object are represented in XML documents generated by NSXMLOutputStream.

Listing 4-14 The `Movies_data.xml` file

```
<?xml version="1.0" encoding="UTF-8"?>
<content xmlns="http://www.apple.com/webobjects/XMLSerialization"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
www.apple.com/webobjects/XMLSerialization
http://www.apple.com/webobjects/5.2/schemas/woxml.xsd">
    <object id="3">
        <class flag="2" id="0"
            name="com.webobjects.foundation.NSMutableArray" suid="-3909373569895711876">
            <super flag="3" id="1"
                name="com.webobjects.foundation.NSArray" suid="-3789592578296478260">
                <field name="objects" type="java.lang.Object[]"/>
            </super>
```

```
        </class>
       <array field="objects" id="5" ignoreEDB="1" length="3" type="java.lang.Object[]">
            <object id="10">
                <class flag="3" id="6" name="Movie" suid="-791832868721905865">
                <field name="releaseDate" type="com.webobjects.foundation.NSTimestamp"/>
                    <field name="studio" type="java.lang.String"/>
                    <field name="title" type="java.lang.String"/>
                </class>
                <string id="11" key="title" xml:space="preserve">Alien</string>
                <string id="12" key="studio" xml:space="preserve">20th Century Fox</
string>                                                                          //1
              <string id="13" ignoreEDB="1" key="release_date" xml:space="preserve">1979-
10-25 07:00:00 Etc/GMT</string>
            </object>
            <object id="14">
                <class idRef="6" name="Movie"/>
                <string id="15" key="title" xml:space="preserve">Blade Runner</string>
              <string id="16" key="studio" xml:space="preserve">Warner Brothers</string>
              <string id="17" ignoreEDB="1" key="release_date" xml:space="preserve">1982-
01-03 08:00:00 Etc/GMT</string>
            </object>
            <object id="18">                                                     //2
                <class idRef="6" name="Movie"/>
                <string id="19" key="title" xml:space="preserve">Star Wars</string>
                <string idRef="12" key="studio"/>
              <string id="20" ignoreEDB="1" key="release_date" xml:space="preserve">1977-
12-29 08:00:00 Etc/GMT</string>
            </object>
        </array>
    </object>
</content>
```

Formatting Serialized Output

The XML documents produced so far are nicely indented to facilitate their comprehension by people. However, most of the time, these documents are intended for applications. Therefore, indentation (and the extra characters it adds to a stream) is not needed. You can determine whether the output produced by an NSXMLOutputStream object is indented using a

com.webobjects.foundation.xml.NSXMLOutputFormat object. NSXMLOutputFormat objects encapsulate formatting information that can be applied to an NSXMLOutputStream object.

Table 4-1 lists the output-format properties you can set with NSXMLOutputFormat.

Table 4-1 Output-format properties accessible through NSXMLOutputFormat

Property	Description	Default value
encoding	Determines the encoding used in the document.	"UTF-8"
indenting	Determines whether the XML document generated is indented.	true
omitXMLDeclaration	Determines whether the XML declaration is omitted from the document.	false
version	Determines the document's XML version.	"1.0"

You can use WebObjects-style accessors to get and set the values of the properties listed in Table 4-1. The value of the indenting property can also be set through one of the constructors of NSXMLOutputFormat: NSXMLOutputFormat(boolean).

Listing 4-15 shows a method that creates an XML stream to a file and sets the indenting and encoding properties for the stream.

Listing 4-15 Setting the indenting and encoding properties of an NSXMLOutputFormat object and applying them to an NSXMLOutputStream object

```
/**
 * Opens an XML output stream to a file.
 *
 * @param filename   fully qualified filename of the
 *                   target or source file; identifies
 *                   the channel to open
 *
 * @return object stream, <code>null</code> when the stream
```

```
 *         could not be created.
 */
public Object xmlOutputStream(String filename) throws IOException {
    BufferedOutputStream file_output_stream;
    NSXMLOutputStream xml_stream;
    NSXMLOutputFormat format;

    // Create an output stream to the file.
    file_output_stream = new BufferedOutputStream(new FileOutputStream(filename));

    // Create object output stream.
    xml_stream = new NSXMLOutputStream(file_output_stream);

    // Set the format of the output document.
    format = new NSXMLOutputFormat(true);  // turn indentation on
    format.setEncoding("UTF-16");          // set encoding to UTF-16
    xml_stream.setOutputFormat(format);    // apply format to the stream

    return xml_stream;
}
```

For more information on NSXMLOutputFormat, see the API documentation.

CHAPTER 5

Transformation of XML Documents

Serializing objects and data into XML documents is a great way of sharing information between applications within an organization. However, communicating that data between companies can be difficult. For example, you can serialize an NSArray containing InventoryItem objects into an XML document and send that document to your business partners over the Internet. But, unless your business partners are also running WebObjects (in fact, they would have to be running the same version of WebObjects that you are running), they will find it difficult to make use of the document. Of course, they can create an XSLT stylesheet that transforms your XML document into a format that they can use, but you can make their job easier by doing the transformation yourself.

Because your application generates XML documents, you're in an excellent position for converting serialized-data documents into a standard format that the recipients of your documents can use. If you're comfortable with XSL Transformations (XSLT), you can create an XSLT file that WebObjects can use to transform the output of XML serialization into other formats.

While this book does not teach XSLT, this chapter gives you an overview of the transformation process. It contains the following sections:

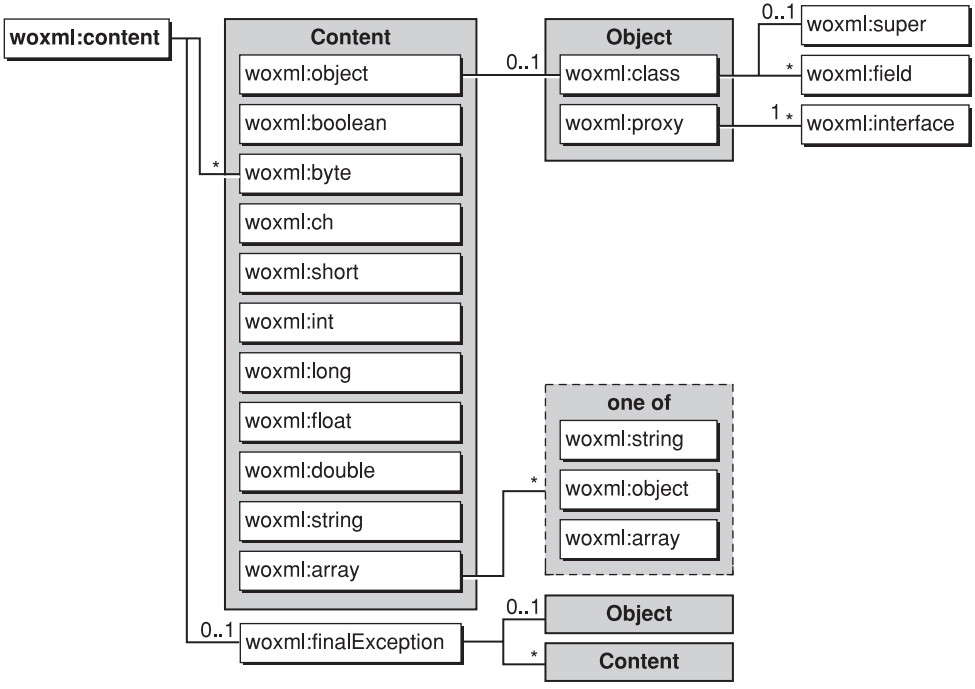
- "Structure of Serialized Data in WebObjects" (page 68) shows you the structure of the XML documents generated by NSXMLOutputStream.
- "XSL Transformations" (page 69) gives an overview of the transformation process.
- "XML Parsers and XSLT Processors" (page 71) explains how WebObjects uses the Java API for XML Processing (JAXP) to communicate with XML parsers and transformers, which allows you to install and use your preferred implementations.

- "Serialization and Transformation Performance" (page 72) touches on performance issues with XML serialization and transformation.

Structure of Serialized Data in WebObjects

The structure of the XML documents created by the WebObjects XML serialization process is described by the `woxml.xsd` and `woxml.dtd` files, which are listed in "XML Schema and DTD Files" (page 85). Figure 5-1 illustrates the structure that the files define, while Listing 5-1 shows an example of a target document.

Figure 5-1 Diagram of the schema for WebObjects XML serialization



Listing 5-1 Example of a target document

```
<?xml version="1.0" encoding="UTF-8"?>
<content xmlns="http://www.apple.com/webobjects/XMLSerialization"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.apple.com/webobjects/XMLSerialization
http://www.apple.com/webobjects/5.2/schemas/woxml.xsd">
    <int>5</int>
    <boolean>true</boolean>
    <ch>u</ch>
    <double>3.14</double>
</content>
```

XSL Transformations

This book does not teach you XSL Transformations (XSLT). There are several books available on the subject that explain the specification and different implementations of it in detail. However, this section explains some segments of the `SimpleTransformation.xsl` script used in this book's transformation-example project. You can find the entire listing of the transformation script in Listing B-3 (page 122).

