An XML Data-Binding Facility for the Java™ Platform

Mark Reinhold
Core Java Platform Group
Java Software

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 U.S.A.

30 July 1999

Sun Microsystems has recently undertaken to provide basic support for XML in the Java Platform. The proposed facilities include both an event-driven, SAX-compliant parser and an implementation of the W3C DOM (Document Object Model) parse-tree API. This is a critical first step, but using these fairly low-level APIs does require a moderately sophisticated understanding of XML.

In order to make XML more easily accessible to a wider developer audience we are therefore looking at ways to connect XML documents more directly to in-memory objects. Such a connection would allow programs that manipulate XML content to be written at the same conceptual level as the content itself, rather than at the level of parser events or parse trees. It would also obviate the need to use low-level APIs such as SAX and DOM in XML-based data-messaging systems, thereby making such systems much easier to create and maintain.

A particularly promising approach along these lines involves compiling, or *binding*, an XML schema into one or more Java classes. These automatically-generated classes handle the translation between XML documents, which must follow the schema, and interrelated instances of the classes. They also ensure that the constraints expressed in the schema are maintained as instances of the classes are manipulated.

This design note reviews the basic concepts of XML and schemas, motivates and defines XML-based data binding, presents an extended example, and then outlines the requirements of a data-binding facility for the Java Platform.

Please send comments on this note to ⟨*xml-binding-comments@java.sun.com*⟩.

Introduction

XML is, essentially, a platform-independent way to structure information. An XML document is a tree of *elements*. An element may have a set of *attributes*, in the form of key-value pairs, and may contain other elements, text, or a mixture thereof. An element may refer to other elements via special attributes, thereby allowing arbitrary graph structures to be represented.

The structure of an XML document need not follow any rules beyond those laid out in the XML specification. To exchange documents in a meaningful way, however, requires that their structure be described and constrained so that the various parties involved will interpret them correctly and consistently. This can be accomplished through the use of a *schema*. A schema contains a set of rules that constrains the structure and content of a document's components, *i.e.*, its elements, attributes, and text. A schema also describes, at least informally and often implicitly, the intended conceptual meaning of a document's components. A schema is, in other words, a specification of the syntax and semantics of a (potentially infinite) set of XML documents. A document is said to be *valid* with respect to a schema if, and only if, it satisfies the constraints described in the schema.

In what language are schemas written? The XML specification itself describes a sublanguage for writing *document-type definitions*, or DTDs. As schemas go, DTDs are fairly weak. They support the definition of simple constraints on structure and content, but provide no facility for expressing data types or complex structural relationships. These deficiencies have motivated various efforts to define more sophisticated schema languages, and there is a W3C Working Group dedicated to defining a standard schema language by the end of 1999. (These newer schema languages are themselves applications of XML, leading to interesting recursive dependencies that will not be explored in any detail here.)

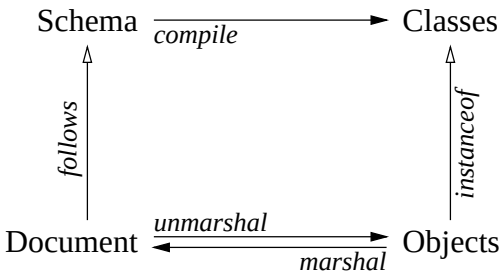
Data binding

Any nontrivial application of XML will, then, be based upon one or more schemas and will involve one or more programs that create, consume, and manipulate documents whose syntax and semantics are governed by those schemas. While it is certainly possible to write such programs using the low-level SAX parser API or the somewhat higher-level DOM parse-tree API, doing so is likely to be tedious and error-prone. The resulting code is also likely to contain many redundancies that will make it difficult to maintain as bugs are fixed and as the schemas evolve.

It would be much easier to write XML-enabled programs if we could simply map the components of an XML document to in-memory objects that represent, in an obvious and useful way, the document's intended meaning according to its schema. Of what classes should these objects be instances? In some cases there will be an obvious mapping from schema components to existing classes, especially for common types such as `String`, `Date`, `Vector`, and so forth. In general, however, classes specific to the schema being

used will be required. Rather than burden developers with having to write these classes we can generate the classes directly from the schema, thereby creating a Java-level *binding* of the schema.

An *XML data-binding facility* therefore contains a *schema compiler*, which translates a schema into a set of schema-specific classes with appropriate access and mutation (*i.e.*, `get` and `set`) methods. It also contains a *marshalling framework*, which supports the *unmarshalling* of XML documents into graphs of interrelated instances of both existing and schema-derived classes and the *marshalling* of such graphs back into XML documents. The unmarshalling process checks incoming XML documents for validity with respect to the schema. Similarly, the compiler generates code into the derived classes to enforce the constraints expressed in the schema, thereby ensuring that only valid documents are generated by the marshalling process.



