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

Early Draft 2
September 28, 2004

Editors:
Marc Hadley
Roberto Chinnici

Comments to: jsr224-spec-comments@sun.com

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

Document Status

This section describes the status of this document at the time of its publication. Other documents may supersede this document; the latest revision of this document can be found on the JSR 224 homepage at `http://www.jcp.org/en/jsr/detail?id=224`.

This is the first Early Draft of JSR 224 (JAX-RPC 2.0). It has been produced by the JSR 224 expert group. Comments on this document are welcome, send them to `jsr224-spec-comments@sun.com`. This draft addresses the following goals and requirements:

- Addition of client side asynchrony
- Improved support for document and message centric usage
- Integration with JAXB
- Improvements to the handler framework
- Improved protocol neutrality

Subsequent versions of this document will address the following additional goals and requirements:

- Support for WS-I Basic Profile 1.1 and Attachments Profile 1.0
- Support for SOAP 1.2
- Support for WSDL 2.0
- Versioning and evolution of web services
- Web services security
- Integration with JSR 181 (Web Services Metadata)
- Service endpoint model
- Runtime services

Contents

1 Introduction **1**

1.1 Goals 1

1.2 Non-Goals 3

1.3 Requirements 3

1.3.1 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 7

1.7 Acknowledgements 7

2 WSDL 1.1 to Java Mapping **9**

2.1 Definitions 9

2.1.1 Extensibility 9

2.2 Port Type 10

2.3 Operation 10

2.3.1 Message and Part 11

2.3.2 Parameter Order and Return Type 14

2.3.3 Holder Classes 14

2.3.4 Asynchrony 15
2.4 Types 19
2.5 Fault 19
2.5.1 Example 20
2.6 Binding 20
2.6.1 General Considerations 20
2.6.2 SOAP Binding 22
2.6.3 MIME Binding 23
2.7 Service and Port 25
2.7.1 Example 25
2.8 XML Names 26
2.8.1 Name Collisions 26

3 Java to WSDL 1.1 Mapping 27

3.1 Java Names 27
3.1.1 Name Collisions 27
3.2 Package 27
3.3 Interface 28
3.3.1 Inheritance 28
3.4 Method 28
3.4.1 One Way Operations 29
3.5 Method Parameters and Return Type 29
3.5.1 Parameter and Return Type Classification 31
3.5.2 Use of JAXB 31
3.6 Service Specific Exception 33
3.7 Bindings 34
3.7.1 Interface 34
3.7.2 Method and Parameters 35
3.8 SOAP HTTP Binding 35
3.8.1 Interface 35
3.8.2 Method and Parameters 36
3.9 Service and Ports 36

4 Client APIs 39

4.1 javax.xml.rpc.ServiceFactory 39

4.1.1 Configuration . . . 39
4.1.2 Factory Usage . . . 40
4.2 javax.xml.rpc.Service . . . 40
4.2.1 Handler Registry . . . 41
4.2.2 Type Mapping Registry . . . 41
4.3 javax.xml.rpc.JAXRPCContext . . . 41
4.3.1 Lifecycle . . . 42
4.3.2 Standard Properties . . . 42
4.3.3 Additional Properties . . . 43
4.4 javax.xml.rpc.Binding . . . 44
4.5 javax.xml.rpc.BindingProvider . . . 44
4.6 javax.xml.rpc.Stub . . . 44
4.6.1 Configuration . . . 45
4.6.2 Dynamic Proxy . . . 46
4.7 javax.xml.rpc.Dispatch . . . 47
4.7.1 Configuration . . . 47
4.7.2 Operation Invocation . . . 47
4.7.3 Operation Response . . . 48
4.7.4 Asynchronous Response . . . 48
4.7.5 Using JAXB . . . 49
4.7.6 Examples . . . 49
4.8 javax.xml.rpc.Call . . . 50
4.8.1 Configuration . . . 50
4.8.2 Operation Invocation . . . 52
4.8.3 Example . . . 52
4.9 Exceptions . . . 53
4.9.1 Protocol Specific Exception Handling . . . 53
4.10 Additional Classes . . . 54

5 Service APIs **55**
5.1 javax.xml.rpc.server.Provider . . . 55
5.1.1 Invocation . . . 55
5.1.2 Lifecycle . . . 56
5.1.3 Configuration . . . 56
5.1.4 Examples . . . 56

5.2 Servlet Based Services . . . 57
5.2.1 Requirements on Service Implementations . . . 57
5.2.2 Servlet Association . . . 57
5.2.3 javax.xml.rpc.server.ServiceLifecycle . . . 58
5.2.4 javax.xml.rpc.server.ServletEndpointContext . . . 59
5.2.5 Packaging and Deployment . . . 60
5.3 EJB Based Services . . . 60

6 Handler Framework **61**
6.1 Architecture . . . 61
6.1.1 Types of Handler . . . 62
6.1.2 Binding Responsibilities . . . 62
6.2 Configuration . . . 63
6.2.1 Programmatic Configuration . . . 63
6.2.2 Deployment Model . . . 65
6.3 Processing Model . . . 65
6.3.1 Handler Lifecycle . . . 65
6.3.2 Handler Execution . . . 66
6.3.3 Handler Implementation Considerations . . . 68
6.4 Message Context . . . 69
6.4.1 javax.xml.rpc.handler.MessageContext . . . 69
6.4.2 javax.xml.rpc.handler.LogicalMessageContext . . . 70
6.4.3 Relationship to JAXRPCContext . . . 70

7 SOAP Binding **71**
7.1 Configuration . . . 71
7.1.1 Programmatic Configuration . . . 71
7.1.2 Deployment Model . . . 72
7.2 Processing Model . . . 73
7.2.1 SOAP `mustUnderstand` Processing . . . 73
7.2.2 Exception Handling . . . 73
7.3 SOAP Message Context . . . 74
7.4 SOAP Transport and Transfer Bindings . . . 74
7.4.1 HTTP . . . 74

8 WSDL 2.0 to Java Mapping **77**

8.1 Definitions . . . 77
8.2 Extensibility . . . 78
8.3 Type Systems . . . 78
8.4 Interfaces . . . 78
8.5 Operations . . . 79
8.5.1 Operations with Signatures . . . 80
8.5.2 Holder Classes . . . 80
8.5.3 Signatureless Operations . . . 81
8.5.4 Fault References . . . 81
8.5.5 Asynchrony . . . 81
8.6 Types . . . 84
8.7 Faults . . . 87
8.8 Services . . . 87
8.9 SOAP 1.2 Binding . . . 88
8.10 XML Names . . . 88
8.10.1 Name Collisions . . . 88

9 Java to WSDL 2.0 Mapping **91**
9.1 Java Names . . . 91
9.1.1 Name Collisions . . . 91
9.2 Packages . . . 92
9.3 Interfaces . . . 92
9.3.1 Inheritance . . . 93
9.4 Methods . . . 93
9.4.1 One Way Operations . . . 94
9.5 Method Parameters . . . 96
9.5.1 Parameters . . . 96
9.5.2 Use of JAXB . . . 96
9.6 Service Specific Exceptions . . . 97
9.7 Bindings . . . 97
9.8 SOAP HTTP Binding . . . 98
9.8.1 Binding component . . . 98
9.9 Services and endpoints . . . 98

A Conformance Requirements **99**

B Change Log **105**

B.1 Changes Since Early Draft 1 105

Bibliography **107**

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 defacto 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 ⇔ XML data binding. JAX-

RPC 2.0 will allow full use of JAXB provided facilities including binding customization and optional schema validation.

SOAP 1.2 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.

WSDL 2.0 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.

WS-I Basic Profile 1.1 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.

A Metadata Facility for the Java Programming Language (JSR 175) JAX-RPC 2.0 will define use of Java annotations[12] to simplify the most common development scenarios for both clients and servers.

Web Services Metadata for the Java Platform (JSR 181) JAX-RPC 2.0 will align with and complement the annotations defined by JSR 181[13].

Implementing Enterprise Web Services (JSR 109) The JSR 109[14] defined `jaxrpc-mapping-info` deployment descriptor provides deployment time Java ⇔ 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 ⇔ WSDL mapping.

Web Services Security (JSR 183) JAX-RPC 2.0 will align with and complement the security APIs defined by JSR 183[15].

JAX-RPC 2.0 will improve support for document/message centric usage:

Asynchrony JAX-RPC 2.0 will add support for client side asynchronous operations.

Non-HTTP Transports 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.

Message Access JAX-RPC 2.0 will simplify client and service access to the messages underlying an exchange.

Session Management JAX-RPC 1.1 session management capabilities are tied to HTTP. JAX-RPC 2.0 will add support for message based session management.

JAX-RPC 2.0 will also address issues that have arisen with experience of implementing and using JAX-RPC 1.0:

Inclusion in J2SE 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.

Handlers 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.

Versioning and Evolution of Web Services JAX-RPC 2.0 will describe techniques and mechanisms to ease the burden on developers when creating new versions of existing services.

Backwards Compatibility of Binary Artifacts JAX-RPC 2.0 will not preclude preservation of backwards binary compatibility between JAX-RPC 1.x and 2.0 implementation runtimes.

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 Relationship To JAXB

JAX-RPC describes the WSDL ⇔ Java mapping, but data binding is delegated to JAXB[10]. The specification must clearly designate where JAXB rules apply to the WSDL ⇔ 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 ⇔ 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 ⇔ 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 ⇔ 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.

Unified Response and Fault Handling The `handleResponse` and `handleFault` methods will be unified and the 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 The session management mechanism must support SOAP based session information.

1.4 Use Cases

1.4.1 Handler Framework

1.4.1.1 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.

1.4.1.2 Message Logging

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

1.4.1.3 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

```java
package com.example.hello;

public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

Non-normative notes are formatted as shown below.

Note: *This is a note.*

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	http://www.w3.org/2003/05/soap-envelope	A normative XML Schema[23, 24] document for the http://www.w3.org/2003/05/soap-envelope namespace can be found at http://www.w3.org/2003/05/soap-envelope.
xsd	http://www.w3.org/2001/XMLSchema	The namespace of the XML schema[23, 24] specification
wsdl	http://schemas.xmlsoap.org/wsdl/	The namespace of the WSDL schema[5]
soap	http://schemas.xmlsoap.org/wsdl/soap/	The namespace of the WSDL SOAP binding schema[23, 24]
jaxb	http://java.sun.com/xml/ns/jaxb	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/...’ and ‘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, notes and sections explicitly marked as ‘Non-Normative’.

1.6 Expert Group Members

The following people have contributed to this specification:

Chavdar Baikov (SAP AG)
Russell Butek (IBM)
Manoj Cheenath (BEA Systems)
Ugo Corda (SeeBeyond Technology Corp)
Glen Daniels (Sonic Software)
Alan Davies (SeeBeyond Technology Corp)
Jim Frost (Art Technology Group Inc)
Kevin R. Jones (Developmentor)
Toshiyuki Kimura (NTT Data Corp)
Doug Kohlert (Sun Microsystems, Inc)
Daniel Kulp (IONA Technologies PLC)
Sunil Kunisetty (Oracle)
Changshin Lee (Tmax Soft, Inc)
Srividya Natarajan (Nokia Corporation)
Bjarne Rasmussen (Novell, Inc)
Sebastien Sahuc (Intalio, Inc.)
Rahul Sharma (Motorola)
Rajiv Shivane (Pramati Technologies)
Richard Sitze (IBM)
Dennis M. Sosnoski (Sosnoski Software)
Christopher St. John (WebMethods Corporation)

1.7 Acknowledgements

Robert Bissett, Arun Gupta, Graham Hamilton, Mark Hapner, Jitendra Kotamraju, Vivek Pandey, Santiago Pericas-Geertsen, Eduardo Pelegri-Llopart, Paul Sandoz, Bill Shannon, and Kathy Walsh (all from Sun Microsystems) have provided invaluable technical input to the JAX-RPC 2.0 specification.

As the specification lead for JAX-RPC 1.0, Rahul Sharma was extremely influential in determining the original direction of this technology.

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. JAXB[10] (see appendix C) defines a standard mapping from a namespace URI to a Java package name. By default, this algorithm is used to map the value of a `wsdl:definitions` element's `targetNamespace` attribute to a Java package name.

◊ *Conformance Requirement (Definitions mapping):* In the absence of customizations, when mapping a WSDL `definitions` element to a Java package, the Java package name is mapped from the value of a `wsdl:definitions` element's `targetNamespace` attribute using the algorithm defined by JAXB[10].

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 (WSDL and XML Schema import directives):* An implementation MUST support the WS-I Basic Profile 1.1[17] 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 (Optional WSDL extensions):* An implementation MAY support mapping of additional 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`):* A mapped SEI MUST extend `java.rmi.Remote`.

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[17] 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.

2.3.1.1 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.

2.3.1.2 Wrapper Style

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

(i) The operation's input and output messages (if present) 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

(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

(v) The wrapper elements only contain child elements, they must not contain other structures such as wildcards (element or attribute), `xsd:choice`, substitution groups (element references are not permitted) or attributes

◇ *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.

◇ *Conformance Requirement (Parameter name clash):* If the mapping results in two Java parameters with the same name and one of those parameters is not mapped to the method return type, see section 2.3.2, then this is reported as an error and requires developer intervention to correct, either by disabling wrapper style mapping, modifying the source WSDL or by specifying a customized parameter name mapping.

2.3.1.3 Example

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

```
<!-- WSDL extract -->
<types>
    <xsd:element name="setLastTradePrice">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="tickerSymbol" type="xsd:string"/>
                <xsd:element name="lastTradePrice" type="xsd:float"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="setLastTradePriceResponse">
        <xsd:complexType>
            <xsd:sequence/>
        </xsd:complexType>
    </xsd:element>
</types>

<message name="setLastTradePrice">
    <part name="setLastTradePrice"
        element="tns:setLastTradePrice"/>
</message>


<message name="setLastTradePriceResponse">
    <part name="setLastTradePriceResponse"
        element="tns:setLastTradePriceResponse"/>
</message>


<portType name="StockQuoteUpdater">
    <operation name="setLastTradePrice">
        <input message="tns:setLastTradePrice"/>
        <output message="tns:setLastTradePriceResponse"/>
    </operation>
</portType>

// non-wrapper style mapping
SetLastTradePriceResponse setLastTradePrice(
    SetLastTradePrice setLastTradePrice) throws RemoteException;

// wrapper style mapping
void setLastTradePrice(String tickerSymbol, float lastTradePrice)
    throws RemoteException;
```

Figure 2.1: 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 message parts are either listed or unlisted. Parameters that are mapped from listed parts are listed; parameters that are mapped from unlisted parts are unlisted.
- Parameters that are mapped from wrapper children (wrapper style mapping only) are unlisted.
- Listed parameters appear first in the method signature in the order in which their corresponding parts are listed in the `parameterOrder` attribute.
- Unlisted parameters either form the return type or follow the listed parameters
- The return type is determined as follows:

  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`.

  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`.

- Unlisted parameters that do not form the return type follow the listed parameters in the following order:
  1. Parameters mapped from `in` and `in/out` parts appear in the same order the corresponding parts appear in the input message.
  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.
  3. Parameters mapped from `out` parts appear in the same order the corresponding parts appear in the output message.
  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.

2.3.3 Holder Classes

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.

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 primitve 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>`.

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

2.3.4 Asynchrony

In addition to the synchronous mapping of `wsdl:operation` described above, a client side 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 client side asynchronous methods should be optional at the users discretion.

◇ *Conformance Requirement (Asynchronous mapping required):* Implementations MUST support the asynchronous mapping.

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

2.3.4.1 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.

2.3.4.2 Operation

Each `wsdl:operation` is mapped to two additional methods in the corresponding 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`[1].

[1] Errors that occur during the invocation are reported when the client attempts to retrieve the results of the operation, see section 2.3.4.5.

2.3.4.3 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.

2.3.4.4 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's 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 in no namespace whose localname 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.

2.3.4.5 Faults

Mapping of WSDL faults to service specific exceptions is identical for both asynchronous and synchronous cases, section 2.5 describes the mapping. However, mapped asynchronous methods do not throw service specific exceptions directly. Instead a `JAXRPCException` is thrown when a client attempts to retrieve the results of an asynchronous method invocation via the `Response.get` method.

◇ *Conformance Requirement (Asynchronous fault reporting):* A WSDL fault that occurs during execution of an asynchronous method invocation MUST be mapped to a `JAXRPCException` thrown when the client calls `Response.get`.

Response is a static generic interface whose get method only throws JAXRPCException. Instead of throwing a service speciifc exception, a Response instance throws a JAXRPCException whose cause is set to an instance of the service specific exception mapped from the corresponding WSDL fault.

◇ *Conformance Requirement (Asychronous fault cause):* A JAXRPCException thrown by Response.get as a result of a WSDL fault MUST have as its cause the service specific exception mapped from the WSDL fault.

2.3.4.6 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.

2.3.4.7 Usage Examples

Synchronous use.

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

Asynchronous polling use.

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

Asynchronous callback use.

