Java™ APIs for WSDL

(JWSDL)

Technical comments to: jsr110-eg-disc@groups.yahoo.com

JSR-110 under Java Community Process

Version 1.1

Editors:
Matthew J. Duftler (duftler@us.ibm.com)
Paul Fremantle (pzf@uk.ibm.com)
John Kaputin (kaputin@uk.ibm.com) – Version 1.1 Maintenance Lead

January 28, 2005

Table of Contents

This is the Specification for the version 1.1 Maintenance Release of the Java™ APIs for WSDL (28 January 2005). It is an update of the version 1.0 Final Release (21 March 2003).

0. What's new in this release?

New features in the version 1.1 Maintenance Release of the Java™ APIs for WSDL [JWSDL]:

1. Code fixes to the WSDL4J Reference Implementation since the release of JWSDL 1.0 have been rolled up into a new version, WSDL4J 1.5. Details are in the WSDL4J change log in CVS.
2. Support for extensibility elements and extensibility attributes in JWSDL is now consistent with the W3C WSDL 1.1 specification.
3. A lightweight schema capability that provides access to XML Schema elements, including those nested via schema imports, includes and redefines.

1. Introduction

The Web Services Description Language [WSDL] is an XML-based language for describing Web services. WSDL allows developers to describe the inputs and outputs to an operation, the set of operations that make up a service, the transport and protocol information needed to access the service, and the endpoints via which the service is accessible.

Java™ APIs for WSDL [JWSDL] is an API for representing WSDL documents in Java. This document, together with the API JavaDocs, is the formal specification for Java Specification Request 110 (JSR-110). JSR-110 is being developed under the Java Community Process (see http://www.jcp.org/jsr/detail/110.jsp).

The expert group that developed this specification was composed of the following individuals:

Name	Company	E-mail
Rahul Bhargava	Netscape Communications	rahul_technical@yahoo.com
Tim Blake	Oracle	Timothy.Blake@oracle.com
Roberto Chinnici	Sun Microsystems, Inc.	roberto.chinnici@sun.com
John P Crupi	Sun Microsystems, Inc.	John.Crupi@Sun.COM
*Matthew J. Duftler	IBM	duftler@us.ibm.com
*Paul Fremantle	IBM	pzf@uk.ibm.com
Pierre Gauthier	Nortel Networks	yaic@nortelnetworks.com
Simon Horrell	Developmentor	simonh@develop.com
Oisin Hurley	IONA Technologies PLC	ohurley@iona.com
Tokuhisa Kadonaga	Fujitsu Limited	kado@sysrap.cs.fujitsu.co.jp
Chris Keller	Silverstream Software	ckeller@silverstream.com
Rajesh Raman	InterKeel	rraman@interkeel.com
Adi Sakala	IONA Technologies PLC	adi.sakala@iona.com
Krishna Sankar	Cisco Systems	ksankar@cisco.com
Miroslav Simek	Systinet	simek@idoox.com

Note: * indicates specification leads.

We borrowed much of the factory mechanism and the set/getFeature mechanism from the JAXP specification, and we would like to acknowledge the JAXP authors for their quality work. We would also like to thank the authors of the WSDL specification for helping us to work through some of the issues that came up. And lastly, thanks to the many folks who adopted this work early, for their feedback and suggestions.

2. Requirements

JWSDL is intended for use by developers of Web services tools and others who need to utilize WSDL documents in Java.

JWSDL is designed to allow users to read, modify, write, create and re-organize WSDL documents in memory. JWSDL is not designed to validate WSDL documents beyond syntactic validity. One use of JWSDL is to develop a tool that validates WSDL semantically.

JWSDL is designed for use in WSDL editors and tools where a partial, incomplete or incorrect WSDL document may require representation.

Although WSDL incorporates XML Schema expressions, JWSDL is not required to parse and represent the contents of schema or schema types. However, JWSDL is required to retrieve all schemas referred to directly or indirectly and present the *org.w3c.dom.Element* that represents each schema so that the JWSDL client application can use a suitable parser to manipulate the schema contents.

WSDL supports extensibility elements and extensibility attributes, which allow the language to be extended. JWSDL must fully support extensibility elements and extensibility attributes.

3. Design Goals

The design goals of this JSR are as follows:

- To specify APIs for reading, writing, creating, and modifying WSDL definitions.
- To specify APIs for reading, writing, creating, and modifying extensibility attributes.
- To specify APIs for reading, writing, creating, and modifying extensibility elements (both those defined in the WSDL specification, and those defined by client applications.)
- To specify interfaces for representing the extensibility elements defined in the WSDL specification.
- To define a mechanism that allows reading, writing, and representing extensibility elements for which no serializers and/or deserializers were defined.
- To specify interfaces for representing XML Schemas as *org.w3c.dom.Element* in a structure that preserves the nesting of schemas within <types>, <import>, <include> and <redefine> tags.
- To define a factory mechanism that allows JWSDL client code to be written independent of any particular JWSDL implementation.
- To specify APIs that are suitable for the building of WSDL tools and runtime infrastructure.
- To define an API that supports WSDL-equivalence of read and written documents. That is, if a document is read into memory, and then written back out, the two documents should be semantically equivalent. XML Processing Instructions and XML Comments may be lost in this process.
- Specify the conformance criteria for JWSDL implementations.

This version of JWSDL supports WSDL v1.1, based on the submission to the W3C dated 15th March 2001 (http://www.w3.org/TR/2001/NOTE-wsdl-20010315). It is expected that changes in the WSDL specification made by the W3C will be reflected in future versions of the JWSDL specification through the workings of the Java Community Process.