XSLT is a declarative language. This means that the transformation of an XML document is expressed as a set of rules or templates that are applied to elements of the source document to create elements of the target document. For example, you can specify a rule that changes every `date` element in a document to an `invoice_date` element.

Listing 5-2 shows the segment of `SimpleTransformation.xsl` that processes `woxml:object` elements.

Listing 5-2 Section of `SimpleTransformation.xsl` that processes `woxml:object` elements

```
<!-- Processes woxml:object elements. -->
<xsl:template name="process_object" match="woxml:object">
```

```
    <!-- extract class name -->
    <xsl:variable name="className">                                          //1
        <xsl:value-of select="woxml:class/@name" />
    </xsl:variable>

    <!-- get base class name -->
    <xsl:variable name="class">                                              //2
        <xsl:call-template name="basename">
            <xsl:with-param name="path" select="$className"/>
        </xsl:call-template>
    </xsl:variable>

    <!-- determine the element name -->
    <xsl:variable name="tag">                                                //3
        <xsl:choose>
            <xsl:when test="$class='NSDictionary' or
                      $class='NSMutableDictionary'">
                <xsl:value-of select="'dictionary'" />
            </xsl:when>
            <xsl:when test="$class='NSArray' or $class='NSMutableArray'">
                <xsl:value-of select="'array'" />
            </xsl:when>
            <xsl:otherwise>
                <xsl:value-of select="$class" />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:variable>

    <!-- create the element -->
    <xsl:element name="{$tag}">                                              //4
        <xsl:choose>
            <xsl:when test="$class='NSDictionary' or
                      $class='NSMutableDictionary'">
                <xsl:call-template name="process_dictionary" />
            </xsl:when>
            <xsl:otherwise>
                <xsl:call-template name="process_object_content" />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:element>
</xsl:template>
```

Here's an explanation of the numbered lines:

1. Gets the class name that the object represents and stores it in a variable called `className`. The rule gets the class name from the `name` attribute of the `woxml:class` element of `woxml:object`.
2. Calls a utility template that extracts the base class name from the fully qualified class name. This base class name is stored in the `class` variable.
3. Determines the name of the element in the target document that corresponds to the `woxml:object` element of the source document. The element name is `dictionary` (when `class` is `'NSDictionary'` or `'NSMutableDictionary'`), `array` (when `class` is `'NSArray'` or `'NSMutableArray'`), or the name of the base class that the `woxml:object` element contains.
4. Creates the element and its contents by invoking one of two templates: `process_dictionary` or `process_object_content`. The `process_dictionary` template creates a `dictionary` element in the target document using either two arrays (one for the keys and another for the values) or a set of `item` elements, each containing a `key` and a `value` element.

For more details on transforming XML documents using `SimpleTransformation.xsl`, see "Transforming XML Documents" (page 75). To learn XSLT, check out *XSLT* (published by O'Reilly) or *XSLT Programmer's Reference* (published by Wrox Press).

XML Parsers and XSLT Processors

An XML parser is software that allows you to read and write XML documents. An XSLT processor converts an XML document into another document, whose format can be XML, HTML, PDF, or any other format supported by the transformer. There are some parsers that can also convert XML documents, such as Microsoft's MSXML3 parser.

One of a parser's duties is to validate the input document, to make sure that it's well formed and that its contents conform to the document's XML Schema file or DTD file. The WebObjects XML schema files are listed in "XML Schema and DTD Files" (page 85). In WebObjects the source document is not validated by default; however, you can turn validation on to debug an application.

WebObjects uses the Java API for XML Processing (JAXP), implemented in the `javax.xml.parsers` and `javax.xml.tranform` packages (including `javax.xml.transform.sax`, `javax.xml.transform.dom`, and `javax.xml.transform.stream`) to instantiate and communicate with the XML parser and XSLT transformer. This allows you to install your preferred parser and transformer for use by your applications. See the API documentation of those packages for additional details. You can also consult Sun's JAXP tutorial, located at http://java.sun.com/xml/jaxp/docs.html.

A standard WebObjects installation includes the Xerces XML parser and the Xalan XSLT processor. However, thanks to JAXP, you can use other parsers and processors if you wish. Just install the pertinent JAR files on your computer, make sure that they are in the Java classpath, and point the `javax.xml.parsers.SAXParserFactory` to the class that implements the factory class. For example, if the JAR file for the Crimson parser is in the classpath, you would add the following line to the `Properties` file of the application project (which you can find under the Resources group) or to the command line to set the property:

```
-D"javax.xml.parsers.SAXParserFactory=
org.apache.crimson.jaxp.SAXParserFactoryImpl"
```

Keep in mind that if you have two parser-factory classes in your classpath, the parser that your application actually uses may not be the one you want. The parser that is loaded last is the one that the application uses. The same applies to the system properties `javax.xml.transform.TransformerFactory` and `javax.xml.parsers.DocumentBuilderFactory`: The application that is loaded last determines the system-wide values of these properties.

Serialization and Transformation Performance

XML serialization is slower than binary serialization because data is converted to XML code while objects are serialized. XML deserialization is slower than binary deserialization because XML documents need to be parsed before their contents can be deserialized. However, the actual speed at which data is serialized and deserialized is highly dependent on disk and network throughput.

To maximize the performance of XML serialization and deserialization in WebObjects, make sure that XML validation is not turned on (it's turned off by default). You turn XML validation on or off by setting the `NSXMLValidation` property in the command line or the `Properties` file:

```
-DNSXMLValidation=<true|false>
```

XML-parsing technology should improve over time. In addition, as mentioned in "XML Parsers and XSLT Processors" (page 71), WebObjects uses JAXP to ensure that a standard API is used to communicate with the parser. This allows you to install and use parsers as they become available.

CHAPTER 6

Transforming XML Documents

In Chapter 4, "Serializing Objects and Data" (page 37), you learned how to serialize and deserialize objects and primitive-type values. This chapter explains how you transform a stream containing serialized data into an XML document using an XSLT script.

The chapter contains the following sections:

- "The Transformation Process" (page 75) contains example code fragments that perform transformations.
- "Creating the Transformation Project" (page 79) shows you how to create the project.
- "Transforming Primitive-Type Values Using Keys" (page 80) explains how to create an XML document from data serialized using keys.
- "Transforming an Array of Movies" (page 82) shows you how to transform an NSArray of custom objects into an XML document.

The Transformation Process

The XMLSerializer class of the Serialization project, introduced in"XML Serialization Example" (page 53) and listed in Listing B-2 (page 115) includes the `transformObject` method, shown in Listing 6-1.

Listing 6-1 The transform method in `XMLSerializer .java`

```
/**
 * Serializes objects and data to a stream, which can also be
 * transformed. The product of the process is written to a file.
 *
 * @param source          object to serialize or transform
 * @param filename        filename of the target document,
 *                        including path and extension
 * @param transformation  type of transformation to perform;
 *                        indicates which transformation script to use.
 *                        When <code>null</code>, no transformation
 *                        is performed, only serialization.
 *
 * @return <code>true</code> when the process succeeds.
 */
public static boolean transformObject(Object source, String filename, String
transformation) {
    boolean success = false;

    try {
        // Create a stream to the output file.
        NSXMLOutputStream stream = (NSXMLOutputStream)openStream(filename, false,
transformation);                                                                    //1

        // Serialize data to XML output stream.
        stream.writeObject(source);

        stream.flush();
        closeStream(filename);

        success = true;
    }

    catch (IOException e) {
        e.printStackTrace();
    }

    return success;
}
```

As you can see, the `transformObject` method opens an output stream (line numbered 1) to a file using the `openStream` method (Listing 6-2), which initializes the XML transformer using the `initializeTransformer` method (1), shown in Listing 6-3 (page 78).

Listing 6-2 The `openStream` method in `XMLSerializer.java`

```
/**
 * Opens a file stream to or from a file and a corresponding
 * output or input object stream.
 * Adds the pair of streams to an internal dictionary for use by
 * the <code>closeStream</code> method.
 *
 * @param filename         fully qualified filename of the
 *                         target or source file; identifies
 *                         the channel to open
 * @param input_stream     indicates whether the stream returned
 *                         is an input stream or an output stream:
 *                         <code>true</code> for an input stream and
 *                        <code>false</code> for an output stream.
 * @param transformation  type of transformation to perform;
 *                        indicates which transformation script to use.
 *                        When <code>null</code>, no transformation
 *                        is performed, only serialization.
 *
 * @return object stream, <code>null</code> when the stream
 *         could not be created.
 */
private static Object openStream(String filename, boolean input_stream, String
transformation) throws IOException {
    BufferedOutputStream file_output_stream = null;
    BufferedInputStream file_input_stream = null;
    Channel channel;
    Object xml_stream = null;

    if (input_stream) {
        // Create an input stream from the file.
        file_input_stream = new BufferedInputStream(new FileInputStream(filename));

        // Create object input stream.
```

```
        xml_stream = new NSXMLInputStream(file_input_stream);

        channel = new Channel(file_input_stream, xml_stream, input_stream);
    } else {
        // Create an output stream to the file.
        file_output_stream = new BufferedOutputStream(new FileOutputStream(filename));

        // Create object output stream.
        if (transformation != null) {
            xml_stream = initializeTransformer(file_output_stream, transformation); //1
        } else {
            xml_stream = new NSXMLOutputStream(file_output_stream);
        }

        // Set the format of the output document (XML).
        NSXMLOutputFormat format = new NSXMLOutputFormat(true);
        ((NSXMLOutputStream)xml_stream).setOutputFormat(format);

        channel = new Channel(file_output_stream, xml_stream, input_stream);
    }
    channels.setObjectForKey(channel, filename);

    return xml_stream;
}
```

