Java™ API for XML-based RPC
JAX-RPC 1.1

Technical comments to: jsr-101-leads@sun.com

JSR-101

Java Community Process (JCP)

Maintenance Release

Version 1.1

JSR-101 Expert Group

Specification Lead: Roberto Chinnici (Sun Microsystems, Inc.)

Java(TM) API for XML-based Remote Procedure Call (JAX-RPC) Specification ("Specification")

Version: 1.1
Status: FCS, Maintenance Release
Release:October 14, 2003

NOTICE; LIMITED LICENSE GRANTS

Sun Microsystems, Inc. (“Sun”) hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under the Sun’s applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation, which shall be understood to include developing applications intended to run on an implementation of the Specification provided that such applications do not themselves implement any portion(s) of the Specification.

Sun also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or patent rights it may have in the Specification to create and/or distribute an Independent Implementation of the Specification that: (i) fully implements the Spec(s) including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/ authorized by the Specification or Specifications being implemented; and (iii) passes the TCK (including satisfying the requirements of the applicable TCK Users Guide) for such Specification. The foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose.

You need not include limitations (i)-(iii) from the previous paragraph or any other particular “pass through” requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to implementations of the Specification (and products derived from them) that satisfy limitations (i)-(iii) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun’s applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation’s compliance with the Spec in question.

For the purposes of this Agreement: “*Independent Implementation*” shall mean an implementation of the Specification that neither derives from any of Sun’s source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun’s source code or binary code materials; and “*Licensor Name Space*” shall mean the public class or interface declarations whose names begin with “java”, “javax”, “com.sun” or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof.

This Agreement will terminate immediately without notice from Sun if you fail to comply with any material provision of or act outside the scope of the licenses granted above.

TRADEMARKS

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

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED “AS IS” SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE

OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

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

LIMITATION OF LIABILITY

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

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

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

REPORT

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

(LFI#128132/Form ID#011801)

Contents

1 Introduction

The RPC (Remote Procedure Call) mechanism enables a remote procedure call from a client to be communicated to a remote server. An example use of an RPC mechanism is in a distributed client/server model. A server defines a service as a collection of procedures that are callable by remote clients. A client calls procedures to access service defined by the server.

In XML based RPC, a remote procedure call is represented using an XML based protocol. SOAP 1.1 specification [3] defines an XML based protocol for exchange of information in a decentralized, distributed environment. SOAP defines a convention for representation of remote procedure calls and responses. This is in addition to the definition of the SOAP envelope and encoding rules.

An XML based RPC server application can define, describe and export a web service as an RPC based service. WSDL (Web Service Description Language) [7] specifies an XML format for describing a service as a set of endpoints operating on messages. An abstract description of such service can be bound to an XML based protocol and underlying transport. A service client can use an RPC based service.

This document specifies Java APIs for XML based RPC (JAX-RPC). This document is the formal specification for JSR-101 [`http://jcp.org/jsr/detail/101.jsp`], which is being worked on under the Java Community Process [`http://jcp.org`].

1.1 Design Goals

The goals of this JSR are as follows:

- Specify APIs for supporting XML based RPC for the Java platform
- Define base level protocol bindings and to not limit other protocol bindings that can be supported with the JAX-RPC programming model.
- Define APIs that are simple to use for development of Java applications that define or use JAX-RPC based services. Simplicity of JAX-RPC APIs is an important element in making the JAX-RPC model easy to understand, implement, maintain and evolve. Part of the simplicity goal will be to follow the standard Java application programming model and concepts as part of the JAX-RPC API specification.
- Support interoperability across heterogeneous platforms and environments. This specification will specify requirements for interoperability as applied to the scope of JAX-RPC.
- Specify conformance and interoperability requirements that are testable for an implementation of the JAX-RPC specification.

- Keep the design of JAX-RPC APIs and mechanisms extensible and modular. This will enable support for future versions of various XML specifications, including XMLP [5]

1.1.1 Expert Group Goals

The goals of simplicity and faster time to market imply that some important features are considered out of scope for the 1.1 version of JAX-RPC specification. However, the JAX-RPC 1.1 specification recognizes that these out of scope features may be implemented by a JAX-RPC implementation.

Proposed out of scope features for the 1.1 version of the JAX-RPC specification include:

- Design of portable stubs and skeletons
- Standard representation of transaction and security context
- Service registration and discovery
- Objects by reference

These features may be addressed by future versions of the JAX-RPC specification.

1.2 JSR-101 Expert Group

- ATG: Mark Stewart
- BEA: Manoj Cheenath
- Cisco Systems: Krishna Sankar
- EDS: Waqar Sadiq
- Fujitsu: Kazunori Iwasa
- HP: Pankaj Kumar
- IBM: Russell Butek, Jim Knutson
- Idoox: Miroslav Simek
- IONA: Dan Kulp
- InterX: Miles Sabin
- iPlanet: Shailesh Bavadekar
- Macromedia: Glen Daniels
- Netdecisions: Steve Jones
- Nortel: Pierre Gauthier
- Novell: Bjarne Rasmussen
- Oracle: Jeff Mischkinsky, Umit Yalcinalp
- Pramati: Amit Khanna
- Software AG: Dietmar Gaertner
- Sun Microsystems: Roberto Chinnici [Maintenance lead]
- WebGain: Steve Marx
- WebMethods: Prasad Yendluri
- Matt Kuntz
- James Strachan

- Shawn Bayern

1.3 Acknowledgments

Art Frechette, Phil Goodwin, Arun Gupta, Marc Hadley, Graham Hamilton, Mark Hapner, Doug Kohlert, Eduardo Pelegri-Llopart, Bill Shannon and Sankar Vyakaranam (all from Sun Microsystems) have provided invaluable technical input to the JAX-RPC 1.1 specification.

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

1.4 Status

This document is a maintenance review draft for the maintenance release of JSR-101 under the Java Community process.

1.5 Notational Conventions

- Diagrams follow the standard UML notation
- Code snippets are not shown in complete form. Refer to the Java docs for complete and detailed description.
- Examples are illustrative (non-prescriptive)

2 JAX-RPC Usecase

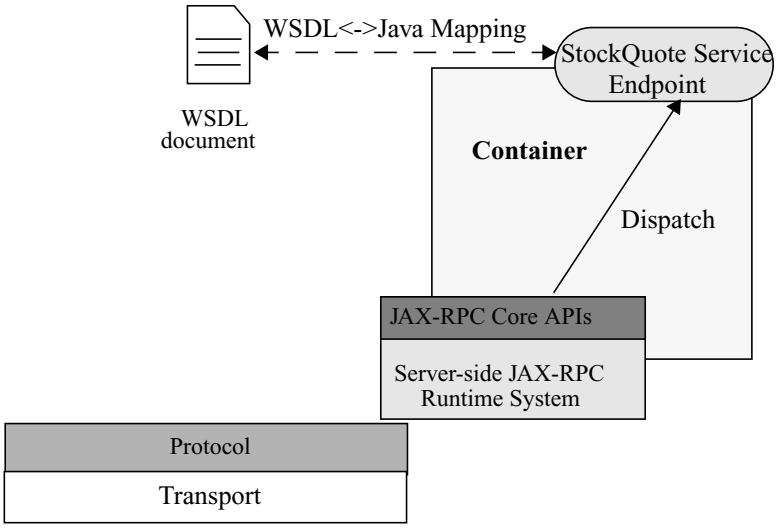
This chapter describes use cases for the JAX-RPC model in a non-prescriptive manner. Later chapters of this document specify requirements and APIs in a prescriptive manner.

2.1 Stock Quote Service

The following description uses a stock quote service example to illustrate JAX-RPC concepts. Note that this use case is used throughout this document to show use cases for the JAX-RPC APIs.

Note that this usecase describes a high level overview of the JAX-RPC concepts. For more complete details, refer to the detailed specification later in this document.

The following diagram shows a server side service endpoint definition of a stock quote service.



2.1.1 Service Endpoint Definition

The example stock quote service is defined and deployed using the Java platform. This service is capable of use by service clients deployed on any platform. JAX-RPC service endpoint definition makes no assumption that the service be only used by a Java based service client. The converse also holds. A Java service client is capable of using an XML based RPC service deployed on any non Java platform.

The example stock quote service endpoint defines and implements the following Java interface.

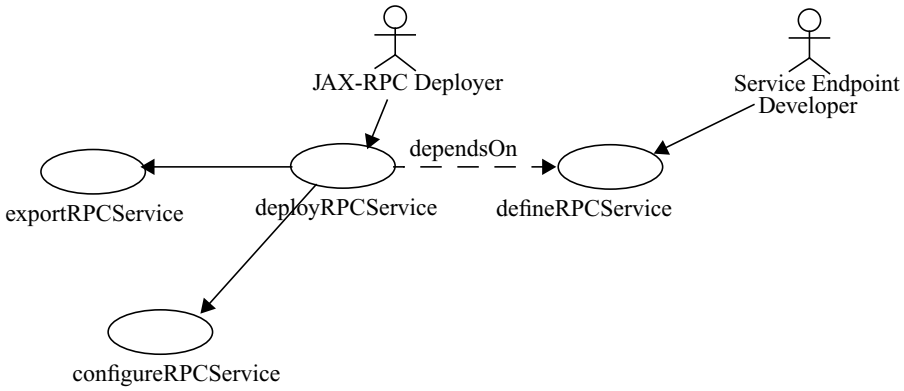
Code Example 1 An example service endpoint interface

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

In this example, stock quote service endpoint definition starts with a Java interface as shown in the above code example. This interface is called a service endpoint interface. Note that the service developer could have started from the stock quote service description in a WSDL document and mapped it to the corresponding Java service endpoint interface.

A JAX-RPC service endpoint can be realized (or implemented) using the J2EE component model. This example uses a stateless session bean for realizing the stock quote service.

The following diagram shows the use case hierarchy view of the JAX-RPC stock quote service. Later chapters of this document specify in detail how these use cases are realized by a JAX-RPC runtime system implementation.



2.1.2 Service Deployment

Once a JAX-RPC service endpoint has been defined and implemented, the JAX-RPC deployer deploys the service on a server-side container based JAX-RPC runtime system. The deployment step depends on the type of component that has been used to realize a JAX-RPC service endpoint.

The example stock quote service endpoint is realized as a stateless session bean and is deployed on an EJB container. The deployment step includes the generation of container specific artifacts (skeleton or tie class) based on the service endpoint interface. A container provided deployment tool provides support for the deployment of JAX-RPC service endpoints.

During the deployment of a JAX-RPC service endpoint, the deployment tool configures one or more protocol bindings for this service endpoint. A binding ties an abstract service endpoint definition to a specific protocol and transport. An example of a binding is SOAP 1.1 protocol binding over HTTP.

Next, the deployment tool creates one or more service endpoints for this JAX-RPC service. Each service endpoint is bound to a specific protocol and transport, and has an assigned endpoint address based on this protocol binding.

2.1.3 Service Description

The deployment tool exports the stock quote service as a WSDL document. The WSDL description of the stock quote service enables service clients (on any platform) to access this service and its endpoints.

A Java-to-WSDL mapping tool (typically part of a container provided deployment tool) maps the example `StockQuoteProvider` service endpoint interface to the following service description in a WSDL document:

Code Example 2 WSDL Description of Stock Quote Service

```
<!-- WSDL Extract... -->
<?xml version="1.0"?>
<definitions name="StockQuoteService"
targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

<types/>
<message name="getLastTradePrice">
   <part name="tickerSymbol" type="xsd:string"/>
</message>
<message name="getLastTradePriceResponse">
   <part name="result" type="xsd:float"/>
</message>

<portType name="StockQuoteProvider">
   <operation name="getLastTradePrice">
      <input message="tns:getLastTradePrice"/>
      <output message="tns:getLastTradePriceResponse"/>
   </operation>
</portType>

<binding name="StockServiceSoapBinding"
       type="tns:StockQuoteProvider">
   <soap:binding style="rpc"
      transport="http:/schemas.xmlsoap.org/soap/http"/>
   <operation name="getLastTradePrice">
   <soap:operation soapAction=""/>
   <input>
      <soap:body use=“encoded”
      namespace="http://example.com/stockquote.wsdl”
      encodingStyle=“http://schemas.xmlsoap.org/soap/encoding”/>
   </input>
   <output>
      <soap:body use=“encoded”
      namespace=""http://example.com/stockquote.wsdl”
      encodingStyle=“http://schemas.xmlsoap.org/soap/encoding”/>
   </output>
   </operation>
</binding>
<service name="StockQuoteService">
   <port name="StockQuoteProviderPort"
```

```
        binding="tns:StockServiceSoapBinding">
    <soap:address location="http://example.com/StockQuoteService"/>
    </port>
</service>
</definitions>
```

In the above WSDL service description, the following are important points to note:

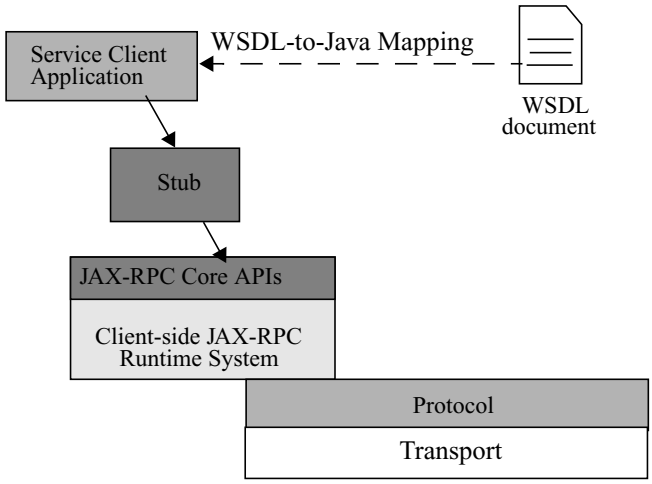
- The `StockQuoteService` includes a single port `StockQuoteProviderPort` with the `StockServiceSoapBinding` binding.
- The binding `StockServiceSoapBinding` binds the `StockQuoteProviderPort` port to the SOAP 1.1 protocol over HTTP.
- The address for the `StockQuoteProviderPort` port is `http://example.com/StockQuoteService`.
- The port type for `StockQuoteProviderPort` is defined as `StockQuoteProvider`. The `StockQuoteProvider` port type includes a single operation `getLastTradePrice` that takes a ticker symbol of the type string and returns a float as the last trade price for this ticker symbol.

Refer to the WSDL specification [7] for complete details.

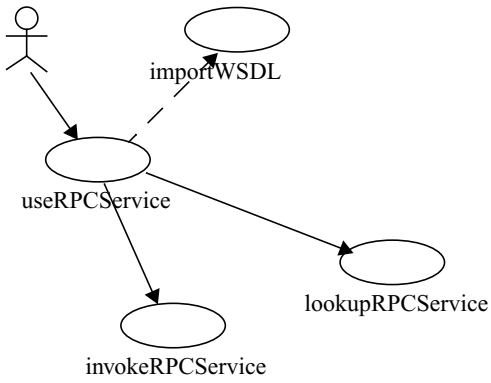
2.1.4 Service Use

A service client uses a JAX-RPC service by invoking remote methods on a service endpoint. Note that a JAX-RPC service client can call a service endpoint that has been defined and deployed on a non-Java platform. The converse is also true.

The following diagram shows how a service client uses stub based model to invoke remote methods on a service endpoint.



The following diagram shows the use case hierarchy for the remote method invocation by a service client.



A service client uses the WSDL document (that describes the stock quote service) to import the stock quote service.

A WSDL-to-Java mapping tool generates client side artifacts (includes stub class, service endpoint interface and additional classes) for the stock quote service and its ports. Note that a service client may use dynamic invocation interface (DII) or a dynamic proxy mechanism instead of a generated stub class to invoke a remote method on a service endpoint.

The JAX-RPC service client programming model describes how a service client looks up and invokes a remote method on a service endpoint. Refer to the chapter 9 (“Service Client Programming Model”) for more details.

The following code snippet shows an illustrative example of how a service client invokes a remote method on the imported stock quote service.

Code Example 3 Invoking a remote method on a JAX-RPC service

```
package com.wombat;
public class ServiceUser {
   // ...
   public void someMethod () {
      com.example.StockQuoteService sqs =
                // ... get access to the StockQuote service
      com.example.StockQuoteProvider sqp =
                sqs.getStockQuoteProviderPort();
      float quotePrice = sqp.getLastTradePrice(“ACME”);
   }
}
```

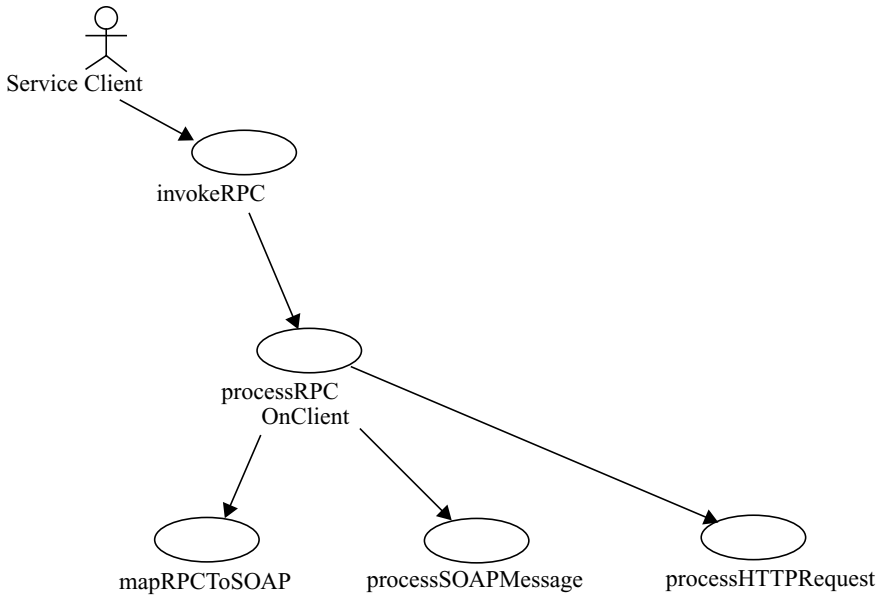
Refer to the chapter 9 for more details on this example.

2.2 JAX-RPC Mechanisms

This section describes a brief overview of the JAX-RPC runtime mechanisms. Note that this description is illustrative. This example assumes SOAP 1.1 protocol with HTTP as the transport.

2.2.1 Service Client

The following use case hierarchy diagram shows how a client-side JAX-RPC runtime system process a remote method invocation on a target service endpoint:



The processing of a remote method call includes the following steps:

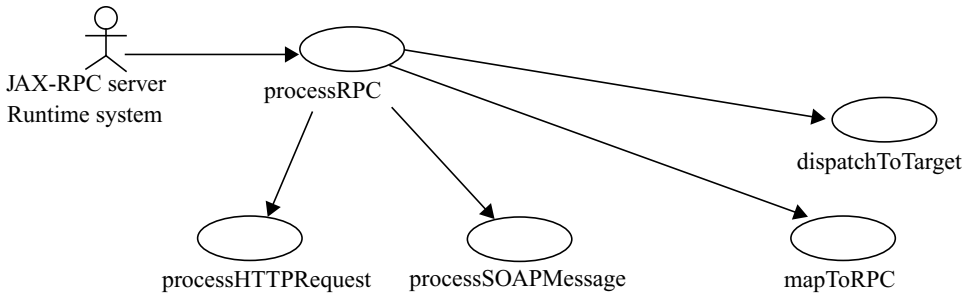
- Mapping of a remote method call to the SOAP message representation: This includes mapping of the parameters, return value and exceptions (for the remote method call) to the corresponding SOAP message; serialization and deserialization based on the mapping between Java types and XML data types.
- Processing of the SOAP message: This includes processing of the SOAP message based on the mapping of this remote method call to the SOAP representation. Encoding style defines how parameters, return value and types in an remote method call are represented in the SOAP message.
- Processing of the HTTP request. This includes transmission of the SOAP request message as part of an HTTP request. A SOAP response message is transmitted as an HTTP response.

2.2.2 Server Side

The diagram shows how server-side JAX-RPC runtime system processes a remote method invocation. Processing of remote method call on the server side includes the following steps:

- Processing of the HTTP request: Server-side JAX-RPC runtime system receives and processes the HTTP request.
- Processing of the SOAP message: JAX-RPC runtime system extracts the SOAP message from the received HTTP request and processes the SOAP message to get access to the SOAP envelope, header, body and any attachments. The processing of the SOAP message uses an XML processing mechanism; examples are streaming parser and SAX based parser.

- Mapping of the SOAP message to a remote method call: JAX-RPC runtime system maps the received SOAP message to a method invocation on the target service endpoint. SOAP body elements carry parameters and return value for this remote call. SOAP header carries any context information that is not part of a remote method signature, but is associated with the remote method call.
- Dispatch to the target JAX-RPC service endpoint: JAX-RPC runtime system invokes method on the target service endpoint based on the mapping of the received remote method invocation. Return value, out parameters and exceptions are carried in the SOAP body and fault elements respectively and are processed as part of the HTTP response.



3 Requirements

This chapter specifies the proposed scope and requirements for the 1.1 version of JAX-RPC specification. These requirements are addressed in detail in the later chapters.

R01 Protocol Bindings

A goal of the JAX-RPC specification is to enable support for multiple protocol bindings that are based on the XML Information Set (Infoset) [20]. For example, SOAP 1.2 messages are specified as XML Infosets. JAX-RPC allows support for binary protocol bindings that are based on the XML infoset but do not carry XML 1.0 documents. Note that the use of term "XML based protocol" in this document is consistent with this goal.

Based on this goal, the JAX-RPC core APIs (defined in the `javax.xml.rpc` package) are defined to be independent of any specific protocol bindings. For the SOAP protocol binding, JAX-RPC specifies APIs in the `javax.xml.rpc.soap` package.