```
class MyPriceHandler implements AsyncHandler<Float> {
    ...
    public void handleResponse(Response<Float> response) {
        Float price = response.get();
        // do something with the result
    }
}

Service service = ...;
StockQuote quoteService = (StockQuote)service.getPort(portName);
MyPriceHandler myPriceHandler = new MyPriceHandler();
quoteService.getPriceAsync(ticker, myPriceHandler);
```

```
<!-- WSDL extract -->
<message name="getPrice">
    <part name="ticker" type="xsd:string"/>
</message>


<message name="getPriceResponse">
    <part name="price" type="xsd:float"/>
</message>


<portType name="StockQuote">
    <operation name="getPrice">
        <input message="tns:getPrice"/>
        <output message="tns:getPriceResponse"/>
    </operation>
</portType>

<!-- Synthesized response bean element -->
<xsd:element name="getPriceResponse">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="price" type="xsd:float"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

// synchronous mapping
public interface StockQuote {
    float getPrice(String ticker) throws RemoteException;
}

// asynchronous mapping
public interface StockQuote {
    float getPrice(String ticker) throws RemoteException;
    Response<Float> getPriceAsync(String ticker);
    Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
}
```

Figure 2.2: Asynchronous operation mapping

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. By default JAX-RPC uses the class based mapping of JAXB but also allows use of the interface based mapping.

◊ *Conformance Requirement (JAXB class mapping):* In the absence of user customizations, an implementation MUST use the JAXB class based mapping (`generateValueClass="true"`, `generateElementType="false"`) when mapping WSDL types to Java.

Editors Note 2.1 *The above annotations are preliminary and subject to change, this section will be updated once the JAXB specification has settled on a set of annotations.*

◊ *Conformance Requirement (JAXB customization use):* An implementation MUST support use of JAXB customizations during mapping.

◊ *Conformance Requirement (JAXB customization clash):* To avoid clashes, if a user customizes the mapping an implementation MUST NOT add the default class based mapping customizations.

In addition, for ease of use, JAX-RPC strips any `IXMLElement<T>` wrapper off the type of a method parameter if the normal JAXB mapping would result in one[2]. E.g. a parameter that JAXB would map to `IXMLElement<Integer>` is instead be mapped to `Integer`.

2.5 Fault

Editors Note 2.2 *The mapping described in this section differs from the mapping provided by JAX-RPC 1.1 by eliminating the generation of exception hierarchies and instead providing an inheritance hierarchy for the type that forms the encapsulated fault information. This minimizes the number of types generated. The expert group seeks feedback on this choice.*

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.

Multiple operations within the same service can define equivalent faults. Faults defined within the same service are equivalent if the values of their `message` attributes are equal.

◊ *Conformance Requirement (Fault equivalence):* An implementation MUST map all equivalent faults within a service to a single Java exception class.

[2] JAXB maps an element declaration to a Java instance that implements IXMLElement.

A `wsdl:fault` element refers to a `wsdl:message` that contains a single part. The global element declaration[3] 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*(String message, *FaultBean* faultInfo)** 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.

***WrapperException*(String message, *FaultBean* faultInfo, 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. The final argument, `cause`, may be used to convey protocol specific fault information, see section 4.9.1.

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

Two `wsdl:fault` child elements of the same `wsdl:operation` that indirectly refer to the same global element declaration are considered to be equivalent since there is no interoperable way of differentiating between their serialized forms.

◇ *Conformance Requirement (Fault equivalence):* At runtime an implementation MAY map a serialized fault into any equivalent Java exception.

2.5.1 Example

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

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.

◇ *Conformance Requirement (Required WSDL extensions):* An implementation MUST support mapping of the WSDL 1.1 specified extension elements for the WSDL SOAP and MIME bindings.

2.6.1 General Considerations

R2209 in WS-I Simple SOAP Binding Profile 1.1[26] 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`.

[3] WS-I Basic Profile[17] R2205 requires parts to refer to elements rather than types.

```
<!-- WSDL extract -->
<types>
    <xsd:schema targetNamespace="...">
        <xsd:element name="faultDetail">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="majorCode" type="xsd:int"/>
                    <xsd:element name="minorCode" type="xsd:int"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
</types>

<message name="operationException">
    <part name="faultDetail" element="tns:faultDetail"/>
</message>


<portType name="StockQuoteUpdater">
    <operation name="setLastTradePrice">
        <input .../>
        <output .../>
        <fault name="operationException"
            message="tns:operationException"/>
    </operation>
</portType>

// fault mapping
class OperationException extends Exception {
    OperationException(String message, FaultDetail faultInfo) {...}
    OperationException(String message, FaultDetail faultInfo,
        Throwable cause) {...}
    FaultDetail getFaultInfo() {...}
}
```

Figure 2.3: Fault mapping

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.

2.6.2.1 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` (see section 2.3.1); the additional part is always mapped using the non-wrapper style.

♢ *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.

Note that the order of headers in a SOAP message is independent of the order of `soap:header` elements in the WSDL binding – see R2751 in WS-I Basic Profile 1.0[8]. This causes problems when two or more headers with the same qualified name are present in a message and one or more of those headers are bound to a method parameter since it is not possible to determine which header maps to which parameter.

♢ *Conformance Requirement (Duplicate headers in binding):* During mapping, an implemention MUST report an error if the binding of an operation includes two or more `soap:header` elements that would result in SOAP headers with the same qualified name.

♢ *Conformance Requirement (Duplicate headers in Message):* During unmarshalling, an implementation MUST generate a runtime error if there is more than one instance of a header whose qualified name is mapped to a method parameter.

2.6.2.2 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.

Unlike a `wsdl:fault` that may only refer to a message containing a single part, a `soap:headerfault` can refer to any single part of a message containing one or more parts. Mapping of `soap:headerfault` elements follows the mapping for `wsdl:fault` elements described in section 2.5 with the following differences:

1. To avoid name clashes, the mapped Exception is named after the part referred to by the `soap:headerfault` rather than its parent message.
2. The global element that is mapped to a Java bean is the global element[4] referred to by the part named in the `soap:headerfault`.
3. For the purposes of duplicate removal during mapping, header faults are consider to be equivalent if the values of their `message` and `part` attributes are equal.

2.6.3 MIME Binding

The presence of a `mime:multipartRelated` binding extension element as a child of a `wsdl:input` or `wsdl:output` element in a `wsdl:binding` indicates that the corresponding messages may be serialized as MIME packages. The WS-I Attachments Profile[27] describes two separate attachment mechanisms, both based on use of the WSDL 1.1 MIME binding[5]:

`wsiap:swaRef` A schema type that may be used in the abstract message description to indicate a reference to an attachment.

`mime:content` A binding construct that may be used to bind a message part to an attachment.

JAXB[10] describes the mapping from the WS-I defined `wsiap:swaref` schema type to Java and, since JAX-RPC inherits this capability, it is not discussed further here. Use of the `mime:content` construct is outside the scope of JAXB mapping and the following subsection describes changes to the WSDL 1.1 to Java mapping that results from its use.

2.6.3.1 `mime:content`

Message parts are mapped to method parameters as described in section 2.3 regardless of whether the part is bound to the SOAP message or to an attachment. JAXB rules are used to determine the Java type of message parts based on the XML schema type referenced by the `wsdl:part`. However, when a message part is bound to a MIME part (using the `mime:content` element of the WSDL MIME binding) additional information is available that provides the MIME type of the data and this can optionally be used to narrow the default JAXB mapping.

♢ *Conformance Requirement (Use of MIME type information):* An implementation MUST provide an option to enable use of the additional metadata in `mime:content` elements when mapping from WSDL to Java.

JAXB defines a mapping between MIME types and Java types. When a part is bound using one or more `mime:content` elements[5] and use of the additional metadata is enabled then the JAXB mapping is customized to use the most specific type allowed by the set of MIME types described for the part in the binding.

Editors Note 2.3 *The relevent section is not yet available in the JAXB specification. The above will be expanded once that section is completed and a reference will be added.*

Figure 2.4 shows an example WSDL and two mapped interfaces: one without using the `mime:content` metadata, the other using the additional metadata to narrow the binding. Note that in the latter the type of the `claimPhoto` method parameter is `Image` rather than the default `byte[]`.

[4] WS-I Basic Profile[17] R2205 requires the part to reference a global element rather than a type

[5] Multiple `mime:content` elements for the same part indicate a set of permissible alternate types.

```
<!-- WSDL extract -->
<wsdl:message name="ClaimIn">
  <wsdl:part name="body" element="types:ClaimDetail"/>
  <wsdl:part name="ClaimPhoto" type="xsd:base64Binary"/>
</wsdl:message>

<wsdl:portType name="ClaimPortType">
  <wsdl:operation name="SendClaim">
    <wsdl:input message="tns:ClaimIn"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="ClaimBinding" type="tns:ClaimPortType">
  <soapbind:binding style="document" transport="..."/>
  <wsdl:operation name="SendClaim">
    <soapbind:operation soapAction="..."/>
    <wsdl:input>
      <mime:multipartRelated>
        <mime:part>
          <soapbind:body parts="body" use="literal"/>
        </mime:part>
        <mime:part>
          <mime:content part="ClaimPhoto" type="image/jpeg"/>
          <mime:content part="ClaimPhoto" type="image/gif"/>
        </mime:part>
      </mime:multipartRelated>
    </wsdl:input>
  </wsdl:operation>
</wsdl:binding>

// Mapped Java interface without mime:content metadata
public interface ClaimPortType {
    public String sendClaim(ClaimDetail detail, byte claimPhoto[]);
}

// Mapped Java interface using mime:content metadata
public interface ClaimPortType {
    public String sendClaim(ClaimDetail detail, Image claimPhoto);
}
```

Figure 2.4: Use of `mime:content` metadata

◇ *Conformance Requirement (MIME type mismatch):* An implementation SHOULD throw a `JAXRPCException` on receipt of a message where the MIME type of a part does not match that described in the WSDL.

◇ *Conformance Requirement (MIME part identification):* An implementation MUST use the algorithm defined in the WS-I Attachments Profile[27] when generating the MIME `Content-ID` header field value for a part bound using `mime:content`.

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 address. On the client side, 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 the `javax.xml.rpc.Service` interface.

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

***ServiceEndpointInterface* `get`*PortName*`()`** 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.

◇ *Conformance Requirement (Failed `get`PortName):* `get`*PortName* MUST throw `javax.xml.rpc.ServiceException` on failure.

The value of *PortName* in the above is derived as follows: the value of the `name` attribute of the `wsdl:port` 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 `get`*PortName* method name.

2.7.1 Example

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

```
<!-- WSDL extract -->
<wsdl:service name="StockQuoteService">
    <wsdl:port name="StockQuoteHTTPPort" binding="StockQuoteHTTPBinding"/>
    <wsdl:port name="StockQuoteSMTPPort" binding="StockQuoteSMTPBinding"/>
</wsdl:service>

// Generated Service Interface
public interface StockQuoteService extends javax.xml.rpc.Service {
    StockQuoteProvider getStockQuoteHTTPPort()
        throws ServiceException;
    StockQuoteProvider getStockQuoteSMTPPort()
        throws ServiceException;
}
```

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. The error may be corrected 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

♢ *Conformance Requirement (Java identifier mapping):* Java identifiers SHOULD be mapped to XML names using the algorithm defined in appendix B of SOAP 1.2 Part 2[4].

3.1.1 Name Collisions

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

♢ *Conformance Requirement (Method name disambiguation):* An implementation MUST support the use of metadata to disambiguate overloaded Java method names when mapped to WSDL.

3.2 Package

A Java package is mapped to a `wsdl:definitions` element and an 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. JAXB[10] (see appendix D) defines a standard mapping from a Java package name to a namespace URI. By default, this algorithm is used to map the package name to the value of the `wsdl:definitions` element's `targetNamespace` attribute.

♢ *Conformance Requirement (Package name mapping):* In the absence of customizations, when mapping a Java package to a WSDL `definitions` element, the Java package name is mapped to the value of the `wsdl:definitions` element's `targetNamespace` attribute using the algorithm defined by JAXB[10].

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 (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[1] (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

Editors Note 3.1 *The mapping described in this section only allows production of wrapper style WSDL descriptions. This simplifies the mapping and targets what is expected to become the dominant usage model. The expert group seeks feedback on this choice.*

[1] WSDL 1.1 does not define any standard representation for constants in a `wsdl:portType` definition.

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:

- A `wsdl:input` element that refers to an associated `wsdl:message` element to describe the operation input.
- (Two-way methods only) an optional `wsdl:output` element that refers to a `wsdl:message` to describe the operation output.
- (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.

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.

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.

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

3.4.1 One Way Operations

Only Java methods whose return type is `void`, that have no parameters that implement `Holder` 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.

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

◊ *Conformance Requirement (One-way mapping errors):* Implementations MUST prevent mapping to one-way operations of methods that do not meet the necessary criteria.

3.5 Method Parameters and Return Type

A Java method's parameters and return type are mapped to components of the global element declarations mapped from the method. Parameters can be mapped to components of the global element declaration

```
// Java
package com.example;
public interface StockQuoteProvider extends java.rmi.Remote {
    float getPrice(String tickerSymbol)
        throws java.rmi.RemoteException, TickerException;
}

<!-- WSDL extract -->
<types>
    <xsd:schema targetNamespace="...">
        <!-- element declarations -->
        <xsd:element name="getPrice"
            type="tns:getPriceType"/>
        <xsd:element name="getPriceResponse"
            type="tns:getPriceResponseType"/>
        <xsd:element name="TickerException"
            type="tns:TickerExceptionType"/>

        <!-- type definitions -->
        ...
    </xsd:schema>
</types>

<message name="getPrice">
    <part name="getPrice" element="tns:getPrice"/>
</message>


<message name="getPriceResponse">
    <part name="getPriceResponse" element="tns:getPriceResponse"/>
</message>


<message name="TickerException">
    <part name="TickerException" element="tns:TickerException"/>
</message>


<portType name="StockQuoteProvider">
    <operation name="getPrice" parameterOrder="tickerSymbol">
        <input message="tns:getPrice"/>
        <output message="tns:getPriceResponse"/>
        <fault message="tns:TickerException"/>
    </operation>
</portType>
```

Figure 3.1: Java interface to WSDL portType mapping

for either the operation input message, operation output message or both. The mapping depends on the parameter classification.

3.5.1 Parameter and Return Type Classification

Method parameters and return type are classified as follows:

`in` The 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.

`out` The 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/out` The 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.

A methods return type is always `out`. For method parameters, holder classes are used to determine the classification. A holder class is a class that implements the `javax.xml.rpc.holders.Holder` interface. A 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 means to specify 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 for the method input (henceforth called the request bean) and one for the method output (henceforth called the response bean). [2]

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.

Two styles of Java to WSDL mapping are supported: wrapped and bare. The two styles differ in how the request and response bean classes are constructed for a method. The two styles are described in the following two subsections.

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

3.5.2.1 Wrapped

A wrapper request bean is generated containing properties for each `in` and `in/out` parameter. A wrapper response bean is generated containing properties for the method return value, each `out` parameter, and `in/out` parameter. Method return values are represented by an `out` property named “return”.

◇ *Conformance Requirement (Wrapper bean naming):* Implementations SHOULD provide a means to specify the localname of the elements generated for the wrapper beans.

◇ *Conformance Requirement (Default wrapper bean names):* In the absence of customizations, the wrapper request bean class MUST be named the same as the method and the wrapper response bean class MUST be named the same as the method with a “Response” suffix.

Figure 3.2 illustrates this representation.

