

# The Java API for XML Based RPC (JAX-RPC) 2.0

*Editors Draft  
April 29, 2004*

Editors:  
Marc Hadley  
Roberto Chinnici

Comments to: [jaxrpc-spec-comments@sun.com](mailto:jaxrpc-spec-comments@sun.com)

*Sun Microsystems, Inc.  
4150 Network Circle  
Santa Clara, CA 95054 USA*



## **Java(TM) API for XML based Remote Procedure Call (JAX-RPC) 2.0 Specification (“Specification”)**

**Version: 2.0**

**Status: editors copy**

**Release: April 29, 2004**

**Copyright 2003 Sun Microsystems, Inc.**

**4150 Network Circle, Santa Clara, California 95054, U.S.A**

**All rights reserved.**

### **NOTICE**

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. (“Sun”) and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, Sun hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Sun’s intellectual property rights to review the Specification only for the purposes of evaluation. This license includes the right to discuss the Specification (including the right to provide limited excerpts of text to the extent relevant to the point[s] under discussion) with other licensees (under this or a substantially similar version of this Agreement) of the Specification. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Sun intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (i) two (2) years from the date of Release listed above; (ii) the date on which the final version of the Specification is publicly released; or (iii) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

### **TRADEMARKS**

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun’s licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

### **DISCLAIMER OF WARRANTIES**

THE SPECIFICATION IS PROVIDED “AS IS” AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SUN. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

### **LIMITATION OF LIABILITY**

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Sun (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

## **RESTRICTED RIGHTS LEGEND**

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

## **REPORT**

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

## **GENERAL TERMS**

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

Neither party may assign or otherwise transfer any of its rights or obligations under this Agreement, without the prior written consent of the other party, except that Sun may assign this Agreement to an affiliated company.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

*(LFI#126151/Form ID#011801)*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Goals . . . . .	1
1.2	Non-Goals . . . . .	3
1.3	Requirements . . . . .	3
1.3.1	Describe Relationship To JAXB . . . . .	3
1.3.2	Standardized WSDL Mapping . . . . .	3
1.3.3	Customizable WSDL Mapping . . . . .	4
1.3.4	Standardized Protocol Bindings . . . . .	4
1.3.5	Standardized Transport Bindings . . . . .	4
1.3.6	Standardized Handler Framework . . . . .	4
1.3.7	Versioning and Evolution . . . . .	5
1.3.8	Standardized Synchronous and Asynchronous Invocation . . . . .	5
1.3.9	Session Management . . . . .	5
1.4	Use Cases . . . . .	5
1.4.1	Handler Framework . . . . .	5
1.5	Conventions . . . . .	6
1.6	Expert Group Members . . . . .	6
1.7	Acknowledgements . . . . .	7
<b>2</b>	<b>WSDL 1.1 to Java Mapping</b>	<b>9</b>
2.1	Definitions . . . . .	9
2.1.1	Extensibility . . . . .	9
2.2	Port Type . . . . .	10
2.3	Operation . . . . .	10
2.3.1	Message and Part . . . . .	10
2.3.2	Parameter Order and Return Type . . . . .	12
2.3.3	Holder Classes . . . . .	14

2.3.4	Asynchrony . . . . .	14
2.4	Types . . . . .	18
2.5	Fault . . . . .	18
2.5.1	Example . . . . .	19
2.6	Binding . . . . .	19
2.6.1	General Considerations . . . . .	19
2.6.2	SOAP Binding . . . . .	20
2.6.3	MIME Binding . . . . .	20
2.7	Service and Port . . . . .	20
2.7.1	Example . . . . .	21
2.8	XML Names . . . . .	21
2.8.1	Name Collisions . . . . .	22
<b>3</b>	<b>Java to WSDL 1.1 Mapping</b>	<b>23</b>
3.1	Java Names . . . . .	23
3.1.1	Name Collisions . . . . .	23
3.2	Package . . . . .	23
3.3	Interface . . . . .	24
3.3.1	Inheritance . . . . .	24
3.4	Method . . . . .	24
3.4.1	One Way Operations . . . . .	25
3.5	Method Parameters . . . . .	25
3.5.1	Parameter Classification . . . . .	25
3.5.2	Use of JAXB . . . . .	27
3.6	Service Specific Exception . . . . .	28
3.7	Bindings . . . . .	28
3.7.1	Interface . . . . .	29
3.7.2	Method and Parameters . . . . .	29
3.8	SOAP HTTP Binding . . . . .	29
3.8.1	Interface . . . . .	29
3.8.2	Method and Parameters . . . . .	30
<b>4</b>	<b>Client APIs</b>	<b>33</b>
4.1	javax.xml.rpc.ServiceFactory . . . . .	33
4.1.1	Configuration . . . . .	33

4.1.2	Factory Usage . . . . .	34
4.1.3	Example . . . . .	34
4.2	javax.xml.rpc.Service . . . . .	34
4.2.1	Handler Registry . . . . .	35
4.2.2	Type Mapping Registry . . . . .	35
4.3	javax.xml.rpc.JAXRPCContext . . . . .	35
4.3.1	Standard Properties . . . . .	36
4.3.2	Additional Properties . . . . .	37
4.4	javax.xml.rpc.Binding . . . . .	37
4.5	javax.xml.rpc.BindingProvider . . . . .	37
4.6	javax.xml.rpc.Stub . . . . .	37
4.6.1	Configuration . . . . .	38
4.6.2	Dynamic Proxy . . . . .	38
4.7	javax.xml.rpc.Dispatch . . . . .	39
4.7.1	Configuration . . . . .	40
4.7.2	Operation Invocation . . . . .	40
4.7.3	Operation Response . . . . .	41
4.7.4	Asynchronous Response . . . . .	41
4.7.5	Using JAXB . . . . .	41
4.7.6	Examples . . . . .	42
4.8	javax.xml.rpc.Call . . . . .	43
4.8.1	Configuration . . . . .	43
4.8.2	Operation Invocation . . . . .	44
4.8.3	Example . . . . .	45
4.9	Exceptions . . . . .	46
4.9.1	Protocol Specific Exception Handling . . . . .	46
4.10	Additional Classes . . . . .	47
<b>A</b>	<b>Conformance Requirements</b>	<b>49</b>
	<b>Bibliography</b>	<b>53</b>





# Chapter 1

## Introduction

A remote procedure call (RPC) mechanism allows a client to invoke the methods of a remote service using a familiar local procedure call paradigm. On the client, the RPC infrastructure manages the task of converting the local procedure call arguments into some standard request representation, communicating the request to the remote service and converting any response back into procedure call return values. On the server, the RPC infrastructure manages the task of converting incoming requests into local procedure calls, converting the result of local procedure calls into responses and communicating responses to the client.

XML[1] is a platform-independent means of representing structured information. XML based RPC mechanisms use XML for the representation of RPC requests and responses and inherit XML's platform independence. SOAP[2, 3, 4] describes one such XML based RPC mechanism and "defines, using XML technologies, an extensible messaging framework containing a message construct that can be exchanged over a variety of underlying protocols".

WSDL[5] is "an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information". WSDL can be considered the de-facto interface definition language(IDL) for XML based RPC.

JAX-RPC 1.0[6] defines APIs and conventions for supporting XML based RPC in the Java™ platform. JAX-RPC 1.1[7] adds support for the WS-I Basic Profile 1.0[8] to improve interoperability between JAX-RPC implementations and with services implemented using other technologies.

JAX-RPC 2.0 (this specification) supersedes JAX-RPC 1.1, extending it as described in the following sections.

### 1.1 Goals

Since the release of JAX-RPC 1.0[6], new specifications and new versions of the standards it depends on have been released. JAX-RPC 2.0 relates to these specifications and standards as follows:

**JAXB** Due primarily to scheduling concerns, JAX-RPC 1.0 defined its own data binding facilities. With the release of JAXB 1.0[9] there is no reason to maintain two separate sets of XML mapping rules in the Java™ platform. JAX-RPC 2.0 will delegate data binding-related tasks to the JAXB 2.0[10] specification that is being developed in parallel with JAX-RPC 2.0.

JAXB 2.0[10] will add support for Java to XML mapping, additional support for less used XML schema constructs and provide bidirectional customization of Java  $\Leftrightarrow$  XML data binding. JAX-

RPC 2.0 will allow full use of JAXB provided facilities including binding customization and optional schema validation.	1 2
<b>SOAP 1.2</b> Whilst SOAP 1.1 is still widely deployed, it's expected that services will migrate to SOAP 1.2[3, 4] now that it is a W3C Recommendation. JAX-RPC 2.0 will add support for SOAP 1.2 whilst requiring continued support for SOAP 1.1.	3 4 5
<b>WSDL 2.0</b> The W3C is expected to progress WSDL 2.0[11] to Recommendation during the lifetime of this JSR. JAX-RPC 2.0 will add support for WSDL 2.0 whilst requiring continued support for WSDL 1.1.	6 7
<b>WS-I Basic Profile 1.1</b> JAX-RPC 1.1 added support for WS-I Basic Profile 1.0. WS-I Basic Profile 1.1 is expected to supersede 1.0 during the lifetime of this JSR and JAX-RPC 2.0 will add support for the additional clarifications it provides.	8 9 10
<b>A Metadata Facility for the Java Programming Language (JSR 175)</b> JAX-RPC 2.0 will define use of Java annotations[12] to simplify the most common development scenarios for both clients and servers.	11 12
<b>Web Services Metadata for the Java Platform (JSR 181)</b> JAX-RPC 2.0 will align with and complement the annotations defined by JSR 181[13].	13 14
<b>Implementing Enterprise Web Services (JSR 109)</b> The JSR 109[14] defined <code>jaxrpc-mapping-info</code> deployment descriptor provides deployment time Java $\Leftrightarrow$ WSDL mapping functionality. In conjunction with JSR 181[13], JAX-RPC 2.0 will complement this mapping functionality with development time Java annotations that control Java $\Leftrightarrow$ WSDL mapping.	15 16 17 18
<b>Web Services Security (JSR 183)</b> JAX-RPC 2.0 will align with and complement the security APIs defined by JSR 183[15].	19 20
JAX-RPC 2.0 will improve support for document/message centric usage:	21
<b>Asynchrony</b> JAX-RPC 2.0 will add support for client side asynchronous operations.	22
<b>Non-HTTP Transports</b> JAX-RPC 2.0 will improve the separation between the XML based RPC framework and the underlying transport mechanism to simplify use of JAX-RPC with non-HTTP transports.	23 24
<b>Message Access</b> JAX-RPC 2.0 will simplify client and service access to the messages underlying an exchange.	25 26
<b>Session Management</b> JAX-RPC 1.1 session management capabilities are tied to HTTP. JAX-RPC 2.0 will add support for message based session management.	27 28
JAX-RPC 2.0 will also address issues that have arisen with experience of implementing and using JAX-RPC 1.0:	29 30
<b>Inclusion in J2SE</b> JAX-RPC 2.0 will prepare JAX-RPC for inclusion in a future version of J2SE. Application portability is a key requirement and JAX-RPC 2.0 will define mechanisms to produce fully portable clients.	31 32 33
<b>Handlers</b> JAX-RPC 2.0 will simplify the development of handlers and will provide a mechanism to allow handlers to collaborate with service clients and service endpoint implementations.	34 35
<b>Versioning and Evolution of Web Services</b> JAX-RPC 2.0 will describe techniques and mechanisms to ease the burden on developers when creating new versions of existing services.	36 37
<b>Backwards Compatibility of Binary Artifacts</b> JAX-RPC 2.0 will not preclude preservation of binary compatibility between JAX-RPC 1.x and 2.0 implementation runtimes.	38 39

## 1.2 Non-Goals

The following are non-goals:

**Pluggable data binding** JAX-RPC 2.0 will defer data binding to JAXB[10], it is not a goal to provide a plug-in API to allow other types of data binding technologies to be used in place of JAXB. However, JAX-RPC 2.0 will maintain the capability to selectively disable data binding to provide an XML based fragment suitable for use as input to alternative data binding technologies.

**SOAP Encoding Support** Use of the SOAP encoding is essentially deprecated in the web services community, e.g. the WS-I Basic Profile[8] excludes SOAP encoding. Instead, literal usage is preferred, either in the RPC or document style.

SOAP 1.1 encoding is supported in JAX-RPC 1.0 and 1.1 but its support in JAX-RPC 2.0 runs counter to the goal of delegation of data binding to JAXB. Therefore JAX-RPC 2.0 will make support for SOAP 1.1 encoding optional and defer description of it to JAX-RPC 1.1.

Support for the SOAP 1.2 Encoding[4] is optional in SOAP 1.2 and JAX-RPC 2.0 will not add support for SOAP 1.2 encoding.

**Backwards Compatibility of Generated Artifacts** JAX-RPC 1.0 and JAXB 1.0 bind XML to Java in different ways. Generating source code that works with unmodified JAX-RPC 1.x client source code is not a goal.