A JAX-RPC runtime system implementation is required to support the SOAP 1.1 with attachments protocol. Refer to the chapter 14 for the interoperability requirements related to the protocol bindings for the JAX-RPC runtime system implementations. SOAP message with attachments [6] defines binding for a SOAP 1.1 message to be carried within a MIME `multipart/related` message.

Note that the required support of SOAP 1.1 with attachments protocol must not preclude or limit use of other protocol bindings and transport in a JAX-RPC runtime system implementation.

Note – The JAX-RPC specification would consider support for the SOAP 1.2 protocol when the SOAP 1.2 W3C specification [5] reaches the final recommendation stage. This would be addressed in the future versions of the JAX-RPC specification.

R02 Transport

A JAX-RPC runtime system implementation is required to support HTTP 1.1 as the transport for SOAP messages. HTTP binding for the SOAP messages is based on the SOAP 1.1 specification [3].

Note that the required support of HTTP 1.1 must not mean that the HTTP transport is the only transport that can be supported by a JAX-RPC runtime system implementation. JAX-RPC core APIs are designed to be transport-neutral. This enables JAX-RPC APIs to be usable with any transport that supports ability to deliver SOAP messages and has a defined protocol binding for the SOAP 1.1 protocol.

JAX-RPC specification does not preclude the use of SOAP binding with a transport that supports security mechanisms. However, the specification of SOAP bindings to transports that support security is outside the scope of the JAX-RPC specification.

A JAX-RPC runtime system implementation is not required to support HTTP/S as the underlying secure transport. Refer to the chapter 14 for the interoperability requirements.

R03 Supported Type Systems

The JAX-RPC specification requires support for the following Java types:

- Java types specified in the section 5.1, “JAX-RPC Supported Java Types”
- Java types specified in the section 7.5, “Mapping between MIME types and Java types”

The JAX-RPC specification requires support for the following XML types:

- XML types specified in section 4.2, “XML to Java Type Mapping”. Refer to the “Appendix: XML Schema Support” for more details on the supported XML types.

R04 XML Encoding for SOAP Messages

The JAX-RPC specification requires support for both encoded and literal representations of a SOAP message representing an RPC call or response.

SOAP 1.1 encoding (also called SOAP 1.1 section 5 encoding) [3] defines rules for the encoding of XML data types. This encoding can be used in conjunction with the mapping of SOAP based RPC calls and responses. The supported data types in the SOAP 1.1 encoding include the following:

- Built-in datatypes specified in the XML Schema Part 2: Datatypes specification [9]. Examples include int, float, string.
- Enumeration as defined in the XML Schema Part 2: Datatypes specification
- Compound types that include Struct and Array
- Serialization and deserialization support for compound types that are neither Struct nor an Array

The use of SOAP 1.1 encoding must not preclude use of any other encoding in a JAX-RPC runtime system implementation. However, the use of a specialized encoding constrains the interoperability of a JAX-RPC runtime system implementation. Refer to the chapter 14 for the interoperability requirements.

R05 JAX-RPC Runtime System

The JAX-RPC runtime system forms the core of a JAX-RPC implementation. JAX-RPC runtime system is a library (part of both service client and server side environments) that provides a set of services required for the JAX-RPC runtime mechanisms.

The JAX-RPC specification identifies the following as the incremental levels of server side implementation of a JAX-RPC runtime system:

- J2SE based JAX-RPC runtime system
- Servlet container based JAX-RPC runtime system
- Standard J2EE container (includes EJB and Web containers) based JAX-RPC runtime system

The JAX-RPC specification identifies a servlet container as a typical implementation of JAX-RPC server side runtime system. Note that the use of the servlet container for the implementation of a JAX-RPC server side runtime system must not preclude any J2SE level implementation of a JAX-RPC runtime system.

The JAX-RPC specification requires a server side JAX-RPC compatible implementation to be either:

- Servlet 2.3 (or higher version) [3] container based JAX-RPC runtime system
- J2EE 1.3 (or higher version) container based JAX-RPC runtime system

The JAX-RPC specification requires a client side JAX-RPC compatible implementation to be based on either J2SE (1.3 version or higher) platform or J2EE (1.3 version or higher) containers. A J2SE based client application is a fully functional and capable client for a JAX-RPC service.

The JAX-RPC specification also supports J2ME (Java 2 platform, Micro Edition) MIDP client as a form of JAX-RPC service client. Note that the programming model specification for the J2ME based JAX-RPC service clients is outside the scope of the JAX-RPC specification.

The JAX-RPC core APIs define the programmatic interface to the JAX-RPC runtime system. Refer to the chapter 8 for the specification of the JAX-RPC core APIs.

R06 Default Type Mapping

The JAX-RPC specification specifies the following standard type mappings:

- Java types to XML datatypes
- XML datatypes to Java types

A JAX-RPC runtime system implementation is required to support these standard type mappings. Refer to the section 4.2, “XML to Java Type Mapping” and section 5.3, “Java to XML Type Mapping” for the specification of the standard type mappings.

A JAX-RPC runtime system implementation is allowed to provide extensions to the standard type mapping. Refer to the requirement R07 for the extensible type mapping support in JAX-RPC.

R07 Extensible Type Mapping

The JAX-RPC specification specifies APIs to support an extensible type mapping and serialization framework. These APIs support development of pluggable serializers and deserializers for mapping between the Java and XML data types. These serializers and deserializers may be packaged as part of a JAX-RPC runtime system implementation or provided by tools vendors or service developers.

A JAX-RPC runtime system implementation uses the extensible type mapping framework to support serialization and deserialization of an extended set of XML and Java data types. The extended set is defined as a super set of the XML and Java data types supported by the standard type mapping specification (refer to the R03).

R08 Service Endpoint Model

The JAX-RPC specification specifies the standard programming model for a service endpoint developed and deployed on a servlet container based JAX-RPC runtime system.

The JAX-RPC specification does not specify a normative J2SE based service endpoint model.

The JAX-RPC specification does not specify the service endpoint model for a JAX-RPC service developed using the standard EJB programming model and deployed on an EJB container. Refer to the JSR-109 [10] and J2EE 1.4 specifications [3] for EJB service endpoint model.

R09 Service Description

The JAX-RPC specification uses the WSDL 1.1 [7] specification for the description of JAX-RPC services. The use of WSDL based service description supports export and import of JAX-RPC services across heterogeneous environments and is required for interoperability.

The standard WSDL-to-Java [refer to the chapter 4] and Java-to-WSDL [refer to the chapter 5] mappings specify the following:

- Mapping between a Java service endpoint interface and abstract WSDL definitions of port type, operation and message
- Binding of abstract WSDL definitions of port type, operations and messages to a specific protocol and transport

A JAX-RPC implementation is not required to support a round-trip mapping between the Java and WSDL based representations of a JAX-RPC service.

R010 Service Registration and Discovery

The JAX-RPC specification considers service registration and discovery as out of scope. The JAX-RPC specification does not address how a JAX-RPC service is registered in a public/private registry and how it is discovered by a service client.

R011 Java API for XML Binding (JAXB)

The JAX-RPC specification does not require the use of JAXB (Java APIs for XML Data Binding) 1.0 [14] for marshalling and unmarshalling of XML data types to and from a Java representation. Note that a JAX-RPC implementation may use JAXB.

A future version of JAX-RPC will consider using JAXB in more integrated manner as the JAXB specification evolves to support XML schema.

R012 Application level Modes of Interaction

The JAX-RPC specification supports the following modes of interaction between a client and service endpoint. Note that these interaction modes are visible as part of the JAX-RPC programming model and are termed application level interaction modes.

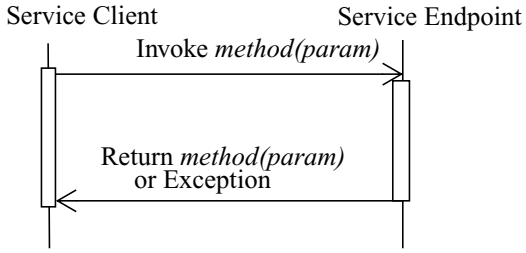
The JAX-RPC specification does not address how a JAX-RPC runtime system implementation provides support for these application level interaction modes. A JAX-RPC runtime system may use more primitive implementation specific interaction modes to implement support for these application level interaction modes.

The JAX-RPC specification requires that any implementation specific mechanisms or implementation level interaction modes must not be exposed to the JAX-RPC programming model.

The JAX-RPC specification does not define any qualities of service QoS (examples: guarantees of message delivery, reliable messaging, use of intermediaries) related to the application level interaction modes. A JAX-RPC runtime system may support such QoS mechanisms. Note that the JAX-RPC specification does not preclude such implementation-specific QoS support.

Synchronous Request-response Mode

A service client invokes a remote method on a target service endpoint and receives a return value or an exception. The client invocation thread blocks while the remote method invocation is processed by the service endpoint. Eventually, the service client gets a return (this may be `void` type) or an exception from the invoked remote method.

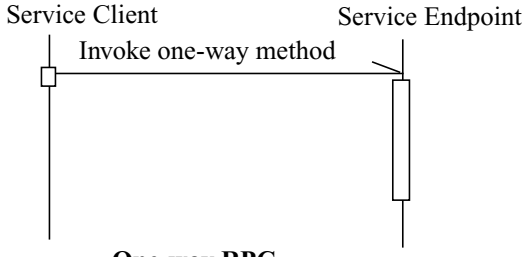


Synchronous Request Response Mode

The JAX-RPC specification does not define how a JAX-RPC runtime system implements support for the synchronous request-response mode in terms of the underlying protocol and transport. Refer to the SOAP 1.2 specification Part 2 [5] for more details on transport message exchange patterns and default HTTP binding.

The JAX-RPC APIs and service client programming model support synchronous request-response mode through both the stub (or dynamic proxy) based model and DII `Call` interface.

One-way RPC Mode



One-way RPC

A service client invokes a remote method on a service endpoint in the one-way mode. The client invocation thread does not block and continues execution without waiting for this remote method invocation to be processed by the service endpoint. The service client does not get any return value, `out` parameters or any remote exception for this method invocation. Note that a JAX-RPC client runtime system may throw an exception during the processing of an one-way RPC call.

The non-blocking behavior, message delivery and processing of the one-way RPC mode depends on the underlying protocol and transport. The JAX-RPC specification does not specify how a JAX-RPC runtime system implements one-way RPC mode in terms of the underlying protocol and transport. For example, HTTP is a request-response protocol. In the one-way RPC mode, the client may handle the HTTP response with either success or error code (but with no entity-body content) as part of the invocation of a one-way RPC. In another case, a JAX-RPC client runtime system may achieve non-blocking behavior for one-way RPC by pipelining multiple HTTP requests without waiting for responses. A J2SE based JAX-RPC client runtime system (targeted for a less restrictive non-managed environment) may choose to create thread for a one-way RPC dispatch.

Note that a client should not rely on any specific guarantees of message delivery and processing semantics or quality of services (QoS) in the one-way RPC mode.

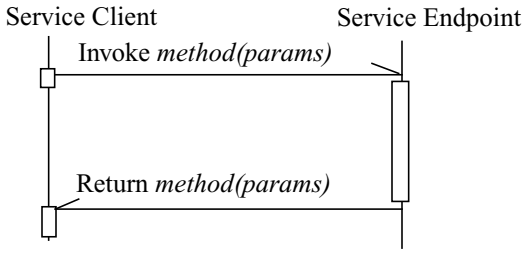
The JAX-RPC specification supports the one-way interaction mode through the DII `Call` interface. Refer to the section 8.2.4, “DII Call Interface” for more details.

The JAX-RPC specification does not specify any standard APIs for the design of asynchronous stubs. This feature will be addressed in the future version of the JAX-RPC specification. This will lead to the support for both one-way and non-blocking RPC interaction modes.

Non-blocking RPC Invocation

A service client invokes a remote method on a service endpoint and continues processing in the same thread without waiting for the return of the remote method invocation. Later, the service client processes the remote method return by performing blocking receive or by polling for the return value. In this case, a service client is responsible for performing the correlation between the remote method call and subsequent response.

A JAX-RPC runtime system is not required to support the non-blocking RPC interaction mode. This interaction mode will be addressed in the future versions of the JAX-RPC specification.



Non-blocking RPC Invocation

R013 Relationship to JAXM and SAAJ

The JAXM (Java API for XML messaging) specification specifies the standard Java APIs to support asynchronous document based messaging based on the SOAP protocol. The JAXM API has two packages:

- The `javax.xml.soap package` specifies API to represent a SOAP message with attachments. This API enables developers to access, create and manipulate a SOAP message. JAXM allows profiles (example: ebXML TRP) to be defined over this base SOAP package. These layered profiles define additional mechanisms, abstractions and conventions on top of the base SOAP abstraction.
- The `javax.xml.messaging` package specifies API for developing clients and message-driven bean based endpoints to support document based asynchronous messaging.

Note that the maintenance release of JAXM 1.1 has created a new specification document named "SOAP with Attachments API for Java (’SAAJ’)" [13] for the `javax.xml.soap` package. The maintenance release also makes the `javax.xml.soap` package independent of the `javax.xml.messaging` package.

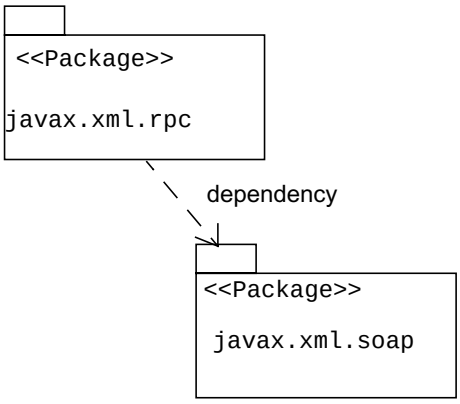
JAX-RPC 1.1 implementations are required to support the SAAJ 1.2 APIs.

In relation to the JAX-RPC specification, the JAXM specification does not define any mapping between WSDL and Java. It also does not define any standard type mapping between the XML data types and Java types.Both JAXM and JAX-RPC use SOAP 1.1 with attachments as the underlying protocol. Refer R01 for the JAX-RPC requirement related to the protocol bindings.

The JAX-RPC specification specifies APIs for the development of SOAP message handlers for SOAP message processing. These SOAP message handlers are based on the `javax.xml.soap` package.

The JAX-RPC specification also defines SOAP protocol binding specific APIs in the `javax.xml.rpc.soap` package. This package uses the Java APIs defined in the `javax.xml.soap` package. JAX-RPC also uses the `javax.xml.soap` APIs to represent mapping of literal fragments carried in a SOAP message.

The following diagram shows the dependency relationship between the `javax.xml.rpc` (specified in JAX-RPC) and `javax.xml.soap` packages:



R014 Parameter Passing semantics

JAX-RPC uses pass by copy semantics for parameter passing in a remote method invocation.

JAX-RPC specification does not define the object-by-reference mode for remote method invocations. Note that the SOAP 1.2 specification[5] does not address objects-by-reference feature as part of its design goals.

R015 Service Context

A remote method call or response may carry context information (termed as *service context*). Examples are service contexts for transaction management (example: unique transaction identifier), security (example: digital signature) or session management.

The JAX-RPC specification specifies a non-normative programming model for the processing of service context. Refer to the chapter 11 (“Service Context”) for more details. Note that the JAX-RPC specification does not (nor intends to, in future) specify the semantic content of the service contexts.

If SOAP is the underlying protocol, service context information is carried in the SOAP header of a SOAP message. Note that neither SOAP 1.1 nor SOAP 1.2 specification defines any standard SOAP header representation for the transaction or security related context.

An explicit goal of the JAX-RPC specification is not to define any SOAP header representation for transaction, security or session related information. A goal of JAX-RPC specification is to leverage work done in other standardization groups for this aspect. An important point to note is that any JAX-RPC specific definition of SOAP headers or session related information is against the design goal of achieving SOAP based interoperability with heterogeneous environments.

R016 SOAP Messages with Attachments

The SOAP 1.1 message with attachments defines a binding for a SOAP message to be carried within a MIME `multipart/related` message. A SOAP message can be transmitted together with attached data in a MIME encoded representation.

The JAX-RPC specification provides support for SOAP message with attachments as the underlying protocol.

A remote method call may include MIME encoded content as a parameter or a return value. A typical use case is passing or return of an XML document or binary image (in JPEG or GIF format) in a remote method call.

Refer to the chapter 7 (“SOAP Message With Attachments”) for more details.

R017 SOAP Message Handler

The JAX-RPC specification specifies the requirements and APIs for the SOAP message handler. A SOAP message handler gets access to the SOAP message that represents either an RPC request or response. A typical use of a SOAP message handler is to process the SOAP header blocks as part of the processing of an RPC request or response.

Note that other types of handlers (for example; stream based handlers, post-binding typed handlers) may also be developed for an implementation of a JAX-RPC runtime system. However, JAX-RPC specification specifies APIs for only the SOAP message handlers. Future versions of the JAX-RPC specification would add support for other types of handlers.

R018 Literal Mode

When the SOAP binding is used, an RPC call with its parameters and return value is assembled inside the body element of a SOAP message. A message part may be either encoded using some encoding rules or may represent a concrete schema definition; the latter is termed literal representation.

The JAX-RPC specification requires support for the literal representation of an RPC request or response in the SOAP body. Refer to the section 6.4, “Literal Representation” for more details.

R019 Application Portability

The JAX-RPC specification requires that service client and service endpoint code be portable across multiple JAX-RPC runtime system implementations.

In the 1.1 version, portable JAX-RPC service client or service endpoint code should not depend on any pluggable vendor-specific serializers and deserializers.

To achieve portability, the JAX-RPC specification does not require portable stubs and skeletons. The stub/skeleton classes and other generated artifacts are generated by a deployment tool (provided with a J2EE container or a JAX-RPC runtime system) during the deployment of a JAX-RPC service endpoint or a service client. In the 1.1 version, these generated artifacts are specific to a JAX-RPC implementation.

4 WSDL/XML to Java Mapping

This chapter specifies the standard mapping of the WSDL definitions to Java representation and mapping of the XML data types to the Java types.

The WSDL/XML to Java mapping specification includes the following:

- Mapping of XML data types to the Java types
- Mapping of abstract definitions of port type, operations and messages to Java interfaces and classes
- Java representation of a `wsdl:port` address specification
- Java representation of a `wsdl:service` definition

This chapter provides illustrative examples of the specified mapping.

4.1 XML Names

XML names in a WSDL document are mapped to the Java identifiers. Refer to the "Appendix: Mapping of XML Names" for more details on the mapping of XML names to Java identifiers.

Any XML names that may be mapped to a reserved Java keyword must avoid any name collision. Any name collisions in the mapped Java code are resolved by prefixing an underscore to the mapped name. Refer to the Java language specification [1] for the list of keywords in the Java language.

4.2 XML to Java Type Mapping

This section specifies the standard type mapping of XML data types to the Java types.

Refer to the "Appendix: XML Schema Support" for the specification of JAX-RPC support for the XML Schema data types as specified in the XML Schema specifications, Part 1 [8] and Part 2 [9], and SOAP 1.1 encoding specification (as specified in the section 5 of the SOAP 1.1 specification].

Note that the rules and format of serialization for XML data types are based on the encoding style. For example, SOAP encoding [4] specifies the default rules of serialization for the simple and compound XML types. Refer to the SOAP specification for more details on the SOAP encoding.

4.2.1 Simple Types

The following table specifies the Java mapping for the built-in simple XML data types. These XML data types are as defined in the XML schema specification [9] and the SOAP 1.1 encoding [http://schemas.xmlsoap.org/soap/encoding/].

TABLE 4-1 Java mapping for the built-in simple XML data types

Simple Type	Java Type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	java.util.Calendar
xsd:date	java.util.Calendar
xsd:time	java.util.Calendar
xsd:anyURI	java.net.URI (J2SE 1.4 only) java.lang.String
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:anySimpleType	java.lang.String

The JAX-RPC specification does not define the standard Java mapping for the xsd:anyType. A JAX-RPC implementation is not required to support the xsd:anyType.

The xsd:anyURI type must be mapped to the java.net.URI class in applications that are intended to run on J2SE 1.4 or later. For compatibility with pre-1.4 environments, JAX-RPC implementations are allowed to map this type to java.lang.String.

Implementations are also required to support the additional data types defined in the XML Schema specification using one of the derivation mechanisms covered in sections 4.2.4, 4.2.5 and 4.2.6 below. Examples of the types covered by this clause include: xsd:token, xsd:nonPositiveInteger, xsd:gDay.

The following types are explicitely excluded from the previous proviso and JAX-RPC implementations are not required to support them: xsd:NOTATION, xsd:ENTITY, xsd:IDREF and their respective derived types.

For clarity, the following table lists the mappings for the remaining data types defined in the XML schema specification. These mappings were derived following the rules given in the preceding paragraphs.

TABLE 4-2 Derived Java mapping for the remaining built-in simple XML data types

Simple Type	Java Type
xsd:duration	java.lang.String
xsd:gYearMonth	java.lang.String
xsd:gYear	java.lang.String
xsd:gMonthDay	java.lang.String
xsd:gDay	java.lang.String
xsd:gMonth	java.lang.String
xsd:normalizedString	java.lang.String
xsd:token	java.lang.String
xsd:language	java.lang.String
xsd:Name	java.lang.String
xsd:NCName	java.lang.String
xsd:ID	java.lang.String
xsd:NMTOKEN	java.lang.String
xsd:NMTOKENS	java.lang.String[]
xsd:nonPositiveInteger	java.math.BigInteger
xsd:negativeInteger	java.math.BigInteger
xsd:nonNegativeInteger	java.math.BigInteger
xsd:unsignedLong	java.math.BigInteger
xsd:positiveInteger	java.math.BigInteger

There are a number of cases in which a built-in simple XML data type must be mapped to the corresponding Java wrapper class for the Java primitive type:

- an element declaration with the `nillable` attribute set to `true`;
- an element declaration with the `minOccurs` attribute set to `0` (zero) and the `maxOccurs` attribute set to `1` (one) or absent;
- an attribute declaration with the `use` attribute set to `optional` or absent and carrying neither the `default` nor the `fixed` attribute;

The following shows examples of each:

```
<xsd:element name="code" type="xsd:int" nillable="true"/>
<xsd:element name="code2" type="xsd:int" minOccurs=″0″/>

<xsd:element name=″description″>
   <xsd:complexType>
      <xsd:sequence/>
      <xsd:attribute name=″code3″ type=″xsd:int″ use=″optional″/>
   </xsd:complexType>
</xsd:element>
```

The element/attribute declarations for `code`, `code2`, `code3` above are all mapped to the `java.lang.Integer` type.