4. Out of Scope

- JWSDL does not provide support for querying/manipulating the contents of XML Schema, other than providing access to the *org.w3c.dom.Element* that represents the schema.
- JWSDL does not provide for validating WSDL documents beyond syntactic validity (see Section 5). One likely use of JWSDL is to develop a tool that validates WSDL semantically.

5. Syntactic Validity

All the details of WSDL syntax are not explicitly defined in the current proposed WSDL specification. This API specification expects the following behaviour from implementations.

Ordering

Implementations must support parsing WSDL that is in the correct order as specified by the WSDL specification and the schema. Implementations *may* support reading incorrectly ordered definitions without errors or exceptions. Implementations must write WSDL documents in the order specified by the WSDL specification.

Extensibility Elements

The WSDL specification only allows extensibility elements under certain elements. Any implementation of JWSDL must enforce that, and illegal extensibility elements will cause an exception. JWSDL also defines the type of each extensibility element through the registration process, and so extensibility elements should only be recognized within the scope in which they are defined to the JWSDL implementation. If an extensibility element that is registered in one place (e.g. Port) is found in another where it is not registered (e.g. Binding), then it should be considered an unknown extensibility element and treated as such.

Extensibility Attributes

The WSDL specification allows only certain elements to contain extensibility attributes. Any JWSDL implementation must enforce this and the illegal use of extensibility attributes must cause an exception. JWSDL allows a ‘type’ to be registered for each extensibility attribute so that the attribute value can be parsed into a suitable object representation (details of these types are in section 12 ‘Extensibility Attributes’). If an extensibility attribute type has not been registered, parsing it will not raise a syntax exception – it will simply default to the QName type.

Referential Integrity

Properly formed WSDL documents should be complete - if there is a reference to an element, then that element should exist. However, during tooling and creation, it may be necessary to manage incomplete WSDL documents. Therefore, implementations should not enforce referential integrity.

6. Factory Mechanism

One of the goals of this JSR is to allow applications to write JWSDL client code, without requiring specific knowledge of the particular implementation being used (with the obvious exception of implementation-provided extensions).

An application first obtains a *WSDLFactory* instance via the static *newInstance* method of *WSDLFactory*. The *newInstance* method uses the following ordered lookup procedure to determine the *WSDLFactory* implementation class to load:

- Check the javax.wsdl.factory.WSDLFactory system property.

- Check the lib/wsdl.properties file in the JRE directory. The key will have the same name as the above system property.
- Use the platform default value (will vary with implementations).

Note: There is also a static *newInstance* method that takes the fully-qualified class name of a factory implementation as an argument, in which case the above procedure is not employed.

Once a *WSDLFactory* instance is obtained, the methods *newDefinition*, *newWSDLReader*, *newWSDLWriter*, or *newPopulatedExtensionRegistry* can be invoked to create the desired objects.

The next several sections contain examples of using these methods to read, write, and programmatically create WSDL definitions.

7. Reading Definitions

An application invokes the *newWSDLReader* method on a *WSDLFactory* to obtain a *WSDLReader*. Once a *WSDLReader* is obtained, one of the various *readWSDL* methods can be used to construct a *Definition* object from a WSDL document. It is recommended that *WSDLReader* implementations employ JAXP in the parsing of WSDL documents, so any JAXP-compliant XML parser can be used.

After obtaining a *WSDLReader* instance, and before invoking *readWSDL*, any desired features should be enabled or disabled by invoking the *setFeature* method. All feature names must be fully-qualified, Java package style. All names starting with javax.wsdl. are reserved for features defined by the JWSDL specification. It is recommended that implementation-specific features be fully-qualified to match the package name of that implementation. For example: com.abc.featureName.

The minimum features that must be supported by any implementation are:

Name	Description	*Default Value*
javax.wsdl.verbose	If set to true, status messages will be displayed.	true
javax.wsdl.importDocuments	If set to true, imported WSDL documents will be retrieved and processed.	true

If the javax.wsdl.verbose feature is enabled, status messages will be sent to the standard output stream (i.e. *System.out*). It is enabled by default.

If the javax.wsdl.importDocuments feature is enabled, imported documents will be retrieved and processed. It is enabled by default. When imported documents are retrieved and processed, the imported items can be returned by queries on the importing *Definition*. That is, when querying a *Definition*, or some item contained in a *Definition*, the returned item may be from a different *Definition*, if other *Definitions* have been imported. Imported *Definitions* may be navigated to by invoking the *getImports* method on the importing *Definition*, and then querying the definition property of the returned *javax.wsdl.Import* objects (will always be *null* if the javax.wsdl.importDocuments feature was disabled). Within any particular *Definition* graph, individual items must only exist once. For example, if multiple *Input* and *Output* objects refer to the same *Message*, all those references must refer to the same *Message* instance.

JWSDL's import logic allows WSDL and XML Schema documents to be imported via the WSDL <import> element. JWSDL is capable of retrieving and processing WSDL documents, but while it can retrieve XML Schema documents, it will not fully process them – instead it just stores them as *org.w3c.dom.Element*.

The following is an example of how to use a *WSDLReader* to construct a *Definition* that represents the WSDL file named sample.wsdl:

```
import javax.wsdl.*;
import javax.wsdl.factory.*;
import javax.wsdl.xml.*;
...
    try
    {
      WSDLFactory factory = WSDLFactory.newInstance();
      WSDLReader reader = factory.newWSDLReader();

      reader.setFeature("javax.wsdl.verbose", true);
      reader.setFeature("javax.wsdl.importDocuments", true);

      Definition def = reader.readWSDL(null, "sample.wsdl");
    }
    catch (WSDLException e)
    {
      e.printStackTrace();
    }
```

The first argument to *readWSDL* is an optional context URI, which can be used to resolve the second argument (also a URI), if the second argument is relative.