**Support for Java versions prior to J2SE 1.5** JAX-RPC 2.0 relies on many of the Java language features added in J2SE 1.5. It is not a goal to support JAX-RPC 2.0 on Java versions prior to J2SE 1.5.

**Service Registration and Discovery** It is not a goal of JAX-RPC 2.0 to describe registration and discovery of services via UDDI or ebXML RR. This capability is provided independently by JAXR[16].

## 1.3 Requirements

### 1.3.1 Describe Relationship To JAXB

JAX-RPC describes the WSDL  $\Leftrightarrow$  Java mapping, but data binding is delegated to JAXB[10]. The specification must clearly designate where JAXB rules apply to the WSDL  $\Leftrightarrow$  Java mapping without reproducing those rules and must describe how JAXB capabilities (e.g. the JAXB binding language) are incorporated into JAX-RPC. JAX-RPC is required to be able to influence the JAXB binding, e.g. to avoid name collisions and to be able to control schema validation on serialization and deserialization.

### 1.3.2 Standardized WSDL Mapping

WSDL is the de-facto interface definition language for XML-based RPC. The specification must specify a standard WSDL  $\Leftrightarrow$  Java mapping. The following versions of WSDL must be supported:

- WSDL 1.1[5] as clarified by the WS-I Basic Profile[8, 17],
- WSDL 2.0[11, 18, 19].

The standardized WSDL mapping will describe the default WSDL  $\Leftrightarrow$  Java mapping. The default mapping may be overridden using customizations as described below.

### 1.3.3 Customizable WSDL Mapping

The specification must provide a standard way to customize the WSDL  $\Leftrightarrow$  Java mapping. The following customization methods will be specified:

**Java Annotations** In conjunction with JAXB[10] and JSR 181[13], the specification will define a set of standard annotations that may be used in Java source files to specify the mapping from Java artifacts to their associated WSDL components. The annotations will support mapping to both WSDL 1.1 and WSDL 2.0.

**WSDL Annotations** In conjunction with JAXB[10] and JSR 181[13], the specification will define a set of standard annotations that may be used either within WSDL documents or as in an external form to specify the mapping from WSDL components to their associated Java artifacts. The annotations will support mapping from both WSDL 1.1 and WSDL 2.0.

The specification must describe the precedence rules governing combinations of the customization methods.

### 1.3.4 Standardized Protocol Bindings

The specification must describe standard bindings to the following protocols:

- SOAP 1.1[2] as clarified by the WS-I Basic Profile[8, 17],
- SOAP 1.2[3, 4].

The specification must not prevent non-standard bindings to other protocols.

### 1.3.5 Standardized Transport Bindings

The specification must describe standard bindings to the following protocols:

- HTTP/1.1[20].

The specification must not prevent non-standard bindings to other transports.

### 1.3.6 Standardized Handler Framework

The specification must include a standardized handler framework that describes:

**Data binding for handlers** The framework will offer data binding facilities to handlers and will support handlers that are decoupled from the SAAJ API.

**Handler Context** The framework will describe a mechanism for communicating properties between handlers and the associated service clients and service endpoint implementations.

**Bidirectional handler chains** Support for bidirectional handler chains that are used for both outgoing and incoming messages will be added to the existing unidirectional handler chains.

**Unified Response and Fault Handling** The `handleResponse` and `handleFault` methods will be unified and the declarative model for handlers will be improved.

### 1.3.7 Versioning and Evolution

The specification must describe techniques and mechanisms to support versioning of service endpoint interfaces. The facilities must allow new versions of an interface to be deployed whilst maintaining compatibility for existing clients.

### 1.3.8 Standardized Synchronous and Asynchronous Invocation

There must be a detailed description of the generated method signatures to support both asynchronous and synchronous method invocation in stubs generated by JAX-RPC. Both forms of invocation will support a user configurable timeout period.

### 1.3.9 Session Management

The specification must describe a standard session management mechanism including:

**Session APIs** Definition of a session interface and methods to obtain the session interface and initiate sessions for handlers and service endpoint implementations.

**HTTP based sessions** The session management mechanism must support HTTP cookies and URL rewriting.

**SOAP based sessions** There must be a standardized way to identify the location of a session identifier in SOAP message content using Java or WSDL annotations.

## 1.4 Use Cases

### 1.4.1 Handler Framework

#### Reliable Messaging Support

A developer wishes to add support for a reliable messaging SOAP feature to an existing service endpoint. The support takes the form of a JAX-RPC handler.

#### Message Logging

A developer wishes to log incoming and outgoing messages for later analysis, e.g. checking messages using the WS-I testing tools.

#### WS-I Conformance Checking

A developer wishes to check incoming and outgoing messages for conformance to one or more WS-I profiles at runtime.

## 1.5 Conventions

The keywords ‘MUST’, ‘MUST NOT’, ‘REQUIRED’, ‘SHALL’, ‘SHALL NOT’, ‘SHOULD’, ‘SHOULD NOT’, ‘RECOMMENDED’, ‘MAY’, and ‘OPTIONAL’ in this document are to be interpreted as described in RFC 2119[21].

For convenience, conformance requirements are called out from the main text as follows:

*Conformance Requirement (Example):* Implementations **MUST** do something.

A list of all such conformance requirements can be found in appendix A.

Java code and XML fragments are formatted as shown in figure 1.1:

Figure 1.1: Example Java Code

```
1 package com.example.hello;
2
3 public class Hello {
4     public static void main(String args[]) {
5         System.out.println("Hello World");
6     }
7 }
```

This specification uses a number of namespace prefixes throughout; they are listed in Table 1.1. Note that the choice of any namespace prefix is arbitrary and not semantically significant (see XML Infoset[22]).

Prefix	Namespace	Notes
env	<a href="http://www.w3.org/2003/05/soap-envelope">http://www.w3.org/2003/05/soap-envelope</a>	A normative XML Schema[23, 24] document for the <a href="http://www.w3.org/2003/05/soap-envelope">http://www.w3.org/2003/05/soap-envelope</a> namespace can be found at <a href="http://www.w3.org/2003/05/soap-envelope">http://www.w3.org/2003/05/soap-envelope</a> .
xsd	<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>	The namespace of the XML schema[23, 24] specification
wsdl	<a href="http://schemas.xmlsoap.org/wsdl/">http://schemas.xmlsoap.org/wsdl/</a>	The namespace of the WSDL schema[5]
soap	<a href="http://schemas.xmlsoap.org/wsdl/soap/">http://schemas.xmlsoap.org/wsdl/soap/</a>	The namespace of the WSDL SOAP binding schema[23, 24]
jaxb	<a href="http://java.sun.com/xml/ns/jaxb">http://java.sun.com/xml/ns/jaxb</a>	The namespace of the JAXB [9] specification

Table 1.1: Prefixes and Namespaces used in this specification.

Namespace names of the general form ‘[http://example.org/...](http://example.org/)’ and ‘[http://example.com/...](http://example.com/)’ represent application or context-dependent URIs (see RFC 2396[20]).

All parts of this specification are normative, with the exception of examples and sections explicitly marked as ‘Non-Normative’.

## 1.6 Expert Group Members

The following people have contributed to this specification:

Chavdar Baikov (SAP AG)	1
Russell Butek (IBM)	2
Manoj Cheenath (BEA Systems)	3
Ugo Corda (SeeBeyond Technology Corp)	4
Glen Daniels (Sonic Software)	5
Alan Davies (SeeBeyond Technology Corp)	6
Jim Frost (Art Technology Group Inc)	7
Kevin R. Jones (Developmentor)	8
Toshiyuki Kimura (NTT Data Corp)	9
Doug Kohlert (Sun Microsystems, Inc)	10
Daniel Kulp (IONA Technologies PLC)	11
Sunil Kunisetty (Oracle)	12
Changshin Lee (Tmax Soft, Inc)	13
Srividya Natarajan (Nokia Corporation)	14
Bjarne Rasmussen (Novell, Inc)	15
Sebastien Sahuc (Intalio, Inc.)	16
Rahul Sharma (Motorola)	17
Rajiv Shivane (Pramati Technologies)	18
Dennis M. Sosnoski (Sosnoski Software)	19
Christopher St. John (WebMethods Corporation)	20

## **1.7 Acknowledgements**

TBD





## Chapter 2

# WSDL 1.1 to Java Mapping

This chapter describes the mapping from WSDL 1.1 to Java. This mapping is used when generating web service interfaces for clients and endpoints from a WSDL 1.1 description.

*Conformance Requirement (WSDL 1.1 support):* Implementations MUST support mapping WSDL 1.1 to Java.

The following sections describe the default mapping from each WSDL 1.1 construct to the equivalent Java construct. In WSDL 1.1, the separation between the abstract port type definition and the binding to a protocol is not complete. Bindings impact the mapping between WSDL elements used in the abstract port type definition and Java method parameters. Section 2.6 describes binding dependent mappings.

## 2.1 Definitions

A WSDL document has a root `wsdl:definitions` element. A `wsdl:definitions` element and its associated `targetNamespace` attribute is mapped to a Java package. There is no standard mapping from the value of a `wsdl:definitions` element's `targetNamespace` attribute to a Java package name.

*Conformance Requirement (Definitions mapping):* Implementations MUST provide a means for the user to specify the Java package name corresponding to the value of a `wsdl:definitions` element's `targetNamespace` attribute when mapping a WSDL `definitions` element to a Java package.

No specific authoring style is required for the input WSDL document; implementations should support WSDL that uses the WSDL and XML Schema import directives.

*Conformance Requirement (Support for WSDL and XML Schema import directives):* Implementations MUST support the WS-I Basic Profile 1.0[8] defined mechanisms (See R2001, R2002 and R2003) for use of WSDL and XML Schema import directives.

### 2.1.1 Extensibility

WSDL 1.1 allows extension elements and attributes to be added to many of its constructs. JAX-RPC specifies the mapping to Java of the extensibility elements and attributes defined for the SOAP and MIME bindings. JAX-RPC does not address mapping of any other extensibility elements or attributes and does not provide a standard extensibility framework though which such support could be added in a standard way. Future versions of JAX-RPC might add additional support for standard extensions as these become available.

*Conformance Requirement (WSDL extension support):* An implementation MAY support mapping of WSDL extensibility elements and attributes not described in JAX-RPC. Note that such support may limit interoperability and application portability.

## 2.2 Port Type

A WSDL port type is a named set of abstract operation definitions. A `wsdl:portType` element is mapped to a Java interface in the package mapped from the `wsdl:definitions` element (see section 2.1 for a description of `wsdl:definitions` mapping). A Java interface mapped from a `wsdl:portType` is called a *Service Endpoint Interface* or SEI for short.

*Conformance Requirement (SEI naming):* In the absence of customizations, the name of an SEI MUST be the value of the `name` attribute of the corresponding `wsdl:portType` element mapped according to the rules described in section 2.8.

*Conformance Requirement (Extending `java.rmi.Remote`):* An SEI MUST extend the `java.rmi.Remote` interface.

An SEI contains Java methods mapped from the `wsdl:operation` child elements of the corresponding `wsdl:portType`, see section 2.3 for further details on `wsdl:operation` mapping. WSDL 1.1 does not support port type inheritance so each generated SEI will contain methods for all operations in the corresponding port type.

## 2.3 Operation

Each `wsdl:operation` in a `wsdl:portType` is mapped to a Java method in the corresponding Java service endpoint interface.

*Conformance Requirement (Method naming):* In the absence of customizations, the name of a mapped Java method MUST be the value of the `name` attribute of the `wsdl:operation` element mapped according to the rules described in section 2.8.

*Conformance Requirement (`RemoteException` required):* A mapped Java method MUST declare `java.rmi.RemoteException` in its `throws` clause.

The WS-I Basic Profile[8] R2304 requires that operations within a `wsdl:portType` have unique values for their `name` attribute so mapping of WS-I compliant WSDL descriptions will not generate Java interfaces with overloaded methods. However, for backwards compatibility, JAX-RPC supports operation name overloading provided the overloading does not cause conflicts (as specified in the Java Language Specification[25]) in the mapped Java service endpoint interface declaration.

*Conformance Requirement (Transmission primitive support):* An implementation MUST support mapping of operations that use the one-way and request-response transmission primitives.

Mapping of notification and solicit-response operations is out of scope.

### 2.3.1 Message and Part

Each `wsdl:operation` refers to one or more `wsdl:message` elements via child `wsdl:input`, `wsdl:output` and `wsdl:fault` elements that describe the input, output and fault messages for the operation

respectively. Each operation can specify one input message, zero or one output message and zero or more fault messages.

Fault messages are mapped to application specific exceptions, see section 2.5. The contents of input and output messages are mapped to Java method parameters using two different styles: non-wrapper style and wrapper style. The two mapping styles are described in the following subsections.

## Non-wrapper Style

A `wsdl:message` is composed of zero or more `wsdl:part` elements. Message parts are classified as follows:

**in** The message part is present only in the operation's input message.

**out** The message part is present only in the operation's output message.

**in/out** The message part is present in both the operation's input message and output message.

Two parts are considered equal if they have the same values for their `name` attribute and they reference the same global element or type. Using non-wrapper style, message parts are mapped to Java parameters according to their classification as follows:

**in** The message part is mapped to a method parameter.

**out** The message part is mapped to a method parameter using a holder class (see section 2.3.3) or is mapped to the method return type.