The following table specifies the mapping of element/attribute declarations of the kind given above for the built-in simple XML types.

TABLE 4-3 Java Mapping for the built-in simple XML data types

Element/attribute declarations in which a value may be omitted	Java Type
xsd:int	java.lang.Integer
xsd:long	java.lang.Long
xsd:short	java.lang.Short
xsd:float	java.lang.Float
xsd:double	java.lang.Double
xsd:boolean	java.lang.Boolean
xsd:byte	java.lang.Byte

The SOAP 1.1 specification indicates that all SOAP encoded elements are `nillable`. So in the SOAP encoded case, a SOAP encoded simple XML type is mapped to the corresponding Java wrapper class for the Java primitive type. An example is mapping of the `soapenc:int` to the `java.lang.Integer`.The following table shows the Java mapping of the SOAP encoded simple types.

TABLE 4-4 Java Mapping for the SOAP encoded XML data types

SOAP Encoded Simple Type	Java Type
soapenc:string	java.lang.String
soapenc:boolean	java.lang.Boolean
soapenc:float	java.lang.Float
soapenc:double	java.lang.Double
soapenc:decimal	java.math.BigDecimal
soapenc:int	java.lang.Integer
soapenc:short	java.lang.Short
soapenc:byte	java.lang.Byte
soapenc:base64	byte[]

4.2.2 Array

An XML array is mapped to a Java array with the operator `[]`. The JAX-RPC specification requires support for the following types of XML array definitions:

- An array derived from the `soapenc:Array` by restriction using the `wsdl:arrayType` attribute. This case is specified in the WSDL 1.1 [7] specification
- An array derived from `soapenc:Array` by restriction as specified in the SOAP 1.1 specification [4]

The type of Java array element is determined based on the schema for the XML array. Note that the array dimension is omitted in the declaration of a Java array. The number of elements in a Java array is determined at the creation time rather than when an array is declared.

The standard type mapping supports XML arrays with multiple dimensions.

Example

The following shows an example of an array derived from the `soapenc:Array` by restriction using the `wsdl:arrayType` attribute. This array maps to the Java `int[]`:

```
<complexType name="ArrayOfInt">
   <complexContent>
      <restriction base="soapenc:Array">
         <attribute ref="soapenc:arrayType"
                    wsdl:arrayType="xsd:int[]"/>
      </restriction>
   </complexContent>
</complexType>
```

The following example shows an XML array formed by the restriction of the `soapenc:Array`.

```
<!-- Schema fragment -->
<complexType name=”ArrayOfPhoneNumbers”>
   <complexContent>
      <restriction base=”soapenc:Array”>
      <sequence>
         <element name=”phoneNumber”
            type=”xsd:string” maxOccurs=”unbounded”/>
      </sequence>
      </restriction>
   </complexContent>
</complexType>
```

The above XML array maps to a `String[]` in the Java mapping. In this example, `java.lang.String` (mapping of the base element type `xsd:string` in the above XML array) is used as the element type in the mapped Java array.

The following example shows a schema fragment and instance for a polymorphic array.

```
<!-- Schema fragment -->
<element name=”myNumbers” type=”soapenc:Array”/>

<!-- Schema instance -->
<myNumbers soapenc:arrayType=”xsd:int[2]”>
   <number>1</number>
   <number>2</number>
</myNumbers>
```

The above XML array maps to a Java array of `java.lang.Object`. The operator `[]` is applied to the mapped Java array. Note that above XML array is not mapped to a Java array of integers, since the type (`xsd:int`) of array members is determined by the inspection of the `soapenc:arrayType` attribute in the schema instance.

An array can also contain struct values. The following schema fragment shows an example:

```
<!-- XML schema fragment -->
<complexType name=”Book”>
   <all>
      <element name=”author” type=”xsd:string”/>
      <element name=”preface” type=”xsd:string”/>
      <element name=”price” type=”xsd:float”/>
```

```
    </all>
</complexType>

<complexType name=″ArrayOfBooks″>
<complexContent>
    <restriction base=″soapenc:Array″>
    <sequence>
        <element name=″book″ type=″tns:Book″ maxOccurs=″unbounded″/>
    <sequence>
    </restriction>
</complexContext>
</complexType>
```

The above XML array maps to `Book[]` in the Java mapping. Refer to the Java mapping of an XML struct for details on how `Book` type has been mapped.

4.2.3 XML Struct and Complex Type

The JAX-RPC specification supports the mapping of the following types of XML struct:

- The `xsd:complexType` with both sequence of elements of either simple or complex type. Refer to the `xsd:sequence` [9]
- The `xsd:complexType` with `xsd:all` [9] based unordered grouping of elements of either simple or complex type
- The `xsd:complexType` with `xsd:simpleContent` used to declare a complex type by extension of an existing simple type

In all cases, attribute uses specified using either the `xsd:attribute` or `xsd:attributeGroup` elements are supported.

An XML struct maps to a JavaBeans class with the same name as the type of the XML struct. If the struct is anonymous, then the name of the nearest enclosing `xsd:element`, `xsd:complexType` or `xsd:simpleType` is used instead.

The mapped JavaBeans class provides a pair of getter and setter methods for each property mapped from the member elements and attributes of the XML struct.

The identifier and Java type of a property in the JavaBeans class is mapped from the `name` and `type` of the corresponding member element (or attribute) in the XML struct. Refer to the section 4.1, “XML Names” for the mapping of XML names to Java identifiers.

Note that, according to JavaBeans conventions, the getter method for a boolean property uses the prefix “is” instead of “get”, e.g. `isRequired()`.

For complex types defined using `xsd:simpleContent` and extending a simple type T, the corresponding JavaBean class will contain an additional property named “`_value`“ and whose type is mapped from the simple type T according to the rules in this specification.

The instances of the mapped JavaBeans class must be capable of marshaling to and from the corresponding XML struct representation.

An element in a complex type with the `maxOccurs` attribute set to a non-negative integer greater than 1 or `unbounded` is mapped to a Java array with a pair of setter and getter methods in the JavaBeans class. The Java type of the array is mapped from the `type` attribute of the XML element. Refer to the following example.

The JAX-RPC specification does not require support for all different combinations of the occurrence constraints (`minOccurs`, `maxOccurs`).

Additionally, the `xsd:any` element can occur within complex type declarations to represent *element wildcards*. In this context, it will result in an additional property on the JavaBean corresponding to the containing complex type. This property will be called "_any" and will have `javax.xml.soap.SOAPElement` as its type, unless the `xml:any` element has a `maxOccurs` attribute with a value greater than 1, in which case its type will be `javax.xml.soap.SOAPElement[]`.

Example

The following example shows a struct of the type `Book` and its schema fragment and instance:

```
<!-- XML Schema fragment -->
<complexType name="Book">
   <sequence>
   <element name="authors" type="xsd:string" maxOccurs="10"/>
   <element name="preface" type="xsd:string"/>
   <element name="price" type="xsd:float"/>
   </sequence>
</complexType>
```

The above XML struct is mapped to a JavaBeans class as follows:

```
// Java
public class Book {
   // ...
   public String[] getAuthors() { ... }
   public void setAuthors(String[] authors) { ... }

   public String getPreface() { ... }
   public void setPreface(String preface) { ... }
   public float getPrice() { ... }
   public void setPrice(float price) { ... }
}
```

Example

The following schema fragment shows a complex type derived by extension from the xsd:string simple type:

```
<!-- XML Schema fragment -->
<complexType name="CountedString">
   <simpleContent>
      <extension base="xsd:string>
         <attribute name="counter" type="xsd:int"/>
      </extension>
   </simpleContent>
</complexType>
```

The complex type above is mapped to the following JavaBeans class:

```
// Java
public class CountedString {
   // ...
   public String get_value() { ... }
   public void set_value(String value) { ... }

   public int getCounter() { ... }
   public void setCounter(int counter) { ... }
}
```

4.2.4 Enumeration

An XML enumeration is a specific list of distinct values appropriate for a base type. The XML Schema Part 2: Datatypes specification supports enumerations for all simple built-in types except for `xsd:boolean`.

An XML enumeration is mapped by default to a Java class with the same name as the enumeration type. If the enumeration is anonymous, then the name of the nearest enclosing `xsd:attribute`, `xsd:element`, `xsd:simpleType` or `xsd:complexType` is used instead.

In order to be compatible with the JAXB 1.0 specification [14], in addition to the default mapping given above JAX-RPC implementations are required to support mapping anonymous enumerations using the rules for simple types derived via restriction given in section 4.2.5.

The mapped Java class declares a `getValue` method, two static data members per label, an integer conversion method and a constructor as follows:

```
//Java
public class <enumeration_name> {
   // ...

   // Constructor
   protected <enumeration_name>(<base_type> value) { ... }

   // One for each label in the enumeration
   public static final <base_type> _<label> = <value>;
   public static final <enumeration_name> <label> =
          new <enumeration_name>(_<label>);

   // Gets the value for a enumerated value
   public <base_type> getValue() {...}

   // Gets enumeration with a specific value
   // Required to throw java.lang.IllegalArgumentException if
   // any invalid value is specified
   public static <enumeration_name> fromValue(<base_type> value) {
                  ... }

   // Gets enumeration from a String
   // Required to throw java.lang.IllegalArgumentException if
   // any invalid value is specified
   public static <enumeration_name> fromString(String value){ ... }

   // Returns String representation of the enumerated value
   public String toString() { ... }

   public boolean equals(Object obj) { ... }
   public int hashCode() { ... }
}
```

All _<label> and <label> used in the mapped Java class (for XML enumeration) are required to be valid Java identifiers. According to the Java language specification, a Java identifier cannot have the same spelling as a Java keyword, `Boolean` literal or `null` literal.

If one or more enumerated values in an XML enumeration cannot map to valid Java identifiers (examples are “3.14”, “int”), the mapped Java class is required to use Java identifiers `value<1..N>` and `_value<1..N>` for `<label>` and `_<label>` (as in the above mapping code snippet) respectively. The numeric suffix starts from 1 and increments by 1 per value in the XML enumeration. Examples are `_value1`, `value1`, `value2` and `_value2`.

Example

The following shows an example of XML enumeration and its schema fragment:

```
<!-- XML Schema fragment -->
<element name=″EyeColor″ type=″EyeColorType″/>
<simpleType name=″EyeColorType″>
<restriction base=″xsd:string″>
   <enumeration value=″green″/>
   <enumeration value=″blue″/>
</restriction>
</simpleType>

<!-- XML Schema instance -->
<EyeColor>green</EyeColor>
```

The following code snippet show the Java mapping for the above XML enumeration:

```
//Java
public class EyeColorType {
   // Constructor
   protected EyeColorType(String value) { ... }

   public static final String _green = “green”;
   public static final String _blue = “blue”;

   public static final EyeColorType green = new EyeColorType(_green);
   public static final EyeColorType blue = new EyeColorType(_blue);

   public String getValue() { ... }
   public static EyeColorType fromValue(String value) { ... }

   public boolean equals(Object obj) { ... }
   public int hashCode() { ... }
   // ... Other methods not shown
}
```

Here’s the same XML enumeration type, this type defined as an anonymous type:

```
<!-- XML Schema fragment -->
<element name=″EyeColor″>
   <simpleType>
      <restriction base=″xsd:string″>
         <enumeration value=″green″/>
         <enumeration value=″blue″/>
      </restriction>
   </simpleType>
</element>
```

The default mapping for this enumeration type is the same as for the non-anonymous case. Additionally, implementations are also required to support mapping this enumeration type to the `java.lang.String` type, so as to be compatible with the JAXB 1.0 specification.

4.2.5 Simple Types Derived By Restriction

The XML Schema specification allows the definition of new simple types obtained by restricting an existing simple type. Restrictions on the set of allowed values are specified using one or more of 12 predefined facets.

A simple type derived by restriction from another simple type, referred to as its *base type*, is mapped to the same Java type that its base type is mapped to. If its base type does not have a standard JAX-RPC mapping (i.e. is unsupported), then the derived type itself is unsupported.

Several built-in types defined in the XML Schema specification are derived by restriction from other types for which JAX-RPC provides a standard mapping. Such types must in consequence be mapped according to the rules in the preceding paragraphs.

Example

The built-in `xsd:normalizedString` type is derived by restriction from `xsd:string`, hence it must be mapped to `java.lang.String`.

Example

The following schema fragment defines a new simple type by restriction of the `xsd:int` type. Consequently, it must be mapped to the Java `int` type.

```
<!-- XML Schema fragment -->
<simpleType name=″OneToTenType″>
   <restriction base=″xsd:int″>
      <minInclusive value=″1″/>
      <maxInclusive value=″10″/>
   </restriction>
</simpleType>
```

4.2.6 Simple Types Derived Using xsd:list

In XML Schema, simple type definitions can specify xsd:list as their derivation mechanism. In this case, they also specify an `item type` which must itself be a simple type.

Simple types defined using `xsd:list` and whose item type has a standard JAX-RPC mapping are mapped to arrays.These types include all built-in XML Schema types defined in the same way, such as `xsd:NMTOKENS`. The component type of the resulting array is mapped from the item type of the simpel type.

Example

The following schema fragment defines a new simple type “list of QNames” which is mapped to the `javax.xml.namespace.QName[]` type.

```
<!-- XML Schema fragment -->
<simpleType name=″QNameList″>
   <list itemType=″xsd:QName″/>
</simpleType>
```

4.3 WSDL to Java Mapping

This section specifies the mapping of a service described in a WSDL document to the corresponding Java representation.

4.3.1 WSDL Document

A WSDL document is mapped to a Java package. The fully qualified name of the mapped Java package is specific to an application and is specified during the WSDL to Java mapping. A WSDL to Java mapping tool is required to support the configuration of the application specific package name during the mapping.

Note that the JAX-RPC specification does not specify the standard mapping of a namespace definition (in a WSDL document) to the corresponding Java package name. However, the JAX-RPC requires that a namespace definition in a WSDL document must be mapped to a unique Java package name. The name of the mapped Java package must follow the package naming conventions defined in the Java Language Specification [1].

The WSDL 1.1 specification allows references to the various WSDL definitions (examples: `portType`, `message`). Such `QName` based references in WSDL are mapped based on the Java package and name scoping conventions.

4.3.2 Extensibility Elements

The WSDL 1.1 specification allows definition of extensibility elements (that may be specific to a binding or technology) under various element definitions.

The JAX-RPC specification specifies mapping of the extensibility elements for SOAP and MIME bindings. Refer to the chapter 6 (“SOAP Binding”) and section 7.4, “WSDL Requirements”. However, the JAX-RPC specification does not address mapping of any vendor specific extensibility elements. A JAX-RPC implementation may support mapping of WSDL extensibility elements at the cost of interoperability and application portability.

4.3.3 WSDL Port Type

A WSDL port type is a named set of abstract operations and messages involved.

A WSDL port type is mapped to a Java interface (termed a Service Endpoint Interface) that extends the `java.rmi.Remote` interface. The mapping of a `wsdl:portType` to a service endpoint interface may use the `wsdl:binding` element. Refer to the section 6.1, “SOAP Binding in WSDL” for details on the use of the `soap:binding` definition in the mapping of a WSDL port type.

The `name` of the Service endpoint interface is mapped from the `name` attribute of the `wsdl:portType` element. Note that a port type name attribute defines a unique name among all the port types defined in an enclosing WSDL document. Refer to the section 4.1, “XML Names” for the mapping of the XML names to Java identifiers.

The mapped Java service endpoint interface contains methods mapped from the `wsdl:operation` elements defined in the `wsdl:portType`. Refer to the section 4.3.4, "WSDL Operation" for the standard mapping of a `wsdl:operation` definition.

Since WSDL does not support inheritance of the port types, the standard Java mapping of the WSDL port type does not define support for the inheritance of the mapped Java interfaces.

Each method of the mapped Java interface is required to declare `java.rmi.RemoteException` in its throws clause. A `RemoteException` is the common superclass for exceptions related to a remote method invocation. Examples are: `java.rmi.MarshalException`, `java.rmi.ConnectException`. Refer to the J2SE [2] documentation for more details on the `RemoteException`.

A method may also throw service specific exceptions based on the mapping of a WSDL faults. Refer to the section 4.3.6, "WSDL Fault" for more details.

Example

The following is an example of a port type definition in a WSDL document:

```
<!-- WSDL Extract -->
<message name="getLastTradePrice">
   <part name="tickerSymbol" type="xsd:string"/>
</message>
<message name="getLastTradePriceResponse">
   <part name="result" type="xsd:float"/>
</message>

<portType name="StockQuoteProvider">
   <operation name="getLastTradePrice"
              parameterOrder="tickerSymbol">
      <input message="tns:getLastTradePrice"/>
      <output message="tns:getLastTradePriceResponse"/>
   </operation>
</portType>
```

The above WSDL port type definition maps to the following Java service endpoint interface:

```
//Java
public interface StockQuoteProvider extends java.rmi.Remote {
   float getLastTradePrice(String tickerSymbol)
          throws java.rmi.RemoteException;
}
```

4.3.4 WSDL Operation

A `wsdl:operation` defined in a `wsdl:portType` maps to a Java method on the mapped Java service endpoint interface. The mapping of a `wsdl:operation` to a Java method may include the use of the `wsdl:binding` element. Refer to the section 6.1, "SOAP Binding in WSDL" for the use of the `soap:binding` element in the mapping of a WSDL operation.

A `wsdl:operation` is named by the `name` attribute. The operation `name` maps to the name of the corresponding method on the mapped Java service endpoint interface. Refer to the section 4.1, "XML Names" for the mapping of XML names to Java identifiers.

Note that the WSDL 1.1 specification does not require that operation names be unique. To support overloading of operations, the `wsdl:binding` element identifies correct operation by providing the `name` attributes of the corresponding `wsdl:input` and `wsdl:output` elements. Refer to the WSDL 1.1 for complete details.

In the WSDL to Java mapping, overloaded operations are mapped to overloaded Java methods provided the overloading does not cause any conflicts in the mapped Java interface declaration and follows the Java Language specification [1].

The JAX-RPC specification supports the mapping of operations with `request-response` and `one-way` transmission primitives. The standard Java mapping of operations defined with other transmission primitives (`notification`, `solicit-response`) is considered out of scope in the JAX-RPC specification.

The message parts in the `wsdl:input` and `wsdl:output` elements defined in an abstract WSDL operation are mapped to parameters on the corresponding Java method signature. The name of the Java method parameter is mapped from the `name` attribute of the corresponding message `part`. The optional `wsdl:fault` element maps to an exception. Refer to the section 4.3.6, “WSDL Fault” for more details on the Java mapping of WSDL faults.

Parameter Passing Modes

The JAX-RPC specification does not support a pass by reference mode for a remote service. JAX-RPC does not support passing of a `java.rmi.Remote` instance in a remote method invocation.

WSDL parameterOrder

According to the WSDL 1.1 specification, a request-response operation may specify a list of parameters using the `parameterOrder` attribute. The value of the `parameterOrder` attribute follows these rules:

- The `parameterOrder` attribute reflects the order of the parameters in the RPC signature
- The return value part is not present in the `parameterOrder` list
- If a part name appears in both the input and output message with the same type, it is an `inout` parameter
- If a part name appears in only the `wsdl:input` message, it is an `in` parameter
- If a part name appears in only the `wsdl:output` message, it is an `out` parameter

The order of parameters in an RPC signature follows these rules:

- Part names are either listed in the `parameterOrder` attribute or are unlisted. If there is no `parameterOrder` attribute, then all part names are considered unlisted.
- If the `parameterOrder` attribute is specified, then all part names from the input message must be listed. The part names from the output message may or may not be listed.
- Listed part names appear first in the method signature in the order that they are listed in the `parameterOrder` attribute.
- Unlisted part names appear following the listed part names in the order in which these parts appear in the message: first, the input message’s part names; next, the output message’s part names. If an unlisted part is a component of an `inout` parameter, then it appears in the order in which its corresponding part appears in the input message (the order of output message parts is ignored for `inout` parameters).
- If there is a single unlisted output part that is not a component of an inout parameter, then it is the return type. Otherwise, the return type is `void`.

The JAX-RPC specification specifies the following rules for the `in`, `out` and `inout` parameter passing modes and return value:

- An `in` parameter is passed as copy. The value of the `in` parameter is copied before a remote method invocation.
- The return value is created as a copy and returned to the caller from a remote method invocation. The caller becomes the owner of the returned object after completion of the remote method invocation.
- The `out` and `inout` parameters are passed by copy. Parameter passing mode for `out` and `inout` parameters uses `Holder` classes. Refer to the section 4.3.5, “Holder Classes”. A service client provides an instance of a `Holder` class that is passed by value for either `out` or `inout` parameter. The contents of the `Holder` class are modified in the remote method invocation and the service client uses the changed contents after the method invocation returns.