8. Navigating Definitions

Let's assume that the sample.wsdl file referred to in the previous section contains the following:

```
<?xml version="1.0"?>

<definitions name="StockQuoteService"
             targetNamespace="urn:xmltoday-delayed-quotes"
             xmlns:tns="urn:xmltoday-delayed-quotes"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

  <message name="getQuoteInput">
    <part name="symbol" type="xsd:string"/>
  </message>

  <message name="getQuoteOutput">
    <part name="quote" type="xsd:float"/>
  </message>

  <portType name="GetQuote">
    <operation name="getQuote">
      <input message="tns:getQuoteInput"/>
      <output message="tns:getQuoteOutput"/>
    </operation>
  </portType>

  <binding name="GetQuoteSoapBinding" type="tns:GetQuote">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getQuote">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="urn:xmltoday-delayed-quotes"/>
```

```
      </input>
      <output>
        <soap:body use="encoded"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="urn:xmltoday-delayed-quotes"/>
      </output>
    </operation>
  </binding>

  <service name="StockQuoteService">
    <port name="StockQuotePort" binding="tns:GetQuoteSoapBinding">
      <soap:address location="http://www.fremantle.org/soap/servlet/rpcrouter"/>
    </port>
  </service>

</definitions>
```

The following is an example of navigating the *Definition* to determine what operations are defined for a particular service:

```
Definition def = reader.readWSDL(null, "sample.wsdl");
String tns = "urn:xmltoday-delayed-quotes";
Service service = def.getService(new QName(tns, "StockQuoteService"));
Port port = service.getPort("StockQuotePort");
Binding binding = port.getBinding();
PortType portType = binding.getPortType();
List operations = portType.getOperations();
Iterator opIterator = operations.iterator();

while (opIterator.hasNext())
{
  Operation operation = (Operation)opIterator.next();

  if (!operation.isUndefined())
  {
    System.out.println(operation.getName());
  }
}
```

Just “getQuote” should be displayed.

The following is an example of navigating the *Definition* to determine what messages are defined in a WSDL definition:

```
Definition def = reader.readWSDL(null, "sample.wsdl");
Map messages = def.getMessages();
Iterator msgIterator = messages.values().iterator();

while (msgIterator.hasNext())
{
  Message msg = (Message)msgIterator.next();

  if (!msg.isUndefined())
  {
    System.out.println(msg.getQName());
  }
}
```

Both the getQuoteInput and getQuoteOutput messages should be listed, within the urn:xmltoday-delayed-quotes namespace.

The “undefined” property defined on the operation and message objects indicates whether the definition for the particular item was found or not. For example: If, within a WSDL document, an `<wsdl:input>` element refers to a message whose definition cannot be found, a placeholder message

object will be created, and its undefined property will be set to true. A similar property also exists on *PortType* and *Binding*. *WSDLWriters* are required to examine this property when determining which items to write out. The default value for the undefined property of *Message*, *Operation*, *PortType*, and *Binding* is true; when creating these items programmatically, the property must be set to false.

9. Writing Definitions

An application invokes the *newWSDLWriter* method on a *WSDLFactory* to obtain a *WSDLWriter*. Once a *WSDLWriter* is obtained, one of the *writeWSDL* methods can be employed to write a Definition out as a WSDL document to either a *java.io.Writer*, or a *java.io.OutputStream*. All *WSDLWriter* implementations must examine the undefined property of *Message*, *Operation*, *PortType*, and *Binding* objects to determine which items should be written out. See the previous section for more information on the undefined property.

WSDLWriters are not required to be capable of writing out *Definitions* created by other JWSDL implementations (although some may have this capability.)

After obtaining a *WSDLWriter* instance, and before invoking *writeWSDL*, any desired features should be enabled or disabled by invoking the *setFeature* method. There are no minimum features that must be supported by implementations.

The following is an example of how to use a *WSDLWriter* to write a *Definition* to *System.out*:

```
WSDLFactory factory = WSDLFactory.newInstance();
WSDLWriter writer = factory.newWSDLWriter();

writer.writeWSDL(def, System.out);
```

If the definition was constructed from the sample.wsdl file, the output should look basically the same as the contents of that file. The formatting of the file may be different, but the elements and attributes will be the same (although they may not appear in the same order).

There is also a *getDocument* method defined on *WSDLWriter*. This method can be used to generate an *org.w3c.dom.Document* from the specified *Definition*.

10. Programmatically Creating Definitions

An application invokes the *newDefinition* method on a *WSDLFactory* to obtain a new instance of a *javax.wsdl.Definition.* Once that definition is obtained, it serves as a factory that can be used to create the rest of the items that will make up the full definition. This specification does not mandate that items be created by the *Definition* they will eventually be added to. Nor does this specification mandate that any item have precisely one parent *Definition* (that is, implementations may allow items to be added to more than one *Definition*.) A particular implementation may choose to require items to be created by the *Definition* they will be added to, and/or to require that an item be added to only one *Definition.* If either of these restrictions is imposed by an implementation, it should be clearly spelled out in that implementation's documentation.

