Java™ APIs for WSDL

(JWSDL)

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

JSR-110 under Java Community Process

Version 1.0

Editors:
Matthew J. Duftler (duftler@us.ibm.com)
Paul Fremantle (pzf@uk.ibm.com)

July 3, 2002

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 schema or schema types.

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

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 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 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 XML Schema. Any children of `<wsdl:types>` elements are treated as extensibility elements.
- 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.

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. All *WSDLReader* implementations must 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 any type of document to be imported. However, JWSDL is only capable of retrieving and processing WSDL documents. If another type of document is imported, such as an XML Schema document, a *javax.wsdl.Import* object will still be created to represent that import; the definition property of that *Import* object will simply be *null*.

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 interfaces and classes:

- 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.

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.

For every WSDL element capable of containing extensibility elements, its corresponding *javax.wsdl.** 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*.

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.

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

12. Extensibility Attributes

There are several methods defined on *javax.wsdl.Part* to enable the representation of extensibility attributes (also called message-typing attributes) that may appear on a `<wsdl:part>` element: *setExtensionAttribute(QName name, QName value)*, *getExtensionAttribute(QName name)*, and *getExtensionAttributes()*. Please see the API documentation for details on how to use these methods. *WSDLReader* and *WSDLWriter* implementations are required to support the use of extensibility attributes on `<wsdl:part>` elements by employing these methods when creating or serializing *javax.wsdl.Parts*.

13. Dependencies

JWSDL requires Java 1.2 or greater, and a JAXP-compliant XML parser.

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.

14. References

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

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