Example

The following is an example of a port type definition in a WSDL document:

```
<!-- WSDL Example -->
<message name=”StockQuoteInput”>
   <part name=”tickerSymbol” type=”xsd:string”/>
</message>
<message name=”StockQuoteOutput”>
   <part name=”lastTradePrice” type=”xsd:float”/>
   <part name=”volume” type=”xsd:int”/>
   <part name=”bid” type=”xsd:float”/>
   <part name=”ask” type=”xsd:float”/>
</message>

<portType name=”StockQuoteProvider”>
   <operation name=”GetStockQuote”
             parameterOrder=”tickerSymbol volume bid ask”>
      <input message=”tns:StockQuoteInput”/>
      <output message=tns:StockQuoteOutput”/>
   </operation>
</portType>
```

The above `wsdl:portType` definition maps to the following Java service endpoint interface:

```
//Java
package com.example;
public interface StockQuoteProvider extends java.rmi.Remote {
   // Method returns last trade price
   float getStockQuote(String tickerSymbol,
                       javax.xml.rpc.holders.IntHolder volume,
                       javax.xml.rpc.holders.FloatHolder bid,
                       javax.xml.rpc.holders.FloatHolder ask)
          throws java.rmi.RemoteException;
}
```

4.3.5 Holder Classes

The JAX-RPC specification requires use of the `Holder` classes as part of the standard Java mapping of a WSDL operation. The use of `Holder` classes enables the mapping to preserve the intended `wsdl:operation` signature and parameter passing semantics.

The JAX-RPC specification includes `Holder` classes for the Java mapping of the simple XML data types (including `xsd:base64Binary`, `xsd.hexBinary`, `soapenc:base64`) in the `javax.xml.rpc.holders` package. Refer to the section 4.2.1, “Simple Types” for the Java mapping of the simple XML data types.

The `Holder` classes for the Java primitive types are defined in the `javax.xml.rpc.holders` package. The name of the `Holder` class is the name of the Java primitive type appended with the suffix `Holder`. The initial letter of the Java type name is capitalized. An example is `javax.xml.rpc.holders.FloatHolder` for the Java primitive type `float`.

The name of the `Holder` class for a Java wrapper class (that wraps a primitive Java type) is derived by appending the suffix `WrapperHolder` to the name of the wrapper class. These holders for the Java wrapper classes are also packaged in the `javax.xml.rpc.holders` package. An example is the `javax.xml.rpc.holders.FloatWrapperHolder` class for the wrapper class `java.lang.Float`.

A WSDL to Java mapping tool generates `Holder` classes for XML data types other than the simple XML data types. Examples are `Holder` classes generated for the complex XML types and extended simple XML data types based on the XML schema specification.

For the complex XML data types, the name of the `Holder` class is constructed by appending `Holder` to the name of the corresponding Java class. These generated `Holder` classes are packaged as part of the generated sub package named `holders` in the WSDL to Java mapping. Refer to the section 4.3.1 for the mapping of the Java package. An example is `com.example.holders.BookHolder`.

The name of the `Holder` class in the Java mapping of an XML array is the name of the complex type (first letter capitalized) appended with the `Holder` suffix. Refer to the example later in this section.

Each `Holder` class provides the following methods and fields:

- A public field named `value`. The type of `value` is the mapped Java type
- A default constructor that initializes the `value` field to a default value. The default values for the Java types are as specified in the Java Language Specification [1]
- A constructor that sets the `value` field to the passed parameter

A standard or generated holder class is required to implement the marker `javax.xml.rpc.holders.Holder` interface. The following code snippet shows the `Holder` interface:

```
package javax.xml.rpc.holders;
public interface Holder {
}
```

A JAX-RPC implementation is required to support serialization and deserialization of the value contained in a `Holder` class. This requirement holds for all types of `Holder` classes irrespective of whether a `Holder` class is generated or is part of the standard `javax.xml.rpc.holders` package.

The following are the standard `Holder` classes specified in the `javax.xml.rpc.holders` package:

- `BigDecimalHolder`
- `BigIntegerHolder`
- `BooleanHolder`
- `BooleanWrapperHolder`
- `ByteArrayHolder`
- `ByteHolder`

- ByteWrapperHolder
- CalendarHolder
- DoubleHolder
- DoubleWrapperHolder
- FloatHolder
- FloatWrapperHolder
- IntHolder
- IntegerWrapperHolder
- LongHolder
- LongWrapperHolder
- ObjectHolder
- QNameHolder
- ShortHolder
- ShortWrapperHolder
- StringHolder

Example

The following is an example of a Holder class for a simple XML data type:

```
//Java
package javax.xml.rpc.holders;
public final class ShortHolder implements Holder {
   public short value;

   public ShortHolder() { }
   public ShortHolder(short value) {
      this.value = value;
   }
}
```

The following is an example of a Holder class for a compound or an extended simple XML type.

```
//Java
package com.example.holders; // Mapped from the WSDL document naming
final public class <Foo>Holder
          implements javax.xml.rpc.holders.Holder {
   public <Foo> value;

   public <Foo>Holder() { ... }
   public <Foo>Holder(<Foo> value) { ... }
}
```

The following shows an example of an array derived from the soapenc:Array by restriction using the wsdl:arrayType attribute. This array maps to the Java int[]:

```
<!-- Example -->
<complexType name="ArrayOfInt">
<complexContent>
   <restriction base="soapenc:Array">
      <attribute ref="soapenc:arrayType"
                 wsdl:arrayType="xsd:int[]"/>
   </restriction>
</complexContent>
</complexType>
```

The holder class for the above array is named ArrayOfIntHolder. The name of the holder class is derived from the name of the complex type by appending suffix Holder..

4.3.6 WSDL Fault

The `wsdl:fault` element (an optional element in a `wsdl:operation`) specifies the abstract message format for any error messages that may be output as a result of a remote operation. According to the WSDL specification, a fault message must have a single part.

A `wsdl:fault` is mapped to either a `java.rmi.RemoteException` (or its subclass), service specific Java exception (described later in this section) or a `javax.xml.rpc.soap.SOAPFaultException`. Refer to the section 6.5, "SOAP Fault" for more details on the Java mapping of a WSDL fault based on the SOAP binding.

Refer to the section 14.3.6, "Mapping of Remote Exceptions" for the mapping between the standard SOAP faults [5] and the `java.rmi.RemoteException`.

Service Specific Exception

A service specific Java exception (mapped from a `wsdl:fault` and the corresponding `wsdl:message`) extends the class `java.lang.Exception` directly or indirectly.

The single message part in the `wsdl:message` (referenced from the `wsdl:fault` element) may be either a type or an element. If the former, it can be either a `xsd:complexType` or a simple XML type.

Each element inside the `xsd:complexType` is mapped to a getter method and a parameter in the constructor of the Java exception. Mapping of these elements follows the standard XML to Java type mapping. The name of the Java exception class is mapped from the `name` attribute of the `xsd:complexType` for the single message part. This naming scheme enables the WSDL to Java mapping to map an `xsd:complexType` derivation hierarchy to the corresponding Java exception class hierarchy. The following section illustrates an example. Refer to the section 4.1, "XML Names" for the mapping of XML names to Java identifiers.

If the single message part in the `wsdl:message` (referenced from the `wsdl:fault` element) has a simple XML type or array, then this element is mapped to a getter method and a parameter in the constructor of the Java exception. In this case, the `name` of the Java exception class is mapped from the `name` attribute of the `wsdl:message` element.

If the single message part in the `wsdl:message` refers to an element, then the type of that element is used to derive the corresponding Java exception class using the rules in the preceding paragraph.

The mapped service specific Java exception is declared as a checked exception in the corresponding Java method mapping for the `wsdl:operation` element. This is in addition to the required `java.rmi.RemoteException`.

Example

The following shows an example of the mapping of a `wsdl:fault` to a service specific Java exception. The `wsdl:message` has a single `part` of type `xsd:string`:

```
<!-- WSDL Extract -->
<message name="InvalidTickerException">
   <part name="tickerSymbol" type="xsd:string"/>
</message>
<portType name="StockQuoteProvider">
   <operation name="getLastTradePrice" ...>
      <input message="tns:getLastTradePrice"/>
      <output message="tns:getLastTradePriceResponse"/>
```

```
      <fault name=”InvalidTickerException”
             message=”tns:InvalidTickerException”/>
   </operation>
</portType>
```

The following is the Java service endpoint interface derived from the above WSDL port type definition. Note that the `getLastTradePrice` method throws the `InvalidTickerException` based on the mapping of the corresponding `wsdl:fault`:

```
package com.example;
public interface StockQuoteProvider extends java.rmi.Remote {
   float getLastTradePrice(String tickerSymbol)
          throws java.rmi.RemoteException,
                 com.example.InvalidTickerException;
}
```

In this example, the `wsdl:fault` element is mapped to a Java exception `com.example.InvalidTickerException` that extends the `java.lang.Exception` class. The name of Java exception is mapped from the name of the `wsdl:message` referenced by the `message` attribute of the `wsdl:fault` element. The following code snippet shows the mapped exception.

```
package com.example;
public class InvalidTickerException extends java.lang.Exception {
   public InvalidTickerException(String tickerSymbol) { ... }
   public getTickerSymbol() { ... }

}
```

Consider another example with the following WSDL extract:

```
<!-- WSDL Extract... -->
<xsd:complexType name="BaseComplexType">
   <xsd:sequence>
      <!-- elements not shown... -->
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ExtendedComplexType">
   <complexContent>
   <extension base="BaseComplexType">
      <xsd:sequence>
      <!-- elements not shown... -->
      </xsd:sequence>
   </extension>
   </complexContent>
</xsd:complexType>

<!-- WSDL fault message... -->
<message name="FaultMessage">
   <part name="info" type="tns:BaseComplexType"/>
</message>
<!-- fault defined within a wsdl:operation... -->
<operation name="...">
   <!-- details not shown -->
   <fault name="FaultMessage" message="tns:FaultMessage"/>
</operation>
```

In this example, the single message `part` in the `wsdl:message` (referenced from the `wsdl:fault`) represents an `xsd:complexType`. The `ExtendedComplexType` derives from the `BaseComplexType`. The above example maps to the following Java exception class hierarchy:

```
package com.example;
public class BaseComplexType extends java.lang.Exception {
   // ...
}
```

```
class ExtendedComplexType extends BaseComplexType {
   // ...
}
```

4.3.7 WSDL Binding

A `wsdl:binding` defines concrete message format and protocol binding for the abstract operations, messages and port types specified in a WSDL document. An example of a binding is the `soap:binding` that defines a binding for SOAP 1.1 service ports.

The JAX-RPC specification does not define a standard Java representation of the `wsdl:binding` element.

4.3.8 WSDL Port

A `wsdl:port` element specifies an address for a service port (or endpoint) based on the specified protocol binding. A `wsdl:port` should have a unique name among all ports defined within an enclosing WSDL document.

In the JAX-RPC service client programming model, a service endpoint (defined using `wsdl:port`) is accessed using an instance of a generated stub class, a dynamic proxy or a `Call` object. Refer to the section 4.3.9, “WSDL Service” for details on how a stub instance and dynamic proxy are created.

4.3.9 WSDL Service

A `wsdl:service` groups a set of service endpoints (or ports), with each service endpoint defined with specific port type, binding and endpoint address.

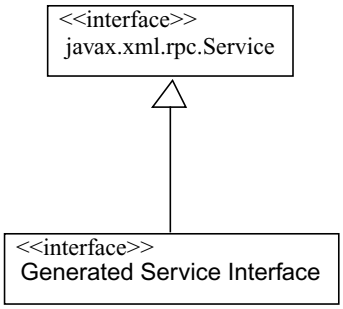
The JAX-RPC specification defines the mapping of a `wsdl:service` element to a service class. A service class acts as a factory of the following:

- Dynamic proxy for a service endpoint. Refer to the section 8.2.3, “Dynamic Proxy”.
- Instance of the type `javax.xml.rpc.Call` for the dynamic invocation of a remote operation on a service endpoint. Refer to the section 8.2.4, “DII Call Interface”.
- Instance of a generated stub class. Refer to the section 8.2.1, “Generated Stub Class”.

A service class implements one of the following interfaces:

- The base `javax.xml.rpc.Service` interface directly, or,
- A generated service interface. This service interface is generated during the WSDL-to-Java mapping and extends the base `javax.xml.rpc.Service` interface. An example of a generated service interface is `com.example.StockQuoteService`.

The following diagram shows the service interface hierarchy.



4.3.10 Service Interface

The following code snippet shows the `javax.xml.rpc.Service` interface:

Code Example 4 javax.xml.rpc.Service interface

```
package javax.xml.rpc;
public interface Service {
   java.rmi.Remote getPort(QName portName,
                           Class serviceEndpointInterface)
                        throws ServiceException;
   java.rmi.Remote getPort(Class serviceEndpointInterface)
                        throws ServiceException;

   Call createCall() throws ServiceException;
   Call createCall(QName portName) throws ServiceException;
   Call createCall(QName portName, String operationName)
                                     throws ServiceException;
   Call createCall(QName portName, QName operationName)
                                     throws ServiceException;
   Call[] getCalls(QName portName) throws ServiceException;

   java.net.URL getWSDLDocumentLocation();
   QName getServiceName();
   java.util.Iterator getPorts() throws ServiceException;
   // ...
}
```

A JAX-RPC runtime system is required to provide the implementation class for the base `javax.xml.rpc.Service` interface. This implementation class is required to support the creation of both dynamic proxies and `Call` objects.

The JAX-RPC specification does not specify the design of a service implementation class. The base `Service` interface is implemented in a vendor specific manner. For example, a `Service` implementation class may be created with a reference to an in-memory representation of the WSDL service description.

The method `getPort(QName, Class)` returns a dynamic proxy or instance of a generated stub class for the specified service endpoint. A JAX-RPC service client uses the returned dynamic proxy or stub instance to invoke operations on the target service endpoint. The parameter `serviceEndpointInterface` specifies the service endpoint interface that must be supported by the created dynamic proxy or stub instance.

The method `getPort(Class)` returns either an instance of a generated stub implementation class or a dynamic proxy. The parameter `serviceEndpoint-Interface` specifies the service endpoint interface that is supported by the returned stub or proxy. In the implementation of this method, the JAX-RPC runtime system takes the responsibility of selecting a protocol binding (and a port) and configuring the stub accordingly. The returned `Stub` instance should not be reconfigured by the client.

This `getPort` method throws `ServiceException` if there is an error in the creation of a dynamic proxy/stub instance or if there is any missing WSDL related metadata as required by this method implementation.

The method `getPorts` returns a list of qualified names (as `javax.xml.namespace.QName`) of ports grouped by this service.

The multiple variants of the method `createCall` create instances of the `javax.xml.rpc.Call`. Refer to the section 8.2.4, “DII Call Interface” and Javadocs for more details on these methods. The `createCall` method throws `ServiceException` if there is any error in the creation of the `Call` object.

The `getCalls` method returns an array of preconfigured `Call` objects for invoking operations on the specified port. There is one `Call` object per operation that can be invoked on the specified port. A pre-configured `Call` object does not need to be configured using the setter methods on the `Call` interface. Each invocation of the `getCalls` method is required to return a new array of preconfigured `Call` objects. This enables `Service` implementation class to avoid side effects of any setter methods that are invoked on the returned `Call` objects.

The `getCalls` method requires the `Service` implementation class to have access to the WSDL related metadata. This method throws `ServiceException` if this `Service` instance does not have access to the required WSDL metadata or if an illegal `portName` is specified.

The `Service` implementation class should implement `java.io.Serializable` and/or `javax.naming.Referenceable` interfaces to support registration in the JNDI namespace.

4.3.11 Generated Service

A WSDL to Java mapping tool is required to generate a service interface based on the mapping of a `wsdl:service` element in the WSDL document. This generated service interface is used for the creation of instances of the generated stub classes.

A JAX-RPC runtime system is required to provide the implementation class for the generated service interface. The design of a generated service implementation class is specific to a vendor’s implementation. A service implementation class should implement `java.io.Serializable` and/or `javax.naming.Referenceable` interfaces to support registration in the JNDI namespace.

A generated service interface is required to follow the design pattern:

Code Example 5 Design pattern for a generated service interface

```
public interface <ServiceName> extends javax.xml.rpc.Service {
   <serviceEndpointInterface> get<Name_of_wsdl:port>()
                        throws ServiceException;

   // ... Additional getter methods
}
```

The name <ServiceName> of the generated service interface is mapped from the name attribute of the corresponding wsdl:service definition. Refer to the section 4.1, “XML Names” for the mapping of the XML names to the Java identifiers.

For each wsdl:port defined in a wsdl:service definition, the generated service interface contains the following methods:

- Required get<Name_of_wsdl:port> method that takes no parameters and returns an instance of the stub class that implements the <serviceEndpointInterface> interface. The <serviceEndpointInterface> interface is mapped from the wsdl:portType and wsdl:binding definitions for this wsdl:port. Refer to the section 4.3.3, “WSDL Port Type”
- Optional get<Name_of_wsdl:port> methods that include parameters specific to the endpoint (or port) configuration. Each such getter method returns an instance of a generated stub class that implements the <serviceEndpointInterface> interface. An example is a getter method that takes security information (example: user name, password) as parameters. These additional getter methods are specific to a JAX-RPC implementation.

All get<Name_of_wsdl:port> methods are required to throw the ServiceException.

The name of the get<Name_of_wsdl:port> methods is obtained by first mapping the name of the wsdl:port to a Java identifier according to the rules in section 4.1, “XML Names”, then treating it as a JavaBean property name for the purpose of adding to it the “get” prefix.

Refer to the section 9.2, “J2EE based Service Client Programming Model” for more details on the use of the generated service class.

Example

The following code snippet shows an example of the generated service interface. This service interface is generated using the WSDL example in the section 2.1.3:

Code Example 6 Example of a Generated Service Interface: StockQuoteService

```
package com.example;
public interface StockQuoteService extends javax.xml.rpc.Service {
   StockQuoteProvider getStockQuoteProviderPort()
                       throws ServiceException;
   // ...
}
```

4.3.12 Name Collisions

Note that the WSDL 1.1 specification allows various element definitions to have the same name within a specified namespace in the WSDL document. This leads to potential name conflicts in the WSDL to Java mapping. To address this issue, a WSDL to Java mapping tool is required to resolve any potential name collisions.

The following table specifies rules for avoiding name collisions in the WSDL to Java mapping. Suffixes are appended to the mapped Java identifiers to resolve the name conflicts. If there are no name collisions, there is no requirement to use these suffixes.

TABLE 4-5 Name Collision Rules

Java definition mapped from WSDL/ XML	Appended Suffix for avoiding name collisions in the mapped Java identifier	Example
Java class based on the WSDL/XML->Java type mapping whose name is derived from that of a xsd:simpleType or an xsd:complexType	*_Type*	XML: <xsd:complexType name="shared"> Java: Shared_*Type*.java
Java enumeration class (see section 4.2.4)	*_Enumeration*	XML: <xsd:simpleType name=”shared”> <xsd:restriction base=”xsd:string”> <xsd:enumeration value=”foo”/> </xsd:restriction> </xsd:simpleType> Java: Shared_*Enumeration*.java
Java class based on the WSDL/XML->Java type mapping whose names is derived from that of a xsd:element	*_Element*	XML: <xsd:element name=”shared”> Java: Shared_*Element*.java
Service Endpoint interface	*_PortType*	XML: <wsdl:portType name="shared"> <wsdl:binding name="shared" ..> Java: Service Endpoint interface: Shared_*PortType*.java
Generated Service interface	*_Service*	XML: <wsdl:service name="shared" ..> Java: Shared_*Service*.java
Exception Class	*_Exception*	Java: Shared_*Exception*.java

5 Java to XML/WSDL Mapping

This chapter specifies the standard mapping from the Java definitions to XML and WSDL definitions. This mapping specifies the following:

- Definition of a JAX-RPC supported Java type
- Service endpoint interface for a JAX-RPC service
- Mapping from a Java service endpoint definition to WSDL definitions
- Mapping from the Java types to the XML data types

Note that the JAX-RPC specification does not require support for a round trip mapping between the WSDL and Java definitions.

5.1 JAX-RPC Supported Java Types

The following are the JAX-RPC supported Java types:

- One of the Java primitive types as specified in the section 5.1.1
- A subset of the standard Java classes (as specified in the J2SE APIs) as specified in the section 5.1.3
- An array of a supported Java type as specified in the section 5.1.2
- An exception class as specified in the section 5.2.1
- A JAX-RPC value type as specified in the section 5.4

A JAX-RPC runtime system implementation must support transmission of the values of a JAX-RPC supported Java type between a service client and service endpoint at runtime. Values of a JAX-RPC supported Java type must be serializable to and from the corresponding XML representation.

Refer to the section 5.3, “Java to XML Type Mapping” for the XML mapping of the JAX-RPC supported Java types.

5.1.1 Primitive Types

The JAX-RPC specification supports the following Java primitive types and the corresponding wrapper Java classes:

- `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`

5.1.2 Java Array

The JAX-RPC specification supports a Java array with members of a supported JAX-RPC Java type. The JAX-RPC specification requires support for Java array of type `java.lang.Object`. Multidimensional Java arrays are also supported. Examples are `int[]` and `String[][]`.

5.1.3 Standard Java Classes

The following standard Java classes are supported by JAX-RPC:

- `java.lang.String`
- `java.util.Date`
- `java.util.Calendar`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `javax.xml.namespace.QName`
- `java.net.URI`

Other standard Java classes (for example: classes in the Java Collection Framework) are mapped using pluggable serializers and deserializers. Refer to the chapter 15 (“Extensible Type Mapping”) for more details on the pluggable serializers and deserializers.

5.1.4 JAX-RPC Value Type

Refer to the section 5.4, “JAX-RPC Value Type” for more details.

5.2 JAX-RPC Service Endpoint Interface

The JAX-RPC specification requires that a JAX-RPC service endpoint interface must follow the following rules:

- Service endpoint interface must extend `java.rmi.Remote` either directly or indirectly
- All methods in the interface must throw `java.rmi.RemoteException`. Methods may throw service specific exceptions in addition to the `RemoteException`.
- Method parameters and return types must be the JAX-RPC supported Java types (refer to the section 5.1, “JAX-RPC Supported Java Types”). At runtime, values of a supported Java type must be serializable to and from the corresponding XML representation.
- Holder classes may be used as method parameters. These `Holder` classes are either generated or those packaged in the standard `javax.xml.rpc.holders` package.
- Service endpoint interface must not include constant (as `public final static`) declarations. WSDL 1.1 specification does not define any standard representation for constants in a `wsdl:portType` definition.