Listing 6-3 The `initializeTransformer` method in `XMLSerializer.java`

```
/**
 * Initializes the transformer.
 *
 * @param file_stream       target file stream
 * @param transformation    type of transformation to perform;
 *                          indicates which transformation file to use
 *
 * @throws IOException when there's a problem initializing the transformer.
 */
private static NSXMLOutputStream initializeTransformer(BufferedOutputStream file_stream,
String transformation) throws IOException {
    NSXMLOutputStream xml_stream = new NSXMLOutputStream(file_stream, new
File(transformationURI(transformation)));                                    //1
```

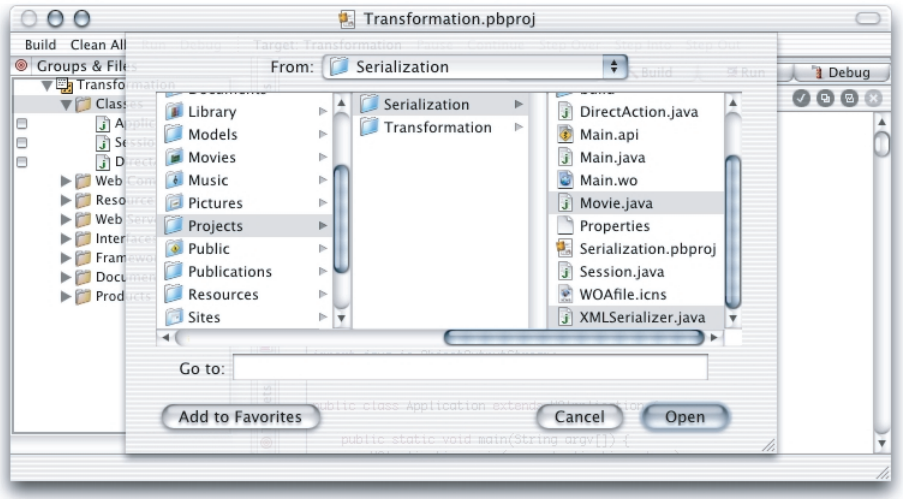
```
    Transformer transformer = ((NSXMLOutputStream)xml_stream).transformer();
    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");

    return xml_stream;
}
```

The `initializeTransformer` method takes a stream to a file in the *file_stream* parameter, which it uses to create the NSXMLOutputStream object (1) that transforms whatever is written to it using the transformation script indicated by the *transformation* parameter. Then it sets an output property of the transformer; in this case it sets the indentation level of the new document.

Creating the Transformation Project

This section shows you how to create the Transformation project, which is based on the Serialization project created in "Serializing Objects and Data" (page 37). (You can avoid all the manual work by copying the Transformation folder in `projects/Transformation/Starter` to your working directory.)

1. In Project Builder, create a WebObjects application project and name it `Transformation`.
2. Add an empty file to the project's Resources group and name it `SimpleTransformation.xsl` and enter the XSLT transformation script in Listing B-3 (page 122) as the file's contents. (Alternatively, you can add the `SimpleTransformation.xsl` file in `projects/Transforming/Starter/Transformation` to your project.) Assign `SimpleTransformation.xsl` to the Application Server target.
3. Copy the `Application.java` file from the Serialization project's folder into the Transformation project's folder.
4. Add `Movie.java` and `XMLSerializer.java` from the Serialization project to the Transformation project. Make sure to copy the files to the Transformation folder and to assign them to the Application Server target.



Transforming Primitive-Type Values Using Keys

This section shows you how to convert a stream generated by NSXMLOutputStream into an XML document in which the element name of each data element is derived from the value of the `key` attribute of each serialized object.

All you have to do is copy the `serializePrimitives` method of `Application.java` and paste it at the bottom of the file. Then edit the method so that it looks like Listing 6-4 (change the numbered lines). Listing B-3 (page 122) shows the XSLT script used to perform the transformation.

Listing 6-4 The `transformPrimitives` method in the Application class

```
/**
 * Transforms a set of primitive values.                                    //1
 *
 * @param filename      identifies the target file including its             //2
 *                      path and extension
```

```
 * @param an_int        value to serialize
 * @param a_boolean     value to serialize
 * @param a_char        value to serialize
 * @param a_double      value to serialize
 */
public void transformPrimitives(filename, int an_int, boolean a_boolean, char a_char,
double a_double) {                                                                    //3
    try {
        // Open an output stream.
        NSXMLOutputStream stream = XMLSerializer.openOutputStream(filename,
XMLSerializer.TRANSFORM_SIMPLE);                                                      //4

        // Write values.
        stream.writeInt(an_int, "my_integer");
        stream.writeBoolean(a_boolean, "my_boolean");
        stream.writeChar(a_char, "my_char");
        stream.writeDouble(a_double, "my_double");

        // Close the stream.
        XMLSerializer.closeStream(filename);
    }

    catch (IOException e) {
        e.printStackTrace();
    }
}
```

Now, add a call to the `transformPrimitives` method to Application's constructor, (for example, the code lines below) and build and run the application.

```
// Transform a set of primitive values.
transformPrimitives("/tmp/PrimitivesTransformed.xml", 5, true, 'u', 3.14);
```

In the transformation process, the document in Listing 6-5 is transformed into the one in Listing 6-6. The first document is not written to a file; it's the source document that the XSLT processor uses to produce the document that is actually written to the file system.

Listing 6-5 The source document: produced by NSXMLOutputStream before transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<content ...>
    <int key="my_integer">5</int>
    <boolean key="my_boolean">true</boolean>
    <ch key="my_char">u</ch>
    <double key="my_double">3.14</double>
</content>
```

Listing 6-6 The target document: `PrimitivesTransformed.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<content>
  <my_integer>5</my_integer>
  <my_boolean>true</my_boolean>
  <my_char>u</my_char>
  <my_double>3.14</my_double>
</content>
```

Transforming an Array of Movies

As you may recall, "Serializing Custom Serializable Objects to an XML Document" (page 61), explains how to serialize a custom object with key values. Listing 4-14 (page 62) shows the document generated. This section explains how to transform that document into another XML document.

Copy the `serializeMovieArray` method of `Application.java` to the end of the file and rename it to `transformMovieArray`. Edit the new method so that it looks like Listing 6-7 (change the numbered lines).

Listing 6-7 The `transformMovieArray` method in `Application.java`

```
/**
 * Transforms a Movie array.                                                    //1
 *
 * @param filename   identifies the target file, including
 *                   its path and extension
 */
public void transformMovieArray(String filename) {                             //2
    // Set the local time zone.
    NSTimeZone timeZone = NSTimeZone.timeZoneWithName("America/Los_Angeles", true);

    // Initialize the array.
    NSMutableArray movies = new NSMutableArray();
    movies.addObject(new Movie("Alien", "20th Century Fox", new NSTimestamp(1979, 10, 25,
0, 0, 0, timeZone)));
    movies.addObject(new Movie("Blade Runner", "Warner Brothers", new NSTimestamp(1982,
1, 3, 0, 0, 0, timeZone)));
    movies.addObject(new Movie("Star Wars", "20th Century Fox", new NSTimestamp(1977, 12,
29, 0, 0, 0, timeZone)));

    // Transform the array.
    XMLSerializer.transformObject(movies, filename, XMLSerializer.TRANSFORM_SIMPLE);//3
}
```

Add the following code lines to Application's constructor and build and run the application.

```
// Transform an array of Movie objects.
transformMovieArray("/tmp/MoviesTransformed.xml");
```

Listing 6-8 shows the product of the transformation. Notice how the transformation script (Listing B-3 (page 122)) replaced the reference to the studio of *Star Wars* (see Listing 4-14 (page 62)) with the correct value (20th Century Fox).

Listing 6-8 The `MoviesTransformed.xml` file

```
<?xml version="1.0" encoding="UTF-8"?>
<content>
  <array>
```

```
    <Movie>
      <title>Alien</title>
      <studio>20th Century Fox</studio>
      <release_date>1979-10-25 07:00:00 Etc/GMT</release_date>
    </Movie>
    <Movie>
      <title>Blade Runner</title>
      <studio>Warner Brothers</studio>
      <release_date>1982-01-03 08:00:00 Etc/GMT</release_date>
    </Movie>
    <Movie>
      <title>Star Wars</title>
      <studio>20th Century Fox</studio>
      <release_date>1977-12-29 08:00:00 Etc/GMT</release_date>
    </Movie>
  </array>
</content>
```

APPENDIX A

XML Schema and DTD Files

The following sections include the listings of the XML Schema file and DTD file used to validate XML documents generated by NSXMLOutputStream.

XML Schema File

Listing A-1 shows the contents of the `woxml.xsd` file, located at http://www.apple.com/webobjects/5.2/schemas/woxml.xsd.