**in/out** The message part is mapped to a method parameter using a holder class.

*Conformance Requirement (Non-wrapped parameter naming):* In the absence of customization, the name of a mapped Java method parameter **MUST** be the value of the `name` attribute of the `wsdl:part` element mapped according to the rules described in sections 2.8 and 2.8.1.

Section 2.3.2 defines rules that govern the ordering of parameters in mapped Java methods and identification of the part that is mapped to the method return type.

## Wrapper Style

A WSDL operation qualifies for wrapper style mapping only if the following criteria are met:

- (i) The operations input and output message each contain only a single part
- (ii) The input message part refers to a global element declaration whose `localname` is equal to the operation name and whose `namespace` name is equal to the target namespace of the WSDL document.
- (iii) The output message part refers to a global element declaration.
- (iv) The elements referred to by the input and output message parts (henceforth referred to as *wrapper* elements) are both complex types defined using the `xsd:sequence` compositor that each contain only element declarations.

**Editors Note 2.1** *Should we relax restriction (iv) for the output wrapper element since it would still be possible to unwrap the input wrapper element and generate an out parameter for the output wrapper element as a whole ?*

*Conformance Requirement (Default mapping mode):* Operations that do not meet the above criteria **MUST** be mapped using non-wrapper style.

In some cases use of the wrapper style mapping can lead to undesirable Java method signatures and use of non-wrapper style mapping would be preferred.

*Conformance Requirement (Disabling wrapper style):* Implementations **MUST** provide a means to disable wrapper style mapping of operations.

Using wrapper style, the child elements of the wrapper element (henceforth called wrapper children) are mapped to Java parameters, wrapper children are classified as follows:

**in** The wrapper child is only present in the input message part's wrapper element.

**out** The wrapper child is only present in the output message part's wrapper element.

**in/out** The wrapper child is present in both the input and output message part's wrapper element.

Two wrapper children are considered equal if they have the same local name and the same type. The mapping depends on the classification of the wrapper child as follows:

**in** The wrapper child is mapped to a method parameter.

**out** The wrapper child is mapped to a method parameter using a holder class (see section 2.3.3) or is mapped to the method return value.

**in/out** The wrapper child is mapped to a method parameter using a holder class.

*Conformance Requirement (Wrapped parameter naming):* In the absence of customization, the name of a mapped Java method parameter **MUST** be the value of the local name of the wrapper child mapped according to the rules described in sections 2.8 and 2.8.1.

## Example

Figure 2.1 shows a WSDL extract and the Java method that results from using wrapper and non-wrapper mapping styles.

### 2.3.2 Parameter Order and Return Type

A `wsdl:operation` element may have a `parameterOrder` attribute that defines the ordering of parameters in a mapped Java method as follows:

- Message parts are either listed or unlisted. If the value of a `wsdl:part` element's `name` attribute is present in the `parameterOrder` attribute then the part is listed, otherwise it is unlisted.
- Parameters that are mapped from listed parts are either listed or unlisted. Parameters that are mapped from listed parts are listed; parameters that are mapped from unlisted parts are unlisted.

```

1  <!-- WSDL extract -->
2  <types>
3      <xsd:element name="setLastTradePrice">
4          <xsd:complexType>
5              <xsd:sequence>
6                  <xsd:element name="tickerSymbol" type="xsd:string"/>
7                  <xsd:element name="lastTradePrice" type="xsd:float"/>
8              </xsd:sequence>
9          </xsd:complexType>
10     </xsd:element>
11
12     <xsd:element name="setLastTradePriceResponse">
13         <xsd:complexType>
14             <xsd:sequence/>
15         </xsd:complexType>
16     </xsd:element>
17 </types>
18
19 <message name="setLastTradePrice">
20     <part name="setLastTradePrice"
21         element="tns:setLastTradePrice"/>
22 </message>
23
24 <message name="setLastTradePriceResponse">
25     <part name="setLastTradePriceResponse"
26         element="tns:setLastTradePriceResponse"/>
27 </message>
28
29 <portType name="StockQuoteUpdater">
30     <operation name="setLastTradePrice">
31         <input message="tns:setLastTradePrice"/>
32         <output message="tns:setLastTradePriceResponse"/>
33     </operation>
34 </portType>
35
36 // non-wrapper style mapping
37 void setLastTradePrice(SetLastTradePrice setLastTradePrice)
38     throws RemoteException;
39
40 // wrapper style mapping
41 void setLastTradePrice(String tickerSymbol, float lastTradePrice)
42     throws RemoteException;

```

Figure 2.1: Wrapper and non-wrapper mapping styles

- Parameters that are mapped from wrapper children (wrapper style mapping only) are unlisted. 1
- Listed parameters appear first in the method signature in the order in which their corresponding parts are listed in the `parameterOrder` attribute. 2  
3
- Unlisted parameters either form the return type or follow the listed parameters 4
- The return type is determined as follows: 5
  - Non-wrapper style mapping** If there is a single unlisted `out` part then it forms the method return type, otherwise the return type is `void`. 6  
7
  - Wrapper style mapping** If there is a single `out` wrapper child then it forms the method return type, if there is an `out` wrapper child with a local name of “return” then it forms the method return type, otherwise the return type is `void`. 8  
9  
10
- Unlisted parameters that do not form the return type follow the listed parameters in the following order: 11  
12
  1. Parameters mapped from `in` and `in/out` parts appear in the same order the corresponding parts appear in the input message. 13  
14
  2. Parameters mapped from `in` and `in/out` wrapper children (wrapper style mapping only) appear in the same order as the corresponding elements appear in the wrapper. 15  
16
  3. Parameters mapped from `out` parts appear in the same order the corresponding parts appear in the output message. 17  
18
  4. Parameters mapped from `out` wrapper children (wrapper style mapping only) appear in the same order as the corresponding wrapper children appear in the wrapper. 19  
20

### 2.3.3 Holder Classes 21

Holder classes are used to support `out` and `in/out` parameters in mapped method signatures. They provide a mutable wrapper for otherwise immutable object references. JAX-RPC 1.1 provides type safe holder classes for the Java types that are mapped from simple XML data types and these are retained in JAX-RPC 2.0 for backwards compatibility when mapping Java interfaces to WSDL. However, for WSDL to Java mapping, JAX-RPC 2.0 adds a new generic holder class (`javax.xml.rpc.holders.GenericHolder<T>`) that should be used instead. 22  
23  
24  
25  
26  
27

Parameters whose XML data type would normally be mapped to a Java primitive type (e.g. `xsd:int` to `int`) are instead mapped to a `GenericHolder` typed on the Java wrapper class corresponding to the primitive type. E.g. an `out` or `in/out` parameter whose XML data type would normally be mapped to a Java `int` is instead mapped to `GenericHolder<java.lang.Integer>`. 28  
29  
30  
31

*Conformance Requirement (Use of `GenericHolder`):* Implementations MUST map `out` and `in/out` method parameters using `javax.xml.rpc.holders.GenericHolder<T>`. 32  
33

### 2.3.4 Asynchrony 34

In addition to the synchronous mapping of `wsdl:operation` described above, an asynchronous mapping is also supported. It is expected that the asynchronous mapping will be useful in some but not all cases and therefore generation of the asynchronous mapped interfaces should be optional at the clients discretion. 35  
36  
37

*Conformance Requirement (Asynchronous mapping required):* An implementation MUST support the asynchronous mapping.

*Conformance Requirement (Asynchronous mapping option):* An implementation MUST provide a means for a client to enable and disable the asynchronous mapping.

## Standard Asynchronous Interfaces

The following standard interfaces are used in the asynchronous operation mapping:

**javax.xml.rpc.Response** A generic interface that is used to group the results of a method invocation with the response context. Response extends Future<T> to provide asynchronous result polling capabilities.

**javax.xml.rpc.AsyncHandler** A generic interface that clients implement to receive results in an asynchronous callback.

## Operation

Each wsdl:operation is mapped to two methods in the corresponding asynchronous service endpoint interface:

**Polling method** A polling method returns a typed Response<ResponseBean> that may be polled using methods inherited from Future<T> to determine when the operation has completed and to retrieve the results. See below for further details on ResponseBean.

**Callback method** A callback method takes an additional final parameter that is an instance of a typed AsyncHandler<ResponseBean> and returns a wildcard Future<?> that may be polled to determine when the operation has completed. The object returned from Future<?>.get() has no standard type. Client code should not attempt to cast the object to any particular type as this will result in non-portable behavior.

*Conformance Requirement (Asynchronous method naming):* In the absence of customizations, the name of the polling and callback methods MUST be the value of the name attribute of the wsdl:operation suffixed with “Async” mapped according to the rules described in sections 2.8 and 2.8.1.

*Conformance Requirement (Failed method invocation):* If there is any error prior to invocation of the operation, an implementation MUST throw a JAXRPCException. Errors that occur during the invocation are reported when the client attempts to retrieve the results of the operation.

## Message and Part

The asynchronous mapping supports both wrapper and non-wrapper mapping styles, but differs in how it maps out and in/out parts or wrapper children:

**in** The part or wrapper child is mapped to a method parameter as described in section 2.3.1.

**out** The part or wrapper child is mapped to a property of the response bean (see below).

**in/out** The part or wrapper child is mapped to a method parameter (no holder class) and to a property of the response bean.

## Response Bean

A response bean is a mapping of an operation's output message, it contains properties for each out and in/out message part or wrapper child.

*Conformance Requirement (Response bean naming):* In the absence of customizations, the name of a response bean MUST be the value of the name attribute of the `wsdl:operation` suffixed with "Response" mapped according to the rules described in sections 2.8 and 2.8.1.

A response bean is mapped from a global element declaration following the rules described in section 2.4. The global element declaration is formed as follows (in order of preference):

- If the operation output message contains a single part and that part refers to a global element declaration then use the referenced global element.
- Synthesize a global element declaration of a complex type defined using the `xsd:sequence` compositor. Each output message part is mapped to a child of the synthesized element as follows:
  - Each global element referred to by an output part is added as a child of the sequence.
  - Each part that refers to a type is added as a child of the sequence by creating an element whose name is the value of the name attribute of the `wsdl:part` element and whose type is the value of the type attribute of the `wsdl:part` element

If the resulting response bean has only a single property then the bean wrapper should be discarded in method signatures.

## Mapping Examples

Figure 2.2 shows an example of the asynchronous operation mapping. Note that the mapping uses `Float` instead of a response bean wrapper (`GetPriceResponse`) since the synthesized global element declaration for the operations output message (lines 17–24) maps to a response bean that contains only a single property.

## Usage Examples

Synchronous use.

```
1 Service service = ...;
2 StockQuote quoteService = (StockQuote)service.getPort(portName);
3 Float quote=quoteService.getPrice(ticker);
```

Asynchronous polling use.

```
1 Service service = ...;
2 StockQuote quoteService = (StockQuote)service.getPort(portName);
3 Response<Float> response=quoteService.getPriceAsync(ticker);
4 while (!response.isDone) {
5     // do something while we wait
6 }
7 Float quote = response.get();
```



```

1  <!-- WSDL extract -->
2  <message name="getPrice">
3      <part name="ticker" type="xsd:string"/>
4  </message>
5
6  <message name="getPriceResponse">
7      <part name="price" type="xsd:float"/>
8  </message>
9
10 <portType name="StockQuote">
11     <operation name="getPrice">
12         <input message="tns:getPrice"/>
13         <output message="tns:getPriceResponse"/>
14     </operation>
15 </portType>
16
17 <!-- Synthesized response bean element -->
18 <xsd:element name="getPriceResponse">
19     <xsd:complexType>
20         <xsd:sequence>
21             <xsd:element name="price" type="xsd:float"/>
22         </xsd:sequence>
23     </xsd:complexType>
24 </xsd:element>
25
26 // synchronous mapping
27 interface StockQuote {
28     float getPrice(String ticker) throws RemoteException;
29 }
30
31 // asynchronous mapping
32 interface StockQuoteAsync {
33     Response<Float> getPriceAsync(String ticker);
34     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
35 }

```

Figure 2.2: Asynchronous operation mapping

Asynchronous callback use.

1

```

1  class MyPriceHandler implements AsyncHandler<Float> {
2      ...
3      public void handleResponse(Response<Float> response) {
4          Float price = response.get();
5          // do something with the result
6      }
7  }
8
9  Service service = ...;
10 StockQuote quoteService = (StockQuote)service.getPort(portName);
11 MyPriceHandler myPriceHandler = new MyPriceHandler();
12 (void)quoteService.getPriceAsync(ticker, myPriceHandler);

```

14

## 2.4 Types

Mapping of XML Schema types to Java is described by the JAXB 2.0 specification[10]. The contents of a `wsdl:types` section is passed to JAXB along with any additional type or element declarations (e.g. see section 2.3.4) required to map other WSDL constructs to Java. E.g. section 2.3.4 defines an algorithm for synthesizing additional global element declarations to provide a mapping from WSDL operations to asynchronous Java method signatures.