The following is an example that programmatically constructs a definition containing two messages and a portType with one operation that uses those two messages:

```
WSDLFactory factory = WSDLFactory.newInstance();
Definition def = factory.newDefinition();
String tns = "urn:xmltoday-delayed-quotes";
String xsd = "http://www.w3.org/2001/XMLSchema";
Part part1 = def.createPart();
```

```
Part part2 = def.createPart();
Message msg1 = def.createMessage();
Message msg2 = def.createMessage();
Input input = def.createInput();
Output output = def.createOutput();
Operation operation = def.createOperation();
PortType portType = def.createPortType();

def.setQName(new QName(tns, "StockQuoteService"));
def.setTargetNamespace(tns);
def.addNamespace("tns", tns);
def.addNamespace("xsd", xsd);

part1.setName("symbol");
part1.setTypeName(new QName(xsd, "string"));
msg1.setQName(new QName(tns, "getQuoteInput"));
msg1.addPart(part1);
msg1.setUndefined(false);
def.addMessage(msg1);

part2.setName("quote");
part2.setTypeName(new QName(xsd, "float"));
msg2.setQName(new QName(tns, "getQuoteOutput"));
msg2.addPart(part2);
msg2.setUndefined(false);
def.addMessage(msg2);

input.setMessage(msg1);
output.setMessage(msg2);
operation.setName("getQuote");
operation.setInput(input);
operation.setOutput(output);
operation.setUndefined(false);
portType.setQName(new QName(tns, "GetQuote"));
portType.addOperation(operation);
portType.setUndefined(false);
def.addPortType(portType);
```

The items created in the above example should match those read from the sample.wsdl file in the earlier examples.

11. Extension Architecture

The extension architecture is designed to allow an application to perform the same basic functions with extensibility elements, as with native WSDL elements. That is, applications are able to read extensions into memory, write extensions back out, query in-memory extensions, and programmatically create extensions. This is made possible by a combination of `javax.wsdl.extensions.*` interfaces and classes:

- The *ElementExtensible* interface is used to represent WSDL elements that may contain extensibility elements.
- The *ExtensibilityElement* interface is used to represent extensions in memory.
- The *ExtensionDeserializer* interface is used to read extensions into memory.
- The *ExtensionSerializer* interface is used to write extensions out.
- The *ExtensionRegistry* class is used to hold the configuration information necessary to determine which serializers and deserializers are to be used for handling which extensions.

Note: The terms “extensibility element” and “extension” are used interchangeably throughout this document.

The WSDL specification describes which WSDL elements may contain extensibility elements (see WSDL 1.1 Schema at http://schemas.xmlsoap.org/wsdl/). All JWSDL implementations are required to support these extensions for the WSDL elements that can contain them and to prohibit the use of extensions in elements that cannot. For every WSDL element capable of containing extensibility elements, its corresponding *javax.wsdl.** interface extends the *ElementExtensible* interface. This interface has two methods to handle the extensions: *addExtensibilityElement* and *getExtensibilityElements*. The *addExtensibilityElement* method takes an instance of an *ExtensibilityElement*, and the *getExtensibilityElements* method returns a *List* whose items are of type *ExtensibilityElement*.

All JWSDL implementations are required to support the WSDL specification-defined extensions. That is, all JWSDL implementations are required to support the "SOAP", "HTTP", and "MIME" extensions. Implementations of the *ExtensibilityElement* interface are provided for each of the WSDL specification-defined extensions in *javax.wsdl.extensions.soap.**, *javax.wsdl.extensions.http.**, and *javax.wsdl.extensions.mime.**. In order to provide support for the specification-defined extensions, implementations are required to implement the *newPopulatedExtensionRegistry* method of *WSDLFactory*. This method must return an instance of an *ExtensionRegistry* with serializers/deserializers registered, and Java types mapped, for all the WSDL specification-defined extensions. The particular serializers and deserializers that each implementation will use to handle these specification-defined extensions are not mandated by this document.

An *ExtensionRegistry* can be set/retrieved on/from a *Definition*, and set/retrieved on/from a *WSDLReader*. To add support for additional extensions, an application must configure the *ExtensionRegistry*. If the *ExtensionRegistry* is being configured for the purpose of reading a document that contains extensibility elements, the configured *ExtensionRegistry* should be set on the *WSDLReader* prior to reading the document. If an *ExtensionRegistry* is set on the *WSDLReader*, the *Definition* constructed by that *WSDLReader* will have that *ExtensionRegistry* set as the value of its extensionRegistry property. In other words, whatever value is assigned to the extensionRegistry property of a *WSDLReader* will be assigned as the value of the extensionRegistry property of all *Definitions* constructed by that *WSDLReader*.

If the *ExtensionRegistry* is being configured for the purpose of writing out a programmatically constructed definition that contains extensions, the configured *ExtensionRegistry* must be set as the value of the extensionRegistry property of the *Definition* prior to handing it off to a *WSDLWriter*.

There are three different types of configuration that can be done on an *ExtensionRegistry*:

- Registering a deserializer for a particular extension.
- Registering a serializer for a particular extension.
- Mapping an implementation class to a particular extension.

In most cases, all three types of configuration will be done for every extension. Every JWSDL implementation is required to do this for all the WSDL specification-defined extensions, when the *newPopulatedExtensionRegistry* method is invoked. If an *ExtensionRegistry* is retrieved by simply invoking *ExtensionRegistry's* zero-argument constructor (i.e. *new ExtensionRegistry*()), it will not have serializers, deserializers, or Java implementation classes mapped for any extensions.