Listing A-1 The `woxml.xsd` file

```
<?xml version="1.0" encoding="US-ASCII"?>
<!--
(c) 2002, Apple Computer, Inc. All rights reserved.
This document and the product to which it pertains are distributed under license
restricting its use, copying, distribution, and decompilation. This document may
be reproduced and distributed but may not be changed without prior written
authorization of Apple Computer, Inc. (Apple) and its licensors, if any. Any
redistribution must retain the above copyright notice and this list of
conditions regarding its use. TO THE EXTENT PERMITTED BY LAW, THIS DOCUMENT IS
PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL APPLE OR ITS LICENSORS, IF
ANY, BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

```
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF
ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Apple and WebObjects are trademarks
of Apple Computer, Inc. registered in the U.S. and other countries.
-->

<!--
This is the schema of WebObjects default XML serialization output.
-->
<schema targetNamespace="http://www.apple.com/webobjects/XMLSerialization"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:woxml="http://www.apple.com/webobjects/XMLSerialization"
        elementFormDefault="qualified">

  <annotation>
    <documentation xml:lang="en">
      Copyright 2002 Apple Computer. All rights reserved.
    </documentation>
  </annotation>

  <!--
  The root element.
  -->
  <element name="content">
    <complexType>
      <sequence>

        <!--
        An unordered list of elements. See "woxml:ContentType" for more details
        about what is allowed in this list.
        -->
        <group ref="woxml:ContentType" minOccurs="0" maxOccurs="unbounded" />

        <!--
        This type describes the exception that caused the serialization process
        to terminate. See "woxml:FinalExceptionType" for more information.
        -->
        <element name="finalException" type="woxml:FinalExceptionType" minOccurs="0"
maxOccurs="1" />
      </sequence>
    </complexType>
```

```
</element>

<!--
This type defines the unordered list of elements that constitute the root element.
It contains eight primitive types and three object types.
-->
<group name="ContentType">
  <choice>
    <element name="boolean" type="woxml:BooleanType" />
    <element name="byte" type="woxml:ByteType" />
    <element name="ch" type="woxml:CharType" />
    <element name="short" type="woxml:ShortType" />
    <element name="int" type="woxml:IntType" />
    <element name="long" type="woxml:LongType" />
    <element name="float" type="woxml:FloatType" />
    <element name="double" type="woxml:DoubleType" />
    <element name="string" type="woxml:StringType" />
    <element name="object" type="woxml:ObjectType" />
    <element name="array" type="woxml:ArrayType" />
  </choice>
</group>

<!--
////////////////////////////////////////////////////////////////////
////////////////// Basic Primitive Types Definition ////////////////
-->

<!--
If an element represents the content of a member that is part of a Java
object, it has field attributes.
-->
<attributeGroup name="FieldAttributes">

  <!--
  Name of the field the element represents.
  -->
  <attribute name="field" type="string" />

  <!--
  Sometimes a field of the same name and type exists somewhere in the
  class hierarchy that the object is an instance of; this attribute
```

```
  identifies the field unambiguously. The absence of this attribute means that the
  field is in the leaf class.
  -->
  <attribute name="classId" type="int" />
</attributeGroup>

<!--
Primitive boolean type.
-->
<complexType name="BooleanType">
  <simpleContent>
    <extension base="boolean">

      <!--
      A key that can be used in XSLT to become the tag name for this content.
      -->
      <attribute name="key" type="string" />
      <attributeGroup ref="woxml:FieldAttributes" />
    </extension>
  </simpleContent>
</complexType>

<!--
Primitive byte type.
-->
<complexType name="ByteType">
  <simpleContent>
    <extension base="byte">

      <!--
      A key that can be used in XSLT to become the tag name for this content.
      -->
      <attribute name="key" type="string" />
      <attributeGroup ref="woxml:FieldAttributes" />
    </extension>
  </simpleContent>
</complexType>

<!--
Primitive char base type. Note that not all Unicode characters are
representable in XML. Notably, \u0000 - \u001f, \u007f, \ufffe and \uffff
```

```
cannot be written natively as XML data. All illegal characters,
including those just mentioned, are written out in the familiar Java
notation \uXXXX. For further explanation of illegal XML characters, consult
the official XML recommendation from W3C.
-->
<simpleType name="char">
  <restriction base="string">

    <!--
    Length could be 6 because of illegal XML chars; for example, \u0001.
    -->
    <minLength value="1" fixed="true"/>
    <maxLength value="6" fixed="true"/>
  </restriction>
</simpleType>

<!--
Primitive char type.
-->
<complexType name="CharType">
  <simpleContent>
    <extension base="woxml:char">

      <!--
      A key that can be used in XSLT to become the tag name for this content.
      -->
      <attribute name="key" type="string" />
      <attributeGroup ref="woxml:FieldAttributes" />
    </extension>
  </simpleContent>
</complexType>

<!--
Primitive short type.
-->
<complexType name="ShortType">
  <simpleContent>
    <extension base="short">

      <!--
      A key that can be used in XSLT to become the tag name for this content.
```

```
      -->
      <attribute name="key" type="string" />
      <attributeGroup ref="woxml:FieldAttributes" />
    </extension>
  </simpleContent>
</complexType>

<!--
Primitive integer type.
-->
<complexType name="IntType">
  <simpleContent>
    <extension base="int">

      <!--
      A key that can be used in XSLT to become the tag name for this content.
      -->
      <attribute name="key" type="string" />
      <attributeGroup ref="woxml:FieldAttributes" />
    </extension>
  </simpleContent>
</complexType>

<!--
Primitive long type.
-->
<complexType name="LongType">
  <simpleContent>
    <extension base="long">

      <!--
      A key that can be used in XSLT to become the tag name for this content.
      -->
      <attribute name="key" type="string" />
      <attributeGroup ref="woxml:FieldAttributes" />
    </extension>
  </simpleContent>
</complexType>

<!--
Primitive float type.
```

```
-->
<complexType name="FloatType">
  <simpleContent>
    <extension base="float">

      <!--
      A key that can be used in XSLT to become the tag name for this content.
      -->
      <attribute name="key" type="string" />
      <attributeGroup ref="woxml:FieldAttributes" />
    </extension>
  </simpleContent>
</complexType>

<!--
Primitive double type.
-->
<complexType name="DoubleType">
  <simpleContent>
    <extension base="double">

      <!--
      A key that can be used in XSLT to become the tag name for this content.
      -->
      <attribute name="key" type="string" />
      <attributeGroup ref="woxml:FieldAttributes" />
    </extension>
  </simpleContent>
</complexType>

<!--
////////////////////////////////////////////////////////////////////////
////////////////////////// Object Types Definition ///////////////////////
-->

<!--
The "id" attribute refers to the identification number of an element
representing an object. It is generated the first time the object is
encountered during serialization. Subsequent references to the same object use
the attribute "idRef" instead of a new element with the complete object
description.
```

```
Both "id" and "idRef" should be declared as type ID and IDREF, respectively.
Unfortunately, the current XML specification states that values of those types
have to start with a letter or underscore character (_). In the name of clarity,
we chose not to use prefixes.

For a null object, neither "id" nor "idRef" are required.
-->
<attributeGroup name="IdAttributes">
  <attribute name="id" type="long" />
  <attribute name="idRef" type="long" />
</attributeGroup>

<!--
Attributes that belong to an element representing an object.
-->
<attributeGroup name="ObjectAttributes">

  <!--
  A key that can be used in XSLT to become the tag name for this content.
  -->
  <attribute name="key" type="string" />
  <attributeGroup ref="woxml:IdAttributes" />
  <attributeGroup ref="woxml:FieldAttributes" />
</attributeGroup>

<!--
This element represents java.lang.String objects. Because of ambiguity related to
\u0009(tab) and \u000a(newline), it uses the more cryptic notation ![CDATA[]]
to ensure that these characters are represented correctly. If a string has no
whitespace, it's simply represented as normal text.