JAXB supports mapping XML types to either Java interfaces or classes. JAX-RPC uses the class based mapping of JAXB.

*Conformance Requirement (JAXB Class Mapping):* An implementation **MUST** use the JAXB class based mapping when mapping WSDL types to Java.

## 2.5 Fault

A `wsdl:fault` element is mapped to a Java exception.

*Conformance Requirement (Exception naming):* In the absence of customizations, the name of a mapped exception **MUST** be the value of the name attribute of the `wsdl:message` referred to by the `wsdl:fault` element mapped according to the rules in sections 2.8 and 2.8.1.

A `wsdl:fault` element refers to a `wsdl:message` that contains a single part. The XML Schema type or global element declaration referred to by that part is mapped to a Java bean, henceforth called a fault bean, using the mapping described in section 2.4. An implementation generates a wrapper exception class that extends `java.lang.Exception` and contains the following methods:

***WrapperException(FaultBean faultInfo, String message, Throwable cause)*** A constructor where *WrapperException* is replaced with the name of the generated wrapper exception and *FaultBean* is replaced by the name of the generated fault bean.

***FaultBean getFaultInfo()*** Getter to obtain the fault information, where *FaultBean* is replaced by the name of the generated fault bean.

## 2.5.1 Example

Figure 2.3 shows an example of the WSDL fault mapping described above.

```

1  <!-- WSDL extract -->
2  <types>
3      <xsd:schema targetNamespace="...">
4          <xsd:element name="faultDetail">
5              <xsd:complexType>
6                  <xsd:sequence>
7                      <xsd:element name="majorCode" type="xsd:int"/>
8                      <xsd:element name="minorCode" type="xsd:int"/>
9                  </xsd:sequence>
10             </xsd:complexType>
11         </xsd:element>
12     </xsd:schema>
13 </types>
14
15 <message name="operationException">
16     <part name="faultDetail" element="tns:faultDetail"/>
17 </message>
18
19 <portType name="StockQuoteUpdater">
20     <operation name="setLastTradePrice">
21         <input .../>
22         <output .../>
23         <fault message="tns:operationException"/>
24     </operation>
25 </portType>
26
27 // fault mapping
28 class OperationException extends Exception {
29     OperationException(FaultDetail faultInfo, String message,
30         Throwable cause) {...}
31     FaultDetail getFaultInfo() {...}
32 }

```

Figure 2.3: Fault mapping

## 2.6 Binding

The mapping from WSDL 1.1 to Java is based on the abstract description of a `wsdl:portType` and its associated operations. However, the binding of a port type to a protocol can introduce changes in the mapping – this section describes those changes in the general case and specifically for the mandatory WSDL 1.1 protocol bindings.

### 2.6.1 General Considerations

R2209 in WS-I Basic Profile 1.1[17] recommends that all parts of a message be bound but does not require it.

*Conformance Requirement (Unbound message parts):* To preserve the protocol independence of mapped operations, an implementation **MUST NOT** ignore unbound message parts when mapping from WSDL 1.1 to Java. Instead an implementation **MUST** generate binding code that ignores `in` and `in/out` parameters mapped from unbound parts and that presents `out` parameters mapped from unbound parts as `null`.

## 2.6.2 SOAP Binding

This section describes changes to the WSDL 1.1 to Java mapping that may result from use of certain SOAP binding extensions.

### Header Binding Extension

A `soap:header` element may be used to bind a part from a message to a SOAP header. As clarified by R2208 in WS-I Basic Profile 1.1[17], the part may belong to either the message bound by the `soap:body` or to a different message:

- If the part belongs to the message bound by the `soap:body` then it is mapped to a method parameter as described in section 2.3.
- If the part belongs to a different message than that bound by the `soap:body` then it may optionally be mapped to an additional method parameter. When mapped to a parameter, the part is treated as an additional unlisted part for the purposes of the mapping described in section 2.3. This additional part does not affect eligibility for wrapper style mapping of the message bound by the `soap:body`, as described in section 2.3.1, however the additional part is mapped using the non-wrapper style, i.e. it is not subject to unwrapping.

*Conformance Requirement (Mapping additional header parts):* An implementation **MUST** provide a means to enable mapping of additional parts bound by a `soap:header` to method parameters. The default is to not map such parts to method parameters.

### Header Fault Binding Extension

A `soap:header` element can contain zero or more `soap:headerfault` elements that describe faults that may arise when processing the header. If the part bound by the `soap:header` is mapped to a method parameter then each child `soap:headerfault` is mapped to an additional exception thrown by the mapped method according to the mapping described in 2.5.

## 2.6.3 MIME Binding

**Editors Note 2.2** *Once the WS-I Attachments Profile 1.0 is finalized, this section will contain a description of the perturbations to the WSDL 1.1 to Java mapping described above that result from the use of the WSDL MIME binding as clarified by WS-I*

## 2.7 Service and Port

A `wsdl:service` is a collection of related `wsdl:port` elements. A `wsdl:port` element describes a port type bound to a particular protocol (a `wsdl:binding`) that is available at particular endpoint ad-

dress. A `wsdl:service` element is mapped to a generated service interface that extends `javax.xml-  
.rpc.Service` (see section 4.2 for more information on the `Service` interface).

*Conformance Requirement (Service interface required):* A generated service interface **MUST** extend `javax-  
.xml.rpc.Service`.

For each port in the service, the generated service interface contains the following methods:

***ServiceEndpointInterface* `getPortName()`** One required method that takes no parameters and returns an instance of a generated stub class or dynamic proxy that implements the mapped service endpoint interface.

***ServiceEndpointInterface* `getPortName(params)`** Zero or more optional additional methods that include parameters specific to the endpoint or port configuration and returns an instance of a generated stub class or dynamic proxy that implements the mapped service endpoint interface. Such additional methods are implementation specific.

*Conformance Requirement (Failed `getPortName`):* `getPortName` **MUST** throw `javax.xml.rpc.Service-  
Exception` on failure.

The value of *PortName* in the above is derived as follows: the value of the `name` attribute of the `wsdl:service` element is first mapped to a Java identifier according to the rules described in section 2.8, this Java identifier is then treated as a `JavaBean` property for the purposes of deriving the `getPortName` method name.

## 2.7.1 Example

The following shows a WSDL extract and the resulting generated service interface.

```
1  <!-- WSDL extract -->
2  <wsdl:service name="StockQuoteService">
3      <wsdl:port name="StockQuoteHTTPPort" binding="StockQuoteHTTPBinding"/>
4      <wsdl:port name="StockQuoteSMTPPort" binding="StockQuoteSMTPBinding"/>
5  </wsdl:service>
6
7  // Generated Service Interface
8  public interface StockQuoteService extends javax.xml.rpc.Service {
9      StockQuoteProvider getStockQuoteHTTPPort()
10         throws ServiceException;
11      StockQuoteProvider getStockQuoteSMTPPort()
12         throws ServiceException;
13  }
```

In the above, `StockQuoteProvider` is the service endpoint interface mapped from the WSDL port type for both referenced bindings.

## 2.8 XML Names

Appendix C of JAXB 1.0[9] defines a mapping from XML names to Java identifiers. JAX-RPC uses this mapping to convert WSDL identifiers to Java identifiers with the following modifications and additions:

**Method identifiers** When mapping `wsdl:operation` names to Java method identifiers, the `get` or `set` prefix is not added. Instead the first word in the word-list has its first character converted to lower case.

**Parameter identifiers** When mapping `wsdl:part` names or wrapper child local names to Java method parameter identifiers, the first word in the word-list has its first character converted to lower case.

## 2.8.1 Name Collisions

WSDL name scoping rules may result in name collisions when mapping from WSDL 1.1 to Java. E.g. a port type and a service are both mapped to Java classes but WSDL allows both to be given the same name. This section defines rules for resolving such name collisions.

The order of precedence for name collision resolution is as follows (highest to lowest);

1. Service endpoint interface
2. Non-exception Java class
3. Exception class
4. Service class

If a name collision occurs between two identifiers with different precedences, the lower precedence item has its name changed as follows:

**Non-exception Java class** The suffix “`_Type`” is added to the class name.

**Exception class** The suffix “`_Exception`” is added to the class name.

**Service class** The suffix “`_Service`” is added to the class name.

If a name collision occurs between two identifiers with the same precedence this is reported as an error and requires developer intervention to correct, either by modifying the source WSDL or by specifying a customized name mapping.

If a name collision occurs between a mapped Java method and a method in `javax.xml.rpc.Stub` (the base class for a mapped SEI), the prefix “`_`” is added to the mapped method.

## Chapter 3

# Java to WSDL 1.1 Mapping

This chapter describes the mapping from Java to WSDL 1.1. This mapping is used when generating web service endpoints from existing Java interfaces.

*Conformance Requirement (WSDL 1.1 support):* Implementations **MUST** support mapping Java to WSDL 1.1.

The following sections describe the default mapping from each Java construct to the equivalent WSDL 1.1 artifact.

### 3.1 Java Names

#### 3.1.1 Name Collisions

**Editors Note 3.1** *This section will contain a list of items that may require name mangling to ensure uniqueness in the generated wsdl.*

WS-I Basic Profile 1.0[8] (see R2304) requires operations within a `wsdl:portType` to be uniquely named – customization of the operation name allows this requirement to be met when a Java SEI contains overloaded methods.

### 3.2 Package

A Java package is mapped to a `wsdl:definitions` element and associated `targetNamespace` attribute. The `wsdl:definitions` element acts as a container for other WSDL elements that together form the WSDL description of the constructs in the corresponding Java package. There is no standard mapping from a Java package name to the value of a WSDL `targetNamespace` attribute.

*Conformance Requirement (Package name mapping):* Implementations **MUST** provide a means for the user to specify the `targetNamespace` value when mapping a Java package to a WSDL `definitions` element.

No specific authoring style is required for the mapped WSDL document; implementations are free to generate WSDL that uses the WSDL and XML Schema import directives.

*Conformance Requirement (Use of WSDL and XML Schema import directives):* Generated WSDL **MUST** comply with the WS-I Basic Profile 1.0[8] restrictions (See R2001, R2002 and R2003) on usage of WSDL and XML Schema import directives.

### 3.3 Interface

A Java service endpoint interface (SEI) is mapped to a `wsdl:portType` element. The `wsdl:portType` element acts as a container for other WSDL elements that together form the WSDL description of the methods in the corresponding Java SEI. An SEI is a Java interface that meets all of the following criteria:

- It extends `java.rmi.Remote` either directly or indirectly
- All of its methods throw `java.rmi.RemoteException` in addition to any service specific exceptions.
- All method parameters and return types are compatible with the JAXB 2.0[10] Java to XML Schema mapping definition.
- No method parameter or return values types implement the `java.rmi.Remote` interface either directly or indirectly.
- It does not include constant declarations<sup>1</sup> (as `public final static`).

*Conformance Requirement (portType naming):* If not customized, the value of the `name` attribute of the `wsdl:portType` element MUST be the name of the service endpoint interface not including the package name.

Figure 3.1 shows an example of a Java SEI and the corresponding `wsdl:portType`.

#### 3.3.1 Inheritance

WSDL 1.1 does not define a standard representation for the inheritance of `wsdl:portType` elements. When mapping an SEI that inherits from another interface, the SEI is treated as if all methods of the inherited interface were defined within the SEI.

*Conformance Requirement (Inheritance flattening):* A mapped `wsdl:portType` element MUST contain WSDL definitions for all the methods of the corresponding Java SEI including all inherited methods.

*Conformance Requirement (Inherited interface mapping):* An implementation MAY map inherited interfaces to additional `wsdl:portType` elements within the `wsdl:definitions` element.

### 3.4 Method

Each public method in a Java SEI is mapped to a `wsdl:operation` element in the corresponding `wsdl:portType` plus one or more `wsdl:message` elements.

*Conformance Requirement (Operation naming):* The value of the `name` attribute of the `wsdl:operation` element SHOULD be the name of the Java method. A valid exception to this rule is when operations are named differently to ensure operation name uniqueness when an SEI contains overloaded methods.

Methods are either one-way or two-way: one way methods have an input but produce no output, two way methods have an input and produce an output. Section 3.4.1 describes one way operations further.