```
float getPrice(String tickerSymbol);

@XmlElement class getPrice {
    public String getTickerSymbol();
}

@XmlElement class getPriceResponse {
    public float getReturn();
}
```

Figure 3.2: Wrapper mode bean representation of an operation

When the JAXB mapping to XML Schema is utilized this results in global element declarations for the mapped request and response beans with child elements for each method parameter according to the parameter classification:

`in` The parameter is mapped to a child element of the global element declaration for the request bean.

`out` The parameter or return value is mapped to a child element of the global element declaration for the response bean.

`in/out` The parameter is mapped to a child element of the global element declarations for the request and response beans.

3.5.2.2 Bare

In order to qualify for use of base mapping mode a Java method must fulfill all of the following criteria:

1. It must have at most one `in` or `in/out` parameter.
2. If it has a return type other than `void` it must have no `in/out` or `out` parameters.
3. If it has a return type of `void` it must have at most one `in/out` or `out` parameter.

If present, the input parameter becomes the request bean, if there is no input parameter then the request bean is an empty Java bean class. If present, the output parameter or return type becomes the response bean, if there is no output parameter or the return type is `void` then the response bean is an empty Java bean class.

Editors Note 3.2 *Doc/lit/bare isn't well specified but since it is based on the method arguments one might expect the default bean (and hence element) naming to be based on parameter name rather than method name. The problem with this is that one can then easily violate R2710 in WS-I BP if one has two methods with the same input argument.*

In the absence of customizations, the request bean class is named the same as the method and the response bean class is named the same as the method with a "Response" suffix.

◇ *Conformance Requirement (Bean naming):* Implementations SHOULD provide a means to specify the localname of the elements generated for the request and response beans.

◇ *Conformance Requirement (Default bean names):* In the absence of customizations, the request bean class MUST be named the same as the method and the response bean class MUST be named the same as the method with a "Response" suffix.

Figure 3.3 illustrates this representation.

Editors Note 3.3 *The JAXB annotations shown in figure 3.3 are preliminary and are subject to change.*

```
float getPrice(String tickerSymbol);

@XmlElement class getPrice {
    @XmlValue public String getValue();
}

@XmlElement class getPriceResponse {
    @XmlValue public float getValue();
}
```

Figure 3.3: Bare mode bean representation of an operation

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 global 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 3.4.

```
<portType name="StockQuoteProvider">
    <operation name="getPrice" parameterOrder="tickerSymbol">
        <input message="tns:getPrice"/>
        <output message="tns:getPriceResponse"/>
        <fault message="tns:unknowntickerException"/>
    </operation>
</portType>

<binding name="StockQuoteProviderBinding">
    <!-- binding specific extensions possible here -->
    <operation name="getPrice">
        <!-- binding specific extensions possible here -->
        <input message="tns:getPrice">
            <!-- binding specific extensions possible here -->
        </input>
        <output message="tns:getPriceResponse">
            <!-- binding specific extensions possible here -->
        </output>
        <fault message="tns:unknowntickerException">
            <!-- binding specific extensions possible here -->
        </fault>
    </operation>
</binding>
```

Figure 3.4: 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 and zero or more `wsdl:port` extensibility elements.

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 contains the qualified name of the corresponding `wsdl:portType` suffixed with “Binding”.

The `wsdl:port` extensibility elements define the binding specific endpoint address for a given port, see section 3.9.

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 3.5 shows an example of a SOAP HTTP binding.

```
<binding name="StockQuoteProviderBinding">
    <soap:binding
        transport="http://schemas.xmlsoap.org/soap/http"
            style="document"/>
    <operation name="getPrice">
        <soap:operation style="document"/>
        <input message="tns:getPrice">
            <soap:body use="literal"/>
        </input>
        <output message="tns:getPriceResponse">
            <soap:body use="literal"/>
        </output>
        <fault message="tns:unknowntickerException">
            <soap:fault use="literal"/>
        </fault>
    </operation>
</binding>
```

Figure 3.5: WSDL SOAP HTTP binding

3.8.1 Interface

Editors Note 3.4 *In line with section 3.4, the mapping described in this section only allows production of document/literal WSDL descriptions. This simplifies the mapping and targets what is expected to become the dominant usage model. The expert group seeks feedback on this choice.*

A Java service endpoint interface (SEI) is mapped to a `soap:binding` child element of the `wsdl:binding` element and a `soap:address` child element of any corresponding `wsdl:port` element (see section 3.9).

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 3.6 shows an example.

3.9 Service and Ports

A Java package is mapped to a single `wsdl:service` element that is a child of the `wsdl:definitions` element mapped from the package (see section 3.2). The value of the `name` attribute of the `wsdl:service` element is not significant but would typically be mapped from the name of the Java package.

A WSDL 1.1 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 address.

♢ *Conformance Requirement (Port selection):* Implementations MUST provide a facility for specifying the ports to generate when mapping from Java to WSDL.

Each desired port is represented by a `wsdl:port` child element of the single `wsdl:service` element mapped from the Java package. The value of the `name` attribute of the `wsdl:port` element is not significant but is typically derived from the name of the binding. The value of the `binding` attribute of the `wsdl:port` element is the same as the value of the `name` attribute of the `wsdl:binding` element to which it refers.

Binding specific child extension elements of the `wsdl:port` element define the endpoint address for a port. E.g. see the `soap:address` element described in section 3.8.1.

```
<types>
    <schema targetNamespace="...">
        <xsd:element name="getPrice" type="tns:getPriceType"/>
        <xsd:complexType name="getPriceType">
            <xsd:sequence>
                <xsd:element name="tickerSymbol" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>

        <xsd:element name="getPriceResponse"
            type="tns:getPriceResponseType"/>
        <xsd:complexType name="getPriceResponseType">
            <xsd:sequence>
                <xsd:element name="return" type="xsd:float"/>
            </xsd:sequence>
        </xsd:complexType>
    </schema>
</types>

<message name="getPrice">
    <part name="getPrice"
        element="tns:getPrice"/>
</message>


<message name="getPriceResponse">
    <part name="getPriceResponse" element="tns:getPriceResponse"/>
</message>


<portType name="StockQuoteProvider">
    <operation name="getPrice" parameterOrder="tickerSymbol">
        <input message="tns:getPrice"/>
        <output message="tns:getPriceResponse"/>
    </operation>
</portType>

<binding name="StockQuoteProviderBinding">
    <soap:binding
        transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="getPrice" parameterOrder="tickerSymbol">
        <soap:operation/>
        <input message="tns:getPrice">
            <soap:body use="literal"/>
        </input>
        <output message="tns:getPriceResponse">
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>
```

Figure 3.6: 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.

Conformance requirements in this chapter use the term 'implementation' to refer to a client side JAX-RPC runtime system.

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 J2SE implementation MUST provide a concrete class that extends `javax.xml.rpc.ServiceFactory`.

4.1.1 Configuration

Editors Note 4.1 *The JAX-RPC 1.1 ServiceFactory provides no API to configure factory properties. The expert group is considering addition of a standard property capability to the ServiceFactory and seeks feedback on this choice.*

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.

Note: *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 uses a `ServiceFactory` to create `Service` instances, the following code illustrates this process.

```
ServiceFactory sf = ServiceFactory.newInstance();
Service s = sf.createService(...);
```

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 as described in section 4.1.2. `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 MUST implement `java.io.Serializable` and `javax.naming.Referenceable` 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 6 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.

◇ *Conformance Requirement (Read-only handler chains):* An implementation MAY prevent changes to handler chains configured by some other means (e.g. via a deployment descriptor) by throwing `UnsupportedOperationException` from the `setHandlerChain` methods of `HandlerRegistry`

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.

4.2.2 Type Mapping Registry

JAX-RPC 1.1 provides 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.

◇ *Conformance Requirement (TypeMappingRegistry support):* An implementation MUST throw `UnsupportedOperationException` from `Service.getTypeMappingRegistry` if the `Service` implementation uses JAXB for data binding.

◇ *Conformance Requirement (TypeMappingRegistry usage):* An implementation MAY use serializers and deserializers registered in the type mapping registry if the `Service` implementation does not use JAXB for data binding.

4.3 javax.xml.rpc.JAXRPCContext

Additional metadata is often required to control 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`, `javax.xml.rpc.Dispatch`, and `javax.xml.rpc.Call` 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. 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.

In some cases, the additional metadata may also need to accompany information exchanges. When this is required, protocol bindings are responsible for annotating outbound protocol data units and extracting metadata from inbound protocol data units.

Note: *An example of the latter usage: 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.*

4.3.1 Lifecycle

The lifecycle of a `JAXRPCContext` is the same as that of the `BindingProvider` with which it is associated, the `JAXRPCContext` is created during `BindingProvider` creation and is destroyed during `BindingProvider` destruction.

4.3.2 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.

Table 4.1: Standard `JAXRPCContext` properties.

Name	Type	Mandatory	Description
`javax.xml.rpc.service.endpoint`			
`.address`	String	Y	The address of the service endpoint as a protocol specific URI. The URI scheme must match the protocol binding in use.
`javax.xml.rpc.binding`			
`.context`	JAXBContext	Y	A `JAXBContext` instance that may be used to perform marshaling of outbound messages and unmarshaling of inbound messages.
`.attachments`	Map<String,DataHandler>	Y	A map of attachments to a message. The key represents a unique identifier for the attachment. The value is a `DataHandler` whose content forms the body of the attachment. Individual bindings describe how to carry attachments with messages.
`javax.xml.rpc.security`			
`.username`	String	Y	Username for HTTP basic authentication.
`.password`	String	Y	Password for HTTP basic authentication.
`javax.xml.rpc.session`			

Continued on next page

Table 4.1 – continued from previous page

Name	Type	Mandatory	Description
`.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 `false`.
`javax.xml.rpc.soap.operation`			
`.style`	String	N	The style of the operation: either `rpc` or `document`. Only used in conjuction with the `Call` API, see section 4.8.
`javax.xml.rpc.soap.http.soapaction`			
`.use`	Boolean	N	Controls whether the `SOAPAction` HTTP header is used in SOAP/HTTP requests. Default value is `false`.
`.uri`	String	N	The value of the `SOAPAction` HTTP header if the `javax.xml.rpc-.soap.http.soapaction.use` property is set to `true`. Default value is an empty string.
`javax.xml.rpc.encodingstyle.namespace`			
`.uri`	String	N	The encoding style specified as a URI. Default value is the URI corresponding to SOAP encoding: `http://schemas.xmlsoap.org-/soap/encoding/`. Retained for backwards compatibility with JAX-RPC 1.1.

◇ *Conformance Requirement (Required `JAXRPCContext` properties):* An implementation MUST support all 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.

4.3.3 Additional Properties

◇ *Conformance Requirement (Additional `JAXRPCContext` properties):* Implementations 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 base 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 7 describes the JAX-RPC SOAP binding.

Clients obtain a `Binding` instance from a stub, dynamic proxy, `Dispatch` or `Call` instance using the `getBinding` method of the `javax.xml.rpc.BindingProvider` interface (see section 4.5).

`Binding` provides methods to manipulate the handler chain (see section 6.2.1) configured on an instance.

◇ *Conformance Requirement (Read-only handler chains):* An implementation MAY prevent changes to handler chains configured by some other means (e.g. via a deployment descriptor) by throwing `UnsupportedOperationException` from the `setHandlerChain` method of `Binding`

4.5 javax.xml.rpc.BindingProvider

The `BindingProvider` interface is the superinterface of `javax.xml.rpc.Stub`, `javax.xml.rpc.Disptach`, and `javax.xml.rpc.Call`. It represents a component that provides a protocol binding for use by clients, figure 4.1 illustrates the class relationships.

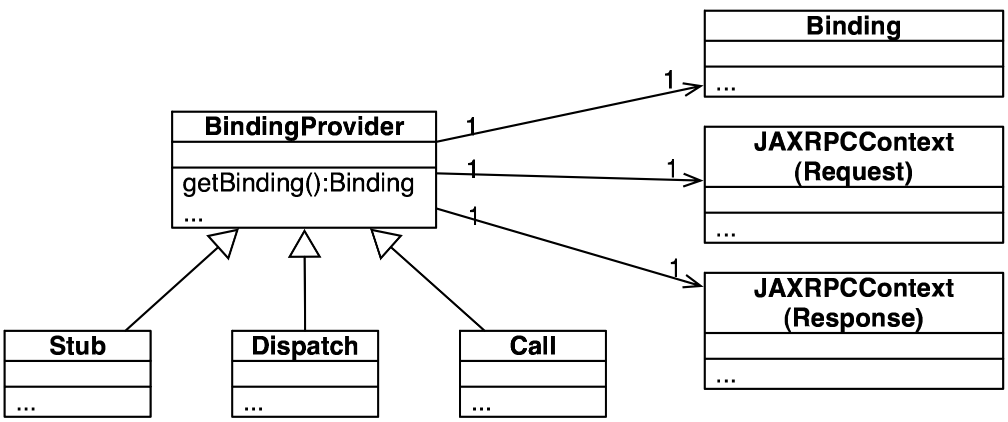

Figure 4.1: Binding Provider Class Relationships

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 `javax.xml.rpc.Stub`.

◇ *Conformance Requirement (Stub class binding):* A generated stub class SHOULD be bound to a particular protocol (and transport if the protocol supports more than one transport).

Note: *JAX-RPC does not standardize the names of generated stub classes. A typical convention is to name them after the protocol binding in the WSDL, i.e.* `BindingName_Stub` *where* BindingName *is the value of the* `name` *attribute of the corresponding WSDL* `binding` *element.*

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 protocol 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`.

```
javax.xml.rpc.Service service = ...;
javax.xml.rpc.Stub stub = (Stub)service.getPort(portName,
    com.example.StockQuoteProvider.class);
stub._setProperty("javax.xml.rpc.session.maintain", Boolean.FALSE);
javax.xml.rpc.JAXRPCContext context = stub.getRequestContext();
context.setProperty("javax.xml.rpc.session.maintain", Boolean.TRUE);
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):* An instance of a dynamic proxy MUST implement `javax.xml.rpc.Stub`.

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 (`Service.getPort` failure):* An implementation MUST throw `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).

4.6.2.1 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", 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 *The name `Dispatch` is a placeholder, other alternatives considered include `Endpoint`, `Post`, `Transmit`, and `Send`. The expert group would welcome alternative suggestions.*

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.

`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 `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 `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 method blocks until the remote operation completes and the results are returned.

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

One-way (`invokeOneWay` methods) The method 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[1] during the remote operation invocation.

See section 7.4.1 for additional SOAP/HTTP requirements.

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.

[1]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.

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 4.3.2). The supplied `JAXBContext` is used to marshall the request object to XML and unmarshall any response.

◇ *Conformance Requirement (Marshalling failure):* If an error occurs when using the supplied `JAXBContext` to marshall a request or unmarshall 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.

4.7.6.1 Synchronous

```
Source reqMsg = ...;
javax.xml.rpc.Service service = ...;
javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
JAXRPCContext reqCtx = disp.getRequestContext();
reqCtx.setProperty(...);
Source resMsg = (Source)disp.invoke(reqMsg);
JAXRPCContext resCtx = disp.getResponseContext();
```

4.7.6.2 Synchronous With JAXB Objects

```
JAXBContext jc = JAXBContext.newInstance("primer.po");
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po = (PurchaseOrder)u.unmarshal(
    new FileInputStream( "po.xml" ) );
javax.xml.rpc.Service service = ...;
javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
JAXRPCContext reqCtx = disp.getRequestContext();
reqCtx.setProperty("javax.xml.rpc.binding.context", jc);
OrderConfirmation conf = (OrderConfirmation)disp.invoke(po);
JAXRPCContext resCtx = disp.getResponseContext();
```

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

4.7.6.3 Asynchronous Polling

```
Source reqMsg = ...;
javax.xml.rpc.Service service = ...;
javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
JAXRPCContext reqCtx = disp.getRequestContext();
reqCtx.setProperty(...);
```

```
Response<Object> res = disp.invokeAsync(reqMsg);
while (!res.isDone()) {
    // do something while we wait
}
Source resMsg = (Source)res.get();
JAXRPCContext resCtx = res.getContext();
```

Note that asynchronous polling is also possible using JAXB objects instead of `Source`.

4.7.6.4 Asynchronous Callback

```
class MyHandler implements AsyncHandler<Object> {
    ...
    public void handleResponse(Response<Object> res) {
        Source resMsg = (Source)res.get();
        JAXRPCContext resCtx = res.getContext();
        // do something with the results
    }
}

Source reqMsg = ...;
javax.xml.rpc.Service service = ...;
javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
JAXRPCContext reqCtx = disp.getRequestContext();
reqCtx.setProperty(...);
MyHandler handler = new MyHandler();
disp.invokeAsync(reqMsg, handler);
```

Note that asynchronous callback is also possible using JAXB objects instead of `Source`.

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):* For backwards compatibility, implementations that support use of the SOAP encoding SHOULD support the creation of a `javax.xml.rpc.Call` instance for ports that use SOAP encoding.

◇ *Conformance Requirement (`createCall` failure):* Implementations MUST throw a `ServiceException` from `Service.createCall` if the port is unsuitable for use with the `Call` API.

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.

	`Service.createCall` Arguments			
	None	`QName port`	`QName port, QName op`	`QName port, String op`
`createService` Arguments				
`QName svc`	U	U	U	U
`URL wsdlDoc, QName svc`	U	U	C	C
`loadService` Arguments				
`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.

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.

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 4.3.2 on page 42 for a list of standard properties)

The `JAXRPCContext` (see section 4.3) interface provides a common container interface for metadata that may be used with `Stub`, `Dispatch`, and `Call`. For backwards compatibility with JAX-RPC 1.1, the `Call` property methods are retained. Setting the value of a property in the request `JAXRPCContext` of a `Call` instance is equivalent to setting the value of the property directly on the `Call` instance.