Example

The following code extract shows an example of a service endpoint interface:

```
// Java
```

```
package com.example;
public interface StockQuoteProvider extends java.rmi.Remote {
   float getLastTradePrice(String tickerSymbol)
             throws java.rmi.RemoteException,
                    com.example.InvalidTickerException;
   public StockQuoteInfo getStockQuote(String tickerSymbol)
             throws java.rmi.RemoteException,
                    com.example.InvalidTickerException;
}
```

5.2.1 Service Specific Exception

A JAX-RPC service endpoint interface that extends the `java.rmi.Remote` interface may declare service specific exceptions in a method signature in addition to the required `java.rmi.RemoteException`.

A service specific exception declared in a remote method signature must be a checked exception. It must extend `java.lang.Exception` either directly or indirectly but must not be a `RuntimeException`.

Example

The following is an example of a service specific Java exception:

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

public class InvalidTickerException extends java.lang.Exception {
   public InvalidTickerException(String tickersymbol) { ... }
   public String getTickerSymbol() { ... }
}
```

5.2.2 Remote Reference Passing

The JAX-RPC specification does not require support for the passing or return of a remote reference across a remote method invocation. The reason is that SOAP [5] specifies object-by-reference as out of scope in both the 1.1 and 1.2 versions. Any support for remote reference passing in the JAX-RPC would be non-standard and hence constrain interoperability.

A service endpoint interface must not include a remote reference as either a parameter or a return type. A Java array or JAX-RPC value type must not include a remote reference as a contained element.

Example

The following is an example of a non-conforming service endpoint interface.

```
// Java
package com.example;
public interface StockBroker extends java.rmi.Remote {
   // ... Remote methods not shown
```

```
}

public interface StockQuoteProvider extends java.rmi.Remote {
   StockBroker getPreferredStockBroker()
                       throws java.rmi.RemoteException;
}
```

In the above example, the return type for the method `getPreferredStockBroker` represents a remote reference to the `StockBroker` service endpoint.

5.2.3 Pass by Copy

The JAX-RPC specification requires support for the pass by copy parameter passing mode for all parameters and return values. This is similar to the parameter passing semantics defined by Java RMI [15].

The value of a parameter object is copied before invoking a remote method on a JAX-RPC service. By default, only non-static and non-transient fields are copied. For return value, a new object is created in the calling virtual machine.

5.3 Java to XML Type Mapping

This section specifies the standard mapping of the Java types to the XML data types.

5.3.1 Java Primitive types

The following table specifies the standard mapping of the Java primitive types to the XML data types:

TABLE 5-1 Mapping of the Java Primitive Types

Java Primitive Type	XML Data Type
`boolean`	`xsd:boolean`
`byte`	`xsd:byte`
`short`	`xsd:short`
`int`	`xsd:int`
`long`	`xsd:long`
`float`	`xsd:float`
`double`	`xsd:double`

In the case of literal element declarations, the Java class for a Java primitive type (example: `java.lang.Integer`) is mapped to an element declaration with the `nillable` attribute set to `true` and with the corresponding built-in XML data type. The following example shows the mapping for the `java.lang.Integer`:

```
<xsd:element name="code" type="xsd:int" nillable="true"/>
<!-- Schema instance -->
<code xsi:nil="true"></code>
```

Note that the SOAP 1.1 specification indicates that all SOAP encoded elements are `nillable`. So in the SOAP encoded case, the Java wrapper class for a Java primitive type is mapped to the corresponding SOAP encoded type. For example, the `java.lang.Integer` maps to `soapenc:int` if the SOAP encoding is being used.

5.3.2 Standard Java Classes

TABLE 5-2 Mapping of Standard Java Classes

Java Class	XML Data Type
`java.lang.String`	`xsd:string`
`java.math.BigInteger`	`xsd:integer`
`java.math.BigDecimal`	`xsd:decimal`
`java.util.Calendar`	`xsd:dateTime`
`java.util.Date`	`xsd:dateTime`
`javax.xml.namespace.QName`	`xsd:QName`
`java.net.URI`	`xsd:anyURI`

5.3.3 Array of Bytes

Both `byte[]` and `Byte[]` are mapped to the `xsd:base64Binary` type.

The mapping of the `java.lang.Byte[]` type to the `xsd:base64Binary` type is now deprecated because it cannot represent `null` values accurately. Instead, `Byte[]` should be mapped to a regular array following the rules in section 5.3.4.

5.3.4 Java Array

The JAX-RPC specification maps a Java array to one of the following XML types:

- An array derived from the `soapenc:Array` using the `wsdl:arrayType` attribute for restriction. This case is specified in the WSDL 1.1 [7] specification
- An array with the `soapenc:arrayType` in the schema instance as specified in the SOAP 1.1 encoding
- An array derived from the `soapenc:Array` by restriction as specified in the SOAP 1.1 specification
- An element in a `xsd:complexType` with the `maxOccurs` attribute set to an integer greater than 1 or `unbounded`. This is one form of mapping used for Java arrays defined in a JAX-RPC value type. Refer to the section 5.4 for more details on the JAX-RPC value types.

Refer to the “Appendix: XML Schema Support” for examples of the above cases.

The JAX-RPC specification requires support for the mapping of the multi-dimensional Java arrays. The member type of a Java array must be a JAX-RPC supported Java type as defined in the section 5.1. The mapped XML array contains elements with XML data type mapped from the corresponding member type of the Java array.

Example

```
// Java
int[] numbers;
```

The above Java array is mapped to the following schema fragment:

```
<!-- Schema fragment -->
<complexType name="numbers">
<complexContent>
    <restriction base="soapenc:Array">
        <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:int[]"/>
    </restriction>
</complexContent>
</complexType>
```

The following example shows another form of XML schema representation and an XML schema instance:

```
<!-- Schema fragment -->
<element name=”numbers” type=”soapenc:Array”/>

<!-- Schema instance -->
<numbers soapenc:arrayType=”xsd:int[3]”>
    <member>1</member>
    <member>2</member>
    <member>3</member>
</numbers>
```

5.4 JAX-RPC Value Type

This section specifies requirements for the JAX-RPC value types.

A JAX-RPC value type is a Java class whose value can be moved between a service client and service endpoint. A Java class must follow these rules to be a JAX-RPC conformant value type:

- Java class must have a public default constructor.
- Java class must not implement (directly or indirectly) the `java.rmi.Remote` interface.
- Java class may implement any Java interface (except the `java.rmi.Remote` interface) or extend another Java class.
- Java class may contain public, private, protected, package-level fields. The Java type of a public field must be a supported JAX-RPC type as specified in the section 5.1, “JAX-RPC Supported Java Types”.
- Java class may contain methods. There are no specified restrictions on the nature of these methods. Refer to the later rule about the JavaBeans properties.
- Java class may contain static or transient fields.
- Java class for a JAX-RPC value type may be designed as a JavaBeans class. In this case, the bean properties (as defined by the JavaBeans introspection) are required to follow the JavaBeans design pattern of setter and getter methods. The Java type of a bean property must be a supported JAX-RPC type as specified in the section 5.1, “JAX-RPC Supported Java Types”.

Example

The following code snippets show valid examples of JAX-RPC value types:

```
// Java
```

```
public class Base {
   public Base() { }
   public int a;
   private int b;
   private int c;

   public int getB() {
      return b;
   }
   public void setB(int b) {
      this.b = b;
   }
   public void someMethod() { ... }
}

public class Derived extends Base {
   public Derived() { }

   public int x;
   private int y;
}
```

5.4.1 XML Mapping

A JAX-RPC value type is mapped to an `xsd:complexType` with either the `xsd:all` or the `xsd:sequence` compositor. Please notice that the XML Schema specification places several restrictions on the contents of a particle that uses the `xsd:all` compositor.

The standard Java to XML mapping of a JAX-RPC value type must follow these rules:

- The `name` attribute of the `xsd:complexType` is mapped from the name of the Java class for the JAX-RPC value type.
- Each public non-final non-transient field in the Java class is mapped to an element in the `xsd:complexType`. The `name` and `type` attributes of the mapped element are mapped from the name and Java type of the public field. The `type` attribute is derived from the Java type of the public field using the type mapping rules in the section 5.3, “Java to XML Type Mapping”.
- Each read/write property (as identified by the `java.beans.Introspector` class) is mapped to an element in the `xsd:complexType`. The `name` attribute for the mapped element is derived from the name of the property. The `type` attribute is mapped from the Java type of the property using the type mapping specified in the section 5.3, “Java to XML Type Mapping”.
- Inheritance of the Java classes is mapped using the derivation of `xsd:complexType` types using the extension mechanism.
- The only fields that are mapped are the non-transient, non-final public fields.
- No methods are mapped.
- There is no standard mapping of indexed properties as reported by the JavaBeans introspection.
- There is no standard mapping for the case when a JavaBean property has the same name as a public field. A Java to XML mapping implementation is required to flag this case as an error.

Note that the pluggable serializers and deserializers may be used to support an advanced custom form of mapping for the JAX-RPC value types. Such custom mapping may add implementation specific additional rules over the above standard XML mapping.

Example

The XML mapping for the `Base` class is as shown below:

```
<complexType name="Base">
   <sequence>
      <element name="a" type="xsd:int"/>
      <element name="b" type="xsd:int"/>
   </sequence>
</complexType>
```

The `Base` XML type includes elements based on the mapping of the public field `a` and the bean property `b` (mapped from the pair of `getB` and `setB` methods). The private field `c` and the method `someMethod` are not mapped.

The `Derived` class is mapped to the following XML schema type:

```
<complexType name="Derived">
   <complexContent>
      <extension base="Base">
         <sequence>
            <element name="x" type="xsd:int"/>
         </sequence>
      </extension>
   </complexContent>
</complexType>
```

The `Derived` XML type extends from the `Base` type. The `Derived` XML type includes element `x` mapped from the public field `x` in the `Derived` Java class. The private field `y` in the `Derived` class is not mapped.

5.4.2 Java Serialization Semantics

The default serialization semantics (termed SOAP serialization semantics) and on-wire representation of a JAX-RPC value type are defined in terms of the XML mapping of a JAX-RPC value type. Based on the standard XML mapping of a JAX-RPC value type (as defined in the section 5.4.1, “XML Mapping”), the default serialized state for a JAX-RPC value type includes only the XML mapping of the public fields and bean properties. For example, the serialized state for the `Base` value type includes only the public field `a` and property `b`. This XML serialized state gets transmitted over the wire between the client and service.

The XML/SOAP serialization semantics and representation for a JAX-RPC value type differ from the standard Java serialization. In the standard Java serialization, the serialized state includes the public, private, protected and package level non-transient fields for a serializable Java class. RMI-IIOP uses the standard Java serialization semantics for its value types.

A JAX-RPC value type is required to conform to the following set of rules if a JAX-RPC value type requires the Java serialization semantics. Note that the following rules are in addition to the rules (in the section 5.4) for a JAX-RPC value type:

- A JAX-RPC value type must implement the `java.io.Serializable` interface
- All public fields in the Java class must be non-transient.
- Each non-public, non-transient field must exactly represent a JavaBeans property, and vice versa.
- Each non-public field that does not exactly represent a JavaBeans property must be marked transient.

A JAX-RPC value type that conforms to the above rules is usable across both JAX-RPC (with different protocol bindings) and RMI-IIOP (that uses the standard Java serialization semantics for its value types) with the same Java serialization semantics.

A JAX-RPC value type that implements the `Serializable` interface but does not conform to all above rules, may not have the Java serialization semantics.

5.5 Java to WSDL Mapping

This section specifies the standard mapping of a JAX-RPC service endpoint definition to a WSDL service description.

Note that the Java to WSDL mapping specifies mapping of the Java definitions to the abstract WSDL definitions (namely: `wsdl:types`, `wsdl:message`, `wsdl:operation` and `wsdl:portType`). The following mapping specification does not specify how the SOAP binding (or binding for any other protocol) is defined in a WSDL document.

5.5.1 Java Identifier

A Java identifier is mapped to an equivalent XML name. Java identfiers are legal XML names. Note that the SOAP 1.2 specification [5] specifies rules for mapping application defined names to XML names.

5.5.2 Java Package

A Java package is mapped to a WSDL document. The mapped WSDL document contains abstract definitions (port type, operations, messages) based on the mapping of Java declarations in the corresponding Java package.

The JAX-RPC specification requires the namespace definitions in the WSDL document to be mapped in an application specific manner. A Java to WSDL mapping tool is required to support the configuration of the namespace definitions in the mapped WSDL document.

Note that the Java to WSDL mapping specification does not require any specific authoring style for the mapped WSDL document. For example, the mapped WSDL definitions may be separated into multiple WSDL documents that are imported across documents.

5.5.3 Service Endpoint Interface

A service endpoint interface (that extends `java.rmi.Remote`) is mapped to the `wsdl:portType` element. The `name` attribute of the `wsdl:portType` has the same name as the service endpoint interface.

Methods defined in a service endpoint interface are mapped to the `wsdl:operation` definitions in the corresponding `wsdl:portType`. Refer to the section 5.5.5, “Methods” for more details on the mapping of methods in a service endpoint interface.

Example

The following shows an example of a service endpoint interface:

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

The above service endpoint interface is mapped to the following `wsdl:portType` definition. Note that the `parameterOrder` attribute is optional based on the WSDL 1.1 specification:

```
<!-- WSDL extract -->
<portType name="StockQuoteProvider">
    <operation name="getLastTradePrice"
              parameterOrder="tickerSymbol">
       <input message="tns:getLastTradePrice"/>
       <output message="tns:getLastTradePriceResponse"/>
    </operation>
</portType>
```

5.5.4 Inherited Service Endpoint interfaces

Each Java interface in the service endpoint inheritance hierarchy is mapped to an equivalent `wsdl:portType` definition. Since the WSDL 1.1 specification does not define a standard representation for the inheritance of the `wsdl:portType` elements, each mapped `wsdl:portType` definition includes mapping of the complete set of inherited Java methods. The support for this mapping is optional.

The following example illustrates this mapping.

Example

In the following example, the service endpoint interfaces `StockQuoteProvider` and `PremiumStockQuoteProvider` are mapped to respective `wsdl:portType` definitions. The port type `StockQuoteProvider` includes a single operation `getLastTradePrice`; while the port type `PremiumStockQuoteProvider` includes two operations `getLastTradePrice` and `getRealtimeLastTradePrice`. The `getLastTradePrice` operation in the `PremiumStockQuoteProvider` port type is mapped from the corresponding inherited method.

```
// Java Code
package com.example;
public interface StockQuoteProvider extends java.rmi.Remote {
    // gets a 20 minute delayed stock price
    float getLastTradePrice(String tickerSymbol)
           throws java.rmi.RemoteException;
}

package com.example;
public interface PremiumStockQuoteProvider extends
                com.example.StockQuoteProvider {
    // gets a realtime stock quote
    float getRealtimeLastTradePrice(String tickerSymbol)
           throws java.rmi.RemoteException;
}
```

The mapping of the StockQuoteProvider interface is the same as shown in the example in the section 5.5.3, “Service Endpoint Interface”.

The PremiumStockQuoteProvider interface is mapped to the following wsdl:portType definition:

```
<!-- WSDL Extract -->
<portType name=”PremiumStockQuoteProvider”>
   <operation name=”getLastTradePrice”
             parameterOrder=”tickerSymbol”>
      <input message=”tns:getLastTradePrice”/>
      <output message=”tns:getLastTradePriceResponse”/>
   </operation>
   <operation name=”getRealtimeLastTradePrice”
             parameterOrder=”tickerSymbol”>
      <input message=”tns:getRealtimeLastTradePrice”/>
      <output message=”tns:getRealtimeLastTradePriceResponse”/>
   </operation>
</portType>
```

5.5.5 Methods

Each method on a Java service endpoint interface is mapped to an equivalent wsdl:operation definition. The mapped wsdl:operation definition preserves the method signature in terms of the parameter ordering.

The standard mapping of a Java method to a wsdl:operation definitions follows these rules:

- The name attribute of the mapped wsdl:operation is the same as the name of the Java method.
- Overloaded Java methods are mapped to multiple wsdl:operation elements. These mapped wsdl:operation elements are named the same (as the name of the overloaded Java methods) or with unique names depending on the mechanism that is used by the server-side JAX-RPC runtime system to dispatch methods to the target service endpoint. For example, a JAX-RPC runtime system may use unique operation names, SOAPAction or an implementation specific mechanism to dispatch methods to the target service endpoint. Note that the JAX-RPC specification does not require use of the SOAPAction.
- A mapped wsdl:operation contains wsdl:input and, unless it is one-way, wsdl:output and optional wsdl:fault(s) based on the mapping of the Java method signature. The message attribute of these wsdl:input and wsdl:output elements uses the qualified name of the corresponding wsdl:message definition. Only Java methods with a void return type and that do not throw any exception other than java.rmi.RemoteException can be mapped to one-way operations. Since not all Java methods that fulfill this requirement are amenable to become one-way operations, Java to WSDL mapping tools are required to provide a facility for specifying metadata used to identify which of the methods should indeed be mapped to one-way operations.
- Each Java parameter (in the Java method signature) is mapped to a message part in the wsdl:message. This wsdl:message corresponds to the wsdl:input element for the mapped wsdl:operation. Each message part has a name attribute that is mapped based on the name of the Java parameter and a type attribute (an XML data type) mapped from the Java type of the parameter. Message parts appear in the same order as in the Java method signature.

- Refer to the section 6.2, “Operation Style attribute” for the requirements related to the mapping of Java methods to the `soap:operation` definitions with `document` and `rpc` styles in the SOAP binding.
- Refer to the section 6.4, “Literal Representation” for details related to the literal representation of SOAP messages for RPC calls and responses. In brief, if literal representation is used for the SOAP message, each message part in the SOAP body references a concrete schema definition using either the `element` or `type` attribute. If the `element` attribute is used, the element referenced by the part appears under the Body element. If the `type` attribute is used, then the referenced type becomes the schema type of the element.
- A Java return type (for a Java method) is mapped to a message part in the `wsdl:message`. This `wsdl:message` corresponds to the `wsdl:output` element for the mapped `wsdl:operation`. The `name` of this message part is not significant.
- For `Holder` classes used as method parameters, a Java to WSDL mapping tool is required to provide a facility for specifying the related mapping metadata. This metadata identifies whether a `Holder` parameter in the method signature is mapped to either the `OUT` or `INOUT` parameter mode. A mapping tool may choose to map all `Holder` parameters to the default `INOUT` parameter mode. Future versions of the JAX-RPC specification would consider specifying a standard approach for specifying such mapping metadata.
- Ordering of Java parameters is represented in the `parameterOrder` attribute in the `wsdl:operation` element. The message part for a return value is not listed in the `parameterOrder` attribute. Refer to the WSDL 1.1 specification for more details on the `parameterOrder` attribute.

Exceptions

- Each service specific Java exception in a remote method signature is mapped to a `wsdl:fault` element. This `wsdl:fault` element references a `wsdl:message` element.
- The `name` attribute of the `wsdl:message` element is the same as the name of the Java exception. The `message` attribute of the `wsdl:fault` element is the qualified name of the `wsdl:message` definition.
- The single message part in the `wsdl:fault` element (as required by the WSDL 1.1 specification) can refer to either a type or an element. If it refers to a type, it is either a `xsd:simpleType` or a `xsd:complexType`. If the latter, the `name` attribute of the `xsd:complexType` is the same as the name of the Java exception class. Similarly, if it refers to an element, the `name` attribute of the `xsd:element` is the same as the name of the Java exception class and its type must follow the rules for the mapping to complex types given in this same section.
- Multiple fields (each with a getter method and corresponding parameter in the constructor) in the Java exception class are mapped to elements of the `xsd:complexType`. The `name` and `type` attribute of each element are mapped from the name and Java type of the corresponding field in the Java exception class. The mapping of these fields follows the mapping specified in the section 5.3, “Java to XML Type Mapping”.
- If a Java exception class has a single field (with a getter method and corresponding parameter in the constructor), this field is mapped (as an option to the previous bullet) to a single message part in the `wsdl:message`. The `name` and `type` of this message `part` are mapped from the name of Java type of the corresponding field.
- The `wsdl:message` part represents the contents of the `detail` element in the SOAP fault.

- Inheritance of Java exceptions is mapped using the derivation of `xsd:complexType` using the extension mechanism. This applies to the `xsd:complexType` used to represent the single message `part` in the `wsdl:message` for the `wsdl:fault`.
- A remote exception (`java.rmi.RemoteException` or its subclasses) is mapped to a standard SOAP fault. Refer to the section 14.3.6, “Mapping of Remote Exceptions” for the standard mapping of remote exceptions.

Example

The following shows an example of a service endpoint interface `StockQuoteProvider` with a single method `getLastTradePrice`:

```
// Java
package com.example;
public interface StockQuoteProvider extends java.rmi.Remote {
    float getLastTradePrice(String tickerSymbol)
          throws java.rmi.RemoteException,
                com.example.InvalidTickerException;
}
public class InvalidTickerException
             extends java.lang.Exception {
    private String tickerSymbol;

    public InvalidTickerException(String tickerSymbol) { ... }
    public String getTickerSymbol() { ... }
}
```

The above Java service endpoint interface is mapped to the following WSDL definitions (using the `type` attribute of `wsdl:part`):