The `wsdl:operation` element corresponding to each method has one or more child elements as follows:

<sup>1</sup>WSDL 1.1 does not define any standard representation for constants in a `wsdl:portType` definition



- A `wsdl:input` element that refers to an associated `wsdl:message` element to describe the operation input. 1 2
- (Two-way methods only) an optional `wsdl:output` element that refers to a `wsdl:message` to describe the operation output. 3 4
- (Two-way methods only) zero or more `wsdl:fault` child elements, one for each exception in addition to the required `java.rmi.RemoteException` thrown by the method, that refer to associated `wsdl:message` elements to describe each fault. See section 3.6 for further details on exception mapping. 5 6 7 8

The value of a `wsdl:message` element's `name` attribute is not significant but by convention it is normally equal to the corresponding operation name for input messages and the operation name concatenated with "Response" for output messages. Naming of fault messages is described in section section 3.6. 9 10 11

Each `wsdl:message` element has a single `wsdl:part` child element that refers, via an `element` attribute, to a global element declaration in the `wsdl:types` section. Figure 3.1 shows an example of mapping a Java interface containing a single method to WSDL 1.1. 12 13 14

Section 3.5 describes the mapping from Java methods and their parameters to corresponding global element declarations in the `wsdl:types` section. 15 16

### 3.4.1 One Way Operations 17

Only Java methods whose return type is `void` and that do not throw any exceptions other than `java.rmi.RemoteException` can be mapped to one-way operations. Not all Java methods that fulfill this requirement are amenable to become one-way operations and automatic choice between two-way and one-way mapping is not possible. 18 19 20 21

*Conformance Requirement (One-way mapping):* Implementations MUST provide a facility for specifying which methods should be mapped to one-way operations. 22 23

*Conformance Requirement (One-way mapping errors):* Implementations MUST reject attempts to map methods that do not meet the necessary criteria to one-way operations. 24 25

## 3.5 Method Parameters 26

A Java method's parameters and return type are mapped to child elements of the global element declarations mapped from the method. Parameters can be mapped to child elements of the global element declaration for either the operation input message, operation output message or both. The mapping depends on the parameter classification. 27 28 29 30

### 3.5.1 Parameter Classification 31

Method parameters are classified as follows: 32

- in** The parameter value is transmitted by copy from a service client to the service endpoint implementation but is not returned from the service endpoint implementation to the client. In WSDL terms, the parameter is mapped to a child element of the global element declaration for the operation input message. 33 34 35 36

```
1  // Java
2  package com.example;
3  public interface StockQuoteProvider extends java.rmi.Remote {
4      float getPrice(String tickerSymbol)
5          throws java.rmi.RemoteException, TickerException;
6  }
7
8  <!-- WSDL extract -->
9  <types>
10     <xsd:schema targetNamespace="...">
11         <xsd:element name="getPrice">
12             <!-- element definition -->
13         </xsd:element>
14
15         <xsd:element name="getPriceResponse">
16             <!-- element definition -->
17         </xsd:element>
18
19         <xsd:element name="TickerException">
20             <!-- element definition -->
21         </xsd:element>
22     </xsd:schema>
23 </types>
24
25 <message name="getPrice">
26     <part name="getPrice" element="tns:getPrice"/>
27 </message>
28
29 <message name="getPriceResponse">
30     <part name="getPriceResponse" element="tns:getPriceResponse"/>
31 </message>
32
33 <message name="TickerException">
34     <part name="TickerException" element="tns:TickerException"/>
35 </message>
36
37 <portType name="StockQuoteProvider">
38     <operation name="getPrice" parameterOrder="tickerSymbol">
39         <input message="tns:getPrice"/>
40         <output message="tns:getPriceResponse"/>
41         <fault message="tns:TickerException"/>
42     </operation>
43 </portType>
```

Figure 3.1: Java interface to WSDL portType mapping

**out** The parameter value is returned by copy from a service endpoint implementation to the client but is not transmitted from the client to the service endpoint implementation. In WSDL terms, the parameter is mapped to a child element of the global element declaration for the operation output message.

**in/out** The parameter value is transmitted by copy from a service client to the service endpoint implementation and is returned by copy from the service endpoint implementation to the client. In WSDL terms, the parameter is mapped to a child element of both the global element declaration for the operation input message and the global element declaration for the operation output message

Holder classes are used to indicate out and in/out method parameters. A holder class is a class that implements the `javax.xml.rpc.holders.Holder` interface. An parameter whose type is a holder class is classified as in/out or out, all other parameters are classified as in.

*Conformance Requirement (Parameter classification):* Implementations **SHOULD** provide a facility for specifying whether a holder parameter is treated as in/out or out, if not specified, the default **MUST** be in/out.

The `javax.xml.rpc.holders.GenericHolder<>` class is provided as a convenient wrapper for any Java class.

### 3.5.2 Use of JAXB

JAXB defines a mapping from Java classes to XML Schema. JAX-RPC uses this mapping to generate XML Schema global element declarations that are referred to from within the WSDL message constructs generated for each operation.

For the purposes of utilizing the JAXB mapping, each method is represented as two Java bean classes: one that contains properties for each in and in/out parameter (henceforth called the request bean) and one that contains properties for the method return value and each out and in/out parameter (henceforth called the response bean).<sup>2</sup>

In the absence of customizations, the request bean class is named the same as the method and the bean response class is named the same as the method with a “Response” suffix. Return values are represented by an out property named “return”. Figure 3.2 illustrates this representation.

```

1  float getPrice(String tickerSymbol)
2
3  class getPrice {
4      String getTickerSymbol();
5  }
6
7  class getPriceResponse {
8      float getReturn();
9  }
```

Figure 3.2: Bean representation of an operation

The beans are generated with the appropriate JAXB customizations to result in a global element declaration for each bean class when mapped to XML Schema by JAXB. The element namespace is the value of the `targetNamespace` attribute of the WSDL definitions element.

<sup>2</sup>Actual generation of Java bean classes is not required, the beans are merely used to define the contractual interface between JAX-RPC and JAXB.

### 3.6 Service Specific Exception

A service specific Java exception is mapped to a `wsdl:fault` element, a `wsdl:message` element with a single child `wsdl:part` element and an XML Schema type or element declaration. The `wsdl:fault` element appears as a child of the `wsdl:operation` element that corresponds to the Java method that throws the exception. The `wsdl:part` element refers to an XML Schema global element declaration that describes the fault.

*Conformance Requirement (Exception naming):* The name of the global element declaration for a mapped exception SHOULD be the name of the Java exception. A valid exception to this rule is when name changes are required to prevent name collisions, see section 3.1.

JAXB defines the mapping from an exception's properties to XML Schema element declarations and type definitions.

### 3.7 Bindings

In WSDL 1.1, an abstract port type can be bound to multiple protocols.

*Conformance Requirement (Binding selection):* Implementations MUST provide a facility for specifying the binding(s) to use in generated WSDL.

Each protocol binding extends a common extensible skeleton structure and there is one instance of each such structure for each protocol binding. An example of a port type and associated binding skeleton structure is shown in figure 3.3.

```

1  <portType name="StockQuoteProvider">
2      <operation name="getPrice" parameterOrder="tickerSymbol">
3          <input message="tns:getPrice"/>
4          <output message="tns:getPriceResponse"/>
5          <fault message="tns:unknowntickerException"/>
6      </operation>
7  </portType>
8
9  <binding name="StockQuoteProviderBinding">
10     <!-- binding specific extensions possible here -->
11     <operation name="getPrice">
12         <!-- binding specific extensions possible here -->
13         <input message="tns:getPrice">
14             <!-- binding specific extensions possible here -->
15         </input>
16         <output message="tns:getPriceResponse">
17             <!-- binding specific extensions possible here -->
18         </output>
19         <fault message="tns:unknowntickerException">
20             <!-- binding specific extensions possible here -->
21         </fault>
22     </operation>
23 </binding>

```

Figure 3.3: WSDL portType and associated binding

The common skeleton structure is mapped from Java as described in the following subsections.

### 3.7.1 Interface

A Java service endpoint interface (SEI) is mapped to a `wsdl:binding` element. The `wsdl:binding` element acts as a container for other WSDL elements that together form the WSDL description of the binding to a protocol of the corresponding `wsdl:portType`.

The value of the `name` attribute of the `wsdl:binding` is not significant, by convention it is the qualified name of the corresponding `wsdl:portType` suffixed with “Binding”.

### 3.7.2 Method and Parameters

Each method in a Java SEI is mapped to a `wsdl:operation` child element of the corresponding `wsdl:binding`. The value of the `name` attribute of the `wsdl:operation` element is the same as the corresponding `wsdl:operation` element in the bound `wsdl:portType`. The `wsdl:operation` element has `wsdl:input`, `wsdl:output` and `wsdl:fault` child elements if they are present in the corresponding `wsdl:operation` child element of the `wsdl:portType` being bound.

## 3.8 SOAP HTTP Binding

This section describes the additional WSDL binding elements generated when mapping Java to WSDL 1.1 using the SOAP HTTP binding.

*Conformance Requirement (SOAP binding support):* Implementations MUST be able to generate SOAP HTTP bindings when mapping Java to WSDL 1.1.

Figure 3.4 shows an example of a SOAP HTTP binding.

```

1  <binding name="StockQuoteProviderBinding">
2    <soap:binding
3      transport="http://schemas.xmlsoap.org/soap/http" style="document" />
4    <operation name="getPrice">
5      <soap:operation style="document" />
6      <input message="tns:getPrice">
7        <soap:body use="literal" />
8      </input>
9      <output message="tns:getPriceResponse">
10       <soap:body use="literal" />
11     </output>
12     <fault message="tns:unknowntickerException">
13       <soap:fault use="literal" />
14     </fault>
15   </operation>
16 </binding>

```

Figure 3.4: WSDL SOAP HTTP binding

### 3.8.1 Interface

A Java service endpoint interface (SEI) is mapped to a `soap:binding` element. The `soap:binding` element is a child of the `wsdl:binding` element. The value of the `transport` attribute of the `soap:binding`

is `http://schemas.xmlsoap.org/soap/http`. The value of the `style` attribute of the `soap:binding` is `document`.

*Conformance Requirement (SOAP binding style required):* Implementations **MUST** include a `style` attribute on a generated `soap:binding`.

### 3.8.2 Method and Parameters

Each method in a Java SEI is mapped to a `soap:operation` child element of the corresponding `wsdl:operation`. The value of the `style` attribute of the `soap:operation` is `document`. If not specified, the value defaults to the value of the `style` attribute of the `soap:binding`. WS-I Basic Profile[8] requires that all operations within a given SOAP HTTP binding instance have the same binding style.

The parameters of a Java method are mapped to `soap:body` child elements of the `wsdl:input` and `wsdl:output` elements for each `wsdl:operation` binding element. The value of the `use` attribute of the `soap:body` is `literal`. Figure 3.5 shows an example.

```

1  <types>
2    <schema targetNamespace="...">
3      <xsd:element name="getPrice">
4        <xsd:complexType>
5          <xsd:sequence>
6            <xsd:element name="tickerSymbol" type="xsd:string"/>
7          </xsd:sequence>
8        </xsd:complexType>
9      </xsd:element>
10
11     <xsd:element name="getPriceResponse">
12       <xsd:complexType>
13         <xsd:sequence>
14           <xsd:element name="return" type="xsd:float"/>
15         </xsd:sequence>
16       </xsd:complexType>
17     </xsd:element>
18   </schema>
19 </types>
20
21 <message name="getPrice">
22   <part name="getPrice"
23     element="tns:getPrice"/>
24 </message>
25
26 <message name="getPriceResponse">
27   <part name="getPriceResponse" element="tns:getPriceResponse"/>
28 </message>
29
30 <portType name="StockQuoteProvider">
31   <operation name="getPrice" parameterOrder="tickerSymbol">
32     <input message="tns:getPrice"/>
33     <output message="tns:getPriceResponse"/>
34   </operation>
35 </portType>
36
37 <binding name="StockQuoteProviderBinding">
38   <soap:binding
39     transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
40   <operation name="getPrice" parameterOrder="tickerSymbol">
41     <soap:operation/>
42     <input message="tns:getPrice">
43       <soap:body use="literal"/>
44     </input>
45     <output message="tns:getPriceResponse">
46       <soap:body use="literal"/>
47     </output>
48   </operation>
49 </binding>

```

Figure 3.5: WSDL definition using document style





# Chapter 4

## Client APIs

This chapter describes the standard APIs provided for client side use of JAX-RPC. These APIs allow a client to configure generated stubs, create dynamic proxies for remote service endpoints and dynamically construct operation invocations.

### 4.1 javax.xml.rpc.ServiceFactory

`ServiceFactory` is an abstract factory class that provides various methods for the creation of `Service` instances (see section 4.2 for details of the `Service` interface).

*Conformance Requirement (Concrete `ServiceFactory` required):* A client side JAX-RPC runtime system, henceforth in this chapter simply called ‘an implementation’, must provide a concrete class that extends `ServiceFactory`.

#### 4.1.1 Configuration

**Editors Note 4.1** *The JAX-RPC 1.1 `ServiceFactory` provides no API to configure factory properties. These would be useful, e.g. in DII without WSDL to set the style of operations. Should we add a generic property capability ?*

The `ServiceFactory` implementation class is set using the system property named `javax.xml.rpc.ServiceFactory` (the constant: `ServiceFactory.SERVICEFACTORY_PROPERTY`).

*Conformance Requirement (Service class loading):* An implementation MUST provide facilities to enable the `ServiceFactory.loadService(Class)` method to succeed provided all the generated artifacts are packaged with the application.