The following examples are concerning a fictitious extensibility element named `<abc:myExt>`, where the prefix "abc" is associated with the namespace URI "urn:def". The Java class created to represent this extension in memory is called *ghi.Abc*, and it implements the *ExtensibilityElement* interface (as it is required to do, in order to be considered an extension). The `<abc:myExt>` extensibility element may only exist as an immediate child of a `<wsdl:service>` element. There is a *ghi.AbcDeserializer* class which implements the *ExtensionDeserializer* interface, and is capable of reading an `<abc:myExt>` element, and populating a new instance of a *ghi.Abc* with the relevant information. There is also a

ghi.AbcSerializer class which implements the *ExtensionSerializer* interface, and is capable of querying an instance of a *ghi.Abc* and serializing the relevant information in the form of a `<abc:myExt>` extensibility element.

```
// Create a new ExtensionRegistry.
ExtensionRegistry extReg = new ExtensionRegistry();
// Register the deserializer.
extReg.registerDeserializer(Service.class,
                            new QName("urn:def", "myExt"),
                            new ghi.AbcDeserializer());
// Register the serializer.
extReg.registerSerializer(Service.class,
                          new QName("urn:def", "myExt"),
                          new ghi.AbcSerializer());
// Map the implementation class to the extension type.
extReg.mapExtensionTypes(Service.class,
                         new QName("urn:def", "myExt"),,
                         ghi.Abc.class);
```

Note that in all three of the above *ExtensionRegistry* method invocations, the Service.class argument indicates that the extension can exist as a child of a `<wsdl:service>` element.

If a WSDL document containing a `<wsdl:service>` element was read in, and the `<wsdl:service>` element contained an `<abc:myExt>` element as an immediate child, the list returned from an invocation of the *getExtensibilityElements* method on that *Service* object would contain one item: a instance of a *ghi.Abc*.

The following is an example of retrieving this *ghi.Abc* object:

```
Definition def = ...
Service svc = def.getService(...);
List extElements = svc.getExtensibilityElements();

ghi.Abc = (ghi.Abc)extElements.get(0);
```

The following is an example of programmatically creating an instance of a *ghi.Abc*, and adding it to a *Service* object:

```
Service svc = def.createService();
ghi.Abc anExt = (ghi.Abc)extReg.createExtension(Service.class,
                                new QName("urn:def", "myExt"));

// Now configure the Abc instance…
// Then add it to the Service object.
svc.addExtensibilityElement(anExt);
```

The *createExtension* method is used to programmatically create extensions so that applications can create extensions without knowing the implementing class. This is particularly relevant when dealing with extensions that have well-known interfaces to represent them, such as the WSDL specification-defined extensions.

The following is an example of programmatically creating an instance of a class which implements the *SOAPBinding* interface, without knowing the implementing class:

```
SOAPBinding soapBinding =
  (SOAPBinding)extReg.createExtension(Binding.class,
    new QName("http://schemas.xmlsoap.org/wsdl/soap/",
              "binding"));
```

Since all JWSDL implementations are required to support the WSDL specification-defined extensions, the above *SOAPBinding* example must work, exactly as shown, with any implementation.

There is one additional item, with respect to extensibility elements, which must be considered: How are unexpected extensibility elements handled when they are encountered?

An "unexpected extensibility element" is an extensibility element for which there are no serializers/deserializers registered. To handle this case, the *ExtensionRegistry* has two properties: defaultSerializer, and defaultDeserializer.

The value of the defaultDeserializer property is an *ExtensionDeserializer* that is to be used to deserialize unexpected extensibility elements. Its default value is an instance of an *UnknownExtensionDeserializer*. The *UnknownExtensionDeserializer* simply wraps the *org.w3c.dom.Element* representing the extensibility element in a new instance of an *UnknownExtensibilityElement*. If the defaultDeserializer property of an *ExtensionRegistry* is set to `null`, an exception will be thrown when an unexpected extensibility element is encountered.

The value of the defaultSerializer property is an *ExtensionSerializer* that is to be used to serialize unexpected extensions that are encountered while writing out definitions. Its default value is an instance of an *UnknownExtensionSerializer*. The *UnknownExtensionSerializer* simply serializes the *org.w3c.dom.Element* that is wrapped in an instance of an *UnknownExtensibilityElement*. If the defaultSerializer property of an *ExtensionRegistry* is set to *null*, an exception will be thrown when an unexpected extension is encountered.

12. Extensibility Attributes

The extension architecture described previously for extensibility elements is also designed to allow an application to perform the same basic functions with extensibility attributes of WSDL elements, as with the native WSDL attributes of those elements. That is, applications are able to read extensibility attributes into memory, write them back out, query them in-memory, and programmatically create them. This is made possible by a combination of `javax.wsdl.extensions.*` interfaces and classes:

- The *AttributeExtensible* interface is used to represent WSDL elements that may contain extensibility attributes.
- The *ExtensionRegistry* class is used to hold the configuration information necessary to determine the type of the extension attributes for each parent WSDL element.

Note: 'extensibility attribute' and 'extension attribute' are used interchangeably.

The WSDL specification describes which WSDL elements may contain extensibility attributes (see WSDL 1.1 Schema at http://schemas.xmlsoap.org/wsdl/). All JWSDL implementations are required to support these extensibility attributes in the WSDL elements that can contain them and to prohibit the use of extensibility attributes in elements that cannot. For every WSDL element capable of containing extensibility attributes, its corresponding *javax.wsdl.** interface extends the *AttributeExtensible* interface.

The *AttributeExtensible* interface has four methods:

- *setExtensionAttribute(QName, Object)* - stores an extensibility attribute of the WSDL element. The QName represents the attribute name and acts as a key to the attribute value represented by the Object.
- *getExtensionAttribute(QName)* – retrieves an extensibility attribute of the WSDL element. It returns an Object containing the attribute value using the QName argument as the key.