◇ *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.

See section 7.4.1 for additional SOAP/HTTP requirements.

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 4.2 shows an example of using a configured `Call` instance

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

Figure 4.2: Use of configured `Call` instance

```
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 4.3: Use of unconfigured `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 4.3 shows an example of using an unconfigured `Call` instance to invoke the same operation as shown in figure 4.2.

Lines 3–5 in figure 4.3 are concerned with configuring the `Call` instance prior to its use in line 7.

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`, may be 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, an implementation MUST reflect the information contained in the `ProtocolException` subclass within the generated protocol level fault.

4.9.1.1 Client Side Example

```
try {
    response = dispatch.invoke(request);
}
catch (RemoteException e) {
    if (e.getCause() != null) {
        if (e.getCause() instanceof SOAPFaultException) {
            SOAPFaultException soapFault =
                (SOAPFaultException)e.getCause();
            QName soap11faultcode = soapFault.getFaultCode();
        }
        ...
    }
}
```

4.9.1.2 Server Side Example

```
public void endpointOperation() throws RemoteException {
    ...
    if (someProblem) {
        SOAPFaultException soapFault = new SOAPFaultException(faultcode,
            faultstring, faultactor, details);
        throw new RemoteException("An error occurred", soapFault);
    }
    ...
}
```

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.

Chapter 5

Service APIs

This chapter describes requirements on JAX-RPC service implementations and standard APIs provided for their use.

5.1 javax.xml.rpc.server.Provider

JAX-RPC services typically implement a native Java service endpoint interface (SEI) or an SEI mapped from a WSDL port type. Section 3.3 describes the requirements that a Java interface must meet to qualify as a JAX-RPC SEI. Section 2.2 describes the mapping from a WSDL port type to an equivalent Java SEI.

Java SEIs provide a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. However, in some cases it is desirable for services to be able to operate at the XML representation level. The `Provider` interface offers an alternative to SEIs and may be implemented by services wishing to work at the XML level.

◇ *Conformance Requirement (Provider support required):* An implementation MUST support `Provider` based service endpoint implementations.

◇ *Conformance Requirement (Provider default constructor):* A `Provider` based service endpoint implementation MUST provide a default constructor.

◇ *Conformance Requirement (Provider implementation):* A `Provider` based service endpoint implementation MUST implement the `Provider` interface.

`Provider` is a low level API that requires services to work with message payloads as XML fragments and hence requires an intimate knowledge of the desired payload structure. `Provider` works in conjunction with `LogicalMessageContext` to offer isolation from protocol level constructs and access to message payloads as either XML or using JAXB APIs.

5.1.1 Invocation

A `Provider` based service instances `invoke` method is called for each message received for the service. The `LogicalMessageContext` parameter may be used to examine the inbound message and associated context and to set an outbound reply message and associated context. The `invoke` method returns `true` to indicate that a response message has been set in the context, `false` otherwise.

Inbound and outbound messages are considered to be separate entities. Setting a reply message in the `LogicalMessageContext` implicitly results in the creation of a new protocol specific message where appropriate.

An `invoke` implementation can throw `ProtocolException` to generate a protocol specific fault or `JAXRPCException`. The JAX-RPC runtime behaviour is the same as described for a handler throwing the exception from its handleMessage method, see section 6.3.2.

5.1.2 Lifecycle

The lifecycle of a `Provider` based service is identical to that of an SEI based service, see section 5.2.3 for details of the lifecycle of services hosted in a servlet container.

5.1.3 Configuration

The JAX-RPC runtime makes certain properties available to a service instance that can be used to determine its configuration, see section 6.4.1.1.

5.1.4 Examples

For clarity, error handling is omitted in the following examples.

Simple echo service, reply message contains the same payload as the input message.

```
public class MyService implements Provider {
    public MyService {
    }

    public boolean invoke(LogicalMessageContext context) {
        return true;
    }
}
```

Simple static reply, reply message contains a fixed acknowlegment element.

```
public class MyService implements Provider {
    public MyService {
    }

    public boolean invoke(LogicalMessageContext context) {
        Source request = context.getMessage();
        ...
        String replyElement = new String("<n:ack xmlns:n=’...’/>");
        StreamSource reply = new StreamSource(new StringReader(replyElement));
        context.setMessage(reply);
        return true;
    }
}
```

Using JAXB to read the input message and set the reply.