To sum up: Schemas describe the structure and meaning of an XML document, in much the same way that a class describes an object in a program. To work with an XML document in a program we would like to map its components directly into a set of objects that reflect the document's meaning according to its schema. We can achieve this by compiling the schema into a set of derived classes that handle all the details of marshalling and unmarshalling and also ensure that only valid documents will be produced and consumed. Data binding thus allows XML-enabled programs to be written at the same conceptual level as the documents they manipulate, rather than at the level of parser events or parse trees.

An example

To illustrate the benefits of data binding, consider the hypothetical problem of writing an order-processing system for a shoe warehouse. This system must accept incoming shoe orders in XML, validate them, and arrange for the specified shoes to be shipped to the requesting retail store. A typical order might look something like this:

```
<ShoeOrder id="4040458" style="Sandal">
  <color>Brown</color>
  <size>9 1/2</size>
  <width>AA</width>
</ShoeOrder>
```

A schema for such a shoe order, written in the draft schema language recently published by the W3C XML Schema Working group, would begin with a declaration of the `ShoeOrder` element type, its attributes, and its subelements:

```
<schema name="ShoeOrder">

  <elementType name="ShoeOrder">

    <attrDecl name="id" required="true">
      <datatypeRef name="ID"/>
    </attrDecl>

    <attrDecl name="style" required="true">
      <datatypeRef name="IDREF"/>
    </attrDecl>

    <model>
      <sequence>
        <elementTypeRef name="color"/>
        <elementTypeRef name="size"/>
        <elementTypeRef name="width"/>
      </sequence>
    </model>
```

These declarations specify that a `ShoeOrder` element has two required attributes, `id` and `style`; the former is an XML element identifier, while the latter is an XML identifier reference. The `model` declaration specifies that the content of a `ShoeOrder` element must be a sequence of the named subelements in the order given. Each subelement is specified here by reference to an element type that will be defined below.

The declaration continues with the definition of a derived data type for colors:

```
    <datatype name="Colors">
      <basetype name="string"/>
      <enumeration>
        <literal>Black</literal>
        <literal>Blue</literal>
        <literal>Brown</literal>
        <literal>Tan</literal>
        <literal>White</literal>
      </enumeration>
    </datatype>
```

This definition says that a color is a string whose value is exactly one of `Black`, `Blue`, `Brown`, `Tan`, or `White`. We can then define the `color` element type that was referred to in the model for the `ShoeOrder` element:

```
    <elementType name="color">
      <datatypeRef name="Colors"/>
    </elementType>
```

Shoe sizes require a more interesting data type definition:

```
<datatype name="Size">
  <basetype name="string"/>
  <lexicalRepresentation>
    <lexical>[1-9][0-9]?( 1/2)?</lexical>
  </lexicalRepresentation>
  <minInclusive>3 1/2</minInclusive>
  <maxInclusive>13</maxInclusive>
</datatype>
```

Here we have constrained a shoe size to be a string matching the regular expression given in the `lexicalRepresentation` declaration. That is, a shoe size is a string beginning with a nonzero digit, possibly followed by another digit which may be zero, possibly followed by a space and the string `1/2` for half sizes. Shoe sizes are further constrained to be no less than `3 1/2` and no greater than `13` according to the usual lexicographic order on strings.

Shoe widths are also defined using a regular expression:

```
<datatype name="Width">
  <basetype name="string"/>
  <lexicalRepresentation>
    <lexical>AAA|AA|[A-E]|EE|EEE</lexical>
  </lexicalRepresentation>
</datatype>
```

This declaration constrains shoe widths to be between `A` and `E` inclusive while also allowing the extreme `AAA`, `AA`, `EE`, and `EEE` widths.

To finish up, we need the element-type declarations for `size` and `width`:

```
<elementType name="size">
  <datatypeRef name="Size"/>
</elementType>

<elementType name="width">
  <datatypeRef name="Width"/>
</elementType>
```

Finally, we have the end tags for the overall schema:

```
  </elementType>

</schema>
```

An XML schema compiler might bind the above schema into a Java class with the following signature:

```
public class ShoeOrder {

    public ShoeOrder(String id, Style style,
                     String color, String size, String width);

    public String getId();
    public void setId(String id);

    public Style getStyle()
    public void setStyle(Style style);

    public String getColor();
    public void setColor(String color);

    public String getSize();
    public void setSize(String size);

    public String getWidth();
    public void setWidth(String width);

    public void marshal(OutputStream out)
        throws IOException;
    public static ShoeOrder unmarshal(InputStream in)
        throws IOException;
}
```

We assume that the schema also defines a `Style` element, which would cause a corresponding `Style` class to be generated.