If the string contains illegal characters, they are represented by the "ch" element
with \uXXXX as the text data. See the definition for "char" above for more details.
Carriage return has to be encoded as <ch>\u000d</ch> because of reasons given
in http://www.w3.org/TR/2000/REC-xml-20001006#sec-line-ends

Examples:

<string id="20">Testing<ch>\u0009</ch>illegal<ch>\u0001</ch>chars</string>
```

```
<string id="39">Well Done!</string>

<string id="42">There is a tab <!CDATA[]]> here</string>

When you serialize a string using the writeUTF method, the corresponding string
element does not have an "id" attribute and, thus, is not referenced by an "idRef"
attribute elsewhere in the document. It is essentially "unshared".
-->
<complexType name="StringType" mixed="true">
  <sequence>
    <element name="ch" type="woxml:CharType" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attributeGroup ref="woxml:ObjectAttributes" />
</complexType>

<!--
An ordinary object, everything that is not a string or an array. If an object
is null, it is represented as an empty element. If an object has already
been written out before, with a unique "id" attribute, it is
represented as an empty element with its "idRef" attribute set to the same value
as the "id" of the original object. This eliminates the circular-object-graph problem.
-->
<complexType name="ObjectType">
  <sequence>

    <!--
    When the element represents an object element, the first child element
    describes the class structure. The class can be a real class or a
    proxy class.
    -->
    <choice minOccurs="0" maxOccurs="1">
      <element name="class" type="woxml:ClassType" />
      <element name="proxy" type="woxml:ProxyType" />
    </choice>

    <!--
    The actual content of the object.
    -->
    <group ref="woxml:ObjectContentType" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
```

```
  <!--
  When "type" is present, it refers to two special, degenerated objects, which
  are instances of java.lang.Class and java.io.ObjectStreamClass. They are
  degenerated because they are represented in a more concise manner (only
  their content is written out). Ordinary objects have their class structure
  written out as well.
  -->
  <attribute name="type" type="string" />
  <attributeGroup ref="woxml:ObjectAttributes" />
</complexType>

<!--
This defines the "class" element, which describes the class structure of an
object. Multiple references to the same class in a document are handled using
the mechanism described in Object Types Definition above.
-->
<complexType name="ClassType">
  <sequence>

    <!--
    A serializable field of a class.
    -->
   <element name="field" type="woxml:FieldType" minOccurs="0" maxOccurs="unbounded" />

    <!--
    The usual superclass description.
    -->
    <element name="super" type="woxml:ClassType" minOccurs="0" />
  </sequence>
  <attributeGroup ref="woxml:IdAttributes" />

  <!--
  This attribute gives more detail about the class, such as whether it is
  Serializable or Externalizable, whether it has overridden the writeObject
  method, and so on.

  The attribute is important only for deserialization using NSXMLInputStream; you
  can ignore it otherwise.
  -->
  <attribute name="flag" type="int" />
```

```
  <!--
  Name of the class. Even when the "idRef" attribute is present, this attribute
  is required for clarity (and perhaps ease of transformation using XSLT).
  -->
  <attribute name="name" type="string" use="required" />

  <!--
  This attribute identifies the unique, original class version of this class. It is
  used for version control and is tied to the SerialVersionUID
  in the Java Binary Serialization specification.

  The attribute is important only for deserialization using NSXMLInputStream; you
  can ignore it otherwise.
  -->
  <attribute name="suid" type="long" />
</complexType>

<!--
A serializable field of a class.
-->
<complexType name="FieldType">
  <attribute name="name" type="string" use="required" />

  <!--
  The class type of the field.
  -->
  <attribute name="type" type="string" use="required" />
</complexType>


<!--
Instead of a regular class, the type of an object could be
java.lang.reflect.Proxy.
-->
<complexType name="ProxyType">
  <sequence>
    <element name="interface" minOccurs="1" maxOccurs="unbounded">
      <complexType>
        <attribute name="name" type="string" use="required"/>
      </complexType>
    </element>
```

```
  </sequence>
  <attributeGroup ref="woxml:IdAttributes" />
</complexType>


<!--
Similar to "woxml:ContentType", with the additional choice "Ignore_EndDataBlock".
-->
<group name="ObjectContentType">
  <choice>
    <group ref="woxml:ContentType" />

    <!--
    This is important only for deserialization using NSXMLInputStream; you
    can ignore it otherwise.
    -->
    <element name="Ignore_EndDataBlock" />
  </choice>
</group>

<!--
This type describes the primitive array object in Java; for example, int[], which
is a legitimate Java object. However, as opposed to an ordinary "object"
element, there is no need for an elaborate class description. Instead, it is
succinctly represented with the "type" attribute.

Primitive types in an array are simply represented as text separated by a
space (0x0020). If the character 0x0020 is present as part of an array of
characters, it is escaped as \u0020.

Examples:
<array id="174" length="2" type="int[]">3 4 </array>
<array id="200" length="6" type="char[]">a b \ \u0020 c d </array>

Multiple references to the same array in a document are handled using
the mechanism described in Object Types Definition above.
-->
<complexType name="ArrayType" mixed="true">
  <sequence>
    <group ref="woxml:ObjectComponentType" minOccurs="0" maxOccurs="1" />
  </sequence>
```

```
    <!--
    Length of the array.
    -->
    <attribute name="length" type="int" />

    <!--
    Array type. If the type is "base64", it means that Base64 encoding was used to
    output an array of bytes.

    Examples:
        int[]                   array of ints
        char[][]                two-dimensional array of chars
        java.lang.String[]      array of Strings
    -->
    <attribute name="type" type="string" />
    <attributeGroup ref="woxml:ObjectAttributes" />
  </complexType>

  <!--
  If the component type of an array object is an object type, each
  component is represented as a "string", "object" or "array" element.
  -->
  <group name="ObjectComponentType">
    <choice>
     <element name="string" type="woxml:StringType" minOccurs="0" maxOccurs="unbounded"
/>
     <element name="object" type="woxml:ObjectType" minOccurs="0" maxOccurs="unbounded"
/>
     <element name="array" type="woxml:ArrayType" minOccurs="0" maxOccurs="unbounded" />
    </choice>
  </group>

  <!--
  This type describes the exception that caused the serialization process
  to terminate.
  If the serialization has an exception that causes the process to abort,
  that exception is considered final and is written out.
  NSXMLInputStream can actually read in this final exception and make
  sense of the failure.
  -->
```

```
  <complexType name="FinalExceptionType">
    <sequence>
      <choice minOccurs="0" maxOccurs="1">
        <element name="class" type="woxml:ClassType" />
        <element name="proxy" type="woxml:ProxyType" />
      </choice>
      <group ref="woxml:ObjectContentType" minOccurs="0" maxOccurs="unbounded" />
    </sequence>
    <attributeGroup ref="woxml:IdAttributes" />
  </complexType>

</schema>
```

DTD Document File

Listing A-2 shows the contents of `woxml.dtd`, located at http://www.apple.com/webobjects/5.2/schemas/woxml.dtd.

Listing A-2 The `woxml.dtd` file

```
<!--
(c) 2002, Apple Computer, Inc. All rights reserved.
This document and the product to which it pertains are distributed under license
restricting its use, copying, distribution, and decompilation. This document may
be reproduced and distributed but may not be changed without prior written
authorization of Apple Computer, Inc. (Apple) and its licensors, if any. Any
redistribution must retain the above copyright notice and this list of
conditions regarding its use. TO THE EXTENT PERMITTED BY LAW, THIS DOCUMENT IS
PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL APPLE OR ITS LICENSORS, IF
ANY, BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF
```

```
ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Apple and WebObjects are trademarks
of Apple Computer, Inc. registered in the U.S. and other countries.
-->

<!--
This is the DTD of WebObjects default XML serialization output. There is an equivalent
XML Schema file (woxml.xsd) that is semantically tighter due to use of namespaces and
type definitions. You should use the XML Schema file whenever possible.
-->

<!--
////////////////////////////////////////////////////////////////////
////////////////////////// Entity Definition ///////////////////////////
-->

<!--
This entity defines the unordered list of elements that constitute the
root element. It contains eight primitive types and three object types.
-->
<!ENTITY % ContentType "(boolean | byte | ch | short | int | long | float | double |
string | object | array)" >

<!--
////////////////////////////////////////////////////////////////////
//////////////////////////// The Root Element //////////////////////////
-->

<!ELEMENT content ((%ContentType;)*, finalException?)>

<!--
The root element has a few XML Schema attributes. We are faking them here.
-->
<!ATTLIST content
    xmlns  CDATA #FIXED "http://www.apple.com/webobjects/XMLSerialization"
    xmlns:xsi CDATA #FIXED "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation CDATA #FIXED "http://www.apple.com/webobjects/XMLSerialization
http://www.apple.com/webobjects/5.2/schemas/woxml.xsd">

<!--
////////////////////////////////////////////////////////////////////
////////////////// Basic primitive types definition ////////////////////
```

```
-->
<!--
Primitive boolean type.
-->
<!ELEMENT boolean (#PCDATA)>

<!--
key: A key that can be used in XSLT to become the tag name for this content.

field:
  Name of the field the element represents.

classId:
  Sometimes a field of the same name and type exists somewhere in the
  class hierarchy that the object is an instance of. The "classId" attribute
  identifies the field unambiguously. The absence of this attribute means that the
  field is in the leaf class.
-->
<!ATTLIST boolean
    key CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>

<!--
Primitive byte type.
-->
<!ELEMENT byte (#PCDATA)>
<!ATTLIST byte
    key CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>

<!--
Primitive char type.
-->
<!ELEMENT ch (#PCDATA)>
<!ATTLIST ch
    key CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>
```

```
<!--
Primitive short type.
-->
<!ELEMENT short (#PCDATA)>
<!ATTLIST short
    key CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>

<!--
Primitive integer type.
-->
<!ELEMENT int (#PCDATA)>
<!ATTLIST int
    key CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>

<!--
Primitive long type.
-->
<!ELEMENT long (#PCDATA)>
<!ATTLIST long
    key CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>

<!--
Primitive float type.
-->
<!ELEMENT float (#PCDATA)>
<!ATTLIST float
    key CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>
<!--
Primitive double type.
-->
<!ELEMENT double (#PCDATA)>
<!ATTLIST double
    key CDATA #IMPLIED
```

```
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>

<!--
//////////////////////////////////////////////////////////////////
/////////////////////// Object Types Definition //////////////////////
-->

<!--
This element represents java.lang.String objects. Because of ambiguity related to
\u0009(tab) and \u000a(newline), it uses the more cryptic notation ![CDATA[]]
to ensure that these characters are represented correctly. If a string has no
whitespace, it's simply represented as normal text.