```
public class MyService implements Provider {
    public MyService {
    }

    public boolean invoke(LogicalMessageContext context) {
        JAXBContent jc = JAXBContext.newInstance(...);
        Object request = context.getMessage(jc);
        ...
        Acknowledgement reply = new Acknowledgement(...);
        context.setMessage(reply,jc);
        return true;
    }
}
```

5.2 Servlet Based Services

This section describes requirements specific to Java Servlet[28] hosted JAX-RPC services.

5.2.1 Requirements on Service Implementations

For a servlet based service, the following requirements apply to the service implementation class:

- It must implement a service endpoint interface, see sections 2.2 and 3.3 for a description of service endpoint interfaces.
- It must have a default public constructor.
- It may implement `javax.xml.rpc.server.ServiceLifecycle`, see section 5.2.3, to manage its lifecycle.
- It may obtain references to resources and enterprise beans using JNDI.

5.2.2 Servlet Association

During deployment, a service implementation class is associated with a servlet. The servlet is provided by the JAX-RPC runtime system and the association is configured in an implementation specific manner. Implementations may associate multiple service implementation classes with a single servlet or may associate each service implementation class with its own dedicated servlet.

The servlet implementation brokers requests and responses between the servlet container and the JAX-RPC runtime. Each `Servlet.service` method is mapped to an invocation of an SEI method on a service implementation class instance. The JAX-RPC runtime inherits the thread model of the servlet.

♢ *Conformance Requirement (One way operations):* For a HTTP based endpoint, processing an incoming request for a one-way operation, an implementation MUST respond with HTTP response code of 200 or 202 as soon as it has identified the incoming request as being one-way and before it dispatches it to a service implementation class instance.

5.2.3 javax.xml.rpc.server.ServiceLifecycle

A service implementation class may optionally implement `ServiceLifecycle` to be notified of lifecycle state changes. For service implementations that implement `ServiceLifecycle`, the JAX-RPC runtime system manages their lifecycle by invoking the `init` and `destroy` methods of `ServiceLifecycle`. Figure 5.1 shows a state transition diagram for the lifecycle of a service implementation instance.

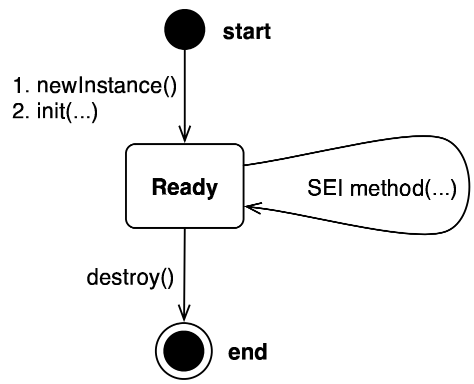

Figure 5.1: Service Lifecycle.

The JAX-RPC runtime system is responsible for loading the service implementation class and instantiating the corresponding service implementation object. The lifecycle of a service implementation instance begins when the JAX-RPC runtime system creates a new instance of the corresponding class and invokes the `ServiceLifecycle.init` method.

◇ *Conformance Requirement (Service initialization):* An implementation is required to call the `init` method of `ServiceLifecycle` prior to invoking any other method on a service implementation instance.

Once the service implementation instance is created and initialized it is placed into the `Ready` state. While in the `Ready` state the JAX-RPC runtime system may invoke SEI methods as required. A JAX-RPC runtime system may load and initialize multiple instances of a service implementation class, each such instance is considered equivalent and a runtime system may invoke a given SEI method on any available instance. A runtime system may invoke multiple SEI methods on the same instance in separate threads.

The lifecycle of a service implementation instance ends when the JAX-RPC runtime system invokes the `ServiceLifecycle.destroy` method.

◇ *Conformance Requirement (Service destruction):* An implementation MUST call the `destroy` method of an instance implementing `ServiceLifecycle` prior to releasing a service implementation instance.

◇ *Conformance Requirement (Service destruction):* An implementation MUST NOT call the `destroy` method of an instance implementing `ServiceLifecycle` while an SEI method is executing is a different thread.

The service implementation instance must release its resources and perform cleanup in the implementation of the `destroy` method. After invocation of the `destroy` method, the service implementation instance will be made available for garbage collection.

◇ *Conformance Requirement (Handler exceptions):* If a service implementation instance throws a `RuntimeException` that is not a subclass of `ProtocolException` (see section 4.9.1), an implementation MUST call its `destroy` method and release it.

5.2.4 javax.xml.rpc.server.ServletEndpointContext

The `ServiceLifecycle.init` method takes a `context` parameter of type `Object`. The actual type of the `context` depends on the type of container into which a service implementation class is deployed.

♢ *Conformance Requirement (Context type):* For a service implementation class deployed in a servlet container, the type of the `context` parameter passed in the `ServiceLifecycle.init` method MUST be `javax.xml.rpc.server.ServletEndpointContext`.

A service implementation class may store the supplied `context` and call its methods during an SEI method invocation.

♢ *Conformance Requirement (Context Implementation Required):* A servlet based JAX-RPC runtime system MUST implement the `ServletEndpointContext` interface.

The following subsections describe the capabilities of `ServletEndpointContext`.

5.2.4.1 HTTP Session

The `getHTTPSession` method returns the current `HTTPSession` instance. When called by a service implementation class instance during an SEI invocation it returns the HTTP session associated with the current request. This method returns `null` if there is no current HTTP session. `getHTTPSession` throws `JAXRPCException` for a non-HTTP based servlet endpoint.

5.2.4.2 Message Context

When called by a service implmentation class instance during an SEI invocation, the `getMessageContext` method returns the message context for the current request as processed by the handler framework, see section 6.4. The type of the message context depends on the protocol binding in use. `getMessageContext` throws `IllegalStateException` if called outside of an SEI invocation.

The message context may be used to inspect and change message context properties and, depending on the type of message context, may also be used to examine the inbound message after processing by handlers.

Editors Note 5.1 *Need to specify the rules for what type of message context is made available here – see issue 12.*

5.2.4.3 Servlet Context

The `getServletContext` method returns the `ServletContext` for the servlet that the service implementation class is associated with, see section 5.2.2.

5.2.4.4 Security

The `getUserPrinciple` and `isUserInRole` methods provide access to the security principle and roles for the currently authenticated user respectively.

5.2.5 Packaging and Deployment

This specification does not define a normative model for packaging and deployment of Java Servlet[28] hosted JAX-RPC services. Refer to Web Services for J2EE[14] for a normative model for packaging and deployment of J2EE based JAX-RPC services.

5.3 EJB Based Services

This specification does not define the service endpoint model for JAX-RPC services developed using the EJB component model. Refer to Web Services for J2EE[14] for full details.

Chapter 6

Handler Framework

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. This chapter describes the handler framework in detail.

◇ *Conformance Requirement (Handler framework support):* An implementation MUST support the handler framework.

6.1 Architecture

The handler framework is implemented by a JAX-RPC protocol binding in both client and server side runtimes. Stubs, ties, dynamic proxies, `Dispatch`, and `Call` instances, known collectively as binding providers, each use protocol bindings to bind their abstract functionality to specific protocols (see figure 6.1). Protocol bindings can extend the handler framework to provide protocol specific functionality; chapter 7 describes the JAX-RPC SOAP binding that extends the handler framework with SOAP specific functionality.

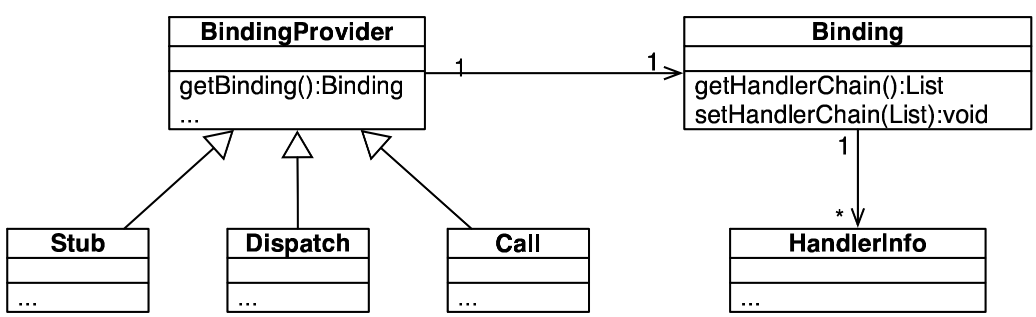

Figure 6.1: Handler architecture

Client and server-side handlers are organized into an ordered list known as a handler chain. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers prior to binding provider processing. Outbound messages are processed by handlers after any binding provider processing.

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties may be used to facilitate communication between individual handlers and between handlers and client and service implementations. Different types of handlers are invoked with different types of message context.

6.1.1 Types of Handler

JAX-RPC 2.0 defines two types of handler:

Logical Handlers that only operate on message context properties and message payloads. Logical handlers are protocol agnostic and are unable to affect protocol specific parts of a message. Logical handlers are handlers that implement `javax.xml.rpc.handler.LogicalHandler`.

Protocol Handlers that operate on message context properties and protocol specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol specific aspects of a message. Protocol handlers are handlers that implement `javax.xml.rpc.handler.Handler` or any sub interface of `javax.xml.rpc.handler.AbstractHandler` except `javax.xml.rpc-.handler.LogicalHandler`.

Figure 6.2 shows the class hierarchy for handlers.

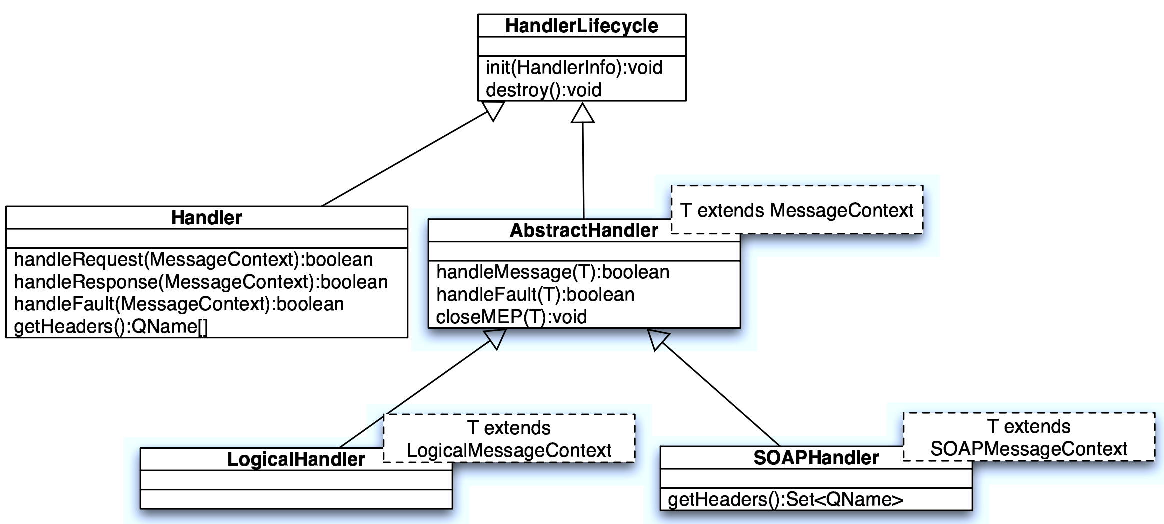

Figure 6.2: Handler class hierarchy

Handlers for protocols other than SOAP are expected to implement an interface that extends `javax.xml-.rpc.handler.AbstractHandler`. JAX-RPC 1.1 defined `javax.xml.rpc.handler.Handler` for SOAP handlers and this is retained for backwards compatibility – new SOAP handlers should implement `SOAPHandler` instead.

6.1.2 Binding Responsibilities

The following subsections describe the responsibilities of the protocol binding when hosting a handler chain.

6.1.2.1 Handler and Message Context Management

The binding is responsible for instantiation, invocation, and destruction of handlers according to the rules specified in section 6.3. The binding is responsible for instantiation and management of message contexts according to the rules specified in section 6.4

◇ *Conformance Requirement (Logical handler support):* All binding implementations MUST support logical handlers (see section 6.1.1) being deployed in their handler chains.

◇ *Conformance Requirement (Other handler support):* Binding implementations MAY support other handler types (see section 6.1.1) being deployed in their handler chains.

6.1.2.2 Message Dispatch

The binding is responsible for dispatch of both outbound and inbound messages after handler processing. Outbound messages are dispatched using whatever means the protocol binding uses for communication. Inbound messages are dispatched to the binding provider. JAX-RPC defines no standard interface between binding providers and their binding.

6.1.2.3 Exception Handling

The binding is responsible for catching runtime exceptions thrown by handlers and respecting any resulting message direction and message type change as described in section 6.3.2.

Outbound exceptions[1] are converted to protocol fault messages and dispatched using whatever means the protocol binding uses for communication. Specific protocol bindings describe the mechanism for their particular protocol, section 7.2.2 describes the mechanism for the SOAP 1.1 binding. Inbound exceptions are passed to the binding provider.

6.2 Configuration

Handler chains may be configured either programmatically or using deployment metadata. The following subsections describe each form of configuration.

6.2.1 Programmatic Configuration

JAX-RPC only defines APIs for programmatic configuration of client side handler chains – server side handler chains are expected to be configured using deployment metadata.

6.2.1.1 javax.xml.rpc.handler.HandlerRegistry

A `Service` instance maintains a handler registry that is referred to when creating stubs, dynamic proxies, `Dispatch` or `Call` instances, known collectively as binding providers. During creation, the registered handlers are added to the binding for the new binding provider. A `Service` instance provides access to a `HandlerRegistry`, via the `Service.getHandlerRegistry` method. The `HandlerRegistry` may

[1] Outbound exceptions are exceptions thrown by a handler that result in the message direction being set to outbound according to the rules in section 6.3.2.

be used to configure handler chains on a per-service, per-port or per-protocol binding basis. Per-service handlers are added to the binding of all created binding providers. Per-port handlers are added to the binding of all binding providers created for a specified port. Per-binding protocol handlers are added to the binding of all binding providers created that use a specific binding type (e.g., SOAP over HTTP).

When a `Service` instance is used to create an instance of a binding provider then the created instance is configured with a snapshot of the applicable handlers configured on the `Service` instance.

◇ *Conformance Requirement (Handler chain snapshot):* Changes to the handlers configured for a `Service` instance MUST NOT affect the handlers on previously created stubs, dynamic proxies, `Dispatch` or `Call` instances.

6.2.1.2 Handler Ordering

The handler chain for a binding is constructed by merging the applicable per-service, per-port or per-protocol binding chains configured for the service instance. The resulting handler order is:

(i) Per-service logical handlers,

(ii) Per-port logical handlers,

(iii) Per-protocol binding logical handlers.

(iv) Per-service protocol handlers,

(v) Per-port protocol handlers,

(vi) Per-protocol binding protocol handlers.

The order of handlers of any given type within a per-service, per-port or per-protocol binding chain is maintained. Figure 6.3 illustrates this.

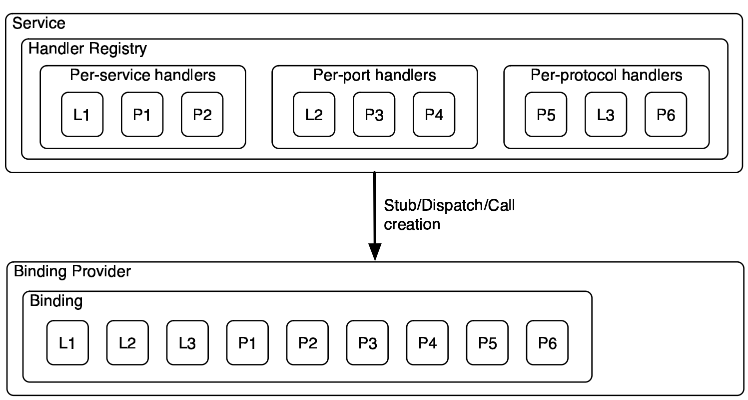

Figure 6.3: Handler ordering, L*n* and P*n* represent logical and protocol handlers respectively.

Section 6.3.2 describes how the handler order relates to the order of handler execution for inbound and outbound messages.

Editors Note 6.1 *Some expert group members expressed a desire to be able to intersperse logical and protocol handlers. The expert groups seeks feedback on the current proposed ordering.*

6.2.1.3 javax.xml.rpc.handler.HandlerInfo

The `HandlerInfo` class represents the configuration data for a specific handler. A handler is passed a `HandlerInfo` instance during initialization (see section 6.3.1).

`HandlerInfo` contains the following properties:

Handler Class The class that implement the handler.

Handler Configuration Configuration information in the form of a `Map`.

Headers Array of `QName` that describes the protocol elements processed by the handler. This is only applicable to protocol handlers for protocols that support such processing.

6.2.1.4 javax.xml.rpc.Binding

The `Binding` interface is an abstraction of a JAX-RPC protocol binding (see section 4.4 for more details). As described above, the handler chain initially configured on an instance is a snapshot of the applicable handlers configured on the `Service` instance at the time of creation. `Binding` provides methods to manipulate the initially configured handler chain for a specific instance.

♢ *Conformance Requirement (Binding handler manipulation):* Changing the handler chain on a `Binding` instance MUST NOT cause any change to the handler chains configured on the `Service` instance used to create the `Binding` instance.

6.2.2 Deployment Model

JAX-RPC defines no standard deployment model for handlers. Such a model is provided by JSR 109[14] “Implementing Enterprise Web Services”.

6.3 Processing Model

This section describes the processing model for handlers within the handler framework.

6.3.1 Handler Lifecycle

The JAX-RPC runtime system manages the lifecycle of handlers by invoking the `init` and `destroy` methods of `HandlerLifecycle`. Figure 6.4 shows a state transition diagram for the lifecycle of a handler.

The JAX-RPC runtime system is responsible for loading the handler class and instantiating the corresponding handler object. The lifecycle of a handler instance begins when the JAX-RPC runtime system creates a new instance of the handler class and invokes the `HandlerLifecycle.init` method.

♢ *Conformance Requirement (Handler initialization):* An implementation is required to call the `init` method of `HandlerLifecycle` prior to invoking any other method on a handler instance.

Once the handler instance is created and initialized it is placed into the `Ready` state. While in the `Ready` state the JAX-RPC runtime system may invoke other handler methods as required. The lifecycle of a handler instance ends when the JAX-RPC runtime system invokes the `HandlerLifecycle.destroy` method.

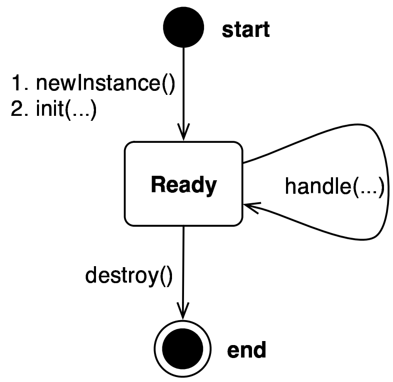

Figure 6.4: Handler Lifecycle.

◊ *Conformance Requirement (Handler destruction):* An implementation MUST call the `destroy` method of `HandlerLifecycle` prior to releasing a handler instance.

The handler instance must release its resources and perform cleanup in the implementation of the `destroy` method. After invocation of the `destroy` method, the handler instance will be made available for garbage collection.

◊ *Conformance Requirement (Handler exceptions):* If a handler instance throws a `RuntimeException` that is not a subclass of `ProtocolException`, an implementation MUST call its `destroy` method and release it.

6.3.2 Handler Execution

As described in section 6.2.1.2, a set of handlers is managed by a binding as an ordered list called a handler chain. Unless modified by the actions of a handler (see below) normal processing involves each handler in the chain being invoked in turn. Each handler is passed a message context (see section 6.4) whose contents may be manipulated by the handler.

For outbound messages handler processing starts with the first handler in the chain and proceeds in the same order as the handler chain. For inbound messages the order of processing is reversed: processing starts with the last handler in the chain and proceeds in the reverse order of the handler chain. E.g., consider a handler chain that consists of six handlers $H_1 \ldots H_6$ in that order: for outbound messages handler H_1 would be invoked first followed by H_2, H_3, $\ldots$, and finally handler H_6; for inbound messages H_6 would be invoked first followed by H_5, H_4, $\ldots$, and finally H_1.

In the following discussion the terms next handler and previous handler are used. These terms are relative to the direction of the message, table 6.1 summarizes their meaning.

Handlers may change the direction of messages and the order of handler processing by throwing an exception or by returning `false` from `handleMessage`, `handleRequest`, `handleResponse` or `handleFault`. The following subsections describe each handler method and the changes to handler chain processing they may cause.

Message Direction	Term	Handler
Inbound	Next	H_{i-1}
	Previous	H_{i+1}
Outbound	Next	H_{i+1}
	Previous	H_{i-1}

Table 6.1: Next and previous handlers for handler H_i.

6.3.2.1 `handleMessage`, `handleRequest` and `handleResponse`

These methods are called for normal message processing. `handleMessage` is called for a hander that implements `AbstractHandler` or a subinterface thereof. `handleRequest` is called for a hander that implements `Handler` for outbound messages on the client side and for inbound messages on the server side. `handleResponse` is called for a hander that implements `Handler` for outbound messages on the server side and for inbound messages on the client side.

Following completion of its work the `handleMessage`, `handleRequest` or `handleResponse` implementation can do one of the following:

Return `true` This indicates that normal message processing should continue. The runtime invokes one of `handleMessage`, `handleRequest` or `handleResponse` on the next handler or dispatches the message (see section 6.1.2.2) if there are no further handlers.

Return `false` This indicates that normal message processing should cease. Subsequent actions depend on whether the message exchange pattern (MEP) in use requires a response to the *message currently being processed*[2] or not:

- **Response** The message direction is reversed, the runtime invokes one of `handleMessage` or `handleResponse` on the current or next[3] handler or dispatches the message (see section 6.1.2.2) if there are no further handlers. For backwards compatibility, the current handler is invoked if it implements the `Handler` interface, otherwise the next handler is invoked.

- **No response** Normal message processing stops, only `close` is called for other handlers in the chain, the message is dispatched (see section 6.1.2.2).

Throw `ProtocolException` or a subclass This indicates that normal message processing should cease. Subsequent actions depend on whether the MEP in use requires a response to the message currently being processed or not:

- **Response** Normal message processing stops, fault message processing starts. The message direction is reversed, if the message is not already a fault message then it is replaced with a fault message[4], and the runtime invokes `handleFault` on the current or next[4] handler or dispatches the message (see section 6.1.2.2) if there are no further handlers. For backwards compatibility, the current handler is invoked if it implements the `Handler` interface, otherwise the next handler is invoked.

- **No response** Normal message processing stops, only `close` is called for other handlers in the chain, the exception is dispatched (see section 6.1.2.3).

[2]For a request-response MEP, if the message direction is reversed during processing of a request message then the message becomes a response message. Subsequent handler processing takes this change into account.

[3]Next in this context means the next handler taking into account the message direction reversal

[4]The handler may have already converted the message to a fault message, in which case no change is made.

Throw any other runtime exception This indicates that normal message processing should cease. Subsequent actions depend on whether the MEP in use includes a response to the message currently being processed or not:

Response Normal message processing stops, only `close` is called for other handlers in the chain, the message direction is reversed, and the exception is dispatched (see section 6.1.2.3).

No response Normal message processing stops, only `close` is called for other handlers in the chain, the exception is dispatched (see section 6.1.2.3).

6.3.2.2 `handleFault`

Called for fault message processing, following completion of its work the `handleFault` implementation can do one of the following:

Return `true` This indicates that fault message processing should continue. The runtime invokes `handleFault` on the next handler or dispatches the fault message (see section 6.1.2.2) if there are no further handlers.

Return `false` This indicates that fault message processing should cease. Fault message processing stops, only `close` is called for other handlers in the chain, the fault message is dispatched (see section 6.1.2.2).

Throw `ProtocolException` or a subclass This indicates that fault message processing should cease. Fault message processing stops, only `close` is called for other handlers in the chain, the exception is dispatched (see section 6.1.2.3).

Throw any other runtime exception This indicates that fault message processing should cease. Fault message processing stops, only `close` is called for other handlers in the chain, the exception is dispatched (see section 6.1.2.3).

6.3.2.3 `close`

A handler's `close` method is called at the conclusion of a message exchange pattern (MEP). It is called just prior to the binding dispatching the final message, fault or exception of the MEP and may be used to clean up per-MEP resources allocated by a handler. The `close` method is only called on handlers that were previously invoked via either `handleMessage` or `handleFault`

◇ *Conformance Requirement (Invoking `close`):* At the conclusion of an MEP, an implementation MUST call the `close` method of each handler that was previously invoked during that MEP via either `handleMessage` or `handleFault`.

◇ *Conformance Requirement (Order of `close` invocations):* Handlers are invoked in the same order that they appear in the handler chain.

6.3.3 Handler Implementation Considerations

Handler instances may be pooled by a JAX-RPC runtime system. All instances of a specific handler are considered equivalent by a JAX-RPC runtime system and any instance may be chosen to handle a particular message. Different handler instances may be used to handle each messages of an MEP. Different threads may be used for each handler in a handler chain, for each message in an MEP or any combination of the two. Handlers should not rely on thread local state to share information. Handlers should instead use the message context, see section 6.4.

6.4 Message Context

Handlers are invoked with a message context that provides methods to access and modify inbound and outbound messages and to manage a set of properties.

Different types of handler are invoked with different types of message context. Sections 6.4.1 and 6.4.2 describe `MessageContext` and `LogicalMessageContext` respectively. In addition, JAX-RPC bindings may define a message context subtype for their particular protocol binding that provides access to protocol specific features. Section 7.3 describes the message context subtype for the JAX-RPC SOAP binding.

6.4.1 javax.xml.rpc.handler.MessageContext

`MessageContext` is the super interface for all JAX-RPC message contexts. It provides methods to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the `setProperty` method to set a value for a specific property in the message context that one or more other handlers in the handler chain may subsequently obtain via the `getProperty` method.

Message context instances are scoped to all handlers of a specific type for an instance of an MEP on a particular endpoint. E.g., all logical handlers will be passed the same message context instance during the execution of a particular MEP instance.

◇ *Conformance Requirement (Message context scope):* A message context instance MUST be shared across all handlers of the same type for a particular instance of an MEP on any particular endpoint.

Properties are scoped to an instance of an MEP on a particular endpoint. E.g., if a logical handler sets the value of a message context property, that property will also be available to any protocol handlers in the chain during the execution of an MEP instance.