- *getExtensionAttributes* – returns all of the extensibility attributes for the WSDL element. It returns a Map of attribute value Objects keyed by attribute QName.
- *getNativeAttributeNames* – returns all of the WSDL-defined attributes for the WSDL element (that is, its standard WSDL attributes not its extensibility attributes).

The *AttributeExtensible* interface also defines several 'types' of extensibility attribute values:

- *STRING_TYPE*
- *QNAME_TYPE*
- *LIST_OF_STRINGS_TYPE*
- *LIST_OF_QNAMES_TYPE*
- *NO_DECLARED_TYPE*.

For JWSDL to be able to parse extensibility attributes into suitable in-memory representations, a JWSDL client application must first configure the *ExtensionRegistry* with one of these *AttributeExtensible* types for each extensibility attribute. JWSDL can then determine the correct in-memory representation (String, QName or List) to store the extensibility attribute's value. If an attribute's type is not registered in the ExtensionRegistry a default representation of QName will be used to store the attribute value and the JWSDL client application may need to further parse the QName into a more useful format. Registration of extensible attribute types places the responsibility for this further parsing with JWSDL, rather than with the JWSDL client. If an extensibility attribute is registered as *NO_DECLARED_TYPE* its object representation will also default to QName.

The *ExtensionRegistry* class provides two configuration methods for extensibility attributes:

- *registerExtensionAttributeType(Class parentType, QName attrName, int attrType)* – is used to associate an attribute's QName with its *AttributeExtensible* type for a given parent WSDL element.
- *queryExtensionAttributeType(Class parentType, QName attrName)* – returns the *AttributeExtensible* type for the attribute identified by QName within the parent WSDL element.

To further understand these features, consider the following WSDL fragment containing the WSDL `<part>` element named `symbol`. The WSDL specification declares that the WSDL <part> element can contain extensibility attributes. `name` and `type` are native WSDL attributes of the `<part>` element and `xyz:extattr` is an extensibility attribute of `<part>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="StockQuoteService"
    targetNamespace="http://wsdl/StockQuoteService/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://wsdl/StockQuoteService/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:foo="http://foo.bar">
    xmlns:xyz="http://xyz.namespace">
    <message name="getQuoteRequest">
        <part name="symbol" type="xsd:string" xyz:extattr="quick brown fox"/>
        <myMsgExtElement/>
    </message>
    ....
    ....
</definitions>
```

The JWSDL client application should first configure the *ExtensionRegistry* with an *AttributeExtensible* type for `extattr`. For example, to register a string type:

```
//register the extensibility attribute type
extReg.registerExtensionAttributeType(
       Part.class,
       new QName("http://xyz.namespace", "extattr"),
       AttributeExtensible.STRING_TYPE);
```

JWSDL implementations of *javax.wsdl.xml.WSDLReader* should query the *ExtensionRegistry* to determine the type of objects required when parsing extensibility attributes:

```
int attrType = extReg.queryExtensionAttributeType(
                      Part.class,
                      new QName("http://xyz.namespace", "extattr"));

if (attrType == AttributeExtensible.QNAME_TYPE)
{
  //instantiate extattr as a QName
}
else if (attrType == AttributeExtensible.LIST_OF_STRINGS_TYPE)
{
  //instantiate extattr as a List of String
}
else if (attrType == AttributeExtensible.LIST_OF_QNAMES_TYPE)
{
  //instantiate extattr as a List of QName
}
else if (attrType == AttributeExtensible.STRING_TYPE)
{
  //instantiate extattr as a String
}
else
{
  //instantiate a QName by default
```

JWSDL will represent the value “quick brown fox” in memory in different ways depending on the type registered in the *ExtensionRegistry*.

If the WSDL is parsed without an *AttributeExtensible* type registered in the *ExtensionRegistry* for `extattr`, or if it has been registered as *QNAME_TYPE* or *NO_DECLARED_TYPE*, `extattr` will be represented as a QName:

```
{http://xyz.namespace}extattr={http://schemas.xmlsoap.org/wsdl/}quick brown fox
```

If `extattr` has been registered as *STRING_TYPE*, it will be represented as a String:

```
{http://xyz.namespace}extattr= “quick brown fox”
```

If `extattr` has been registered as *LIST_OF_STRINGS_TYPE*, it will be represented as a list of Strings:

```
{http://xyz.namespace}extattr=[“quick”, “brown”, “fox”]
```

If `extattr` has been registered as *LIST_OF_QNAMES_TYPE*, it will be represented as a list of QNames:

```
{http://xyz.namespace}extattr=
[{http://schemas.xmlsoap.org/wsdl/}quick,
{http://schemas.xmlsoap.org/wsdl/}brown,
{http://schemas.xmlsoap.org/wsdl/}fox]
```

Note that in these examples the WSDL document’s default namespace, `{http://schemas.xmlsoap.org/wsdl/}`, is used for the QName representations of `extattr`‘s value

because the attribute value was not qualified with a namespace in the WSDL. If we had written the WSDL as say,

```
<part name="symbol" type="xsd:string" xyz:extattr="foo:quick brown fox"/>
```

the attribute value QNames would have been qualified with the corresponding namespace:

```
{http://xyz.namespace}extattr={http://foo.bar/}quick brown fox
```

The *javax.wsdl.Part* interface is declared as:

```
public interface Part extends java.io.Serializable, AttributeExtensible
```

WSDLReader implementations and JWSDL client applications should use the *AttributeExtensible* methods on *Part* to manipulate its extensibility attributes. The following code examples assume `partElement` is an instance of *Part*, `qname` is the qualified name of the extensibility attribute (eg: `{http://xyz.namespace}extattr`) and `parsedValue` is a String, QName or List that represents "quick brown fox":

```
//Get the native WSDL-defined attributes of <part> (ie: 'name' and 'type')
List nativeAttributeNames = partElement.getNativeAttributeNames();

//After parsing 'extattr' into a String, QName or List, store it in the Part
partElement.setExtensionAttribute(qname, parsedValue);

//Retrieve 'extattr' from the Part, assuming parsedValue was a String
String extattrValue = (String) partElement.getExtensionAttribute(qname);

//Get all extensibility attributes for the Part
Map extAttributes = partElement.getExtensionAttributes();
```

The *ExtensionRegistry* is not used to serialize extensibility attributes. Strings, QNames and Lists of Strings or QNames can be easily serialized programmatically, so JWSDL implementations of *javax.wsdl.xml.WSDLWriter* just need to examine the type of object that represents the extensibility attribute in-memory and convert it to a string accordingly. The following code sample assumes `attrValue` is an Object reference to the extensibility attribute value:

```
if (attrValue instanceof String)
{
  //serialize it as is
}
else if (attrValue instanceof QName)
{
  //serialize the QName value to a string
}
else if (attrValue instanceof List)
{
  //If it contains Strings, concatenate them with spaces as appropriate
  //If it contains QName, convert their values to strings then concatenate
{
```

JWSDL client applications can create elements with extensibility attributes programmatically simply by instantiating the attributes as Strings, QNames or Lists of Strings or QNames, setting them on the parent element, then letting the WSDLWriter.writeWSDL method handle the serialization into WSDL. If the extensibility attributes are just being created programmatically (that is, they are not being parsed from existing WSDL) there is no need to register their types in the *ExtensionRegistry*.

13. XML Schema Support

JWSDL has a 'lightweight' schema architecture designed to access schema elements, including schemas that have been defined in-line in the WSDL <types> element and those that have been nested using WSDL <import> tags or Schema <import>, <include> or <redefine> tags. As stated previously in the Requirements and Design Goals sections, JWSDL is not a schema parser and this mechanism does not provide full access to or manipulation of the schema contents. It is simply a schema navigation mechanism that allows a JWSDL client application to access the *org.w3c.dom.Element* that represents each schema in the nested schema tree. The client application must then use a suitable parser to access the contents of the schema *Element*.

JWSDL defines three interfaces in *javax.wsdl.extensions.schema.**:

- *Schema* - represents a schema element. It provides access to the *org.w3c.dom.Element* representation of the schema and references to any direct child schemas nested via Schema <import>, <include> or <redefine> tags.
- *SchemaReference* - represents the 'link' from the parent schema to an immediate child schema. It provides a reference to the child schema, which in turn is represented as a *Schema*. The schema <include> and <redefine> elements have an id and a schemaLocation attribute and both elements are represented as a *SchemaReference.*
- *SchemaImport* – is an extension of *SchemaReference* that adds a namespace property and is used to represent schema <import> elements, which include a namespace attribute as well as id and schemaLocation.

JWSDL implementations must parse all schemas nested within or below a WSDL document into a 'linked chain' of *Schemas* and *SchemaReferences* or *SchemaImports* that mirrors the nested schema tree in the WSDL and schema documents. Although this not a full schema parsing capability, JWSDL implementations must still access the <import>, <include> and <redefine> tags within the schema *Element* to build up this in-memory representation of the nested schema tree.

Schema has methods to support five properties; documentBaseURI, the *org.w3c.dom.Element* and schema imports, includes and redefines:

- *getDocumentBaseURI* and *setDocumentBaseURI* handle the full location URI of the schema
- *getElement* and *setElement* handle the *org.w3c.dom.Element* that represents the schema
- *createImport* returns a *SchemaImport*
- *createInclude* and *createRedefine* both return a *SchemaReference*
- *addImport*, *addInclude* and *addRedefine* are used to store *SchemaImport* or *SchemaReference* objects
- *getImports* returns a Map of *SchemaImport* objects keyed by the namespace attribute of the <import> tag
- *getIncludes* and *getRedefines* both return a List of *SchemaReference* objects

SchemaReference has methods to support three properties; id, schemaLocation and referencedSchema:

- *getId* and *setId* handle the id attribute used on the import, include or redefine
- *getSchemaLocationURI* and *setSchemaLocationURI* handle the schema URI specified in the schemaLocation attribute on the import, include or redefine
- *getReferencedSchema* and *setReferencedSchema* handle the schema referred to by the import, include or redefine

SchemaImport adds methods to support the namespace property:

- *getNamespaceURI* and *setNamespaceURI* handle the namespace attribute used on the import.

The schema support in JWSDL is defined using the same extension architecture described previously for the standard WSDL extensions; Soap, HTTP and MIME. So, *Schema* is an *ExtensibilityElement*:

```
Schema extends ExtensibilityElement, Serializable
```

and it is stored, retrieved and manipulated in JWSDL in the same way as SOAP, HTTP and MIME extensions. In addition to registering these standard extensions, the *ExtensionRegistry* returned by the *newPopulatedExtensionRegistry* method of *WSDLFactory* must also have a schema serializer /deserializer registered and a Java type mapped for the *Schema* implementation. As with the standard extensions, the particular serializer and deserializer are not mandated by this document.

Schema can only be referred to in WSDL using the <types> and <import> elements. Within <types>, <schema> is used as an extensibility element, so *javax.wsdl.Types* extends *ElementExtensible* which provides the methods for storing and retrieving the *Schema* objects as *ExtensibilityElement*s (see section 11. 'Extension Architecture' for more details). Within a WSDL <import> the schema is referred to by the location attribute, not as an extensibility element. JWSDL handles this using the same in-memory representation used for importing WSDL documents; that is, *javax.wsdl.Import* refers a *Definition* which in the case if a WSDL-imported schema will contain a *Types* object with a *Schema* object in its list of extensibility elements.

JWSDL implementations are required to provide the XML Schema parsing described in this section, however they may also add their own schema parsing behaviour to provide full XML Schema parsing capability or to support other types of schema. However, this would be implementation specific and is not part of the JWSDL specification. To do this, implementations could either extend the *Schema* interface or provide a new interface by extending *ExtensibilityElement* directly.

As an example of these features, consider a WSDL file called TravelCo.wsdl with an in-line schema that imports Flight.xsd, includes Hotel.xsd and redefines Address.xsd. The schema defined in Flight.xsd includes Address.xsd and the schema defined in Hotel.xsd imports Address.xsd. This WSDL document also has a WSDL import of Address.xsd.

The WSDL fragment for TravelCo.wsdl is:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="TravelCo"
    targetNamespace="http://travelco.com"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://travelco.com"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://travelco.com/schema/Flight">

    <!-- 'types' contains an inline Schema with nested Schemas -->
    <types>
      <xsd:schema>
        <xsd:element name="Custname" type="xsd:string"/>
        <xsd:element name="Response" type="xsd:string"/>
        <xsd:import schemaLocation="Flight.xsd"
                    namespace="http://travelco.com/schema/Flight"/>
        <xsd:include schemaLocation="Hotel.xsd"/>
        <xsd:redefine schemaLocation="Address.xsd">
          <xsd:complexType name="address_struct">
            <xsd:complexContent>
              <xsd:extension base="address_struct">
                <xsd:sequence>
                  <xsd:element name="Country" type="string"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
```

```
        </xsd:redefine>
      </xsd:schema>
    </types>
    ...
    <!-- A WSDL import of a Schema, as opposed to a Schema import -->
    <import location="Address.xsd" namespace="http://travelco.com/schema/Address"/>
    ...
    ...
</definitions>
```

The Flight.xsd fragment is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://travelco.com/schema/Flight"
    targetNamespace="http://travelco.com/schema/Flight">
       <xs:include schemaLocation="Address.xsd"/>
       <xs:element name="FlightResRQ">
              ...
       </xs:element>
        <xs:element name="FlightResRS" type="xsd:string"/>
</xs:schema>
```

The Hotel.xsd fragment is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://travelco.com/schema/Hotel"
    targetNamespace="http://travelco.com">
       <xs:import schemaLocation="Address.xsd"
                  namespace="http://travelco.com/schema/Address"/>
       <xs:element name="HotelResRQ">
              ...
       </xs:element>
</xs:schema>
```

Address.xsd does not have any child schemas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://travelco.com/schema/Address">
       <xs:complexType name="address_struct">
              ...
       </xs:complexType>
       <xs:element name="Address" type="address_struct"/>
</xs:schema>
```

The graph of *javax.wsdl.** and *javax.wsdl.extensions.schema.** objects is depicted by this diagram:

```
>Definition                                <<< TravelCo.wsdl
 |
 |--->Types                                (the <types> element)
 |     |
 |      --->Schema                         (the in-line <schema> within <types>)
 |          |
 |          |--->SchemaImport              (the schema <import> of Flight.xsd)
 |          |     |
 |          |      --->Schema              <<< Flight.xsd
 |          |          |
 |          |           --->SchemaReference (the schema <include> of Address.xsd)
 |          |               |
 |          |                --->Schema    <<< Address.xsd
 |          |
 |          |--->SchemaReference           (the schema <include> of Hotel.xsd)
 |          |     |
 |          |      --->Schema              <<< Hotel.xsd
 |          |          |
 |          |           --->SchemaImport   (the schema <import> of Address.xsd)
 |          |               |
 |          |                --->Schema    <<< Address.xsd
 |          |
 |           --->SchemaReference           (the schema <redefine> of Address.xsd)
 |               |
 |                --->Schema               <<< Address.xsd
 |
 |--->Import                                (the WSDL <import> of Address.xsd)
       |
        --->Definition                      (a dummy definition)
            |
             --->Types                      (a dummy <types> to hold the schema)
                    |
                     --->Schema            <<< Address.xsd
```

Note, JWSDL handles WSDL imports by having *javax.wsdl.Import* refer to a *Definition* which in turn contains the instantiation of the imported WSDL. In this case, it is a Schema being imported, not a WSDL definition, but the *Import* interface still expects a *Definition*, so JWSDL creates a 'dummy' *Definition* and *Types* to store the *Schema*.

14. Dependencies

JWSDL requires Java 1.2 or greater, and the *org.w3c.dom* interfaces.

JWSDL also depends on the *javax.xml.namespace.QName* class. It has been recognized by the various concerned groups that a common representation of qualified names is necessary. As with JAX-RPC [JAX-RPC], this specification temporarily employs the *javax.xml.namespace.QName* class because no common *QName* representation is specified as of yet. It is expected that when a common *QName* representation is defined, JWSDL will become dependent on the new definition.

15. References

[WSDL] http://www.w3.org/TR/wsdl

[WSDL Schema] http://schemas.xmlsoap.org/wsdl/

[JAX-RPC] http://www.jcp.org/jsr/detail/101.jsp