If the string contains illegal characters, they are represented by the "ch" element
with \uXXXX as the text data. See the definition for "char" above for more details.
Carriage return has to be encoded as <ch>\u000d</ch> because of reasons given
in http://www.w3.org/TR/2000/REC-xml-20001006#sec-line-ends

Examples:

  <string id="20">Testing<ch>\u0009</ch>illegal<ch>\u0001</ch>chars</string>

  <string id="39">Well Done!</string>

  <string id="42">There is a tab <!CDATA[]]> here</string>

When you serialize a string using the writeUTF method, the corresponding string
element does not have an "id" attribute and, thus, is not referenced by an "idRef"
attribute elsewhere in the document. It is essentially "unshared".
-->
<!ELEMENT string (#PCDATA | ch)*>


<!--
key: A key that can be used in XSLT to become the tag name for this content.

id, idRef:
  The "id" attribute refers to the identification number of an element
  representing an object. It is generated when the object is encountered the
  first time during serialization. Subsequent references to the same object use
```

```
  the attribute "idRef" instead of a new element with the complete object
  description.

  Both "id" and "idRef" should be declared as type ID and IDREF respectively.
  Unfortunately, the current XML specification insists that values of those types
  have to start with a letter or an underscore character (_). In the name of clarity,
  we chose not to use prefixes.

  For a null object, neither "id" nor "idRef" are required.

field:
  Name of the field the element represents.

classId:
  Sometimes a field of the same name and type exists somewhere in the
  class hierarchy that the object is an instance of. The "classId" attribute
  identifies the field unambiguously. The absence of this attribute means that the
  field is in the leaf class.
-->
<!ATTLIST string
    key CDATA #IMPLIED
    id CDATA #IMPLIED
    idRef CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>
<!--
An ordinary object, everything that is not a string or an array. If an object
is null, it will be represented as an empty element. If an object has already
been written out before, with a unique "id" attribute, it is
represented as an empty element with its "idRef" attribute set to
the same value as the "id" of the original object. This eliminates the
circular-object-graph problem.

When the element represents an object element, the first child element
describes the class structure. The class can be a real class or a
proxy class.
-->
<!ELEMENT object ((class | proxy)?, (%ContentType; | Ignore_EndDataBlock)*)>

<!--
type:
```

```
   When type is present, it refers to two special, degenerated objects, which
   are instances of java.lang.Class and java.io.ObjectStreamClass. They are
   degenerated because they are represented in a more concise manner (only
   their content is written out). Ordinary objects have their class structure
   written out as well.
-->
<!ATTLIST object
    type CDATA #IMPLIED
    key CDATA #IMPLIED
    id CDATA #IMPLIED
    idRef CDATA #IMPLIED
    field CDATA #IMPLIED
    classId CDATA #IMPLIED>

<!--
This defines the "class" element, which describes the class structure of an
object. Multiple references to the same class in a document are handled using
the mechanism described in Object Types Definition above.
-->
<!ELEMENT class (field*, super?)>

<!--
flag:
   This attribute gives more detail about the class, such as whether it is
   Serializable or Externalizable, whether it has overridden the writeObject
   method, and so on.

   The attribute is important only for deserialization using NSXMLInputStream; you
   can ignore it otherwise.

name:
   Name of the class. Even if the "idRef" attribute is present, this
   attribute is required for clarity (and perhaps ease of transformation using
   XSLT).

suid:
   This attribute identifies the unique, original class version of this class.
   It is used for version control and is tied to the SerialVersionUID
   in the Java Binary Serialization specification.

   The attribute is important only for deserialization using NSXMLInputStream; you
```

```
   can ignore it otherwise.
-->
<!ATTLIST class
    id CDATA #IMPLIED
    idRef CDATA #IMPLIED
    flag CDATA #IMPLIED
    name CDATA #IMPLIED
    suid CDATA #IMPLIED>
<!--
A serializable field of a class.
-->
<!ELEMENT field EMPTY>

<!--
type:
   The class type of the field.
-->
<!ATTLIST field
    name CDATA #REQUIRED
    type CDATA #REQUIRED>

<!--
The usual superclass description.
-->
<!ELEMENT super (field*, super?)>
<!ATTLIST super
    id CDATA #IMPLIED
    idRef CDATA #IMPLIED
    flag CDATA #IMPLIED
    name CDATA #IMPLIED
    suid CDATA #IMPLIED>

<!--
Instead of a regular class, the type of an object could be
java.lang.reflect.Proxy.
-->
<!ELEMENT proxy (interface+)>
<!ATTLIST proxy
    id CDATA #IMPLIED
    idRef CDATA #IMPLIED>
```

```
<!ELEMENT interface EMPTY>
<!ATTLIST interface
    name CDATA #REQUIRED>

<!--
This element describes the primitive array object in Java; for example, int[], which is
a legitimate Java object. However, as opposed to an ordinary "object"
element, there is no need for an elaborate class description. Instead, it is
succinctly represented with the "type" attribute.

Primitive types in an array are simply represented as text separated by a
space (0x0020). If the character 0x0020 is present as part of an array of
characters, it is escaped as \u0020.

Examples:
<array id="174" length="2" type="int[]">3 4 </array>
<array id="200" length="6" type="char[]">a b \ \u0020 c d </array>

Multiple references to the same array in a document are handled using
the mechanism described in Object Types Definition above.
-->
<!ELEMENT array (#PCDATA | string | object | array)*>

<!--
length:
   Length of the array.

type:
  Array type. If the type is "base64", it means that Base64 encoding was used to
  output an array of bytes.

  Examples:
    int[]            array of ints
    char[][]    two-dimensional array of chars
    java.lang.String[]array of Strings
-->
<!ATTLIST array
    key CDATA #IMPLIED
    id CDATA #IMPLIED
    idRef CDATA #IMPLIED
    field CDATA #IMPLIED
```

```
    classId CDATA #IMPLIED
    length CDATA #IMPLIED
    type CDATA #IMPLIED>
<!--
This element is important only for deserialization using NSXMLInputStream; you
can ignore it otherwise.
-->
<!ELEMENT Ignore_EndDataBlock EMPTY>

<!--
This type describes the exception that caused the serialization process
to terminate.
If the serialization has an exception that causes the process to abort,
that exception is considered final and is written out.
NSXMLInputStream can actually read in this final exception and make
sense of the failure.
-->
<!ELEMENT finalException ((class | proxy)?, (%ContentType; | Ignore_EndDataBlock)*)>
<!ATTLIST finalException
    id CDATA #IMPLIED
    idRef CDATA #IMPLIED>
```

APPENDIX A

XML Schema and DTD Files

Code Listings

This appendix contains the listings of the example serialization utility classes used in Chapter 4, "Serializing Objects and Data" (page 37), and Chapter 6, "Transforming XML Documents" (page 75), and the example transformation script, which is also used in the transformation of XML documents.

BinarySerialization.java

Listing B-1 shows the implementation of the `BinarySerialization.java` example class.

Listing B-1 `BinarySerializer.java` class

```
import com.webobjects.appserver.*;
import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.ClassNotFoundException;
```

```
/**
 * Manages serialization and deserialization of objects
 * to and from binary files.
 */
public class BinarySerializer {
    /**
     * Encapsulates a file stream and an object stream (a channel).
     */
    private static class Channel {
        protected Object file_stream;
        protected Object object_stream;
        protected boolean input_stream;

        Channel(Object file_stream, Object object_stream, boolean input_stream) {
            this.file_stream = file_stream;
            this.object_stream = object_stream;
            this.input_stream = input_stream;
        }
    }

    /**
     * Stores open channels.
     */
    private static NSMutableDictionary channels = new NSMutableDictionary();

    /**
     * Directory in which serialized data intended for
     * deserialization is stored.
     */
    private static final String FILE_PREFIX = "/tmp/";

    /**
     * Suffix (including extension) of files used to store serialized data.
     */
    private static final String FILE_SUFFIX = "_data.binary";

    /**
     * Serializes data to a file.
     *
     * @param source          object to serialize
     * @param identifier      file identifier for deserialization
```

```
     *                        (name of the file without the extension)
     *
     * @return <code>true</code> when the process succeeds.
     */
    public static boolean serializeObject(Object source, String identifier) {
    ObjectOutputStream binary_stream;
        String filename = FILE_PREFIX + identifier + FILE_SUFFIX;
        boolean success = false;

        try {
            // Create a stream to the output file.
            binary_stream = (ObjectOutputStream)BinarySerializer.openStream(filename,
false);

            // Serialize data to output stream.
            binary_stream.writeObject(source);

            // Close the stream.
            binary_stream.flush();
            closeStream(filename);

            success = true;
        }

        catch (IOException e) {
            e.printStackTrace();
        }

        return success;
    }

    /**
     * Deserializes data from a file.
     *
     * @param identifier      file identifier (name of the file
     *                        without the extension)
     *
     * @return deserialized object.
     */
    public static Object deserializeObject(String identifier) {
    ObjectInputStream binary_stream;
```