◇ *Conformance Requirement (Message context property scope):* Properties in a message context MUST be shared across all handler invocations for a particular instance of an MEP on any particular endpoint.

6.4.1.1 Standard Message Context Properties

Editors Note 6.2 *This section describes a small number of standard properties. Others may be added to this list. The expert group seeks feedback on suitable additional candidate properties.*

The following standard properties are defined:

`javax.xml.rpc.handler.message.outbound` This property specifies the message direction and is of type boolean. The value of this property is `true` for outbound messages, `false` for inbound messages.

`javax.xml.rpc.handler.context.request` See section 6.4.3.

`javax.xml.rpc.handler.context.response` See section 6.4.3.

javax.xml.rpc.service.provider.wsdl Provides access to the WSDL for the endpoint. Type is `InputStream`.

javax.xml.rpc.service.provider.service The name of the service being invoked in the WSDL. Type is `QName`.

javax.xml.rpc.service.provider.port The name of the port over which the current message was received in the WSDL. Type is `QName`.

◊ *Conformance Requirement (Additional `MessageContext` properties):* Implementations MAY support additional implementation specific properties not listed above. Such properties MUST NOT use the `javax.xml.rpc` prefix in their names.

Implementation specific properties are discouraged as they limit application portability. Handlers can interact using application specific properties. User defined properties must not use the prefix `javax.xml.rpc` in their names.

6.4.2 javax.xml.rpc.handler.LogicalMessageContext

Logical handlers (see section 6.1.1) are passed a message context of type `LogicalMessageContext` when invoked. `LogicalMessageContext` extends `MessageContext` with methods to obtain and modify the message payload, it does not provide access to the protocol specific aspects of a message. A protocol binding defines what component of a message are available via a logical message context. E.g., the SOAP binding, see section 7.1.1.2, defines that a logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers.

6.4.3 Relationship to JAXRPCContext

Client side binding providers have methods to access `JAXRPCContext` (see section 4.3) instances for outbound and inbound messages. These contexts are made available to handlers as message context properties named as follows:

javax.xml.rpc.handler.context.request The value of this property corresponds to the value of `BindingProvider.getRequestContext`. Its type is `JAXRPCContext`.

javax.xml.rpc.handler.context.response The value of this property corresponds to the value of `BindingProvider.getResponseContext`. Its type is `JAXRPCContext`.

Handlers may manipulate the values of properties within these two contexts and these properties are made available to client applications. 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.

Chapter 7

SOAP Binding

This chapter describes the JAX-RPC SOAP binding and its extensions to the handler framework (described in chapter 6) for SOAP message processing.

7.1 Configuration

A SOAP binding instance requires SOAP specific configuration in addition to that described in section 6.2. The additional information can be configured either programmatically or using deployment metadata. The following subsections describe each form of configuration.

7.1.1 Programmatic Configuration

JAX-RPC only defines APIs for programmatic configuration of client side SOAP bindings – server side bindings are expected to be configured using deployment metadata.

7.1.1.1 SOAP Roles

SOAP 1.1[2] and SOAP 1.2[3, 4] use different terminology for the same concept: a SOAP 1.1 *actor* is equivalent to a SOAP 1.2 *role*. This specification uses the SOAP 1.2 terminology.

An ultimate SOAP receiver always plays the following roles:

Next In SOAP 1.1, the next role is identified by the URI http://schemas.xmlsoap.org/soap/actor/next. In SOAP 1.2, the next role is identified by the URI http://www.w3.org/2003/05/soap-envelope/role/next.

Ultimate receiver In SOAP 1.1 the ultimate receiver role is identified by omission of the `actor` attribute from a SOAP header. In SOAP 1.2 the ultimate receiver role is identified by the URI http://www.w3-.org/2003/05/soap-envelope/role/ultimateReceiver or by omission of the `role` attribute from a SOAP header.

♢ *Conformance Requirement (SOAP required roles):* An implementation of the SOAP binding MUST act in the following roles: next and ultimate receiver.

A SOAP 1.2 endpoint never plays the following role:

None In SOAP 1.2, the none role is identified by the URI http://www.w3.org/2003/05/soap-envelope/role-/none.

◇ *Conformance Requirement (SOAP required roles):* An implementation of the SOAP binding MUST NOT act in the none role.

The `javax.xml.rpc.SOAPBinding` interface is an abstraction of the JAX-RPC SOAP binding. It extends `javax.xml.rpc.Binding` with methods to configure additional SOAP roles played by the endpoint.

◇ *Conformance Requirement (Default role visibility):* An implementation MUST include the required next and ultimate receiver roles in the `Set` returned from `SOAPBinding.getRoles`.

◇ *Conformance Requirement (Default role persistence):* An implementation MUST add the required next and ultimate receiver roles to the roles configured with `SOAPBinding.setRoles`.

◇ *Conformance Requirement (None role error):* An implementation MUST throw `JAXRPCException` if a client attempts to configure the binding to play the none role via `SOAPBinding.setRoles`.

7.1.1.2 SOAP Handlers

The handler chain for a SOAP binding is configured as described in section 6.2.1. The handler chain may contain handlers of the following types:

Logical Logical handlers are handlers that implement `javax.xml.rpc.handler.LogicalHandler` either directly or indirectly. Logical handlers have access to the context of the SOAP body via the logical message context.

SOAP SOAP handlers are handlers that implement `javax.xml.rpc.handler.Handler` or `javax-.xml.rpc.handler.soap.SOAPHandler`.

◇ *Conformance Requirement (Incompatible handlers):* An implementation MUST either: (i) throw `JAXRPC-Exception` when attempting to configure an incompatible handler using the `setHandlerChain` method of `HandlerRegistry` or (ii) throw `ServiceException` when attempting to create a binding provider using one of the `Service` methods when an incompatible handler has been configured.

◇ *Conformance Requirement (Incompatible handlers):* Implementations MUST throw a `JAXRPCException` when attempting to configure an incompatible handler using `Binding.setHandlerChain`.

◇ *Conformance Requirement (Logical handler access):* An implementation MUST allow access to the contents of the SOAP body via a logical message context.

7.1.1.3 SOAP Headers

The SOAP headers processed by a handler are configured using the `HandlerInfo` for the handler (see section 6.2.1.3).

7.1.2 Deployment Model

JAX-RPC defines no standard deployment model for handlers. Such a model is provided by JSR 109[14] “Implementing Enterprise Web Services”.

7.2 Processing Model

The SOAP binding implements the general handler framework processing model described in section 6.3 but extends it to include SOAP specific processing as described in the following subsections.

7.2.1 SOAP `mustUnderstand` Processing

The SOAP protocol binding performs the following additional processing on inbound SOAP messages prior to the start of normal handler invocation processing (see section 6.3.2). Refer to the SOAP specification[2, 3, 4] for a normative description of the SOAP processing model. This section is not intended to supercede any requirement stated within the SOAP specification, but rather to outline how the configuration information described above is combined to satisfy the SOAP requirements:

1. Obtain the set of SOAP roles for the current binding instance. This is returned by `SOAPBinding-.getRoles`.
2. Obtain the set of `HandlerInfos` deployed on the current binding instance. This is returned by `Binding.getHandlerChain`.
3. Identify the set of header qualified names (QNames) that the binding instance understands. This is the set of all header `QNames`:
   (a) corresponding to parameters mapped to the operation by the original WSDL, and
   (b) obtained from `HandlerInfo.getHeaders()` for each `HandlerInfo` in the set obtained in step 2.
4. Identify the set of must understand headers in the inbound message that are targeted at this node. This is the set of all headers with a `mustUnderstand` attribute whose value is `1` or `true` and an `actor` or `role` attribute whose value is in the set obtained in step 1.
5. For each header in the set obtained in step 4, the header is understood if its QName is in the set identified in step 3.
6. If every header in the set obtained in step 4 is understood, then the node understands how to process the message. Otherwise the node does not understand how to process the message.
7. If the node does not understand how to process the message, then neither handlers nor the endpoint are invoked and instead the binding generates a SOAP must understand exception. Subsequent actions depend on whether the message exchange pattern (MEP) in use requires a response to the message currently being processed or not:

   Response The message direction is reversed and the binding dispatches the SOAP must understand exception (see section 7.2.2).

   No response The binding dispatches the SOAP must understand exception (see section 7.2.2).

7.2.2 Exception Handling

A binding is responsible for catching runtime exceptions thrown by handlers and following the processing model described in section 6.3.2. A binding is responsible for converting the exception to a fault message subject to further handler processing if the following criteria are met:

1. A handler throws a `ProtocolException` from `handleMessage`, `handleRequest` or `handleResponse`
2. The MEP in use includes a response to the message being processed
3. The current message is not already a fault message (the handler might have undertaken the work prior to throwing the exception).

If the above criteria are met then the exception is converted to a SOAP fault message as follows:

- If the exception is an instance of `SOAPFaultException` then the fields of the exception are serialized to a new SOAP fault message. The current message is replaced by the new SOAP fault message.
- If the exception is of any other type then a new SOAP fault message is created to reflect a server class of error for SOAP 1.1[2] or a receiver class of error for SOAP 1.2[3] .

Handler processing then resumes as described in section 6.3.2.

If the criteria for converting the exception to a fault message subject to further handler processing are not met then the exception is handled as follows depending on the current message direction:

Outbound A new SOAP fault message is created to reflect a server class of error for SOAP 1.1[2] or a receiver class of error for SOAP 1.2[3] and the message is dispatched.

Inbound The exception is passed to the binding provider.

7.3 SOAP Message Context

SOAP handlers are passed a `SOAPMessageContext` when invoked. `SOAPMessageContext` extends `MessageContext` with methods to obtain and modify the SOAP message payload.

7.4 SOAP Transport and Transfer Bindings

SOAP[2, 4] can be bound to multiple transport or transfer protocols. This section describes requirements pertaining to the supported protocols for use with SOAP.

7.4.1 HTTP

◊ *Conformance Requirement (SOAP 1.1 HTTP Binding Support):* An implementation MUST support the HTTP binding of SOAP 1.1[2] as clarified by the WS-I Basic Profile[17].

◊ *Conformance Requirement (SOAP 1.2 HTTP Binding Support):* An implementation MUST support the HTTP binding of SOAP 1.2[4].

7.4.1.1 One-way Operations

HTTP interactions are request-response in nature. When using HTTP as the transfer protocol for a one-way SOAP message, implementations wait for the HTTP response even though there is no SOAP message in the HTTP response entity body.

◇ *Conformance Requirement (One-way operations):* When invoking one-way operations, an implementation of the SOAP/HTTP binding MUST block until the HTTP response is received or an error occurs.

Note that completion of the HTTP request simply means that the transmission of the request is complete, not that the request was accepted or processed.

7.4.1.2 Security

Section 4.3.2 defines two standard `JAXRPCContext` properties (`javax.xml.rpc.security.username` and `javax.xml.rpc.security.password`) that may be used to configure authentication information.

◇ *Conformance Requirement (HTTP basic authentication support):* An implementation of the SOAP/HTTP binding MUST support HTTP basic authentication.

◇ *Conformance Requirement (Authentication properties):* A client side implementation MUST support use of the the standard properties `javax.xml.rpc.security.username` and `javax.xml.rpc.security-.password` to configure HTTP basic authentication.

7.4.1.3 Session Management

Section 4.3.2 defines a standard `JAXRPCContext` propertu (`javax.xml.rpc.session.maintain`) that may be used to control whether a client side runtime will join a session initiated by a service.

A SOAP/HTTP binding implementation can use three HTTP mechanisms for session management:

Cookies To initiate a session a service includes a cookie in a message sent to a client. The client stores the cokkie and returns it in subsequest messages to the service.

URL rewriting To initiate a session a service directs a client to a new URL for subsequent interactions. The new URL contains an encoded session identifier.

SSL The SSL session ID is used to track a session.

R1120 in WS-I Basic Profile 1.1[17] allows a service to use HTTP cookies. However, R1121 recommends that a service should not rely on use of cookies for state management.

◇ *Conformance Requirement (URL rewriting support):* An implementation MUST support use of HTTP URL rewriting for state management.

◇ *Conformance Requirement (Cookie support):* An implementation SHOULD support use of HTTP cookies for state management.

◇ *Conformance Requirement (SSL session support):* An implementation MAY support use of SSL session based state management.

Chapter 8

WSDL 2.0 to Java Mapping

This chapter describes the mapping from WSDL 2.0 to Java. This mapping is used when generating web service interfaces for clients and endpoints from a WSDL 2.0 document.

◇ *Conformance Requirement (WSDL 2.0 support):* Implementations MUST support mapping WSDL 2.0 to Java.

The following sections describe the default mapping from each WSDL 2.0 construct to the equivalent Java construct. The basis for this mapping is the WSDL 2.0 Working Draft dated ”3 August 2004” ([11], [18], [19]), which was adopted by the W3C Web Services Description Working Group as a Last Call Draft.

The mapping of a WSDL 2.0 document to a set of Java artifacts is described in terms of the WSDL 2.0 component model. This approach is in the spirit of the WSDL 2.0 specification. It has the benefit of freeing the mapping from any dependencies on the serialized form of a WSDL 2.0 document, opening the door for supporting alternative serializations in the future.

◇ *Conformance Requirement (WSDL 2.0 processor conformance):* Implementations MUST be conformant WSDL 2.0 processors.

In the rest of this chapter, ”WSDL” refers to ”WSDL 2.0”. The ”wsdl” and ”wrpc” namespace prefixes are assumed to be bound per the WSDL 2.0 specification (i.e. to the ”http://www.w3.org/2004/08/wsdl” and ”http://www.w3.org/2004/08/wsdl/rpc” namespaces respectively).

8.1 Definitions

A Definitions component acts as a container for the Interface, Binding and Service components, either defined directly in the WSDL document being processed or imported from another WSDL document.

In order to facilitate the mapping of its sub-components, while processing a Definitions component it is required that a mapping of namespace names to Java packages be in effect. Such a mapping MUST associate one Java package name to every namespace name that is the value of the {target namespace} property of an Interface or Service component reachable from the Definitions component itself.

There is no standard mapping from the value of a namespace name 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 namespace name that appears as the value of the {target namespace} property of an Interface or Service component reachable from the Definitions component.

8.2 Extensibility

WSDL has an open content model, allowing extensibility via elements and attributes. Besides the `wrpc:signature` predefined extension, JAX-RPC does not specify the mapping of extensibility elements or attributes, nor of any component model properties derived from them according to the specification for the extensions in question.

WSDL also supports a general extensibility facility in the form of Feature and Property components. JAX-RPC presently ignores these components. Naturally, JAX-RPC tools that process WSDL documents MUST obey the processor conformance requirements described in the WSDL specification; in particular, they MUST honor all rules concerning mandatory extensions, including features and properties.

◇ *Conformance Requirement (WSDL extensions and F&P):* An implementation MAY support mapping of WSDL element extensions, attribute extensions, Feature components and Property components not described in JAX-RPC. Note that such support may limit interoperability and application portability.

Editors Note 8.1 *Should we also require support for the Application Data Feature and Module?*

8.3 Type Systems

The WSDL specification mandates support for XML Schema as its type system description language. It also contains non-normative examples showing how support for other types systems (namely, DTDs and Relax NG) can be added to the language via extensibility.

◇ *Conformance Requirement (Other type systems):* An implementation MAY support type systems other than XML Schema.

8.4 Interfaces

An Interface component is mapped to a Java interface. The resulting interface is defined in the package whose name is mapped from the value of the {target namespace} property of the component, according to the mapping from namespace names to Java packages associated with the Definitions component under which the Interface component being mapped is found.

A Java interface mapped from an Interface component 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} property of the corresponding Interface component, mapped according to the rules described in section 8.10.

◇ *Conformance Requirement (Extending `java.rmi.Remote`):* A mapped SEI MUST extend `java.rmi.Remote`.

WSDL supports multiple inheritance for interfaces. Consequently, an SEI will extend all the SEIs that are mapped from the Interface components extended by its corresponding Interface component.

◇ *Conformance Requirement (Extended Interfaces):* A SEI MUST extend all the SEIs that are mapped from the Interface Components listed under the {extended interfaces} property of the Interface Component it is mapped from.

An SEI contains Java methods mapped from the Interface Operation components listed under the {operations} property of the corresponding Interface component, see section 8.5 for further details.

An Interface component also contains a set of Interface Fault components, specified under its {faults} property. Such components are mapped to Java exception classes, see section 8.7 below.

8.5 Operations

Each Interface Operation component is mapped to a Java method in the corresponding Java service endpoint interface.

Note that in WSDL, an Interface component's {operations} property contains not only all the Operation components defined directly under the component in question, but also all the Operation components defined by the Interface components the latter extends. When mapping an Interface Operation component to a Java method, only Interface Operation components that are not already defined by an Interface component extended by the Interface component being mapped must be mapped. This avoids having the definition of a Java SEI explicitely list all the methods it inherits from its super-interfaces.

◇ *Conformance Requirement (Method naming):* In the absence of customizations, the name of a mapped Java method MUST be the value of the {name} property of the Operation component, mapped according to the rules described in section 8.10.

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

Every WSDL Operation component must declare which message exchange pattern (MEP) it belongs to. JAX-RPC supports two of the eight predefined MEPs, In-Only and In-Out. Support for other MEPs may be added by implementations.

◇ *Conformance Requirement (Supported MEPs):* An implementation MUST support mapping of operations that use the In-Only and In-Out message exchange patterns.

Among the Message Reference components listed under the {message references} property of the Interface Operation component being mapped, there must be exactly one component whose {direction} is `in` and whose {message label} is `in`. For simplicity, this component will be referred to as the "Input Message Reference component" for the Operation component being mapped. Similarly, in the case of an In-Out Operation component, there must be exactly one Message Reference component whose {direction} is `out` and whose {message label} is "out". This component will be referred to as the "Output Message Reference component" for the Operation component being mapped.

An Interface Operation component's {style} property contains a set of URIs, each one identifying a particular style that the operation follows. JAX-RPC requires support for the predefined "RPC style".

◇ *Conformance Requirement (RPC style):* An implementation MUST support the RPC style and `wrpc:signature` extension as defined by the WSDL specification. In particular, an implementation MUST check that an Interface Operation component tagged with the RPC style does indeed follow all the corresponding rules.

Normally, the RPC style is used in conjunction with the `wrpc:signature` extension. An Interface Operation component thus marked can be mapped to a Java method in a way that preserves as much as possible the original intent of the author of the WSDL document being processed. On the other hand, Interface Operation for which no signature is available are mapped in a generic way, reminiscent of non-wrapper-style document-literal operations in WSDL 1.1.

8.5.1 Operations with Signatures

An Interface Operation component which follows the RPC style and which has a {rpc-signature} property, MUST be mapped according to the following rules.

◇ *Conformance Requirement (Disabling RPC style):* Implementations MUST provide a means to disable the RPC-style-with-signature mapping of operations, in which case the rules in 8.5.3 apply.

Construct the formal signature of the operation per the rules in the WSDL specification. It will be of the form $f([d_0]a_0, [d_1]a_1, \ldots) \Rightarrow (r_0, r_1, \ldots)$, where the d_i tokens identify the direction of a parameter (`#in`, `#out` or `#inout`) and all the a_i, r_i are qualified names of children of the request and/or response messages. Per the WSDL specification rules, it is possible to associate to each a_i, r_i precisely one XML schema type.

If there is more than one return type r_i, then the mapped method is the one obtained by disregarding the signature and using the generic mapping rules instead, see 8.5.3.

Otherwise, each $[d_i]a_i$ parameter is mapped to a method parameter whose type is determined as follows, based on the value of its direction, d_i:

`#in` The mapping of the type of a_i.

`#out` The holder type corresponding to the mapping of the type of a_i.

`#inout` The holder type corresponding to the mapping of the type of a_i.

◇ *Conformance Requirement (Parameter naming):* In the absence of customization, the name of a mapped Java method parameter MUST correspond to the local part of the qualified name of a_i, mapped according to the rules described in sections 8.10 and 2.8.1.

Moreover, the r_i types are mapped to the return type of the mapped method according to whether:

there is no return type The return type of the method is `void`.

there is only one return type, r_0 The return type of the method is mapped from r_0.

8.5.2 Holder Classes

Holder classes are used to support `#out` and `#inout` 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.

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 `#inout` parameter whose XML data type would normally be mapped to a Java `int` is instead mapped to `GenericHolder<java.lang.Integer>`.

◇ *Conformance Requirement (Use of `GenericHolder`):* Implementations MUST map any `#out` and `#inout` method parameters using `javax.xml.rpc.holders.GenericHolder<T>`.

8.5.3 Signatureless Operations

An Interface Operation component which does not belong to the RPC style or which doesn't have an {rpc-signature} property (meaning that the original `wsdl:operation` element wasn't annotated with a `wrpc:signature` extension attribute), MUST be mapped according to the following rules.

Depending on the value of the Input Message Reference component's {message content model} property:

`#element` The mapped method has one parameter, whose type is mapped from the global element declaration referred to by the {element} property of the Input Message Reference component.

`#any` The mapped method has one parameter, whose type is mapped from the `xsd:anyType` urtype.

`#none` The mapped method has zero parameters.

If the operation is In-Out, depending on the value of the Output Message Reference component's {message content model} property:

`#element` The mapped method has a return type mapped from the global element declaration referred to by the {element} property of the Input Message Reference component.

`#any` The mapped method has a return type mapped from the `xsd:anyType` urtype.

`#none` The mapped method has a `void` return type.

If the operation is In-Only, the mapped method has a `void` return type.

◇ *Conformance Requirement (Parameter naming):* In the absence of customization, the name of the single mapped Java method parameter, if present, MUST be `"in"`.

8.5.4 Fault References

Each Fault Reference component listed under the {fault references} property of an Interface Operation component results in an additional exception thrown by the mapped method. The type of the exception is mapped from the Interface Fault component referred to by the {fault reference} property of the Fault Reference component being mapped (see 8.7).

8.5.4.1 Example

Figure 8.1 shows a WSDL extract and the corresponding Java service interface.

8.5.5 Asynchrony

In addition to the synchronous mapping of `wsdl:operation` described above, a client side 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 client side asynchronous mapped interfaces should be optional at the users discretion.

◇ *Conformance Requirement (Asynchronous mapping required):* Implementations MUST support the asynchronous mapping.

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