The generated `ShoeOrder` class handles all the details of marshalling and unmarshalling. To accept an order over a network connection, for example, and enter it into the warehouse database we merely need this code:

```
public void acceptOrder(Socket s) throws IOException {
    ShoeOrder so = ShoeOrder.unmarshal(s.getInputStream());
    WarehouseDB.enter(so);
}
```

Similarly, the following method retrieves a shoe order by number and transmits it over the given socket:

```
public void sendOrder(String id, Socket s) throws IOException {
    ShoeOrder so = WarehouseDB.lookup(id);
    so.marshal(s.getOutputStream());
}
```

Finally, the various `set` methods in the generated `ShoeOrder` class perform complete validity checking in order to ensure that marshalled instances will be valid with respect to

the original XML schema. Each of the following statements would cause an appropriate runtime exception to be thrown because they violate the schema:

```
so.setColor("Red");
so.setSize("5 3/4");
so.setWidth("Z");
```

Schema-derived classes need not be used in their original form. In some settings it may be convenient to extend such classes in order to add application-specific data and behavior. The `ShoeOrder` class, for example, could be extended by a `WarehouseShoeOrder` subclass that adds fields and methods that are needed only within the warehouse order-processing system. It could be specialized in a different way by a `StoreShoeOrder` subclass for use in the retail-store order-entry systems. This style of programming may benefit from development tools that can keep track of the various application-specific subclasses and recheck their validity as the `ShoeOrder` schema, and hence the `ShoeOrder` class, evolves.

Requirements

A data-binding facility for the Java Platform will have two major components: A marshalling framework and a schema compiler. Herewith a preliminary sketch of the requirements of each component.

Marshalling framework The marshalling framework will be a platform extension that establishes conventions for annotating classes with the necessary metadata. This metadata, perhaps in the form of the `marshal` and `unmarshal` methods shown above, will define the translation between an external XML document and an internal instance of the annotated class. (This approach is reminiscent of the `encode` and `decode` operations for transmissible types in the Argus programming language.)

The marshalling framework will also contain basic interfaces for the marshalling and unmarshalling operations as well as the necessary low-level support services.

The marshalling framework must be useful for applications other than XML data binding. As part of the Swing work, for instance, we are experimenting with an XML-based mechanism for archiving graphs of JavaBeans™ from within graphical application-builder tools. This mechanism, as well as more general-purpose runtime archiving mechanisms, should be able to use the same marshalling framework as the XML data-binding facility.

Ideally the marshalling framework will not be specific to XML. It seems unwise to tie such a general framework to a specific data format, especially since we may want to support other formats in the future. This implies that the metadata conventions and interfaces must be carefully designed so as to be independent of XML. Because this goal may be very difficult to achieve, it is a desideratum rather than a hard requirement.

Note that the marshalling framework is not in any way intended to displace the object-serialization mechanism which is already a central part of the Java Platform.

Schema compiler The schema compiler will be a command-line tool rather than an extension to the platform itself, though it may also be exposed in a public but non-platform API for direct use by development tools.

Exactly which of the many extant XML schema languages the compiler will support is an open question. The standard currently under development by the W3C's XML Schema Working Group will almost certainly be worth supporting. There are a number of other schema languages, some of which have been deployed, that may be worth supporting if there is demand. These include DCD, DDML, SOX, and XML-Data. Finally, the DTD sublanguage of XML is itself a simple schema language that is already in widespread use and may therefore be worth supporting.

A variety of schema-translation strategies are possible. The simplest translation results in roughly one Java class for each nontrivial schema component. A more sophisticated translation might produce interfaces or abstract classes reflecting the structures and types expressed in schema together with related classes containing the metadata and constraint-checking code. Precisely which strategy or strategies should be used by the compiler is an open question.

Given that there is not (yet) a universal schema language, and with the strong possibility that still more such languages will be invented in the future, the schema compiler should be engineered in a modular fashion that will allow support for new languages to be added as required.

References

W3C XML Schema Working Group	`http://www.w3.org/XML/Group/Schemas.html`
XML Schema Working Draft	
Part 1: Structures	`http://www.w3.org/1999/05/06-xmlschema-1`
Part 2: Datatypes	`http://www.w3.org/1999/05/06-xmlschema-2`
Other schema languages	
DCD	`http://www.w3.org/TR/NOTE-dcd`
DDML	`http://www.w3.org/TR/NOTE-ddml`
SOX	`http://www.marketsite.net/xml/download/sox20.pdf`
XML-Data	`http://www.w3.org/TR/1998/NOTE-XML-data-0105`
Argus	`http://www.pmg.lcs.mit.edu/Argus.html`
Java Object Serialization	`ftp://ftp.java.sun.com/docs/jdk1.2/serial-spec-JDK1.2.pdf`
XML and Java Technology	`http://java.sun.com/xml`