```
        String filename = FILE_PREFIX + identifier + FILE_SUFFIX;
        Object object = null;

        try {
            // Create a stream to the input file.
            binary_stream = (ObjectInputStream)openStream(filename, true);

            // Deserialize data from input stream.
            object = (Object)binary_stream.readObject();

            // Close the stream.
            closeStream(filename);
        }

        catch (IOException e) {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        return object;
    }

    /**
     * Opens an output stream.
     *
     * @param filename          fully qualified filename of the
     *                          target or source file; identifies
     *                          the channel to open.
     */
   public static ObjectOutputStream openOutputStream(String filename) throws IOException
{
        return (ObjectOutputStream)openStream(filename, false);
    }

    /**
     * Opens an input stream.
     *
     * @param filename           fully qualified filename of the
     *                           target or source file; identifies
```

```
     *                           the channel to open.
     */
    public static ObjectInputStream openInputStream(String filename) throws IOException {
        return (ObjectInputStream)openStream(filename, true);
    }

    /**
     * Opens a file stream to or from a file and a corresponding
     * output or input object stream.
     * The method adds the pair of streams to an internal dictionary
     * for use by the <code>closeStream</code> method.
     *
     * @param filename           fully qualified filename of the
     *                           target or source file; identifies
     *                           the channel to open.
     * @param input_stream       indicates whether the stream returned
     *                           is an input stream or an output stream:
     *                           <code>true</code> for an input stream and
     *                           <code>false</code> for an output stream.
     *
     * @return object stream, <code>null</code> when the stream could not
     *         be created.
     */
    private static Object openStream(String filename, boolean input_stream) throws
IOException {
        BufferedOutputStream file_output_stream = null;
        BufferedInputStream file_input_stream = null;
        Channel channel;
        Object binary_stream = null;

        if (input_stream) {
            // Create an input stream from the file.
            file_input_stream = new BufferedInputStream(new FileInputStream(filename));

            // Create object-input stream.
            binary_stream = new ObjectInputStream(file_input_stream);

            channel = new Channel(file_input_stream, binary_stream, input_stream);
        } else {
            // Create an output stream to the file.
```

```
            file_output_stream = new BufferedOutputStream(new
FileOutputStream(filename));

            // Create object-output stream.
            binary_stream = new ObjectOutputStream(file_output_stream);

            channel = new Channel(file_output_stream, binary_stream, input_stream);
        }
        channels.setObjectForKey(channel, filename);

        return binary_stream;
    }

    /**
     * Closes an object stream and its corresponding file stream.
     *
     * @param filename             fully qualified filename of the
     *                             target or source file; identifies
     *                             the streams to close.
     */
    public static void closeStream(String filename) throws IOException {
    Channel channel = (Channel)channels.objectForKey(filename);

    if (channel.input_stream) {
        ((ObjectInputStream)channel.object_stream).close();
        ((BufferedInputStream)channel.file_stream).close();
    } else {
        ((ObjectOutputStream)channel.object_stream).close();
        ((BufferedOutputStream)channel.file_stream).close();
    }

    channels.removeObjectForKey(filename);
    }
}
```

XMLSerializer.java

Listing B-2 shows the implementation of the `XMLSerializer.java` example class.

Listing B-2 `XMLSerializer.java` class

```
import com.webobjects.appserver.WOApplication;
import com.webobjects.appserver.WOResourceManager;
import com.webobjects.eocontrol.*;
import com.webobjects.foundation.*;
import com.webobjects.foundation.xml.*;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import javax.xml.transform.Transformer;

/**
 * Manages serialization and deserialization of objects
 * to and from XML files.
 */
public class XMLSerializer extends Object {
    /**
     * Encapsulates a file stream and an object stream (a channel).
     */
    private static class Channel {
        protected Object file_stream;
        protected Object object_stream;
```

```
    protected boolean input_stream;

    Channel(Object file_stream, Object object_stream, boolean input_stream) {
        this.file_stream = file_stream;
        this.object_stream = object_stream;
        this.input_stream = input_stream;
    }
}

/**
 * Identifier for a simple transformation.
 */
public static final String TRANSFORM_SIMPLE = "SimpleTransformation";

/**
 * Directory where serialized data is stored.
 */
private static final String FILE_PREFIX = "/tmp/";

/**
 * Suffix (including extension) of files used to store serialized data.
 */
private static final String FILE_SUFFIX = "_data.xml";

/**
 * Stores open channels.
 */
private static NSMutableDictionary channels = new NSMutableDictionary();

/**
 * Serializes data to a file.
 *
 * @param source          object to serialize
 * @param identifier      file identifier for deserialization
 *                        (name of the file without its path or extension)
 *
 * @return <code>true</code> when the process succeeds.
 */
public static boolean serializeObject(Object source, String identifier) {
    String filename = FILE_PREFIX + identifier + FILE_SUFFIX;
```

```
    boolean success = transformObject(source, filename, null);

    return success;
}

/**
 * Deserializes data from a file.
 *
 * @param identifier      file identifier
 *                        (name of the file without the extension)
 *
 * @return deserialized object.
 */
public static Object deserializeObject(String identifier) {
    String filename = FILE_PREFIX + identifier + FILE_SUFFIX;
    Object object = null;

    try {
        // Create a stream from the input file.
       NSXMLInputStream stream = (NSXMLInputStream)openStream(filename, true, null);

        // Deserialize data from input stream.
        object = stream.readObject();

        // Close stream
        closeStream(filename);
    }

    catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    return object;
}
```

```
    /**
     * Serializes objects and data to a stream, which can also be
     * transformed. The product of the process is written to a file.
     *
     * @param source          object to serialize or transform
     * @param filename        filename of the target document,
     *                        including path and extension
     * @param transformation  type of transformation to perform;
     *                        indicates which transformation script to use.
     *                        When <code>null</code>, no transformation
     *                        is to be performed, only serialization.
     *
     * @return <code>true</code> when the process succeeds.
     */
    public static boolean transformObject(Object source, String filename, String
transformation) {
        boolean success = false;

        try {
            // Create a stream to the output file.
            NSXMLOutputStream stream = (NSXMLOutputStream)openStream(filename, false,
transformation);

            // Serialize data to object output stream.
            stream.writeObject(source);

            stream.flush();
            closeStream(filename);

            success = true;
        }

        catch (IOException e) {
            e.printStackTrace();
        }

        return success;
    }

    /**
     * Opens an output stream.
```

```
     *
     * @param filename         fully qualified filename of the target
     *                         or source file; identifies the channel to open.
     * @param transformation   type of transformation to perform
     *                         (indicates which transformation file to use)
     */
    public static NSXMLOutputStream openOutputStream(String filename, String
transformation) throws IOException {
        return (NSXMLOutputStream)openStream(filename, false, transformation);
    }

    /**
     * Opens an input stream.
     *
     * @param filename   fully qualified filename of the target
     *                   or source file; identifies the channel to open.
     */
    public static NSXMLInputStream openInputStream(String filename) throws IOException {
        return (NSXMLInputStream)openStream(filename, true, null);
    }

    /**
     * Opens a file stream to or from a file and a corresponding
     * output or input object stream.
     * Adds the pair of streams to an internal dictionary for use by
     * the <code>closeStream</code> method.
     *
     * @param filename       fully qualified filename of the
     *                         target or source file; identifies
     *                         the channel to open
     * @param input_stream     indicates whether the stream returned
     *                         is an input stream or an output stream:
     *                         <code>true</code> for an input stream and
     *             <code>false</code> for an output stream.
     * @param transformation   type of transformation to perform;
     *                         indicates which transformation script to use.
     *                         When <code>null</code> no transformation
     *                         is performed, only serialization.
     *
     * @return object stream, <code>null</code> when the stream
     *         could not be created.
```

```
     */
    private static Object openStream(String filename, boolean input_stream, String
transformation) throws IOException {
        BufferedOutputStream file_output_stream = null;
        BufferedInputStream file_input_stream = null;
        Channel channel;
        Object xml_stream = null;

        if (input_stream) {
            // Create an input stream from the file.
            file_input_stream = new BufferedInputStream(new FileInputStream(filename));

            // Create object-input stream.
            xml_stream = new NSXMLInputStream(file_input_stream);

            channel = new Channel(file_input_stream, xml_stream, input_stream);
        } else {
            // Create an output stream to the file.
          file_output_stream = new BufferedOutputStream(new FileOutputStream(filename));

            // Create object-output stream.
            if (transformation != null) {
                xml_stream = initializeTransformer(file_output_stream, transformation);
            } else {
                xml_stream = new NSXMLOutputStream(file_output_stream);
            }