An implementation can either use a consistent naming convention for generated service implementation classes or allow an application developer to specify sufficient configuration information to locate `Service` implementation classes. Examples of such configuration information include:

- System properties
- Properties or XML-based configuration files that are looked up as resources via the `getResource` or `getResources` methods of `java.lang.ClassLoader`
- User and system preference and configuration data retrieved via the `java.util.prefs` facilities.

### 4.1.2 Factory Usage

A J2SE service client should use `ServiceFactory` to create `Service` instances. Alternatively, a J2SE based service client may use the JNDI naming context to lookup a service instance.

A J2EE-based service client should not use `ServiceFactory` to create `Service` instances. Instead, J2EE-based service clients should use JNDI to lookup an instance of a `Service` class as specified in JSR-109[14]. Moreover, packaging implementation-specific artifacts (including classes and configuration information) with a J2EE application is strongly discouraged as this makes the application non-portable.

### 4.1.3 Example

The following shows a typical use of `ServiceFactory` to create a `Service` instance.

```
1 ServiceFactory sf = ServiceFactory.newInstance();
2 Service s = sf.createService(wsdlDoc, serviceName);
```

## 4.2 javax.xml.rpc.Service

`Service` is an abstraction that represents a WSDL service. A WSDL service is a collection of related ports, each of which consists of a port type bound to a particular protocol and available at particular endpoint address.

`Service` instances are created using methods on a `ServiceFactory` instance or are obtained via JNDI. `Service` instances provide facilities to:

- Create an instance of a generated stub via one of the `getPort` methods. See section 4.6 for information on stubs.
- Create a dynamic proxy via one of the `getPort` methods. See section 4.6.2 for information on dynamic proxies.
- Create a `Dispatch` instance via the `createDispatch` method. See section 4.7 for information on the `Dispatch` interface.
- Create a `Call` instance via one of the `createCall` methods. See section 4.8 for information on the `Call` interface.
- Create a new port via the `createPort` method. Such ports only include binding and endpoint information and are thus only suitable for creating `Dispatch` or `Call` instances which do not require WSDL port type information.
- Configure per-service, per-port and per-protocol message handlers, see section 4.2.1.

*Conformance Requirement (Service completeness):* A `Service` implementation must be capable of creating dynamic proxies, `Dispatch` instances, `Call` instances and new ports.

`Service` implementations can either implement `javax.xml.rpc.Service` directly or can implement a generated service interface (see section 2.7) that extends `javax.xml.rpc.Service`.

*Conformance Requirement (Service capabilities):* A `Service` implementation SHOULD implement the `java.io.Serializable` and `javax.naming.Referenceable` interfaces to support registration in JNDI.

### 4.2.1 Handler Registry

JAX-RPC provides a flexible plug-in framework for message processing modules, known as handlers, that may be used to extend the capabilities of a JAX-RPC runtime system. Chapter ?? describes the handler framework in detail. A `Service` instance provides access to a `HandlerRegistry`, via the `getHandlerRegistry` method, that may be used to configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a `Service` instance is used to create an instance of a generated stub, a dynamic proxy, or a `Dispatch` or `Call` instance then the created instance is configured with a snapshot of the applicable handlers configured on the `Service` instance. Subsequent changes to the handlers configured for a `Service` instance do not affect the handlers on previously created stubs, dynamic proxies, `Dispatch` or `Call` instances.

Handler execution order for outbound messages is:

- (i) Per-service handlers,
- (ii) Per-port handlers,
- (iii) Per-protocol binding handlers.

Handler execution order is reversed for inbound messages.

### 4.2.2 Type Mapping Registry

JAX-RPC 1.1 provided a registry accessible via `Service.getTypeMappingRegistry` that can be used to register non-portable serializers and deserializers for Java types. In JAX-RPC 2.0, all data binding is delegated to JAXB 2.0[10] and JAXB provides its own different mechanisms for customizing the mapping between XML and Java data types. The type mapping registry is not used when JAXB is used for data binding but is retained for backwards compatibility with JAX-RPC 1.1 when alternate data binding methods are used, e.g. in implementations that continue to support the SOAP encoding.

## 4.3 *javax.xml.rpc.JAXRPCContext*

Additional metadata is often required to control and annotate information exchanges. This metadata forms the context of an exchange and the `JAXRPCContext` interface provides programmatic access to this metadata.

Clients obtain a `JAXRPCContext` instance from the methods of `javax.xml.rpc.BindingProvider`. `javax.xml.rpc.Stub` and `javax.xml.rpc.Dispatch` extend `BindingProvider` to provide contextual information to clients.

There are separate contexts for the request and response phases of an operation invocation. The request context may be manipulated by a client prior to an operation invocation. The response context is created by a protocol binding and made available to the client when an operation completes. Protocol bindings are responsible for annotating outbound protocol data units when the metadata from the request context is required to be transferred with a request. Protocol bindings are responsible for extracting metadata from inbound protocol data units when that data is required to be available to clients in the response context. E.g. a handler in a SOAP binding might introduce a header into a SOAP request message to carry metadata from

the request `JAXRPCContext` and might add metadata to the response `JAXRPCContext` from the contents of a header in a response SOAP message.

A context takes the form of a set of named properties. JAX-RPC defines a set of standard properties and implementations can add additional properties.

### 4.3.1 Standard Properties

Table 4.1 lists a set of standard properties that may be set on a `JAXRPCContext` instance and shows which properties are optional for implementations to support.

Name	Type	Mandatory	Description
<b>javax.xml.rpc.service.endpoint</b>			
.address	String	Y	The address of the service endpoint as a protocol specific URI. The URI scheme must match the protocol binding in use.
<b>javax.xml.rpc.binding</b>			
.context	JAXBContext	Y	A <code>JAXBContext</code> instance that may be used to perform marshaling of outbound messages and unmarshaling of inbound messages.
<b>javax.xml.rpc.security</b>			
.username	String	Y	Username for HTTP basic authentication.
.password	String	Y	Password for HTTP basic authentication.
<b>javax.xml.rpc.session</b>			
.maintain	Boolean	Y	Used by a client to indicate whether it is prepared to participate in a service endpoint initiated session. The default value is <code>false</code> .
<b>javax.xml.rpc.soap.operation</b>			
.style	String	N	The style of the operation: either <code>rpc</code> or <code>document</code> .
<b>javax.xml.rpc.soap.http.soapaction</b>			
.use	Boolean	N	Controls whether the <code>SOAPAction</code> HTTP header is used in SOAP/HTTP requests. Default value is <code>false</code> .
.uri	String	N	The value of the <code>SOAPAction</code> HTTP header if the <code>javax.xml.rpc.soap.http.soapaction.use</code> property is set to <code>true</code> . Default value is an empty string.
<b>javax.xml.rpc.encodingstyle.namespace</b>			
.uri	String	N	The encoding style specified as a URI. Default value is the URI corresponding to SOAP encoding: <code>http://schemas.xmlsoap.org/soap/encoding/</code> .

Table 4.1: Standard `JAXRPCContext` properties.

*Conformance Requirement (Required `JAXRPCContext` properties):* An implementation **MUST** support the properties shown as mandatory in table 4.1.

Note that properties shown as mandatory are not required to be present in any particular context, however if present they must be honored.

*Conformance Requirement (Optional JAXRPCContext properties):* An implementation MAY support the properties shown as optional in table 4.1.

### 4.3.2 Additional Properties

*Conformance Requirement (Additional JAXRPCContext properties):* An implementation MAY support additional implementation specific properties not listed in table 4.1. Such properties MUST NOT use the javax.xml.rpc prefix in their names.

Implementation specific properties are discouraged as they limit application portability. Applications and binding handlers can interact using application specific properties.

## 4.4 javax.xml.rpc.Binding

The javax.xml.rpc.Binding interface acts as a marker interface for JAX-RPC protocol bindings. Bindings to specific protocols extend Binding and add methods to configure aspects of that protocol binding's operation. Chapter ?? describes the JAX-RPC SOAP binding.

Clients obtain a Binding instance from a stub, dynamic proxy or Dispatch instance using the getBinding method of the javax.xml.rpc.BindingProvider interface that is extended by javax.xml.rpc.Stub and javax.xml.rpc.Dispatch.

## 4.5 javax.xml.rpc.BindingProvider

The javax.xml.rpc.BindingProvider interface is the superinterface of javax.xml.rpc.Stub and javax.xml.rpc.Dispatch. It represents a component that provides a protocol binding for use by clients. The BindingProvider interface provides methods to obtain the Binding, request context and response context.

## 4.6 javax.xml.rpc.Stub

WSDL to Java mapping implementations generate strongly typed Java interfaces for services described in WSDL, see chapter 2. Implementations also allow generation of client side stub classes and server side tie classes that implement the mapping between Java and the protocol binding described in the WSDL. Generated stub classes implement the Java interface generated from the WSDL and also implement the Stub interface.

*Conformance Requirement (Implementing Stub):* Client-side generated stubs MUST implement the javax.xml.rpc.Stub interface.

*Conformance Requirement (Stub class binding):* A generated stub class SHOULD be bound to a particular protocol and transport (if applicable).

Generated stub classes are either named after the protocol binding in the WSDL (*BindingName\_Stub* where *BindingName* is the value of the name attribute of the corresponding WSDL binding element) or using some other implementation specific naming scheme.

## 4.6.1 Configuration

The `Service` interface provides two methods for obtaining instances of generated stub classes or dynamic proxies:

**`getPort(Class sei)`** Returns an instance of a generated stub class or dynamic proxy, the `Service` instance is responsible for selecting the port (protocol binding and endpoint address).

**`getPort(QName port, Class sei)`** Returns an instance of a generated stub class or dynamic proxy for the endpoint specified by `port`. Note that the namespace component of `port` is the target namespace of the WSDL definitions document,

Both methods throw `javax.xml.rpc.ServiceException` on failure. Generated service interfaces (see section 2.7) contain additional methods for acquiring instances of generated stub classes or dynamic proxies equivalent to the second `getPort` method above.

Stub classes are not required to be dynamically configurable for different protocols bindings but may support configuration of certain aspects of their operation. Stub classes support two forms of configuration:

**Static** The WSDL binding from which the stub class was generated contains static information including the protocol binding and service endpoint address.

**Dynamic** A Stub instance provides methods (inherited from `javax.xml.rpc.BindingProvider`) to dynamically query and change the values of properties in its request and response contexts. Section 4.3 describes the format of request and response contexts.

*Conformance Requirement (Stub configuration):* An implementation **MUST** throw a `JAXRPCException` if a client attempts to set an unsupported optional property or if an implementation detects an error in the value of a property.

The `JAXRPCContext` (see section 4.3) interface provides a common container interface for metadata that may be used with `Stub` and `Dispatch` (see section 4.7) instances. For backwards compatibility with JAX-RPC 1.1, the `Stub` property methods are retained. Setting the value of a property in the request `JAXRPCContext` of a `Stub` instance is equivalent to setting the value of the property directly on the `Stub` instance. E.g. in the following code fragment, line 7 would print `true`.

```

1  javax.xml.rpc.Service service = ...;
2  javax.xml.rpc.Stub stub = service.getPort(portName,
3      com.example.StockQuoteProvider.class);
4  stub._setProperty("javax.xml.rpc.session.maintain", new Boolean(false));
5  javax.xml.rpc.JAXRPCContext context = stub.getRequestContext();
6  context.setProperty("javax.xml.rpc.session.maintain", new Boolean(true));
7  System.out.println(stub._getProperty("javax.xml.rpc.session.maintain"));

```

## 4.6.2 Dynamic Proxy

In addition to statically generated stub classes, JAX-RPC also provides dynamic proxy generation support. Dynamic proxies provide access to service endpoint interfaces at runtime without requiring static generation of a stub class. See `java.lang.reflect.Proxy` for more information on dynamic proxies.

*Conformance Requirement (Dynamic proxy support):* An implementation MUST support generation of dynamic proxies.

*Conformance Requirement (Implementing Stub required):* Dynamic proxy instances MUST implement the `javax.xml.rpc.Stub` interface.

A dynamic proxy is created using the `getPort` method of a `Service` instance. The `serviceEndpointInterface` parameter specifies the interface that will be implemented by the generated proxy. The service endpoint interface provided by the client needs to conform to the WSDL to Java mapping rules specified in chapter 2 (WSDL 1.1). Generation of a dynamic proxy can fail if the interface doesn't conform to the mapping or if any WSDL related metadata is missing from the `Service` instance.

*Conformance Requirement (Failed `Service.getPort`):* An implementation MUST throw a `javax.xml.rpc.ServiceException` if generation of a dynamic proxy fails.

An implementation is not required to fully validate the service endpoint interface provided by the client against the corresponding WSDL definitions and may choose to implement any validation it does require in an implementation specific manner (e.g. lazy and eager validation are both acceptable).