```
<!-- WSDL extract -->
<types>
    <xsd:element name="transferAmount">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="account" type="xsd:string"/>
                <xsd:element name="amount" type="xsd:double"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="transferAmountResponse">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="updatedAmount" type="xsd:double"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</types>

<interface name="AccountManager">
    <operation name="transferAmount"
               style="http://www.w3.org/2004/08/wsdl/style/rpc">
               rpc:signature="account #in amount #in updatedAmount #return">
        <input element="tns:transferAmount"/>
        <output element="tns:transferAmountResponse"/>
    </operation>
    <operation name="transferAmount2">
        <input element="tns:transferAmount"/>
        <output element="tns:transferAmountResponse"/>
    </operation>
</portType>

// mapped Java interface

public interface AccountManager extends Remote {

  // the operation this method comes from was RPC-style with signature information
  double transferAmount(String account, double amount)
      throws RemoteException;

  // the operation this method comes from was not RPC-style
  TransferAmountResponse transferAmount2(TransferAmount transferAmount)
      throws RemoteException;
}
```

Figure 8.1: Mapping of WSDL operations

8.5.5.1 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.

8.5.5.2 Operations

Each Operation component is mapped to two methods in the corresponding asynchronous service endpoint interface:

Polling method A polling method returns a typed Response<*ResponseType*> 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 *ResponseType*.

Callback method A callback method takes an additional final parameter that is an instance of a typed AsyncHandler<*ResponseType*> 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} property of the Interface Operation component suffixed with "Async" mapped according to the rules described in sections 8.10 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.

8.5.5.3 Operations with Signatures

An Interface Operation component which follows the RPC style and which has a {rpc-signature} property, MUST be mapped according to the following rules.

♢ *Conformance Requirement (Disabling RPC style):* Implementations MUST provide a means to disable the RPC-style-with-signature mapping of operations, in which case the rules in 8.5.5.4 apply.

The mapped method signature is determined in a different way than the synchronous one:

1. Start with the value of the {rpc-signature} property, a list of pairs of a qualified name and a token: $[(q_i, t_i), \ldots]$.
2. Filter the elements in the list, retaining only those whose token is either #in or #inout: $[(q'_j, t'_j), \ldots]$.

3. The mapped method has a parameter for each pair (q'_j, t'_j). In the absence of customizations, the parameter name is mapped from the local part of q'_j according to the rules described in sections 8.10 and 2.8.1. The parameter type is mapped from the XML Schema type uniquely associated with q'_j.

4. The `ResponseType` is mapped from the type of the global element declaration referred to by the Output Message Reference component. If the value of the {rpc-signature} property is such that there is exactly one pair $(q_R, \#return)$ marked with the `#return` token and no pair marked with the `#out` or `#inout` tokens, then the `ResponseType` is the type mapped from the XML Schema type associated with the q_R qualified name.

Notice that in the asynchronous method mapping, holder classes are not used. Also, the response type is usually mapped directly from the global element definition for the output message of the operation, except when the output message has a single child. This special case ensures that when there are no output parameters, the `ResponseType` of an asynchronous method is the same as the return type of the corresponding synchronous one.

8.5.5.4 Signatureless Operations

Depending on the value of the Input Message Reference component's {message content model} property:

`#element` The mapped method has a first parameter, whose type is mapped from the global element declaration referred to by the {element} property of the Input Message Reference component.

`#any` The mapped method has a first parameter, whose type is mapped from the `xsd:anyType` urtype.

`#none` The mapped method does not have any parameters except the one required by the callback method.

Depending on the value of the Output Message Reference component's {message content model} property:

`#element` The mapped method has a `ResponseType` mapped from the global element declaration referred to by the {element} property of the Input Message Reference component.

`#any` The mapped method has a `ResponseType` mapped from the `xsd:anyType` urtype.

`#none` The mapped method has a `ResponseType` of `?`.

8.5.5.5 Example

Figure 8.2 shows a WSDL extract (the same used in 8.1) and figure 8.3 shows the corresponding Java service endpoint interfaces, both with and without asynchronous support.

8.6 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.

Editors Note 8.2 *What about the `xsd:import` under `wsdl:types`?*

```
<!-- WSDL extract -->
<types>
    <xsd:element name="transferAmount">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="account" type="xsd:string"/>
                <xsd:element name="amount" type="xsd:double"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="transferAmountResponse">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="updatedAmount" type="xsd:double"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</types>

<interface name="AccountManager">
    <operation name="transferAmount"
               style="http://www.w3.org/2004/08/wsdl/style/rpc">
               rpc:signature="account #in amount #in updatedAmount #return">
        <input element="tns:transferAmount"/>
        <output element="tns:transferAmountResponse"/>
    </operation>
    <operation name="transferAmount2">
        <input element="tns:transferAmount"/>
        <output element="tns:transferAmountResponse"/>
    </operation>
</portType>

```

Figure 8.2: Asynchronous mapping of WSDL operations

```
// mapped Java interface (synchronous)

public interface AccountManager extends Remote {

  double transferAmount(String account, double amount)
      throws RemoteException;

  TransferAmountResponse transferAmount2(TransferAmount transferAmount)
      throws RemoteException;
}

// mapped Java interface (asynchronous)

public interface AccountManageAsync extends Remote {

  Response<Double> transferAmountAsync(String account, double amount)
      throws RemoteException;

  Future<?> transferAmountAsync(String account, double amount, AsyncHandler<Double>)
      throws RemoteException;

  Response<TransferAmountResponse> transferAmount2Async(TransferAmount transferAmount)
      throws RemoteException;

  Future<?> transferAmount2Async(TransferAmount transferAmount,
                                 AsyncHandler<TransferAmountResponse>)
      throws RemoteException;

}

```

Figure 8.3: Asynchronous mapping of WSDL operations

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.

8.7 Faults

An Interface Fault component is mapped to a Java exception class. The generated exception class is defined in the package whose name is mapped from the value of the {target namespace} property of the component, according to the mapping from namespace names to Java packages associated with the Definitions component under which the Interface Fault component being mapped is found.

♢ *Conformance Requirement (Exception naming):* In the absence of customizations, the name of a mapped exception MUST be the value of the {name} propertu of the Interface Fault component, mapped according to the rules in sections 8.10 and 2.8.1.

An Interface Fault component's {element} property refers to an XML Schema global element declaration. In turn, the global element declaration is mapped to a Java bean, henceforth called a fault bean, using the mapping described in section 8.6. An implementation generates a wrapper exception class that extends `java.lang.Exception` and contains the following methods:

***WrapperException*`(String message,` *FaultBean* `faultInfo)`** 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.

***WrapperException*`(String message,` *FaultBean* `faultInfo, 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. The final argument, `cause`, may be used to convey protocol specific fault information, see section 4.9.1.

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

8.8 Services

A WSDL Service component comprises a set of Endpoint components, all implementing the same Interface. An Endpoint component associates some address information to a Binding component, which in turn describes the concrete message format and transmission protocol used for communication with the Endpoint.

On the client side, 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 the `javax.xml.rpc.Service` interface.

For each Endpoint component in the {endpoints} property of the Service component, the generated client side service interface contains the following methods:

***ServiceEndpointInterface* `get`*EndpointName*`()`** 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* `get`*EndpointName*`(`*params*`)`** Zero or more optional additional methods that include parameters specific to the endpoint 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 `get`*EndpointName*):* `get`*EndpointName* MUST throw `javax.xml-.rpc.ServiceException` on failure.

The value of *EndpointName* in the above is obtained by starting with the value of the {name} property of the corresponding Endpoint component, then mapping it to a a Java identifier according to the rules described in section 8.10, finally treating this Java identifier as a JavaBean property for the purposes of deriving the `get`*EndpointName* method name.

8.9 SOAP 1.2 Binding

The WSDL specification defines a normative binding for SOAP 1.2 over HTTP (see [11]). JAX-RPC implementations MUST support this binding. They MAY also support additional bindings for other protocols as well as for SOAP 1.2 over non-HTTP transports.

◇ *Conformance Requirement (SOAP 1.2 over HTTP):* An implementation MUST support the SOAP 1.2 Binding over the HTTP protocol as defined by the WSDL specification.

◇ *Conformance Requirement (Other bindings):* An implementation MAY support other bindings.

8.10 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 Operation component 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 parameter 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.

8.10.1 Name Collisions

WSDL name scoping rules may result in name collisions when mapping from WSDL to Java. E.g., an interface 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. The error may be corrected 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 9

Java to WSDL 2.0 Mapping

This chapter describes the mapping from Java to WSDL 2.0. This mapping is used when generating web service endpoints from existing Java interfaces.

◇ *Conformance Requirement (WSDL 2.0 support):* Implementations MUST support mapping Java to WSDL 2.0.

The following sections describe the default mapping from each Java construct to the equivalent WSDL 2.0 component. The basis for this mapping is the WSDL 2.0 Working Draft dated ”3 August 2004” ([11], [18], [19]) which was adopted by the W3C Web Services Description Working Group as a Last Call Draft.

The mapping is described in terms of the WSDL 2.0 component model. The resulting component model MUST be serialized to XML 1.0 using the serialization described in the WSDL 2.0 specification.

◇ *Conformance Requirement (WSDL 2.0 serialization):* Implementations MUST support the XML 1.0-based serialization of a WSDL 2.0 component model.

In the rest of this chapter, ”WSDL” refers to ”WSDL 2.0”. The ”wsdl” and ”wrpc” namespace prefixes are assumed to be bound per the WSDL 2.0 specification (i.e. to the ”http://www.w3.org/2004/08/wsdl” and ”http://www.w3.org/2004/08/wsdl/rpc” namespaces respectively).

9.1 Java Names

◇ *Conformance Requirement (Java identifier mapping):* Java identifiers SHOULD be mapped to XML names using the algorithm defined in appendix B of SOAP 1.2 Part 2[4].

9.1.1 Name Collisions

Like WS-I Basic Profile 1.0[8] (see R2304), WSDL 2.0 requires the Operation components within an Interface component to be uniquely named – support for customization of the operation name allows this requirement to be met when a Java SEI contains overloaded methods.

◇ *Conformance Requirement (Method name disambiguation):* An implementation MUST support the use of metadata to disambiguate overloaded Java method names when mapped to WSDL.

9.2 Packages

A Java package is mapped to a namespace name, which in turn will be used as the value of the {target namespace} property of the Interface and Service components that result from processing a Java type that belongs to that package.

All generated components will live inside a Definitions component, which upon serialization will be mapped to one or more WSDL documents. Although tools are free to generate multiple WSDL documents for a given Java package (using suitable `wsdl:import` and/or `xsd:import` statements), there MUST be one root WSDL document that imports all others, so that it can be identified as being *the* WSDL document corresponding to a given Java package.

There is no standard mapping from a Java package name to a namespace name.

◇ *Conformance Requirement (Package name mapping):* Implementations MUST provide a means to specify the namespace name when mapping a Java package to a WSDL document.

9.3 Interfaces

A Java service endpoint interface (SEI) is mapped to an Interface component. 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[1] (as `public final static`)

◇ *Conformance Requirement (Interface naming):* If not customized, the value of the {name} property of the Interface component MUST be the name of the service endpoint interface not including the package name.

The Interface component mapped from a Java SEI has the following properties:

{**name**} Name as specified above.

{**target namespace**} The namespace name corresponding to the Java package the mapped interface lives in.

{**extended interfaces**} The set of Interface components determined according to the rules in 9.3.1.

{**operations**} The set of Interface Operation components determined according to 9.4.

{**faults**} The set of Interface Fault components determined according to 9.6.

Figure 9.1 shows an example of a Java SEI and the corresponding WSDL Interface component, as serialized to a `wsdl:interface`.

[1]WSDL 2.0 does not define any standard representation for constants in an Interface component.

9.3.1 Inheritance

Java interface inheritance is directly mapped to WSDL interface inheritance. When mapping a SEI that inherits from another remote interface, the extended interface is in turn mapped to a WSDL Interface component. The Interface component corresponding to the original SEI will then list the newly created Interface component in its {extended interfaces} property.

♢ *Conformance Requirement (Inheritance):* A mapped Interface component's {extended interfaces} property MUST contain Interface components corresponding to all interfaces extended by the interface it was mapped from and that satisfy the conditions given above to be a SEI.

9.4 Methods

Each public method in a Java SEI is mapped to an Interface Operation component in the {operations} property of the Interface component that corresponds to the SEI.

♢ *Conformance Requirement (Interface Operation naming):* The value of the {name} property of the Interface Operation component 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 9.4.1 describes one way operations further.

By default, the generated Interface Operation component is tagged with the RPC style and with a `wrpc:signature` extension attribute that allows a WSDL processor to reconstruct the original method signature.

♢ *Conformance Requirement (Disabling RPC style):* Implementations MUST provide a means to disable the RPC-style-with-signature mapping of methods.

The Interface Operation component corresponding to each method has the following properties:

{**target namespace**} The namespace name (a URI) corresponding to the Java package the interface defining the method being mapped lives in.

{**name**} The name as determined per the above rules.

{**message exchange pattern**} The predefined MEP URI for an in-only or in-out operation, according to whether the method is one-way or two-way.

{**message references**} One or two Message Reference components, depending on whether the method is one-way or two-way. The construction of the input and ouput Message Reference components for an operation is described below.

{**fault references**} Zero or more Fault Reference components, constructed according to the exception mapping rules below.

{**style**} The 1-element list containg the RPC-style URI, i.e. "http://www.w3.org/2004/08/wsdl/style/rpc", unless the RPC-style-with-signature mapping has been disabled for the method being processed.

{**rpc-signature**} Computed from the method according to the rules defined below, unless the RPC-style-with-signature mapping has been disabled for the method being processed.

The input Message Reference component for the Interface Operation component mapped from a Java method has the following properties:

{**message label**} `in`.

{**direction**} `in`.

{**message content model**} `#element`.

{**element**} A global element declaration constructed from the arguments to the method per the rules below.

The output Message Reference component for the Interface Operation component mapped from a Java method has the following properties:

{**message label**} `out`.

{**direction**} `out`.

{**message content model**} `#element`.

{**element**} A global element declaration constructed from the arguments and return type of the method per the rules below.

For each exception (in addition to the required `java.rmi.RemoteException`) thrown by the method being mapped, the {fault references} property of the Operation component will contain a Fault Reference component with the following properties:

- A {fault reference} property containing the Interface Fault component to which the exception in question is mapped to.
- A {message label} property with value `out`.
- A {direction} property with value `out`.

9.4.1 One Way Operations

Only Java methods whose return type is `void`, that have no parameters that implement `Holder` 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.

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

♢ *Conformance Requirement (One-way mapping errors):* Implementations MUST prevent mapping to one-way operations of methods that do not meet the necessary criteria.