            // Set the format of the output document (XML).
            NSXMLOutputFormat format = new NSXMLOutputFormat(true);
            ((NSXMLOutputStream)xml_stream).setOutputFormat(format);

            channel = new Channel(file_output_stream, xml_stream, input_stream);
        }
        channels.setObjectForKey(channel, filename);

        return xml_stream;
    }

    /**
     * Closes an object stream and its corresponding file stream.
     *
```

```
     * @param filename   fully qualified filename of the
     *                   target or source file; identifies
     *                   the channel to close
     */
    public static void closeStream(String filename) throws IOException {
        Channel channel = (Channel)channels.objectForKey(filename);

        if (channel.input_stream) {
            ((NSXMLInputStream)channel.object_stream).close();
            ((BufferedInputStream)channel.file_stream).close();
        } else {
            ((NSXMLOutputStream)channel.object_stream).close();
            ((BufferedOutputStream)channel.file_stream).close();
        }

        channels.removeObjectForKey(filename);
    }

    /**
     * Computes the URI of a transformation file.
     *
     * @param transformation   type of transformation (does not
     *                         include the .xsl extension);
     *                         for example, "SimpleTransformation"
     *
     * @return relative path to the transformation file.
     */
    private static String transformationURI(String transformation) {
        WOApplication application = WOApplication.application();
        WOResourceManager resource_manager = application.resourceManager();
        String transformationURI =
resource_manager.pathForResourceNamed("SimpleTransformation" + ".xsl", null, null);

        return transformationURI;
    }

    /**
     * Initializes the transformer.
     *
     * @param file_stream       target file stream
     * @param transformationtype of transformation to perform;
```

```
     *                          indicates which transformation script to use
     *
     * @throws IOException when there's a problem initializing
     *         the transformer.
     */
    private static NSXMLOutputStream initializeTransformer(BufferedOutputStream
file_stream, String transformation) throws IOException {
        NSXMLOutputStream xml_stream = new NSXMLOutputStream(file_stream, new
File(transformationURI(transformation)));
        Transformer transformer = ((NSXMLOutputStream)xml_stream).transformer();
        transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");

        return xml_stream;
    }
}
```

SimpleTransformation.xsl

Listing B-3 shows an example of an XSLT file that is used by an XML transformer or XSLT processor to transform an XML document generated by NSXMLOutputStream into another XML document.

Listing B-3 SimpleTransformation.xsl file

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:woxml="http://www.apple.com/webobjects/XMLSerialization"
    exclude-result-prefixes="woxml"
    version="1.0">

<xsl:output method="xml" encoding="UTF-8" omit-xml-declaration="no" indent = "yes"/>


<!-- ** Constants ** -->
<!-- Indicates how dictionaries are encoded: key-value or two-array. -->
```

```
<xsl:variable name="dictionary_encoding">
    <xsl:value-of select="'key-value'" />
</xsl:variable>

<!-- ** Utilities ** -->
<!-- Gets the base class name from a fully-qualified class name. -->
<xsl:template name="basename">
  <xsl:param name="path"/>
  <xsl:choose>
     <xsl:when test="contains($path, '.')">
        <xsl:call-template name="basename">
           <xsl:with-param name="path" select="substring-after($path, '.')"/>
        </xsl:call-template>
     </xsl:when>
     <xsl:otherwise>
        <xsl:value-of select="$path"/>
     </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- ** Element Processing ** -->
<!-- Processes the tree root. -->
<xsl:template match="/">
    <xsl:element name="content">
        <xsl:apply-templates select="woxml:content" />
    </xsl:element>
</xsl:template>
<!-- Processes the root element (woxml:content). -->
<xsl:template match="woxml:content">
    <xsl:apply-templates select="*"/>
</xsl:template>

<!-- Processes woxml:object elements. -->
<xsl:template name="process_object" match="woxml:object">
    <!-- extract class name -->
    <xsl:variable name="className">
        <xsl:value-of select="woxml:class/@name" />
    </xsl:variable>

    <!-- get base class name -->
    <xsl:variable name="class">
```

```
        <xsl:call-template name="basename">
            <xsl:with-param name="path" select="$className"/>
        </xsl:call-template>
    </xsl:variable>

    <!-- determine the element name -->
    <xsl:variable name="tag">
        <xsl:choose>
            <xsl:when test="$class='NSDictionary' or $class='NSMutableDictionary'">
                <xsl:value-of select="'dictionary'" />
            </xsl:when>
            <xsl:when test="$class='NSArray' or $class='NSMutableArray'">
                <xsl:value-of select="'array'" />
            </xsl:when>
            <xsl:otherwise>
                <xsl:value-of select="$class" />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:variable>

    <!-- create the element -->
    <xsl:element name="{$tag}">
        <xsl:choose>
            <xsl:when test="$class='NSDictionary' or $class='NSMutableDictionary'">
                <xsl:call-template name="process_dictionary" />
            </xsl:when>
            <xsl:otherwise>
                <xsl:call-template name="process_object_content" />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:element>
</xsl:template>

<!-- Processes the content of a woxml:object element. -->
<xsl:template name="process_object_content">
    <xsl:apply-templates select="*" />
</xsl:template>

<!-- Processes woxml:class elements. -->
<xsl:template match="woxml:class" />
```

```
<!-- Processes woxml:array elements. -->
<xsl:template match="woxml:array">
    <xsl:for-each select="woxml:object">
        <xsl:call-template name="process_object" />
    </xsl:for-each>
</xsl:template>

<!-- Processes primitive-type and woxml:string elements. -->
<xsl:template match="woxml:boolean|woxml:byte|woxml:ch|woxml:short|woxml:int|
                     woxml:long|woxml:float|woxml:double|woxml:string">
    <!-- determine the element name -->
    <xsl:variable name="element_name">
        <xsl:choose>
            <xsl:when test="@key">
                <xsl:value-of select="@key" />
            </xsl:when>
            <xsl:when test="@field">
                <xsl:value-of select="@field" />
            </xsl:when>
            <xsl:otherwise>
                <xsl:value-of select="name()" />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:variable>

    <!-- store possible reference to another element -->
    <xsl:variable name="ref">
        <xsl:value-of select="@idRef" />
    </xsl:variable>

    <!-- create the element -->
    <xsl:element name="{$element_name}">
        <xsl:choose>
            <xsl:when test="string(number($ref))='NaN'">
                <!-- $ref is not a number, therefore there's no reference -->
                <xsl:value-of select="." />
            </xsl:when>
            <xsl:otherwise>
                <!-- $ref is a number and, by extension, a reference -->
                <xsl:value-of select="//*[@id=$ref]" />
            </xsl:otherwise>
```

```
        </xsl:choose>
    </xsl:element>
</xsl:template>

<!-- Processes woxml:object elements that contain a NSDictionary or NSMutableDictionary.
-->
<xsl:template name="process_dictionary">
    <xsl:choose>
        <xsl:when test="$dictionary_encoding='key-value'">
            <!-- output the two arrays as key-value pairs within item elements -->
            <xsl:for-each select="woxml:array[1]">
                <xsl:for-each select="*">
                    <xsl:variable name="current_position">
                        <xsl:value-of select="position()" />
                    </xsl:variable>

                    <xsl:element name="item">
                        <xsl:element name="key">
                            <xsl:apply-templates select="." />
                        </xsl:element>

                        <xsl:element name="value">
                            <xsl:apply-templates select="ancestor::*[position()=2]/
child::woxml:array[2]/child::*[position()=$current_position]" />
                        </xsl:element>
                    </xsl:element>
                </xsl:for-each>
            </xsl:for-each>
        </xsl:when>
        <xsl:otherwise>
            <!-- output the two arrays on separate nodes -->
            <xsl:for-each select="woxml:array">
                <xsl:element name="array">
                    <xsl:apply-templates select="*"/>
                </xsl:element>
            </xsl:for-each>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Document Revision History

Table C-1 describes the revisions to *Inside WebObjects: XML Serialization*.

Table C-1 Document revision history

Date	Notes
October 2002	First version of *Inside WebObjects: XML Serialization*.

APPENDIX C

Document Revision History

Glossary

DTD (document type definition) File that describes the structure of an XML document.

JAXP (Java API for XML Processing) Specification that provides API for processing XML documents.

NSXMLInputStream WebObjects class that deserializes untransformed XML documents produced by NSXMLOutputStream into objects.

NSXMLOutputFormat WebObjects class that encapsulates format properties for an NSXMLOutputStream object.

NSXMLOutputStream WebObjects class that serializes objects and data into XML documents.

Project Builder Application used to manage the development of a WebObjects application or framework.

schema File that describes the structure of an XML document. This file can be a DTD file or an XML Schema file.

SGML (Standard Generalized Markup Language) Language that allows the creation of sharable documents with a formal type and element structure.

URI (Uniform Resource Identifier) The Web naming and addressing technology. A URI is a string of characters that identify a resource. Some typical URI schemes are HTTP and FTP.

XML (Extensible Markup Language) Markup language used to represent structured information in a standard way.

XML namespaces Specification that allows qualifying element names by associating element-name prefixes to URIs.

XML parser Software engine that reads and writes XML documents.

XML Schema Specification used to describe the structure of XML documents. XML Schema is more powerful than document type definition (DTD) because it includes facilities to specify the data type of elements and it is based on XML.

XSLT (Extensible Stylesheet Language Transformations Specification that allows the conversion of an XML document into another XML document or any other type of document.

XSLT stylesheet File written in XSLT that specifies how a source document is to be converted into another document.

XSLT transformer Software that converts an XML document into another document using an XSLT stylesheet.