## Example

The following example shows the use of a dynamic proxy to invoke a method (`getLastTradePrice`) on a service endpoint interface (`com.example.StockQuoteProvider`). Note that no statically generated stub class is involved.

```
1  javax.xml.rpc.Service service = ...;
2  java.rmi.Remote proxy = service.getPort(portName,
3      com.example.StockQuoteProvider.class)
4  javax.xml.rpc.Stub stub = (javax.xml.rpc.Stub)proxy;
5  javax.xml.rpc.JAXRPCContext context = stub.getRequestContext();
6  context.setProperty("javax.xml.rpc.session.maintain", new Boolean(true));
7  com.example.StockQuoteProvider sqp = (com.example.StockQuoteProvider)proxy;
8  sqp.getLastTradePrice("ACME");
```

Lines 1–3 show how the dynamic proxy is generated. Lines 4–6 cast the proxy to a `Stub` and perform some dynamic configuration of the stub. Lines 7–8 cast the the proxy to the service endpoint interface and invoke the method.

## 4.7 javax.xml.rpc.Dispatch

**Editors Note 4.2** *Dispatch is a placeholder, other alternatives considered include `Endpoint`, `Post`, `Transmit` and `Send`. Alternative suggestions welcome.*

XML based RPC represents RPC invocations and any associated responses as XML messages. The higher level JAX-RPC APIs are designed to hide the details of converting between Java objects and their XML representations but in some cases operating at the XML representation level is desirable. The `Dispatch` interface provides support for this mode of interaction.

*Conformance Requirement (Dispatch support):* Implementations MUST support the `javax.xml.rpc.Dispatch` interface.

Dispatch is a low level API that requires clients to construct XML based RPC message payloads as XML fragments and requires an intimate knowledge of the desired payload structure. For convenience the Dispatch API also provides a means for direct use of JAXB objects. This allows clients to use JAXB objects generated from an XML Schema to create and manipulate XML representations and to use these objects with JAX-RPC without requiring an intermediate XML serialization.

### 4.7.1 Configuration

Dispatch instances are obtained using the `createDispatch` method of a `Service` instance. Dispatch instances are not thread safe, use of the same instance in multiple threads requires considerable care.

Dispatch instances are not required to be dynamically configurable for different protocol bindings but may support configuration of certain aspects of their operation. Dispatch instances support two forms of configuration:

**Static** The WSDL binding from which the Dispatch instance was generated contains static information including the protocol binding and service endpoint address.

**Dynamic** A Dispatch instance provides methods (inherited from `javax.xml.rpc.BindingProvider`) to dynamically query and change the values of properties in its request and response contexts. Section 4.3 describes the format of request and response contexts.

*Conformance Requirement (Dispatch configuration failure):* An implementation **MUST** throw a `JAXRPCException` if a client attempts to set an unsupported optional property or if an implementation detects an error in the value of a property.

### 4.7.2 Operation Invocation

A Dispatch instance supports three invocation modes:

**Synchronous request response (`invoke` methods)** The operation invocation blocks until the remote operation completes and the results are returned.

**Asynchronous request response (`invokeAsync` methods)** The operation invocation returns immediately, any results are provided either through a callback or via a polling object.

**One-way (`invokeOneWay` methods)** The operation invocation is logically non-blocking, subject to the capabilities of the underlying protocol, no results are returned.

*Conformance Requirement (Failed Dispatch.invoke):* When an operation is invoked using an `invoke` method, an implementation **MUST** throw a `JAXRPCException` if there is any error in the configuration of the Dispatch instance or a `java.rmi.RemoteException` if an error occurs during the remote operation invocation.

*Conformance Requirement (Failed Dispatch.invokeAsync):* When an operation is invoked using an `invokeAsync` method, an implementation **MUST** throw a `JAXRPCException` if there is any error in the configuration of the Dispatch instance. Errors that occur during the invocation are reported when the client attempts to retrieve the results of the operation.



*Conformance Requirement (Failed Dispatch.invokeOneWay):* When an operation is invoked using an `invokeOneWay` method, an implementation **MUST** throw a `JAXRPCException` if there is any error in the configuration of the `Dispatch` instance or if an error is detected<sup>1</sup> during the remote operation invocation.

*Conformance Requirement (One-way operations):* When invoking one-way operations where the binding is SOAP/HTTP an implementation **MUST** block until the HTTP response is received or an error occurs. Completion of the HTTP request simply means that the transmission of the request is complete, not that the request was accepted or processed.

### 4.7.3 Operation Response

The following interfaces are used to obtain the results of an operation invocation:

**javax.xml.rpc.Response** A generic interface that is used to group the results of an invocation with the response context. `Response` extends `java.util.concurrent.Future<T>` to provide asynchronous result polling capabilities.

**javax.xml.rpc.AsyncHandler** A generic interface that clients implement to receive results in an asynchronous callback.

### 4.7.4 Asynchronous Response

`Dispatch` supports two forms of asynchronous invocation:

**Polling** The `invokeAsync` method returns a `Response` that may be polled using the methods inherited from `Future<T>` to determine when the operation has completed and to retrieve the results.

**Callback** The client supplies an `AsyncHandler` and the runtime calls the `handleResponse` method when the results of the operation are available. The `invokeAsync` method returns a wildcard `Future` (`Future<?>`) that may be polled to determine when the operation has completed. The object returned from `Future<?>.get()` has no standard type. Client code should not attempt to cast the object to any particular type as this will result in non-portable behaviour.

In both cases, errors that occur during the invocation are reported via an exception when the client attempts to retrieve the results of the operation.

*Conformance Requirement (Reporting asynchronous errors):* An implementation **MUST** throw a `JAXRPCException` from `Response.get` if the operation invocation failed.

### 4.7.5 Using JAXB

All of the `Dispatch` invoke method variants permit use with either XML fragments (using `javax.xml.transform.Source` instances) or JAXB objects. When using `Dispatch` with JAXB objects, the request context must contain a `JAXBContext` as the value of the standard `javax.xml.rpc.binding.context` property, see section 4.3.1. The supplied `JAXBContext` is used to marshall the request object to XML and unmarshall any response.

<sup>1</sup>The invocation is logically non-blocking so detection of errors during operation invocation is dependent on the underlying protocol in use. For SOAP/HTTP it is possible that certain HTTP level errors may be detected.

*Conformance Requirement (Marshalling failure):* If an error occurs when using the supplied `JAXBContext` to marshal a request or unmarshal a response, an implementation **MUST** throw a `JAXRPCException` whose cause is set to the original `JAXBException`.

## 4.7.6 Examples

The following examples demonstrate use of `Dispatch` methods in the synchronous, asynchronous polling and asynchronous callback modes. For ease of reading, error handling has been omitted.

### Synchronous

```

1 Source reqMsg = ...;
2 javax.xml.rpc.Service service = ...;
3 javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
4 JAXRPCContext reqCtx = disp.getRequestContext();
5 reqCtx.setProperty(...);
6 Source resMsg = (Source)disp.invoke(reqMsg);
7 JAXRPCContext resCtx = disp.getResponseContext();

```

### Synchronous With JAXB Objects

```

1 JAXBContext jc = JAXBContext.newInstance( "primer.po" );
2 Unmarshaller u = jc.createUnmarshaller();
3 PurchaseOrder po = (PurchaseOrder)u.unmarshal(
4     new FileInputStream( "po.xml" ) );
5 javax.xml.rpc.Service service = ...;
6 javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
7 JAXRPCContext reqCtx = disp.getRequestContext();
8 reqCtx.setProperty("javax.xml.rpc.binding.context", jc);
9 OrderConfirmation conf = (OrderConfirmation)disp.invoke(po);
10 JAXRPCContext resCtx = disp.getResponseContext();

```

In the above example `PurchaseOrder` and `OrderConfirmation` are interfaces pre-generated by JAXB from the schema document ‘primer.po’.

### Asynchronous Polling

```

1 Source reqMsg = ...;
2 javax.xml.rpc.Service service = ...;
3 javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
4 JAXRPCContext reqCtx = disp.getRequestContext();
5 reqCtx.setProperty(...);
6 Response<Object> res = disp.invokeAsync(reqMsg);
7 while (!res.isDone()) {
8     // do something while we wait
9 }
10 Source resMsg = (Source)res.get();
11 JAXRPCContext resCtx = res.getContext();

```

## Asynchronous Callback

```

1  class MyHandler implements AsyncHandler<Object> {
2      ...
3      public void handleResponse(Response<Object> res) {
4          Source resMsg = (Source)res.get();
5          JAXRPCContext resCtx = res.getContext();
6          // do something with the results
7      }
8  }
9
10 Source reqMsg = ...;
11 javax.xml.rpc.Service service = ...;
12 javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
13 JAXRPCContext reqCtx = disp.getRequestContext();
14 reqCtx.setProperty(...);
15 MyHandler handler = new MyHandler();
16 (void)disp.invokeAsync(reqMsg, handler);

```

## 4.8 javax.xml.rpc.Call

The `Call` interface provides support for dynamically constructing operation invocations either with or without a WSDL description of the service endpoint. `Call` was originally designed for use with RPC style operation invocations using SOAP encoding. It is difficult to use with the now more popular RPC and document style operation invocations using literal XML. Use of the `Dispatch` interface (see section 4.7) is strongly recommended for new client applications, the `Call` interface is only retained for backwards compatibility with earlier version of JAX-RPC.

*Conformance Requirement (Call support):* Implementations **MUST** support the `javax.xml.rpc.Call` interface.

### 4.8.1 Configuration

`Call` instances are obtained using one of the `createCall` methods of a `Service` instance. A `Call` instance can be created in one of two states depending on the combination of how the `Service` instance was obtained and which `createCall` variant was used:

**Unconfigured** The client is responsible for configuring `Call` instance prior to use

**Configured** The `Call` instance is ready for use

Table 4.2 shows the state for each combination. Combinations shown as U/C may result in either unconfigured or configured `Call` instances, the result is implementation dependent.

Unconfigured `Call` instances require configuration using the appropriate setter methods prior to use of the `invoke` and `invokeOneWay` methods. `Call` instances are mutable, a client may change the configuration of an existing instance and re-use it.

*Conformance Requirement (Unconfigured Call):* An implementation **MUST** throw a `JAXRPCException` if the `invoke` or `invokeOneWay` methods are called on an unconfigured instance.

		Service.createCall Arguments		
	None	QName port	QName port, QName op	QName port, String op
<b>createService Arguments</b>				
QName svc	U	U	U	U
URL wsdlDoc, QName svc	U	U	C	C
<b>loadService Arguments</b>				
Class si	U	U	U/C	U/C
URL wsdlDoc, Class si, Properties p	U	U	U/C	U/C
URL wsdlDoc, QName svc, Properties p	U	U	U/C	U/C

Table 4.2: Call states resulting from combinations of Service creation and createCall variants.

Setter methods are provided to configure:

- The name of the operation to invoke
- The names, types and modes of the operation parameters
- The operation return type
- The name of the port type
- The endpoint address
- Additional properties (see section 4.3.1 on page 36 for a list of standard properties).

**Conformance Requirement (Call configuration):** An implementation **MUST** throw a `JAXRPCException` if a client attempts to set an unknown or unsupported optional property or if an implementation detects an error in the value of a property.

A client can determine the level of configuration a `Call` instance requires by use of the `isParameterAndReturnSpecRequired` method. This method returns `false` for operations that only require the operation name to be configured, `true` for operations that require operation name, parameter and return types to be configured.

## 4.8.2 Operation Invocation

When an operation is invoked, the `Call` instance checks that the passed parameters match the number, order and types of parameters configured in the instance.

**Conformance Requirement (Misconfigured invocation):** When an operation is invoked, an implementation **MUST** throw a `JAXRPCException` if the `Call` instance is incorrectly configured or if the passed parameters do not match the configuration.

A `Call` instance supports two invocation modes:

**Synchronous request response (invoke method)** The operation invocation blocks until the remote operation completes and the results are returned.

**One-way (`invokeOneWay` method)** The operation invocation is logically non-blocking, subject to the capabilities of the underlying protocol, no results are returned.

*Conformance Requirement (Failed `invoke`):* When an operation is invoked using the `invoke` method, an implementation **MUST** throw a `java.rmi.RemoteException` if an error occurs during the remote invocation.

*Conformance Requirement (Failed `invokeOneWay`):* When an operation is invoked using the `invokeOneWay` method, an implementation **MUST** throw a `JAXRPCException` if an error occurs during the remote invocation.

*Conformance Requirement (One-way operations):* When invoking one-way operations where the binding is SOAP/HTTP, an implementation **MUST** block until the HTTP response is received or an error occurs. Completion of the HTTP request simply means that the transmission of the request is complete, not that the request was accepted or processed.

Once an operation has been invoked the values of the operation's output parameters may be obtained using the following methods:

**`getOutputParams`** The returned `Map` contains the output parameter values keyed by name. The type of the key is `String` and the type of the value depends on the mapping between XML and Java types.

**`getOutputValues`** The returned `List` contains the values of the output parameters in the order specified for the operation. The type of the value depends on the mapping between XML and Java types.