```
// Java
package com.example;
public interface StockQuoteProvider extends java.rmi.Remote {
    float getPrice(String tickerSymbol)
        throws java.rmi.RemoteException, TickerException;
}

<!-- WSDL extract -->
<types>
    <xsd:schema targetNamespace="...">
        <xsd:element name="getPrice" type="tns:getPriceType"/>
        <xsd:complexType name="getPriceType">
            <xsd:sequence>
                <xsd:element name="tickerSymbol" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>

        <xsd:element name="getPriceResponse" type="tns:getPriceResponseType"/>
        <xsd:complexType name="getPriceResponseType">
            <xsd:sequence>
                <xsd:element name="return" type="xsd:float"/>
            </xsd:sequence>
        </xsd:complexType>

        <xsd:element name="TickerException" type="tns:TickerExceptionType"/>
        <xsd:complexType name="TickerExceptionType">
            <xsd:sequence>
                <xsd:element name="message" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:schema>
</types>

<interface name="StockQuoteProvider">
  <fault name="TickerException" element="tns:TickerException"/>
  <operation name="getPrice"
             style=" http://www.w3.org/2004/08/wsdl/rpc"
             wrpc:signature="tickerSymbol #in return #return">
    <input element="xs-tns:getPrice"/>
    <output element="xs-tns:getPriceResponse"/>
    <outfault ref="wsdl-tns:TickerException"/>
  </operation>
</interface>
```

Figure 9.1: Java interface to WSDL Interface component mapping

9.5 Method Parameters

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.

9.5.1 Parameters

Method parameters are classified as follows:

`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 component's input Message Reference component.

`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 component's output Message Reference component.

`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 component's input Message Reference component and the global element declaration for the Operation component's output Message Reference component.

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. A 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 means to specify 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.

9.5.2 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). [2]

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

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 9.2 illustrates this representation.

```
float getPrice(String tickerSymbol);

class getPrice {
    public String getTickerSymbol();
}

class getPriceResponse {
    public float getReturn();
}
```

Figure 9.2: 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.

9.6 Service Specific Exceptions

A service specific Java exception is mapped to a Interface Fault component defined under the {faults} property of the Interface component that corresponds to the Java interface whose method throws the exception being mapped. It is possible for the same exception to be thrown by methods defined on different interfaces, in which case there will be multiple Interface Fault components being defined, one per interface (under the corresponding Interface components).

The Interface Fault component mapped from a service specific Java excxeption has the following properties:

{**target namespace**} The namespace name (a URI) corresponding to the Java package the interface using the exception being mapped lives in.

{**name**} The unqualified name of the exception class.

{**element**} A global element declaration mapped per the rules below.

◇ *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 9.1.

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

9.7 Bindings

In WSDL, an interface 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.

9.8 SOAP HTTP Binding

This section describes the binding components to be produced when mapping Java service endpoint interfaces to an endpoint bound to SOAP 1.2 over HTTP.

◇ *Conformance Requirement (SOAP binding support):* Implementations MUST be able to generate SOAP 1.2 HTTP bindings when mapping Java to WSDL.

9.8.1 Binding component

A Java service endpoint interface (SEI) is mapped to a Binding component with the following properties:

{**name**} The unqualified name of the interface being mapped, with the suffix ”Binding” appended to it.

{**target namespace**} The namespace name corresponding to the Java package the mapped interface lives in.

{**type**} `"http://www.w3.org/2004/08/wsdl/soap12"`.

{**soap underlying protocol**} `"http://www.w3.org/2003/05/soap/bindings/HTTP/"`.

Editors Note 9.1 *The default binding rules for the SOAP binding do not cover in-only operations. Given that there is no other suitable SOAP MEP, does that mean that they can only be bound using the HTTP binding (no SOAP)?*

9.9 Services and endpoints

A WSDL 2.0 service may contain multiple endpoints, possibly bound to different protocols but all implementing the same interface.

◇ *Conformance Requirement (Service and endpoint selection):* Implementations MUST provide a facility for specifying the services and endpoints to generate when mapping from Java to WSDL.

A Service component mapped from a given Java service endpoint interface has the following properties:

{**name**} In the absence of customization, the unqualified name of the Java interface being mapped with the suffix ”Service” appended to it.

{**target namespace**} The namespace name corresponding to the Java package the interface being mapped lives in.

{**interface**} The Interface component mapped from the Java interface being mapped.

{**endpoints**} At least one Endpoint component.

Appendix A

Conformance Requirements

2.1 WSDL 1.1 support 9
2.2 Definitions mapping 9
2.3 WSDL and XML Schema import directives 9
2.4 Optional WSDL extensions 10
2.5 SEI naming 10
2.6 Extending `java.rmi.Remote` 10
2.7 Method naming 10
2.8 `RemoteException` required 10
2.9 Transmission primitive support 10
2.10 Non-wrapped parameter naming 11
2.11 Default mapping mode 12
2.12 Disabling wrapper style 12
2.13 Wrapped parameter naming 12
2.14 Parameter name clash 12
2.15 Use of `GenericHolder` 15
2.16 Asynchronous mapping required 15
2.17 Asynchronous mapping option 15
2.18 Asynchronous method naming 15
2.19 Failed method invocation 15
2.20 Response bean naming 16
2.21 Asynchronous fault reporting 16
2.22 Asychronous fault cause 17
2.23 JAXB class mapping 19
2.24 JAXB customization use 19

2.25 JAXB customization clash 19
2.26 Exception naming 19
2.27 Fault equivalence 19
2.28 Fault equivalence 20
2.29 Required WSDL extensions 20
2.30 Unbound message parts 20
2.31 Mapping additional header parts 22
2.32 Duplicate headers in binding 22
2.33 Duplicate headers in Message 22
2.34 Use of MIME type information 23
2.35 MIME type mismatch 25
2.36 MIME part identification 25
2.37 `Service` interface required 25
2.38 Failed `get`*PortName* 25
3.1 WSDL 1.1 support 27
3.2 Java identifier mapping 27
3.3 Method name disambiguation 27
3.4 Package name mapping 27
3.5 WSDL and XML Schema import directives 28
3.6 portType naming 28
3.7 Inheritance flattening 28
3.8 Inherited interface mapping 28
3.9 Operation naming 29
3.10 One-way mapping 29
3.11 One-way mapping errors 29
3.12 Parameter classification 31
3.13 Wrapper bean naming 32
3.14 Default wrapper bean names 32
3.15 Bean naming 33
3.16 Default bean names 33
3.17 Exception naming 33
3.18 Binding selection 34
3.19 SOAP binding support 35
3.20 SOAP binding style required 36

3.21 Port selection . . . 36
4.1 Concrete `ServiceFactory` required . . . 39
4.2 `Service` class loading . . . 39
4.3 Service completeness . . . 40
4.4 Service capabilities . . . 40
4.5 Read-only handler chains . . . 41
4.6 TypeMappingRegistry support . . . 41
4.7 TypeMappingRegistry usage . . . 41
4.8 Required `JAXRPCContext` properties . . . 43
4.9 Optional `JAXRPCContext` properties . . . 43
4.10 Additional `JAXRPCContext` properties . . . 43
4.11 Read-only handler chains . . . 44
4.12 Implementing `Stub` . . . 45
4.13 Stub class binding . . . 45
4.14 `Stub` configuration . . . 45
4.15 Dynamic proxy support . . . 46
4.16 Implementing `Stub` required . . . 46
4.17 `Service.getPort` failure . . . 46
4.18 `Dispatch` support . . . 47
4.19 `Dispatch` configuration failure . . . 47
4.20 Failed `Dispatch.invoke` . . . 48
4.21 Failed `Dispatch.invokeAsync` . . . 48
4.22 Failed `Dispatch.invokeOneWay` . . . 48
4.23 Reporting asynchronous errors . . . 48
4.24 Marshalling failure . . . 49
4.25 `Call` support . . . 50
4.26 `createCall` failure . . . 50
4.27 Unconfigured `Call` . . . 51
4.28 `Call` configuration . . . 51
4.29 Misconfigured invocation . . . 52
4.30 Failed `invoke` . . . 52
4.31 Failed `invokeOneWay` . . . 52
4.32 Missing invocation . . . 52
4.33 Protocol specific fault generation . . . 53

4.34 Protocol specific fault consumption 54
5.1 Provider support required 55
5.2 Provider default constructor 55
5.3 Provider implementation 55
5.4 One way operations 57
5.5 Service initialization 58
5.6 Service destruction 58
5.7 Service destruction 58
5.8 Handler exceptions 58
5.9 Context type 59
5.10 Context Implementation Required 59
6.1 Handler framework support 61
6.2 Logical handler support 63
6.3 Other handler support 63
6.4 Handler chain snapshot 64
6.5 Binding handler manipulation 65
6.6 Handler initialization 65
6.7 Handler destruction 66
6.8 Handler exceptions 66
6.9 Invoking `close` 68
6.10 Order of `close` invocations 68
6.11 Message context scope 69
6.12 Message context property scope 69
6.13 Additional `MessageContext` properties 70
7.1 SOAP required roles 71
7.2 SOAP required roles 72
7.3 Default role visibility 72
7.4 Default role persistence 72
7.5 None role error 72
7.6 Incompatible handlers 72
7.7 Incompatible handlers 72
7.8 Logical handler access 72
7.9 SOAP 1.1 HTTP Binding Support 74
7.10 SOAP 1.2 HTTP Binding Support 74

7.11 One-way operations . . . 75
7.12 HTTP basic authentication support . . . 75
7.13 Authentication properties . . . 75
7.14 URL rewriting support . . . 75
7.15 Cookie support . . . 75
7.16 SSL session support . . . 75
8.1 WSDL 2.0 support . . . 77
8.2 WSDL 2.0 processor conformance . . . 77
8.3 Definitions mapping . . . 77
8.4 WSDL extensions and F&P . . . 78
8.5 Other type systems . . . 78
8.6 SEI naming . . . 78
8.7 Extending `java.rmi.Remote` . . . 78
8.8 Extended Interfaces . . . 78
8.9 Method naming . . . 79
8.10 `RemoteException` required . . . 79
8.11 Supported MEPs . . . 79
8.12 RPC style . . . 79
8.13 Disabling RPC style . . . 80
8.14 Parameter naming . . . 80
8.15 Use of `GenericHolder` . . . 80
8.16 Parameter naming . . . 81
8.17 Asynchronous mapping required . . . 81
8.18 Asynchronous mapping option . . . 81
8.19 Asynchronous method naming . . . 83
8.20 Failed method invocation . . . 83
8.21 Disabling RPC style . . . 83
8.22 JAXB Class Mapping . . . 87
8.23 Exception naming . . . 87
8.24 `Service` interface required . . . 87
8.25 Failed `get`*EndpointName* . . . 88
8.26 SOAP 1.2 over HTTP . . . 88
8.27 Other bindings . . . 88
9.1 WSDL 2.0 support . . . 91

9.2 WSDL 2.0 serialization 91
9.3 Java identifier mapping 91
9.4 Method name disambiguation 91
9.5 Package name mapping 92
9.6 Interface naming 92
9.7 Inheritance 93
9.8 Interface Operation naming 93
9.9 Disabling RPC style 93
9.10 One-way mapping 94
9.11 One-way mapping errors 94
9.12 Parameter classification 96
9.13 Exception naming 97
9.14 Binding selection 97
9.15 SOAP binding support 98
9.16 Service and endpoint selection 98

Appendix B

Change Log

B.1 Changes Since Early Draft 1

- Added chapter 5 Service APIs.
- Added chapter 8 WSDL 2.0 to Java Mapping.
- Added chapter 9 Java to WSDL 2.0 Mapping.
- Added mapping from Java to `wsdl:service` and `wsdl:port`, see sections 3.7.1, 3.8.1 and 3.9.
- Fixed section 2.4 to allow use of JAXB interface based mapping.
- Added support for document/literal/bare mapping in Java to WSDL mapping, see section 3.5.
- Added conformance requirement to describe the expected behaviour when two or more faults refer to the same global element, see section 2.5.
- Added resolution to issue regarding binding of duplicate headers, see section 2.6.2.1.
- Added use of JAXB ns URI to Java package name mapping, see section 2.1.
- Added use of JAXB package name to ns URI mapping, see section 3.2.
- Introduced new typographic convention to clearly mark non-normative notes.
- Removed references to J2EE and JNDI usage from ServiceFactory description, see section 4.1.2.
- Clarified relationship between TypeMappingRegistry and JAXB, see section 4.2.2.
- Emphasized control nature of context properties, added lifecycle subsection, see section 4.3.
- Clarified fixed binding requirement for stubs, see section 4.6.
- Added section for SOAP proocol bindings 7.4. The HTTP subsection of this now contains much of the mterial from the JAX-RPC 1.1 ‘Runtime Services’ chapter.
- Clarified that async methods are added to the regular sync SEI when async mapping is enabled rather than to a separate async-only SEI, see section 2.3.4.
- Added support for WSDL MIME binding, see section 2.6.3.

- Clarified that fault mapping should only generate a single exception for each equivalent set of faults, see section 2.5.
- Added property for message attachments, see section 4.3.2.
- Removed element references to anonymous type as valid for wrapper style mapping (this doesn't prevent substitution as orignally thought), see section 2.3.1.2.
- Removed implementation specific methods from generated service interfaces, see section 2.7.
- Clarrified behaviour under fault condition for asynchronous operation mapping, see section 2.3.4.5.

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. Final Material, WS-I, April 2004. See http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.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, Jeffrey Schlimmer, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Working Draft, W3C, August 2004. See http://www.w3.org/TR/2004/WD-wsdl20-20040803.

[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.

[14] Jim Knutson and Heather Kreger. Web Services for J2EE. JSR, JCP, September 2002. See http://jcp.org/en/jsr/detail?id=109.

[15] Nataraj Nagaratnam. Web Services Message Security APIs. JSR, JCP, April 2002. See http://jcp.org/en/jsr/detail?id=183.

[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.

[17] Keith Ballinger, David Ehnebuske, Chris Ferris, Martin Gudgin, Canyang Kevin Liu, Mark Nottingham, Jorgen Thelin, and Prasad Yendluri. Basic Profile Version 1.1. Final Material, WS-I, August 2004. See http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html.

[18] Martin Gudgin, Amy Lewis, and Jeffrey Schlimmer. Web Services Description Language (WSDL) Version 2.0 Part 2: Predefined Extensions. Working Draft, W3C, August 2004. See http://www.w3.org/TR/2004/WD-wsdl20-extensions-20040803.

[19] Hugo Haas, Philippe Le Hégaret, Jean-Jacques Moreau, David Orchard, Jeffrey Schlimmer, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 3: Bindings. Working Draft, W3C, August 2004. See http://www.w3.org/TR/2004/WD-wsdl20-bindings-20040803.

[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.

[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.

[22] John Cowan and Richard Tobin. XML Information Set. Recommendation, W3C, October 2001. See http://www.w3.org/TR/2001/REC-xml-infoset-20011024/.

[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] 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/.

[25] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification - second edition. Book, Sun Microsystems, Inc, 2000. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[26] Mark Nottingham. Simple SOAP Binding Profile Version 1.0. Working Group Draft, WS-I, August 2004. See http://www.ws-i.org/Profiles/SimpleSoapBindingProfile-1.0-2004-08-24.html.

[27] Chris Ferris, Anish Karmarkar, and Canyang Kevin Liu. Attachments Profile Version 1.0. Final Material, WS-I, August 2004. See http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html.

[28] Danny Coward and Yutaka Yoshida. Java Servlet Specification Version 2.4. JSR, JCP, November 2003. See http://java.sun.com/products/servlet.