```
<!-- WSDL Extract -->
<message name=”getLastTradePrice”>
   <part name=”tickerSymbol” type=”xsd:string”/>
</message>
<message name="getLastTradePriceResponse">
   <part name="result" type="xsd:float"/>
</message>
<message name="InvalidTickerException">
   <part name="tickerSymbol" type="xsd:string"/>
</message>
<portType name=”StockQuoteProvider”>
   <operation name=”getLastTradePrice”
             parameterOrder=”tickerSymbol”>
      <input message=”tns:getLastTradePrice”/>
      <output message=”tns:getLastTradePriceResponse”/>
      <fault name="InvalidTickerException"
             message="tns:InvalidTickerException"/>
   </operation>
</portType>
```

It may also be mapped to the following WSDL definitions (using the element attribute of `wsdl:part`):

```
<!-- WSDL Extract -->
<types>
   <schema xmlns=”...”
          targetNamespace=”...”>
      <element name=”InvalidTickerException”>
          <complexType>
             <element name=”tickerSymbol” type=”xsd:string”/>
          </complexType>
      </element>
   </schema>
```

```
</types>

<message name=”getLastTradePrice”>
   <part name=”tickerSymbol” type=”xsd:string”/>
</message>
<message name="getLastTradePriceResponse">
   <part name="result" type="xsd:float"/>
</message>
<message name="InvalidTickerException">
   <part name="exception" element="ttns:InvalidTickerException"/>
</message>
<portType name=”StockQuoteProvider”>
   <operation name=”getLastTradePrice”
             parameterOrder=”tickerSymbol”>
      <input message=”tns:getLastTradePrice”/>
      <output message=”tns:getLastTradePriceResponse”/>
      <fault name="InvalidTickerException"
             message="tns:InvalidTickerException"/>
   </operation>
</portType>
```

6 SOAP Binding

This chapter specifies the JAX-RPC support for the SOAP 1.1 binding.

Note that this section uses extracts from the WSDL 1.1 specification. In all cases, WSDL specification should override any WSDL extracts that have been used in this chapter.

6.1 SOAP Binding in WSDL

The `soap:binding` element in the WSDL identifies that the SOAP protocol is used for binding the abstract WSDL definitions.

The JAX-RPC specification requires support for the following cases (termed `operation modes`) for an operation with the SOAP binding. Later sections of this chapter specify more details on these supported operation modes:

- Operation with the `rpc` style and `encoded` use (`rpc/encoded`)
- Operation with the `rpc` style and `literal` use (`rpc/literal`)
- Operation with the `document` style and `literal` use (`document/literal`)

Refer to the WSDL 1.1 [7] specification for more details on the `document` and `rpc` operation styles.

A JAX-RPC implementation is required to support the above operation modes for the mapping of a WSDL based service description to the corresponding Java representation.

A JAX-RPC implementation may choose to support the optional operation mode of `document/encoded`.

6.2 Operation Style attribute

The `style` attribute (specified per `soap:operation` element or as a default in the `soap:binding` element) indicates whether an operation is `rpc` or `document` oriented. In the JAX-RPC programming model, both `rpc` and `document` style operations are mapped to the corresponding remote methods on a service endpoint interface.

A JAX-RPC client side implementation is required to support use of services that follow either the WSDL 1.1 or JAX-RPC (refer to the beginning of this section) specified `document` and `rpc` operation style requirements. Note that the WSDL 1.1 specification does not require a wrapper element for the `document` style operations and assumes the use of `SOAPAction`.

The JAX-RPC specification requires that the above requirements based on the operation style should be hidden from the JAX-RPC programming model. A JAX-RPC implementation should take the responsibility for the appropriate representation of a SOAP message based on the operation style.

6.3 Encoded Representation

The JAX-RPC specification requires support for `rpc` style operations with the `encoded` use. Support for `document` style operations with `encoded` use is optional. In the `encoded` use, each message part in the SOAP body references an abstract type using the `type` attribute. These types are serialized according to the encodings identified in the `encodingStyle` attribute of the `soap:body` element. An example of such encoding is the SOAP 1.1 encoding [4].

Refer to the chapter 14 for the related interoperability requirements.

6.4 Literal Representation

The JAX-RPC specification requires support for `rpc` and `document` style operations with the `literal` use. Note that the `literal` use is defined on the `soap:body` element in the WSDL.

If the use is `literal`, each message part in the SOAP Body element references a concrete schema definition using either the `element` or `type` attribute. If the `element` attribute is used, the element referenced by the message part appears under the Body element (with or without wrapper depending on the operation style). If the `type` attribute is used, the type referenced by the part becomes the schema type of the enclosing element (Body element for the `document` style or part accessor element for the `rpc` style). WSDL 1.1 specification allows (but does not require) use of an optional `encodingStyle` attribute for encoding in the literal case.

The JAX-RPC specification requires support for the `element` attribute in a message part with literal representation and referenced by a `document` style operation. It also requires support for the `type` attribute in a message part with literal representation and referenced by an `rpc` style operation. However, use of any specific encoding style with the literal mode is not required. A JAX-RPC implementation may use the optional `encodingStyle` attribute (in a `soap:body` element with the `literal` use) to apply specific encoding rules.

6.4.1 Java Mapping of Literal Representation

The Java mapping for a message part (either a parameter or return value) with literal representation depends on whether the JAX-RPC specifies a standard Java mapping for the XML type of this message part. The “Appendix: XML Schema Support” specifies a subset of the XML schema data types that a JAX-RPC implementation is required to support. Refer to the section 4.2, “XML to Java Type Mapping” for the specified mapping of this subset of XML schema.

When mapping `document` style operations, in addition to the regular mapping, JAX-RPC implementations are required to support the so-called "wrapper" style, in which the logical parameters of an operation are wrapped inside a `xsd:sequence` element named after the operation.

In order to qualify as using the "wrapper" style, an operation must fulfill the following conditions:

- its input and output messages (if present) must contain exactly one part;
- such a part must refer to an element named after the operation;
- such an element (a *wrapper*) must be of a complex type defined using the `xsd:sequence` compositor and containing only elements declarations.

In this case, implementations must be able to discard the wrapper elements and treat their children as the actual parameters of the operation. Please refer to section 6.4.3 below for an example.

If there is no standard Java mapping for an XML schema type, a message part with literal representation is considered and mapped as a document fragment. The XML to Java mapping uses the interface `javax.xml.soap.SOAPElement` to represent a literal message part in the Java mapping of a `wsdl:operation` element. For example, a parameter or a return type (for a remote operation) represented as a literal message part is mapped to a `SOAPElement` in the Java method signature.

When a `SOAPElement` is declared and passed as parameter to the remote method call, the JAX-RPC runtime system implementation is required to embed the XML representation of the `SOAPElement` instance as part of the underlying SOAP message representation. This embedding of `SOAPElement` instance is internal to the implementation of a JAX-RPC runtime system.

In order to make it possible to use other data binding frameworks (e.g. JAXB) in conjunction with JAX-RPC, it must be possible to selectively turn the standard JAX-RPC type mapping off on a per-part basis and revert to the default mapping, namely `SOAPElement`. JAX-RPC tools are required to provide a facility to specify metadata for this purpose.

Refer to the section 6.4.2, "SOAPElement" for more details on the `SOAPElement` API.

Any other form of mapping between the Java and XML data types for an operation with the literal use is specific to a JAX-RPC implementation. A portable application should not rely on any non-standard JAX-RPC implementation specific mapping. For example, a JAX-RPC implementation may use a Java framework for XML data type binding to map the literal parts. In this case, each literal part or its elements is represented by an instance of a Java class generated by the data binding framework. In the JAX-RPC specification, this form of mapping is specific to a JAX-RPC implementation. Note that the JAXB 1.0 specification is defining standard Java APIs for the XML data binding. Future versions of the JAX-RPC specification will consider use of the JAXB 1.0 APIs.

6.4.2 SOAPElement

The following code snippet shows the `javax.xml.soap.SOAPFactory`. A `SOAPFactory` is used to create a `SOAPElement`, `Detail` and `Name` objects. Refer to the JAXM 1.1 and SAAJ [13] specification for complete details on the `SOAPFactory`.

```
package javax.xml.soap;
public abstract class SOAPFactory {
   public abstract SOAPElement createElement(Name name)
               throws SOAPException;
```

```
    public abstract SOAPElement createElement(String localName)
                  throws SOAPException;
    public abstract SOAPElement createElement(String localName,
                        String prefix,String uri)
                  throws SOAPException;
    public abstract Detail createDetail() throws SOAPException;

    public abstract Name createName(String localName,
                  String prefix, String uri)throws SOAPException;
    public abstract Name createName(String localName)
                           throws SOAPException;

    public static SOAPFactory newInstance()
                  throws SOAPException { ... }
}
```

The `SOAPFactory` in the `javax.xml.soap` package follows the abstract factory pattern. An instance of the `SOAPFactory` is created using the static `newInstance` method.

The variants of the `createElement` method create a `SOAPElement`. The created `SOAPElement` instance represents a document fragment and can be used as parameter to a remote method call. The JAX-RPC specification requires the use of the `SOAPFactory` to create `SOAPElement` instances that are passed in the remote method invocation.

The method `createName` creates an instance of the `javax.xml.soap.Name`. The method `createDetail` creates an instance of the `javax.xml.soap.Detail`.

The following code snippet shows the `javax.xml.soap.SOAPElement` interface. This interface is used to represent a literal message part when mapped as a document fragment.

Refer to the JAXM and SAAJ specifications [13] for complete details on the `SOAPElement`:

```
package javax.xml.soap;
public interface SOAPElement extends Node {
    SOAPElement addChildElement(Name name)
                  throws SOAPException;
    SOAPElement addChildElement(String localName)
                  throws SOAPException;
    SOAPElement addChildElement(String localName,
                                    String prefix)
                  throws SOAPException;
    SOAPElement addChildElement(String localName,
                                    String prefix,String uri)
                  throws SOAPException;
    SOAPElement addChildElement(SOAPElement element)
                  throws SOAPException;

    SOAPElement addTextNode(String text) throws SOAPException;
    SOAPElement addAttribute(Name name, String value)
                  throws SOAPException;
    SOAPElement addNamespaceDeclaration(String prefix,
                                            String uri)
                  throws SOAPException;

    String getAttributeValue(Name name);
    Iterator getAllAttributes();
    String getNamespaceURI(String prefix);
    Iterator getNamespacePrefixes();
    Name getElementName();
    boolean removeAttribute(Name name);
    boolean removeNamespaceDeclaration(String prefix);
```

```
    Iterator getChildElements();
    Iterator getChildElements(Name name);
    void setEncodingStyle(String encodingStyle)
                throws SOAPException;
    String getEncodingStyle();
}
```

A `SOAPElement` instance represents the contents in the SOAP Body element for either an RPC request or response. Once a new `SOAPElement` is created using the `SOAPFactory`, the various add methods are used to populate this document fragment.

6.4.3 Example

The following shows schema fragment for this example. Let the namespace prefix be "s" for the following schema definitions:

```
<types>
<schema attributeFormDefault=”qualified”
       elementFormDefault=”qualified”
       targetNamespace=”...”>
<element name="DoExample">
    <complexType>
        <sequence>
            <element name="value1" type="xsd:string"/>
            <element name="value2" type="xsd:string"/>
        </sequence>
    </complexType>
</element>

<element name="DoAnotherExample">
    <complexType>
        <sequence>
            <element name="field1" type="xsd:string"/>
            <element name="field2" type="xsd:string"/>
            ... Other elements with types that do not have standard mapping
            ... defined by JAX-RPC
        </sequence>
        <attribute name="generic" type="xsd:string" use="required"/>
        <anyAttribute namespace="##other”/>
    </complexType>
</element>

<element name="DoExampleResponse">
    <complexType>
        <sequence>
            <element name=”result” type=”xsd:string”/>
        </sequence>
    </complexType>
</element>
</schema>
</types>
```

The following WSDL extract shows the definitions for the messages, port type and binding:

```
<!-- WSDL extract -->
<message name="DoExample">
    <part name="body" element="s:DoExample"/>
</message>
<message name="DoAnotherExample">
    <part name="body" element="s:DoAnotherExample"/>
</message>
```

```
<message name="DoExampleResponse">
   <part name="result" element="s:DoExampleResponse"/>
</message>

<portType name="ExampleSoap">
   <operation name="DoExample">
      <input message="tns:DoExample"/>
      <output message="tns:DoExampleResponse"/>
   </operation>
   <operation name="DoAnotherExample">
      <input message="tns:DoAnotherExample"/>
      <output message="tns:DoExampleResponse"/>
   </operation>
</portType>
...
<binding name="ExampleSoap" type="tns:ExampleSoap">
<soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="DoExample">
<soap:operation soapAction="" style="document"/>
   <input message="tns:DoExample">
       <soap:body use="literal" parts="body" namespace="..."/>
   </input>
   <output message="tns:DoExampleResponse">
      <soap:body use="literal" parts="result" namespace="..."/>
   </output>
</operation>
<operation name="DoAnotherExample">
   ... same for the operation DoAnotherExample
</operation>
</binding>
```

The above example uses `document` style and `literal` use.

Based on the specification in the sections 6.2 and 6.4, the above WSDL definitions are mapped to the following service endpoint interface.

```
// Java
public interface ExampleSoap extends java.rmi.Remote {
   String doExample(String field1, String field2)
                throws java.rmi.RemoteException;
   DoExampleResponse doAnotherExample(SOAPElement doAnotherExample)
                throws java.rmi.RemoteException;
}
```

This example assumes that the return type is mapped to a `String` for both methods. Refer to the section 4.3.4, “WSDL Operation” for more details on how WSDL operations are mapped.

In the above example mapping, the `part` (named `body`) in the `message` named `DoExample` is mapped as a wrapper. The elements `field1` and `field2` are mapped as separate parameters with the corresponding Java types based on the standard XML to Java type mapping.

Another possible mapping of the `doExample` method is as follows. In this example, the `part` named `body` in the `DoExample` `message` is mapped as an `xsd:complexType`:

```
   DoExampleResponse doExample(DoExample body)
                        throws java.rmi.RemoteException;
```

The JAX-RPC specification does not specify a standard Java mapping for a `xsd:complexType` with the `xsd:anyAttribute`. So the message part in the `DoAnotherExample` message is mapped as a `SOAPElement`.

The following code snippet shows the client code for invoking the `doAnotherExample` method:

```
ExampleSoap ib = // ... get the Stub instance
SOAPFactory sef = SOAPFactory.newInstance();
SOAPElement request = sef.createElement(...);

// populate the SOAPElement with attributes and child elements
// ...
String response = ib.doAnotherExample(request);
```

6.5 SOAP Fault

This section specifies the Java mapping of the SOAP faults.

The `soap:fault` element in the WSDL specifies the contents of the `detail` element of a SOAP fault. The `name` attribute relates the `soap:fault` element to the `wsdl:fault` element. The `wsdl:fault` element (an optional element in a `wsdl:operation`) specifies the abstract message format for any error messages that may be output as a result of a remote operation.

The `soap:fault` element is patterned after the `soap:body` element in terms of the literal or encoded use. According to the WSDL 1.1 specification, the `soap:fault` element must contain only a single message part.

A SOAP fault is mapped to either a `javax.xml.rpc.soap.SOAPFaultException`, a service specific exception class or `RemoteException`.

SOAPFaultException

The `SOAPFaultException` exception represents a SOAP fault. This exception is thrown from the Java method mapped from the corresponding `wsdl:operation`. Refer to the section 4.3.4, “WSDL Operation” for the mapping of WSDL operations.

The message part in the SOAP fault maps to the contents of `detail` element accessible through the `getDetail` method on the `SOAPFaultException`. The method `createDetail` on the `javax.xml.soap.SOAPFactory` creates an instance of the `javax.xml.soap.Detail`.

The `faultstring` provides a human-readable description of the SOAP fault. The `faultcode` element provides an algorithmic mapping of the SOAP fault.

The following code snippet shows the `SOAPFaultException`:

```
package javax.xml.rpc.soap;
public class SOAPFaultException extends java.lang.RuntimeException {
   public SOAPFaultException(QName faultcode,
                  String faultstring,
                  String faultactor,
                  javax.xml.soap.Detail detail) { ... }

   public QName getFaultCode() { ... }
   public String getFaultString() { ... }
   public String getFaultActor() { ... }
   public javax.xml.soap.Detail getDetail() { ... }
}
```

6.6 SOAP Headerfault

The `soap:headerfault` element is used to specify SOAP header faults, that is, faults pertaining to the processing of header elements in the message. According to the SOAP specification, such faults must be carried within the Header element of the SOAP envelope instead of the Body.

Unlike `soap:fault`, which refers to a `wsdl:fault`, headerfaults refer directly to a message part. The JAX-RPC specification only requires support for header faults with literal use and referring (through their message part) to an `xsd:element`. When the header fault is transmitted, this element will appear within the `SOAP-ENV:Header` element (and *not* inside the detail element of a `SOAP-ENV:Fault`), while the `SOAP-ENV:Body` will contain a `SOAP-ENV:Fault` element whose `detail` is either empty or unspecified.

Aside from their different serialization, SOAP headerfaults may be mapped to Java exceptions in the same way that regular faults are, with the caveat that the mapping applies only when the corresponding `soap:header` has been mapped to a method argument according to the explicit service context rules (see section 11.2.2).

7 SOAP Message With Attachments

The JAX-RPC specification supports the use of MIME encoded content in a remote method call to a target service endpoint. This chapter specifies details on the following related aspects:

- SOAP Message with Attachments
- WSDL 1.1 MIME Binding
- Mapping between MIME types and Java types

7.1 SOAP Message with Attachments

A JAX-RPC runtime system implementation must use the SOAP message with attachments [6] protocol to support MIME encoded parameters or return value in a remote method call.

A SOAP message package with attachments is constructed using the MIME `multipart/related` type. The primary SOAP 1.1 message is carried in the root body part of the `multipart/related` structure. The primary SOAP 1.1 message may reference additional entities (termed as attachment or MIME parts) in the message package.

Both the header entries and body of the primary SOAP 1.1 message refer to the MIME parts in the message package using the `href` attributes. A referenced MIME part contains either a `Content-ID` or `Content-Location` MIME header. The SOAP messages with Attachment specification [6] specifies more details on the resolution of URI references.

7.2 Java Types

A remote method in a Java service endpoint interface may use the following Java types to represent MIME encoded content:

- Java classes based on the standard Java mapping of MIME types. Refer to the section section 7.5, “Mapping between MIME types and Java types” for more details.
- Java class `javax.activation.DataHandler` for content with any MIME type

The JAX-RPC specification uses the JavaBeans Activation Framework [18] to support various MIME content types. The `DataHandler` class provides a consistent interface to the data represented in various MIME types. A `DataHandler` can be instantiated with data and a specific MIME type using the constructor `DataHandler(Object obj, String mime_type)`.

The data represented by a `DataHandler` may be retrieved as a Java object or an `InputStream`. The method `DataHandler.getContentType` returns the MIME type of the encapsulated data. The `DataHandler.getContent` method returns a Java object based on the MIME type of the encapsulated data.

A `DataHandler` class uses the `DataContentHandler` interface to convert between a stream and specific Java object based on the MIME type.

7.3 MIME Types

The JAX-RPC specification does not define any restrictions on the MIME types. For example, the content in an attachment (for a SOAP message with attachments) may be text, binary image or a complex XML document.

The JAX-RPC runtime system determines the MIME type of a MIME part carried in the SOAP message package by using the following:

- The MIME type defined in the `mime:content` element for this MIME part in the WSDL MIME binding
- The `Content-Type` of the MIME part in the SOAP message package

The MIME type of an attachment part in the SOAP message is required to conform to the MIME type in the MIME binding for an operation.

7.4 WSDL Requirements

A JAX-RPC implementation is required to support mapping between Java and WSDL/ XML for remote method calls with the MIME encoded content.

A JAX-RPC implementation may use the MIME binding (as specified in the WSDL 1.1 specification [7]) to support remote method calls with the MIME encoded content. In the mapped WSDL, MIME elements appear under `wsdl:input` and `wsdl:output` elements in the `wsdl:binding`. The `mime:multipartRelated` element aggregates a set of MIME parts (or attachment parts) into one message of MIME type "`multipart/related`". The `mime:part` element describes each attachment part of such `multipart/related` message.

The `mime:content` elements appear within each `mime:part` to specify the concrete MIME type for this part. If more than one `mime:content` element appears inside a `mime:part`, these are considered alternatives.

The `type` attribute of the `mime:content` element specifies the MIME type for an attachment part. The `part` attribute is used to specify the name of the message part. The `type` attribute may use `*` wild cards to specify a set of acceptable MIME types. An example is `image/*` for all image types.

The `parts` attribute in the `wsdl:input` and `wsdl:output` elements (defined per `operation` in the `wsdl:binding`) lists parts that appear in the SOAP body portion of the message. The unlisted parts of a message appear in the attachment parts of the SOAP message in the case of the `multipart/related` MIME binding.

7.5 Mapping between MIME types and Java types

The following table specifies the standard Java mapping for a subset of the MIME types.

TABLE 7-1 Mapping of MIME Types

MIME Type	Java Type
`image/gif`	`java.awt.Image`
`image/jpeg`	`java.awt.Image`
`text/plain`	`java.lang.String`
`multipart/*`	`javax.mail.internet.MimeMultipart`
`text/xml or application/xml`	`javax.xml.transform.Source`

The Java to WSDL/XML and WSDL/XML to Java mapping for the MIME types is required to conform to the above mapping. This mapping is reflected in the mapped Java method signatures and WSDL description. A WSDL/XML to Java mapping tool is required to provide an option to map the above set of MIME types to the `javax.activation.DataHandler` class. The `DataHandler` class provides methods to get access to the stream representation of the data for a MIME type.

A Java to WSDL mapping tool is required to provide a facility for specifying metadata related to the above mapping between Java and MIME types. This metadata identifies whether a Java type is mapped to a MIME type (using the WSDL MIME binding) or is mapped to an XML schema type (based on the section 4.2, “XML to Java Type Mapping”). For example, a `java.lang.String` can be mapped to either an `xsd:string` or MIME type `text/plain`. The mapping metadata identifies a specific mapping.

If a MIME type is mapped to the `javax.activation.DataHandler`, the `getContent` method of the `DataHandler` class must return instance of the corresponding Java type for a specific MIME content type.

A JAX-RPC implementation is required to support the above MIME types (as specified in the TABLE 7-1) and provide implementation of the required `javax.activation.DataContentHandler` classes.

Due to limitations in the platform, a JAX-RPC implementation is required only to support decoding images of type `image/gif`, but not encoding of the same.

A JAX-RPC implementation is not required to support MIME types beyond that specified in the above table. These additional MIME types may be mapped and supported using the `javax.activation.DataHandler` class and Java Activation Framework.

8 JAX-RPC Core APIs

This chapter specifies JAX-RPC APIs that support the JAX-RPC runtime mechanisms. These APIs are packaged in the `javax.xml.rpc` package. The chapter 12 specifies the SOAP message handler APIs.

8.1 Server side APIs

It is important to note that the JAX-RPC specification does not define and require any standard APIs for the EJB container based implementation of a JAX-RPC server-side runtime system. Refer to the chapter 10 for API specified for the servlet endpoint model.

The internals of a J2EE container based JAX-RPC runtime system are implementation specific.

8.2 Client side APIs

The JAX-RPC specifies the following client side APIs:

- The `javax.xml.rpc.Stub` interface
- The `javax.xml.rpc.Call` interface for the dynamic invocation of a JAX-RPC service
- The `javax.xml.rpc.Service` interface
- The `javax.xml.rpc.ServiceFactory` class
- The `javax.xml.rpc.JAXRPCException` class

A JAX-RPC runtime system implementation is required to implement the above APIs.

The chapter 9 describes the programming model for the use of these APIs by a service client.

8.2.1 Generated Stub Class

A WSDL to Java mapping tool generates a stub class during the import of a service described in a WSDL document. The JAX-RPC specification does not require that a stub class be generated only through the mapping of a WSDL document. For example, a stub class may be generated using a service endpoint interface and additional information about the protocol binding.

All generated stub classes are required to implement the `javax.xml.rpc.Stub` interface. An instance of a stub class represents a client side proxy or stub instance for the target service endpoint.

The following code snippet shows the `javax.xml.rpc.Stub` interface:

```
package javax.xml.rpc;
public interface Stub {
   // ... Methods specified later
}
```

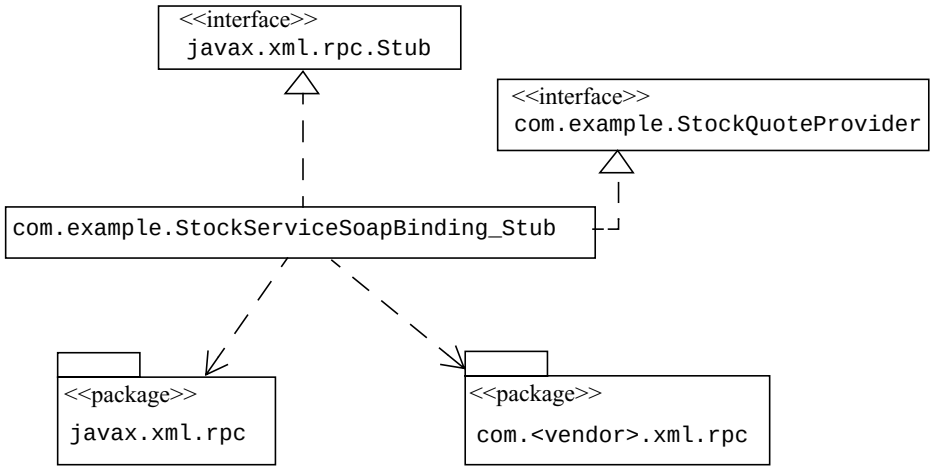
A generated stub class is required to implement a service endpoint interface. The name of a generated stub class is either `<BindingName>_Stub` or is implementation specific. In the former case, the name of the generated stub class is derived from the `name` attribute of the corresponding `wsdl:binding` element in the imported WSDL service description. The `wsdl:binding` element binds an abstract `wsdl:portType` to a specific protocol and transport.

Note that the JAX-RPC specification does not require that a generated stub class be binding and transport neutral. A stub class should be bound to a specific protocol and transport.

Example

The following diagram shows an illustrative example of the stub class hierarchy. The diagram shows the generated `StockServiceSoapBinding_Stub` class that implements the `javax.xml.rpc.Stub` and `com.example.StockQuoteProvider` interfaces where the `StockQuoteProvider` interface is the service endpoint interface.

The diagram also shows the generated stub class using the standard `javax.xml.rpc` package and a vendor specific JAX-RPC implementation package. For example, a generated stub class may be based on either the standard APIs that are specified in the `javax.xml.rpc` package or vendor specific JAX-RPC implementation APIs.



The following code snippet shows how a J2SE based service client uses a stub instance. This example uses a generated stub class:

```
com.example.StockQuoteProvider_Stub sqp = // get the Stub instance
float quote = sqp.getLastTradePrice("ACME");
```

8.2.2 Stub Configuration

A stub instance must be configured such that a service client can use this stub instance to invoke a remote method on the target service endpoint. A stub instance may be configured in one of the following manners:

- Static configuration based on the WSDL description of a target service endpoint. Examples of such configuration include protocol binding specified using the `wsdl:binding` and `soap:binding` elements; target service endpoint address specified using the `wsdl:port` element.
- Dynamic runtime configuration using the `javax.xml.rpc.Stub` API. This section specifies further details of this type of configuration.

The JAX-RPC specification does not require support for protocol and transport neutral stub classes that are dynamically configurable for different protocols and transports. A stub class should be bound to a specific protocol and transport.

The `javax.xml.rpc.Stub` interface provides an extensible property mechanism for the dynamic configuration of a stub instance. The following code snippet shows the `Stub` interface:

```
package javax.xml.rpc;
public interface Stub {
    // ...
    void _setProperty(String name, Object value);
    Object _getProperty(String name);
    java.util.Iterator _getPropertyNames();
}
```

The `_setProperty` method sets the value of a property configured on the stub instance. This method throws the `javax.xml.rpc.JAXRPCException` in the following cases:

- If an optional standard property name is specified, however this `Stub` implementation class does not support the configuration of this property.
- If an invalid or unsupported property name is specified or if a value of mismatched property type is passed. An example is an illegal property with the standard property name prefix `javax.xml.rpc`.
- If there is any error in the configuration of a valid property.

Note that the `_setProperty` method may not check validity of a configured property value. An example is the standard property for the target service endpoint address that is not checked for validity in the `_setProperty` method. In this case, any stub configuration errors are detected at the remote method invocation.

The `_getProperty` method gets the value of a property configured on the stub instance. This method should throw the `javax.xml.rpc.JAXRPCException` if an invalid or unsupported property name is passed.

The `_getPropertyNames` returns the names of the supported configurable properties for this stub class.

Standard Properties

The JAX-RPC specification specifies a standard set of properties that are protocol and transport independent. These standard properties may be passed to the `_getProperty` and `_setProperty` methods.

A stub class is not required to support these standard properties. The names and types of these properties are as follows. This table also specifies whether or not a `Stub` implementation is required to support a specific standard property:

TABLE 8-1 Standard Properties for Stub Configuration

Name of Property	Type of Property	Description and Required/Optional support
`javax.xml.rpc.security.auth.username`	`java.lang.String`	User name for authentication **Required** to support HTTP Basic Authentication. Refer to the section 13.1.1 for more details.
`javax.xml.rpc.security.auth.password`	`java.lang.String`	Password for authentication **Required** to support HTTP Basic Authentication. Refer to the section 13.1.1 for more details.
`javax.xml.rpc.service.endpoint.address`	`java.lang.String`	Target service endpoint address. The URI scheme for the endpoint address specification must correspond to the protocol/transport binding for this stub class. **Required**.
`javax.xml.rpc.session.maintain`	`java.lang.Boolean`	This `boolean` property is used by a service client to indicate whether or not it wants to participate in a session with a service endpoint. If this property is set to `true`, the service client indicates that it wants the session to be maintained. If set to `false`, the session is not maintained. The default value for this property is `false`. Refer to the section 13.2, “Session Management” for more details. **Required** to support session management.

Note that the standard JAX-RPC properties are required to be prefixed by the `javax.xml.rpc` package name. Any vendor implementation specific properties must be defined prefixed by a vendor specific package name.

8.2.3 Dynamic Proxy

The JAX-RPC specification requires support for the dynamic proxy for invoking a service endpoint. A dynamic proxy class supports a service endpoint interface dynamically at runtime without requiring any code generation of a stub class that implements a specific service endpoint interface.

Refer to the J2SE [2] documentation for more details on the dynamic proxy APIs `java.lang.reflect.Proxy` and `java.lang.reflect.InvocationHandler`.

The `getPort` method on the `javax.xml.rpc.Service` interface provide support for the creation of a dynamic proxy. The `serviceEndpointInterface` parameter specifies the service endpoint interface that is supported by the created dynamic proxy. The service endpoint interface must conform to the JAX-RPC specification.:

```
package javax.xml.rpc;
public interface Service {
   java.rmi.Remote getPort(QName portName,
                           Class serviceEndpointInterface)
                       throws ServiceException;
   // ...
}
```

This method throws `ServiceException` if there is an error in creation of the dynamic proxy or if there is any missing WSDL related metadata that is required. The JAX-RPC specification does not require that the `getPort` method fully validate the passed `serviceEndpointInterface` against the corresponding WSDL definitions. A JAX-RPC implementation may choose to perform either lazy or eager validation in an implementation specific manner.

A dynamic proxy is required to also support the `javax.xml.rpc.Stub` interface. This provides support for the configuration of a dynamic proxy. Refer to the section 8.2.2, “Stub Configuration” for additional details.

Example

The following example shows the use of a dynamic proxy for invoking an operation on the target service endpoint. Note that this example does not use any generated stub class that implements the `StockQuoteProvider` service endpoint interface:

```
javax.xml.rpc.Service service = //... access a Service instance
com.example.StockQuoteProvider sqp =
   (com.example.StockQuoteProvider)service.getPort(
                portName, StockQuoteProvider.class);
float price = sqp.getLastTradePrice("ACME");
```

8.2.4 DII Call Interface

The `javax.xml.rpc.Call` interface provides support for the dynamic invocation of an operation on the target service endpoint. The `javax.xml.rpc.Service` interface acts as a factory for the creation of `Call` instances.

A client side JAX-RPC runtime system implementation is required to implement the `Call` interface.

The `createCall` method on the `Service` interface may create a `Call` object with an internal implementation specific in-memory model of the WSDL description of the service. The `portName` in the `createCall` method identifies the target service endpoint.

The `createCall` method with no parameters creates an empty `Call` object that needs to be configured using the setter methods on the `Call` interface.

The following code snippet shows the `Service` and `Call` interfaces. Refer to the Javadocs for a complete API specification:

```
package javax.xml.rpc;
public interface Service {
   Call createCall() throws ServiceException;
   Call createCall(QName portName) throws ServiceException;
   Call createCall(QName portName, String operationName)
                                        throws ServiceException;
   Call createCall(QName portName, QName operationName)
                                        throws ServiceException;
   Call[] getCalls(QName portName) throws ServiceException;
   // ... Other methods not shown here
}

// Typesafe Enumeration for ParameterMode
public class ParameterMode {
   private final String mode;
   private ParameterMode(String mode) {
      this.mode = mode;
   }
   public String toString() { return mode; }
   public static final ParameterMode IN = new ParameterMode("IN");
   public static final ParameterMode OUT = new ParameterMode("OUT");
   public static final ParameterMode INOUT  =
                                  new ParameterMode("INOUT");
}

public interface Call {
   // ...
   boolean isParameterAndReturnSpecRequired(QName operation);
   void addParameter(String paramName,
                          QName xmlType,
                          ParameterMode parameterMode);
   void addParameter(String paramName,
                          QName xmlType, Class javaType
                          ParameterMode parameterMode);
   QName getParameterTypeByName(String paramName);
   void setReturnType(QName xmlType);
   void setReturnType(QName xmlType, Class javaType);
   public QName getReturnType();
   void removeAllParameters();

   QName getOperationName();
   void setOperationName(QName operationName);

   QName getPortTypeName();
   void setPortTypeName(QName portType);

   String getTargetEndpointAddress();
   void setTargetEndpointAddress(String address);

   void setProperty(String name, Object value);
   Object getProperty(String name);
   boolean removeProperty(String name);
   java.util.Iterator getPropertyNames();

   // Remote Method Invocation methods
   Object invoke(QName operationName, Object[] inputParams)
```

```
                          throws java.rmi.RemoteException;
    Object invoke(Object[] inputParams)
                          throws java.rmi.RemoteException;
    void invokeOneWay(Object[] inputParams);

    java.util.Map getOutputParams();
    java.util.List getOutputValues();
}
```

Once a `Call` instance is created, various setter and getter methods are used to configure this `Call` instance. The configuration of a `Call` instance includes the following:

- Name of a specific operation
- Port type for the target service endpoint
- Address of the target service endpoint
- Properties specific to the binding to a protocol and transport: example, SOAPAction URI for the SOAP binding to HTTP. The standard properties are specified later in this section.
- Name, type and mode (`IN`, `INOUT`, `OUT`) of the parameters. These properties are optional.
- Return type

A setter method should throw `javax.xml.rpc.JAXRPCException` in the following cases:

- If an optional standard property name is specified, however this `Call` implementation class does not support the configuration of this property.
- If an invalid or unsupported property name is specified or if a value of mismatched property type is passed. An example is an illegal property name with the standard property name prefix `javax.xml.rpc`.
- If there is any error in the configuration of a valid property.

Note that a setter method may not check validity of a configured property value. An example is the standard property for the target service endpoint address that is not checked for validity. In this case, any configuration errors are detected at the invoke time.

In terms of the specification of parameters and return value, a `Call` implementation class is required to support the following cases:

- The `Call` implementation class determines the types of the parameters and return value from the invocation of `addParameter` and `setReturnType` methods in the client code. The method `isParameterAndReturnSpecRequired` is required to return `true` in this case.
- The `Call` implementation class determines the types and modes of the parameters in an implementation specific manner. For example, a `Call` implementation class may determine parameter types by using Java reflection on parameters, using WSDL service description and the configured type mapping registry. In this case, the client code is not required to invoke any `addParameter` and `setReturnType` methods before calling the `invoke` method. The method `isParameterAndReturnSpec-Required` is required to return `false` in this case. The methods `addParameter`, `removeAllParameters` and `setReturnType` may throw `JAXRPCException` if invoked in this case.

8.2.4.1 Invocation Modes

The `Call` implementation class is required to support the following invocation modes:

- Synchronous request-response mode: The `invoke` method calls a specified operation using a synchronous request-response interaction mode. The `invoke` method takes as parameters the object values corresponding to the parameter types. The implementation of the `invoke` method checks whether the passed parameter values match the number, order and types of parameters specified in the corresponding WSDL specification of the same operation. The `invoke` method is required to throw exception `java.rmi.RemoteException` if there is any error in the remote method invocation. The method throws `JAXRPCException` if the `Call` object is not configured properly or if parameters and return type are incorrectly specified.
- One-way mode: The `invokeOneWay` method invokes a remote method using the one-way interaction mode. The client thread does not block waiting for the completion of the processing of this remote method invocation on the service endpoint. When the protocol in use is SOAP/HTTP, the client should block until an HTTP response code has been received or an error occurs. Reception of a response code simply means that the transmission of the request is complete, and not that the request was accepted or processed. This method must not throw any remote exceptions. This method is required to throw a `JAXRPCException` if there is an error in the configuration of the `Call` object (example: a non-void return type has been incorrectly specified for the one-way call) or if there is any error during the invocation of the one-way remote call.

The method `getOutputParams` returns a `java.util.Map` of {name, value} for the output parameters for the last invoked operation. The parameter names in the returned `Map` are of the type `java.lang.String`. The type of a value depends on the mapping between the Java and XML types. This method is required to throw `JAXRPCException` if invoked for a one-way operation or if invoked before any `invoke` method has been called.

The method `getOutputValues` returns a list (of Java type `java.util.List`) of values for the output parameters. This method is required to throw `JAXRPCException` if invoked for a one-way operation or if invoked before any `invoke` method has been called.

8.2.4.2 Standard Properties

The JAX-RPC specification specifies a standard set of properties that may be passed to the `Call.setProperty` method. A `Call` implementation class may also support additional set of properties. However, the specification of these additional properties is out of scope for JAX-RPC.

The names and types of these properties are as follows. The following table also specifies whether or not a `Call` implementation is required to support a specific standard property:

TABLE 8-2 Standard Properties for the Call Interface

Name of Property	Type of Property	Description
`javax.xml.rpc.security.auth.username`	`java.lang.String`	User name for Authentication **Required** to support HTTP Basic Authentication. Refer to the section 13.1.1 for more details.
`javax.xml.rpc.security.auth.password`	`java.lang.String`	Password for Authentication. **Required** to support HTTP Basic Authentication. Refer to the section 13.1.1 for more details.
`javax.xml.rpc.session.maintain`	`java.lang.Boolean`	This `boolean` property is used by a service client to indicate whether or not it wants to participate in a session with a service endpoint. If this property is set to `true`, the service client indicates that it wants the session to be maintained. If set to `false`, the session is not maintained. The default value for this property is `false`. Refer to the section 13.2, "Session Management" for more details. **Required** to support session management.
`javax.xml.rpc.soap.operation.style`	`java.lang.String`	"`rpc`" if the operation style is `rpc`; "`document`" if the operation style is `document`. Note that a `Call` implementation may choose to not allow setting of this property. In this case, the `setProperty` method throws `JAXRPCException`. **Optional**

TABLE 8-2 Standard Properties for the Call Interface

Name of Property	Type of Property	Description
`javax.xml.rpc.soap.http.soapaction.use`	`java.lang.Boolean`	This boolean property indicates whether or not `SOAPAction` is to be used. The default value of this property is `false` indicating that the `SOAPAction` is not used. **Optional**
`javax.xml.rpc.soap.http.soapaction.uri`	`java.lang.String`	Indicates the `SOAPAction` URI if the `javax.xml.rpc.soap.http.soapaction.use` property is set to `true`. **Optional**
`javax.xml.rpc.encodingstyle.namespace.uri`	`java.lang.String`	Encoding style specified as a namespace URI. The default value is the SOAP 1.1 encoding `http://schemas.xmlsoap.org/soap/encoding/` **Optional**

Note that the standard properties are prefixed by the `javax.xml.rpc` package name. Any vendor implementation specific properties must be defined prefixed by a vendor specific package name.

8.2.4.3 Example

The following code example shows a typical use of the `Call` interface:

```
javax.xml.rpc.Service service = //... get a Service instance
javax.xml.rpc.Call call = service.createCall(
                                portName, “<operationName>”);
call.addParameter(“param1”, <xsd:string>,
                  ParameterMode.IN);
call.addParameter(“param2”, <xsd:string>,
                  ParameterMode.OUT);
call.setReturnType(<xsd:int>);
Object[] inParams = new Object[] {“<SomeString>”};
Integer ret = (Integer) call.invoke(inParams);
Map outParams = call.getOutputParams();
String outValue = (String)outParams.get(“param2”);
```

An alternative way to invoke the same remote method using the DII `Call` interface is as follows. In this case, the `Call` implementation class takes the responsibility of determining the corresponding types for the parameters and return value:

```
javax.xml.rpc.Service service = //... get a Service instance
javax.xml.rpc.Call call = service.createCall(
                             portName, “<operationName>”);
Object[] params = new Object[] {“<SomeString>”};
Integer ret = (Integer) call.invoke(params);
String outValue = (String)call.getOutputParams().get(“param2”);
```

8.2.5 Abstract ServiceFactory

The `javax.xml.rpc.ServiceFactory` is an abstract class that provides a factory for the creation of instances of the type `javax.xml.rpc.Service`. This abstract class follows the abstract static factory design pattern. This enables a J2SE based client to create a `Service` instance in a portable manner without using the constructor of the `Service` implementation class.

```
package javax.xml.rpc;
public abstract class ServiceFactory {
   protected ServiceFactory() { ... }
   public static ServiceFactory newInstance()
                  throws ServiceException { ...}
   public abstract Service createService(
                         java.net.URL wsdlDocumentLocation,
                         QName serviceName)
                  throws ServiceException;
   public abstract Service createService(
                         QName serviceName)
                  throws ServiceException;
   public abstract Service loadService(Class serviceInterface)
                  throws ServiceException;
   public abstract Service loadService(
                         java.net.URL wsdlDocumentLocation,
                         Class serviceInterface,
                         java.util.Properties properties)
                  throws ServiceException;
   public abstract Service loadService(
                         java.net.URL wsdlDocumentLocation,
                         QName serviceName,
                         java.util.Properties properties)
                  throws ServiceException;
}
```

A client-side JAX-RPC runtime system is required to provide implementation of the abstract `ServiceFactory` class. The `ServiceFactory` implementation class is set using the system property named `SERVICEFACTORY_PROPERTY`.

A JAX-RPC implementation that does not use a consistent naming convention for generated service implementation classes must allow an application developer to specify sufficient configuration information so that the `ServiceFactory.loadService(Class)` method will succeed provided all the generated artifacts are packaged with the application. Examples of configuration information include: properties- or XML-based configuration files that are looked up as resources using the `java.lang.ClassLoader getResource`/`getResources` APIs, system properties and the preferences APIs (java.util.prefs) introduces in J2SE 1.4.

A J2SE service client should use this model to get access to the `Service` object. A J2SE based service client may use the JNDI naming context to lookup a service instance.

A J2EE-based service client should not use the `ServiceFactory` APIs to access a service. Moreover, packaging implementation-specific artifacts (including classes and configuration information) with an application is strongly discouraged as it would make the application non-portable. Instead, J2EE-based service clients should use JNDI to lookup an instance of a `Service` class as specified in JSR-109 [10].

Example

The following code snippet shows the use of the `ServiceFactory`.