*Conformance Requirement (Missing invocation):* An implementation **MUST** throw `JAXRPCException` if `getOutputParams` or `getOutputValues` is called prior to `invoke` or following `invokeOneWay`.

### 4.8.3 Example

As described in section 4.8.1, a `Call` instance can be either configured or unconfigured. Use of a configured instance is simpler since the `Call` instance takes the responsibility of determining the corresponding types for the parameters and return value. Figure 4.1 shows an example of using a configured `Call` instance

```
1  javax.xml.rpc.Service service = ...
2  javax.xml.rpc.Call call = service.createCall( portName, operationName);
3  Object[] inParams = new Object[] { "hello world!" };
4  Integer ret = (Integer) call.invoke(inParams);
5  Map outParams = call.getOutputParams();
6  String outValue = (String)outParams.get( "param2" );
```

Figure 4.1: Use of configured `Call` instance

Alternatively, the `Call` instance can be unconfigured or only partially configured. In this case the client is responsible for configuring the call prior to use. Figure 4.2 shows an example of using an unconfigured `Call` instance to invoke the same operation as shown in figure 4.1.

Lines 3–5 in figure 4.2 are concerned with configuring the `Call` instance prior to its use in line 7.

```

1  javax.xml.rpc.Service service = ...
2  javax.xml.rpc.Call call = service.createCall( portName, operationName);
3  call.addParameter("param1", <xsd:string>, ParameterMode.IN);
4  call.addParameter("param2", <xsd:string>, ParameterMode.OUT);
5  call.setReturnType(<xsd:int>);
6  Object[] inParams = new Object[] {"hello world!"};
7  Integer ret = (Integer) call.invoke(inParams);
8  Map outParams = call.getOutputParams();
9  String outValue = (String)outParams.get("param2");

```

Figure 4.2: Use of unconfigured Call instance

## 4.9 Exceptions

The following standard exceptions are defined by JAX-RPC.

**javax.xml.rpc.ServiceException** A checked exception that is thrown by methods in the `ServiceFactory` and `Service` interfaces.

**javax.xml.rpc.JAXRPCException** A runtime exception that is thrown by methods in service client APIs when errors occur during local processing. `java.rmi.RemoteException` is thrown when errors occurs during processing of the remote method invocation.

**javax.xml.rpc.ProtocolException** A base class for exceptions related to a specific protocol binding. Subclasses are used to communicate protocol level fault information to clients and may be used on the server to control the protocol specific fault representation.

**javax.xml.rpc.soap.SOAPFaultException** A subclass of `ProtocolException`, used to carry SOAP 1.1 specific information.

### 4.9.1 Protocol Specific Exception Handling

*Conformance Requirement (Protocol specific fault generation):* When throwing an exception as the result of a protocol level fault, an implement **MUST** set the cause of that exception to be an instance of the appropriate `ProtocolException` subclass. For SOAP 1.1 the appropriate `ProtocolException` subclass is `SOAPFaultException`.

*Conformance Requirement (Protocol specific fault consumption):* When an implementation catches an exception thrown by a service endpoint implementation and the cause of that exception is an instance of the appropriate `ProtocolException` subclass for the protocol in use then an implementation **MUST** reflect the information contained in the `ProtocolException` subclass within the generated protocol level fault.

#### Client Side Example

```

1  try {
2      response = dispatch.invoke(request);
3  }
4  catch (RemoteException e) {
5      if (e.getCause() != null && e.getCause() instanceof SOAPFaultException) {
6          SOAPFaultException soapFault = (SOAPFaultException)e.getCause();

```

```

7         QName soap11faultcode = soapFault.getFaultCode();      1
8         ...                                                         2
9     }                                                             3
10 }                                                                 4

```

### Server Side Example

```

1  public void endpointOperation() throws RemoteException {          6
2      ...                                                         7
3      if (someProblem) {                                          8
4          SOAPFaultException soapFault = new SOAPFaultException(faultcode, faultstring,
5              faultactor, details);                                10
6          throw new RemoteException("An error occurred", pde);    11
7      }                                                           12
8      ...                                                         13
9  }                                                                 14

```

## 4.10 Additional Classes

The following additional classes are defined by JAX-RPC:

```

javax.xml.rpc.NamespaceConstants Contains constants for common XML namespace prefixes and
    URIs.
javax.xml.rpc.encoding.XMLType Contains constants for the QNames of the supported set of XML
    schema types and SOAP encoding types.

```





## Conformance Requirements 2

2.1	WSDL 1.1 support . . . . .	9	3
2.2	Definitions mapping . . . . .	9	4
2.3	Support for WSDL and XML Schema import directives . . . . .	9	5
2.4	WSDL extension support . . . . .	10	6
2.5	SEI naming . . . . .	10	7
2.6	Extending <code>java.rmi.Remote</code> . . . . .	10	8
2.7	Method naming . . . . .	10	9
2.8	<code>RemoteException</code> required . . . . .	10	10
2.9	Transmission primitive support . . . . .	10	11
2.10	Non-wrapped parameter naming . . . . .	11	12
2.11	Default mapping mode . . . . .	12	13
2.12	Disabling wrapper style . . . . .	12	14
2.13	Wrapped parameter naming . . . . .	12	15
2.14	Use of <code>GenericHolder</code> . . . . .	14	16
2.15	Asynchronous mapping required . . . . .	15	17
2.16	Asynchronous mapping option . . . . .	15	18
2.17	Asynchronous method naming . . . . .	15	19
2.18	Failed method invocation . . . . .	15	20
2.19	Response bean naming . . . . .	16	21
2.20	JAXB Class Mapping . . . . .	18	22
2.21	Exception naming . . . . .	18	23
2.22	Unbound message parts . . . . .	20	24
2.23	Mapping additional header parts . . . . .	20	25
2.24	Service interface required . . . . .	21	26

2.25	Failed <code>getPortName</code> . . . . .	21	1
3.1	WSDL 1.1 support . . . . .	23	2
3.2	Package name mapping . . . . .	23	3
3.3	Use of WSDL and XML Schema import directives . . . . .	23	4
3.4	<code>portType</code> naming . . . . .	24	5
3.5	Inheritance flattening . . . . .	24	6
3.6	Inherited interface mapping . . . . .	24	7
3.7	Operation naming . . . . .	24	8
3.8	One-way mapping . . . . .	25	9
3.9	One-way mapping errors . . . . .	25	10
3.10	Parameter classification . . . . .	27	11
3.11	Exception naming . . . . .	28	12
3.12	Binding selection . . . . .	28	13
3.13	SOAP binding support . . . . .	29	14
3.14	SOAP binding style required . . . . .	30	15
4.1	Concrete <code>ServiceFactory</code> required . . . . .	33	16
4.2	Service class loading . . . . .	33	17
4.3	Service completeness . . . . .	34	18
4.4	Service capabilities . . . . .	34	19
4.5	Required <code>JAXRPCContext</code> properties . . . . .	36	20
4.6	Optional <code>JAXRPCContext</code> properties . . . . .	36	21
4.7	Additional <code>JAXRPCContext</code> properties . . . . .	37	22
4.8	Implementing <code>Stub</code> . . . . .	37	23
4.9	<code>Stub</code> class binding . . . . .	37	24
4.10	<code>Stub</code> configuration . . . . .	38	25
4.11	Dynamic proxy support . . . . .	39	26
4.12	Implementing <code>Stub</code> required . . . . .	39	27
4.13	Failed <code>Service.getPort</code> . . . . .	39	28
4.14	Dispatch support . . . . .	39	29
4.15	Dispatch configuration failure . . . . .	40	30
4.16	Failed <code>Dispatch.invoke</code> . . . . .	40	31
4.17	Failed <code>Dispatch.invokeAsync</code> . . . . .	40	32
4.18	Failed <code>Dispatch.invokeOneWay</code> . . . . .	41	33
4.19	One-way operations . . . . .	41	34

---

4.20	Reporting asynchronous errors . . . . .	41	1
4.21	Marshalling failure . . . . .	42	2
4.22	Call support . . . . .	43	3
4.23	Unconfigured Call . . . . .	43	4
4.24	Call configuration . . . . .	44	5
4.25	Misconfigured invocation . . . . .	44	6
4.26	Failed invoke . . . . .	45	7
4.27	Failed invokeOneWay . . . . .	45	8
4.28	One-way operations . . . . .	45	9
4.29	Missing invocation . . . . .	45	10
4.30	Protocol specific fault generation . . . . .	46	11
4.31	Protocol specific fault consumption . . . . .	46	12



# Bibliography

- [1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Recommendation, W3C, October 2000. See <http://www.w3.org/TR/2000/REC-xml-20001006>.
- [2] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. Note, W3C, May 2000. See <http://www.w3.org/TR/SOAP/>.
- [3] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. Recommendation, W3C, June 2003. See <http://www.w3.org/TR/2003/REC-soap12-part1-20030624>.
- [4] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 2: Adjuncts. Recommendation, W3C, June 2003. See <http://www.w3.org/TR/2003/REC-soap12-part2-20030624>.
- [5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Note, W3C, March 2001. See <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
- [6] Rahul Sharma. The Java API for XML Based RPC (JAX-RPC) 1.0. JSR, JCP, June 2002. See <http://jcp.org/en/jsr/detail?id=101>.
- [7] Roberto Chinnici. The Java API for XML Based RPC (JAX-RPC) 1.1. Maintenance JSR, JCP, August 2003. See <http://jcp.org/en/jsr/detail?id=101>.
- [8] Keith Ballinger, David Ehnebuske, Martin Gudgin, Mark Nottingham, and Prasad Yendluri. Basic Profile Version 1.0. Board Approval Draft, WS-I, July 2003. See <http://www.ws-i.org/Profiles/Basic/2003-06/BasicProfile-1.0-BdAD.html>.
- [9] Joseph Fialli and Sekhar Vajjhala. The Java Architecture for XML Binding (JAXB). JSR, JCP, January 2003. See <http://jcp.org/en/jsr/detail?id=31>.
- [10] Joseph Fialli and Sekhar Vajjhala. The Java Architecture for XML Binding (JAXB) 2.0. JSR, JCP, August 2003. See <http://jcp.org/en/jsr/detail?id=222>.
- [11] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Working Draft, W3C, June 2003. See <http://www.w3.org/TR/2003/WD-wsdl12-20030611>.
- [12] Joshua Bloch. A Metadata Facility for the Java Programming Language. JSR, JCP, August 2003. See <http://jcp.org/en/jsr/detail?id=175>.

- [13] Jim Trezzo. Web Services Metadata for the Java Platform. JSR, JCP, August 2003. See <http://jcp.org/en/jsr/detail?id=181>. 1  
2
- [14] Jim Knutson and Heather Kreger. Web Services for J2EE. JSR, JCP, September 2002. See <http://jcp.org/en/jsr/detail?id=109>. 3  
4
- [15] Nataraj Nagaratnam. Web Services Message Security APIs. JSR, JCP, April 2002. See <http://jcp.org/en/jsr/detail?id=181>. 5  
6
- [16] Farrukh Najmi. Java API for XML Registries 1.0 (JAXR). JSR, JCP, June 2002. See <http://www.jcp.org/en/jsr/detail?id=93>. 7  
8
- [17] Keith Ballinger, David Ehnebuske, Chris Ferris, Martin Gudgin, Marc Hadley, Anish Karmarkar, Canyang Kevin Liu, Mark Nottingham, Jorgen Thelin, and Prasad Yendluri. Basic Profile Version 1.1. Working Group Draft, WS-I, July 2003. Not yet published. 9  
10  
11
- [18] Martin Gudgin, Amy Lewis, and Jeffrey Schlimmer. Web Services Description Language (WSDL) Version 2.0 Part 2: Message Patterns. Working Draft, W3C, June 2003. See <http://www.w3.org/TR/2003/WD-wsdl12-patterns-20030611>. 12  
13  
14
- [19] Jean-Jacques Moreau and Jeffrey Schlimmer. Web Services Description Language (WSDL) Version 2.0 Part 3: Bindings. Working Draft, W3C, June 2003. See <http://www.w3.org/TR/2003/WD-wsdl12-bindings-20030611>. 15  
16  
17
- [20] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. RFC, IETF, March 1997. See <http://www.ietf.org/rfc/rfc2396.txt>. 18  
19
- [21] S. Bradner. RFC 2119: Keywords for use in RFCs to Indicate Requirement Levels. RFC, IETF, March 1997. See <http://www.ietf.org/rfc/rfc2119.txt>. 20  
21
- [22] John Cowan and Richard Tobin. XML Information Set. Recommendation, W3C, October 2001. See <http://www.w3.org/TR/2001/REC-xml-infoaset-20011024/>. 22  
23
- [23] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. Recommendation, W3C, May 2001. See <http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/>. 24  
25  
26
- [24] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. Recommendation, W3C, May 2001. See <http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/>. 27  
28
- [25] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification - second edition. Book, Sun Microsystems, Inc, 2000. 29  
30  
31  
[http://java.sun.com/docs/books/jls/second\\_edition/html/j.title.doc.html](http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html).