```
Service service = ServiceFactory.newInstance().createService(...);
```

8.2.6 ServiceException

The `javax.xml.rpc.ServiceException` is thrown from the methods defined in the `javax.xml.rpc.Service` and `javax.xml.rpc.ServiceFactory` APIs. The following code snippet shows the `ServiceException`:

```
package javax.xml.rpc;
public class ServiceException extends java.lang.Exception {
    public ServiceException() { ... }
    public ServiceException(String message) { ... }
    public ServiceException(Throwable cause) { ... }
    public ServiceException(String message, Throwable cause) { ... }
    public Throwable getLinkedCause() { ... }
}
```

8.2.7 JAXRPCException

The `javax.xml.rpc.JAXRPCException` is thrown from the core APIs to indicate exceptions related to the JAX-RPC runtime mechanisms. A `JAXRPCException` is mapped to a `java.rmi.RemoteException` if the former is thrown during the processing of a remote method invocation.

```
package javax.xml.rpc;
public class JAXRPCException extends java.lang.RuntimeException {
    public JAXRPCException() { ... }
    public JAXRPCException(String message) { ... }
    public JAXRPCException(Throwable cause) { ... }
    public JAXRPCException(String message, Throwable cause) { ... }
    public Throwable getLinkedCause() { ... }
}
```

8.2.8 Additional Classes

The JAX-RPC specification specifies the following additional classes for commonly used constants:

- `javax.xml.rpc.NamespaceConstants` class for common XML namespace prefixes and URIs
- `javax.xml.rpc.encoding.XMLType` class for `QNames` of the supported set of XML schema types and SOAP encoded types

9 Service Client Programming Model

This chapter describes the programming model used by a JAX-RPC service client. This chapter specifies the following details:

- Requirements for the JAX-RPC client programming model
- Programming model for a J2EE container based service client
- Programming model for a J2SE based service client

9.1 Requirements

The JAX-RPC specification specifies the following requirements for the service client programming model:

- Service client programming model must be independent of how a service endpoint is realized on the server side. A service client must invoke a service endpoint using the same client programming model irrespective of whether a service endpoint has been defined on the J2EE platform or on a non-Java platform.
- Service client environment should be capable of importing a WSDL document and generating a Java based client side representation for a service described in this document. A client side representation includes classes generated based on the mapping of the WSDL definitions to the corresponding Java representation.
- Service client programming model must not be exposed or tied to a specific protocol, transport or any JAX-RPC implementation specific mechanism. For example, a JAX-RPC service client should not be exposed to how a JAX-RPC client side runtime system invokes a remote method using a specific implementation level interaction mode and connection management.

The JAX-RPC specification does not address discovery of a JAX-RPC service from a service registry. A JAX-RPC implementation is not required to support the Java APIs for XML Registries (JAXR 1.0).

9.2 J2EE based Service Client Programming Model

Note – This section is non-prescriptive and is not required to be implemented by a client side JAX-RPC 1.1 implementation. The standard JAX-RPC client programming model for J2EE would be specified in the JSR-109, J2EE 1.4, EJB 2.1 and Servlet 2.4 specifications. This section provides input to these specifications.

This section describes the programming interface and deployment descriptor that allows a J2EE component (either a servlet, EJB) or a J2EE application client to act as a service client and invoke a service endpoint. The programming model approach in this section is consistent with the J2EE programming model for looking up external resources.

9.2.1 Component Provider

The component provider references an external service using a logical name called *service reference*. The component provider uses the service reference to get access to the service ports as follows:

- The component provider assigns an entry in the component's environment to the service reference. The component provider uses the deployment descriptor to declare a service reference. Refer to the next section for more details on the deployment descriptor.
- The component provider looks up an instance of a service class using the JNDI namespace. Refer to the section 4.3.9, "WSDL Service" for the details on the service class. Note that the service class should implement the `javax.naming.Referenceable` and/or `java.io.Serializable` interface to support registration and lookup from the JNDI namespace.
- The component provider uses an appropriate method on the looked up service instance to get one or more proxy objects for the target service endpoint. A proxy for a service endpoint can be either a dynamic proxy or an instance of a generated stub class or a `javax.xml.rpc.Call` object.

Example

The following code snippet illustrates how a component provider looks up a service, gets a stub instance for a service endpoint and invokes a remote method on the service endpoint:

Code Example 7 J2EE Programming Model: Getting a stub instance

```
Context ctx = new InitialContext();
com.example.StockQuoteService sqs =
   ctx.lookup("java:comp/env/StockQuoteService");
com.example.StockQuoteProvider sqp =
          sqs.getStockQuoteProviderPort();
float quotePrice = sqp.getLastTradePrice("ACME");
```

In the typical case, the component provider calls the getter method on the service (`getStockQuoteProviderPort` method in the above example) with no parameters. Refer to the Code Example 1 and Code Example 6 for the code snippets for `StockQuoteProvider` and `StockQuoteService`.

The following example shows the creation of a dynamic proxy.

Code Example 8 J2EE Programming Model: Creating a dynamic proxy

```
Context ctx = new InitialContext();
javax.xml.rpc.Service sqs = ctx.lookup(
                        "java:comp/env/DynamicService");
com.example.StockQuoteProvider sqp =
   (com.example.StockQuoteProvider)sqs.getPort(
                portName,
                StockQuoteProvider.class);
float quotePrice = sqp.getLastTradePrice("ACME");
```

Note that the JAX-RPC specification does not specify a standard JNDI subcontext for the service references.

9.2.2 Deployment Descriptor

The component provider declares all the service references in the deployment descriptor (for a specific J2EE component) using the `service-ref` element. Each `service-ref` contains the following elements:

- `description`: This element describes the referenced service. This information is meant for the deployer.
- `service-ref-name`: The name of the logical reference used in the component code. This name is relative to the `java:comp/env` subcontext. For example, the name is `StockQuoteService` in the Code Example 7 rather than the `java:comp/env/StockQuoteService`.
- `service-ref-type`: This element indicates the fully qualified name of the service class returned by the JNDI lookup. The Code Example 7 has `com.example.StockQuoteService` as the `service-ref-type`, while the Code Example 8 has `javax.xml.rpc.Service` as the `service-ref-type`.
- `type-mapping`: This optional element specifies the requirements for the pluggable serializers and deserializers.

9.2.3 Deployer

The deployer performs the following steps for each service reference declared in a component's deployment descriptor:

- The deployer links a service reference to the actual representation and configuration of the corresponding service. Such linking of a service reference is specific to the implementation of a container. For example, the deployer may choose to link the logical service reference to the imported WSDL based description of the service. However, JAX-RPC does not mandate any specific linking mechanism.
- The deployer also provides and configures required configuration information for the service instance and service endpoint proxies. For example, this configuration information includes target endpoint address, properties specific to a protocol and underlying transport, security information and type mapping registry.
- The deployer ensures that the configuration of a service and service proxy is based on the protocol binding specified in the WSDL description of this service. Refer to the section 8.2.2, "Stub Configuration" for more details.

9.3 J2SE based Service Client Programming Model

A J2SE based service client uses one of the following approaches for the invocation of a JAX-RPC service endpoint:

- The service client uses the generated stub classes. A WSDL to Java mapping tool imports a WSDL based service description and maps it to the corresponding client side Java representation. The generated client side artifacts may include serializers, deserializers, holders and utility classes.
- The service client uses the `javax.xml.rpc.Service` interface to create a dynamic proxy for the target service endpoint.
- The service client uses the `javax.xml.rpc.Service` interface to create a `Call` object. Next, the service client uses the `javax.xml.rpc.Call` interface to dynamically invoke an operation on the target service endpoint.

10 Service Endpoint Model

This chapter specifies the service endpoint model for a JAX-RPC service developed and deployed on a servlet container based JAX-RPC runtime system.

The JAX-RPC specification does not specify the service endpoint model for a JAX-RPC service developed using the EJB component model. Refer to the JSR-109 and EJB 2.1 specifications for the EJB model for JAX-RPC service endpoints.

10.1 Service Developer

This section specifies the role of a service developer in the definition of a servlet based JAX-RPC service endpoint component.

First, the service developer performs one of the following steps depending on whether or not the service developer has the WSDL document for a service endpoint:

- The service developer generates a service endpoint interface from a WSDL document using a WSDL to Java mapping tool. Refer to the section 4.3, “WSDL to Java Mapping” for more details.
- The service developer defines a service endpoint interface that conforms to the specification in the section 5.2, “JAX-RPC Service Endpoint Interface”. In this case, the developer may not have the WSDL document for this service.

Next, the service developer develops a service endpoint component by providing a service endpoint class as follows:

- The service endpoint class is required to implement a service endpoint interface.
- The service endpoint class is required to have a default public constructor.
- The service endpoint class may implement the `ServiceLifecycle` interface. Refer to the section 10.1.1, “JAX-RPC Service Endpoint Lifecycle” for more details on the lifecycle of the JAX-RPC service endpoint class.
- The service endpoint class is allowed to obtain references to resources and enterprise beans by using JNDI to lookup these resources. The deployment elements (`env-entry`, `ejb-ref`, `ejb-local-ref`, `resource-ref` and `resource-env-ref`) are specified in the web application deployment descriptor. Refer to the Servlet 2.3 and J2EE 1.3 specifications for more details on the web application environment that is also available to the service endpoint class.

10.1.1 JAX-RPC Service Endpoint Lifecycle

The service endpoint class may implement the following `ServiceLifecycle` interface:

```
package javax.xml.rpc.server;
public interface ServiceLifecycle {
    void init(Object context) throws ServiceException;
    void destroy();
}
```

If the service endpoint class implements the `ServiceLifeycle` interface, the servlet container based JAX-RPC runtime system is required to manage the lifecycle of the corresponding service endpoint instances. The lifecycle of a service endpoint instance is realized through the implementation of the `init` and `destroy` methods of the `ServiceLifecycle` interface.

The JAX-RPC runtime system is responsible for loading and instantiation of service endpoint instances. The JAX-RPC runtime system may choose to load and instantiate a service endpoint instance during the startup of the runtime system or when a service endpoint instance is needed to service an RPC request. The JAX-RPC runtime system loads the service endpoint class using the Java class loading facility. After service endpoint class is loaded, the JAX-RPC runtime system instantiates the class.

After the service endpoint instance is instantiated, the JAX-RPC runtime system is required to initialize the endpoint instance before any requests can be serviced. The JAX-RPC runtime system is required to invoke the `ServiceLifecycle.init` method (if this interface is implemented by the endpoint class) for the initialization of the service endpoint instance. The service endpoint instance uses the `init` method to initialize its configuration and setup access to any external resources. The `context` parameter in the `init` method enables the endpoint instance to access the endpoint context provided by the underlying JAX-RPC runtime system.

Once a service endpoint instance has been initialized (and in a method ready state), the JAX-RPC runtime system may dispatch multiple remote method invocations to the service endpoint instance. These method invocations must correspond to the remote methods in the service endpoint interface implemented by the service endpoint class.

Instances of a specific service endpoint class are considered equivalent when not servicing a client invoked remote method invocation. A service endpoint instance does not maintain any client specific state across remote method invocations. So service endpoint instances are capable of pooling by the JAX-RPC runtime system. The JAX-RPC runtime system can invoke a remote method on any available and method ready instances of a specific service endpoint class.

The JAX-RPC runtime system is required to invoke the `destroy` method when the runtime system determines that the service endpoint instance needs to be removed from service of handling remote invocations. For example, the JAX-RPC runtime may remove an endpoint instance from service when the runtime system is shutting down or managing memory resources.

The service endpoint class releases its resources and performs cleanup in the implementation of the `destroy` method.

After successful invocation of the `destroy` method, a service endpoint instance is ready for garbage collection. The JAX-RPC runtime system should not dispatch any remote method invocations to a destroyed endpoint instance. The JAX-RPC runtime system is required to instantiate and initialize a new service endpoint instance for servicing new remote method invocations.

10.1.2 Servlet based Endpoint

During the deployment of a service endpoint component on a servlet container based JAX-RPC runtime system, a service endpoint class is associated with a servlet. The associated servlet class is provided by the JAX-RPC runtime system (not by service endpoint developer) during the deployment. This association is configured in a manner specific to a JAX-RPC runtime system and its deployment tool. For example, a JAX-RPC deployment tool may configure a 1-1 association between a servlet class and service endpoint class. The associated servlet class corresponds to the configured transport binding for the service endpoint. For example, the servlet class `javax.servlet.http.HttpServlet` is used for the HTTP transport.

The associated servlet typically takes the responsibility of handling transport specific processing of an RPC request and for initiating dispatch to the target service endpoint instance. Each `Servlet.service` method maps to a single remote method invocation on the target service endpoint instance. The thread model (whether single threaded or concurrent) for the remote method invocation on the service endpoint instance depends on the runtime system specific servlet associated with the corresponding endpoint class. The Servlet specification [3] provides facility for both concurrent and single threaded model (the latter through the `SingleThreadModel` interface) for the `service` method on a servlet.

When processing an incoming SOAP request for a one-way operation, the associated servlet is required to send back an 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 the target service endpoint instance.

10.1.3 ServletEndpointContext

For service endpoint components deployed on a servlet container based JAX-RPC runtime system, the `context` parameter in the `ServiceLifecycle.init` method is required to be of the Java type `javax.xml.rpc.server.ServletEndpointContext`. The `ServletEndpoint Context` provides an endpoint context maintained by the underlying servlet container based JAX-RPC runtime system.

Note that the JAX-RPC specification specifies the standard programming model for a servlet based endpoint. The goal of JAX-RPC specification is not to define a more generic abstraction for the endpoint context or session that is independent of any specific component model, container and protocol binding. Such generic abstractions and endpoint model are outside the scope of the JAX-RPC specification.

The following code snippet shows the `ServletEndpointContext` interface.

```
package javax.xml.rpc.server;
public interface ServletEndpointContext {
   public java.security.Principal getUserPrincipal();
   public boolean isUserInRole(String role);
   public javax.xml.rpc.handler.MessageContext getMessageContext();
   public javax.servlet.http.HttpSession getHttpSession();
   public javax.servlet.ServletContext getServletContext();
}
```

A servlet container based JAX-RPC runtime system is required to implement the `ServletEndpointContext` interface. The JAX-RPC runtime system is required to provide appropriate session, message context, servlet context and user principal information per method invocation on service endpoint instances.

The `getHttpSession` method returns the current HTTP session (as a `javax.servlet.http.HTTPSession`). When invoked by the service endpoint instance within a remote method implementation, the `getHttpSession` returns the HTTP session associated currently with this method invocation. This method is required to return `null` if there is no HTTP session currently active and associated with this service endpoint instance. An endpoint class should not rely on an active HTTP session being always there; the underlying JAX-RPC runtime system is responsible for managing whether or not there is an active HTTP session.

The `getHttpSession` method throws `JAXRPCException` if it is invoked by a non HTTP bound endpoint. The JAX-RPC specification does not specify any transport level session abstraction for non-HTTP bound endpoints.

The method `getMessageContext` returns the `MessageContext` targeted for this endpoint instance per remote method invocation. This enables the service endpoint instance to access the `MessageContext` propagated by request `HandlerChain` (and its contained `Handler` instances) to the target endpoint instance and to share any SOAP message processing related context. The endpoint instance can access and manipulate the `MessageContext` and share the SOAP message processing related context with the response `HandlerChain`. Refer to the section 12.1, “JAX-RPC Handler APIs” for more details on the SOAP message handler APIs.

The JAX-RPC runtime system is required to set the appropriate `MessageContext` based on the `HandlerChain` associated with the target endpoint. If there is no associated `MessageContext`, this method returns `null`. The `getMessageContext` method is required to throw the `java.lang.IllegalStateException` if this method is invoked outside a remote method implementation by an endpoint class.

The method `getServletContext` returns the `ServletContext` associated with the web application that contain this endpoint. According to the Servlet specification, there is one context per web application (deployed as a `WAR`) per JVM. A servlet based service endpoint component is deployed as part of a web application.

The method `getUserPrincipal` returns a `java.security.Principal` instance that contains the name of the authenticated user for the current method invocation on the endpoint instance. This method returns `null` if there is no associated principal yet. The underlying JAX-RPC runtime system takes the responsibility of providing the appropriate authenticated principal for a remote method invocation on the service endpoint instance.

The method `isUserInRole` returns a `boolean` indicating whether the authenticated user for the current method invocation on the endpoint instance is included in the specified logical “role”. This method returns false if the user has not been authenticated. The determination of whether the authenticated user is included in the specified role must be carried out according to the Servlet specification.

10.2 Packaging and Deployment Model

The JAX-RPC specification does not specify any normative model for packaging and deployment of service endpoints on a servlet container based JAX-RPC runtime system. A JAX-RPC 1.1 implementation is allowed to have a vendor-specific deployment and packaging model for servlet based service endpoints.

The J2EE 1.4 specification [3] and JSR-109 [10] would standardize the standard web service deployment model for both EJB and servlet based service endpoints.

11 Service Context

The JAX-RPC specification allows service context to be associated with a remote call to a service endpoint. For example, a service context may carry information corresponding to the SOAP header entries.

This chapter describes the client programming model for the management and processing of service context. Note that the JAX-RPC specification does not (nor intends to, in future) specify the semantic content of the service contexts.

This chapter describes non-prescriptive guidelines for the mapping of the service context.

11.1 Context Definition

The concrete format of the service context and its transmission between a service client and service endpoint depends on the protocol binding associated with the JAX-RPC call. For example, if SOAP over HTTP binding is used, the service context is transmitted in the SOAP Header. Another example is Basic Authentication information that is carried in the HTTP request header.

The JAX-RPC programming model is independent of whether service context is carried in the XML based protocol or the underlying transport.

Note that the SOAP specification [4] does not define any standard formats for the SOAP headers. Refer to the chapter 14 (“Interoperability”) for the implication of service context propagation on the interoperability.

The WSDL 1.1 specification includes a `soap:header` element in the SOAP binding. A `soap:header` element defines an header entry transmitted in the SOAP Header. Multiple `soap:header` elements can be defined per operation for both input and output. Note that the WSDL 1.1 specification does not require that SOAP headers be listed exhaustively in the SOAP binding.

An exported WSDL document (for a JAX-RPC service endpoint definition) includes `soap:header` elements for both input and output of each operation. These `soap:header` elements may be defined specific to either a service endpoint definition or a JAX-RPC runtime system or a server-side J2EE container.

On the service client side, a WSDL to Java mapping tool maps `soap:header` elements to either implicit or explicit service context. A client side JAX-RPC runtime system or generated stub class may add service context for a remote method call. This service context information is in addition to that specified in the WSDL defined `soap:header` elements.

11.2 Programming Model

The JAX-RPC specification classifies service context as follows based on the JAX-RPC programming model:

- Implicit service context
- Explicit service context

11.2.1 Implicit Service Context

This form of service context is managed and propagated implicitly by generated stubs or client and server side JAX-RPC runtime systems. A service client or service implementation does not need to programmatically manage an implicit service context.

Example

A J2EE container with the JAX-RPC runtime system implicitly propagates the security context with a remote method invocation. The propagation of the security context does not require any explicit programming by the service client.

11.2.2 Explicit Service Context

Explicit service context is represented in the form of additional parameters appended following the service endpoint defined parameters in the remote method signature. The type of a service context parameter depends on whether the service context is of `IN`, `OUT` or `INOUT` type. Explicit service context is typically used to represent application specific service context.

An explicit service context parameter may be either based on the mapping of a `soap:header` element in the WSDL service description or generated specific to a client-side JAX-RPC implementation. The name of the Java method parameter is mapped from the `name` of the `part` referenced in the `soap:header` element. If a `soap:header` is mapped to a Java method argument, and only in this case, any of its `soap:headerfault`(s) may be mapped to service specific exceptions (see section 6.6).

A Java to WSDL mapping tool is required to provide a facility for specifying metadata related to the mapping of the explicit service context. This metadata identifies whether a specific parameter or return type in the Java method signature is mapped to a SOAP header element instead of an element in the SOAP body. Future versions of the JAX-RPC specification would consider specifying a standard approach for the specification of such mapping metadata.

Example

The following Java interface shows example of an explicit service context. In this example, the `context` is a service context parameter appended after the service defined parameters of the `getStockQuote` method. This context parameter is of `inout` type and is mapped to a `Holder` class. The generated stub class for this service endpoint interface represents this `context` parameter as a SOAP header entry:

```
package com.example;
public interface StockQuoteProvider extends java.rmi.Remote {
   // Method returns last trade price
```

```
    float getStockQuote(String tickerSymbol, StringHolder context)
          throws java.rmi.RemoteException;
}
```

Since processing of a SOAP header can generate SOAP faults, a method with an explicit context parameter may throw a `javax.xml.rpc.soap.SOAPFaultException` that extends the `java.lang.RuntimeException`.

11.3 Processing of Service Context

A JAX-RPC runtime system should be capable of processing both inbound or outbound service contexts. A J2EE container based JAX-RPC runtime system typically processes the infrastructure related service context. Examples are transaction and security contexts. The JAX-RPC runtime system uses the container provided services (example: transaction manager) to process the service context.

The processing of an inbound service context depends on the semantics of the service context and also on the consumer of the service context-whether context consumer is server-side JAX-RPC runtime system, J2EE container or the target service endpoint implementation.

The JAX-RPC specification does not specify any standard server-side model for the processing of the service context.

12 SOAP Message Handlers

This chapter specifies the requirements and APIs for SOAP message handlers. A SOAP message handler gets access to the SOAP message that represents either an RPC request or response. A typical use of a SOAP message handler is to process the SOAP header blocks as part of the processing of an RPC request or response. A few typical examples of handlers are:

- Encryption and decryption handler
- Logging and auditing handler
- Caching handler

SOAP message handlers are tied to web service endpoints (either on client or server) and are used to provide additional SOAP message processing facility as an extension to these components.

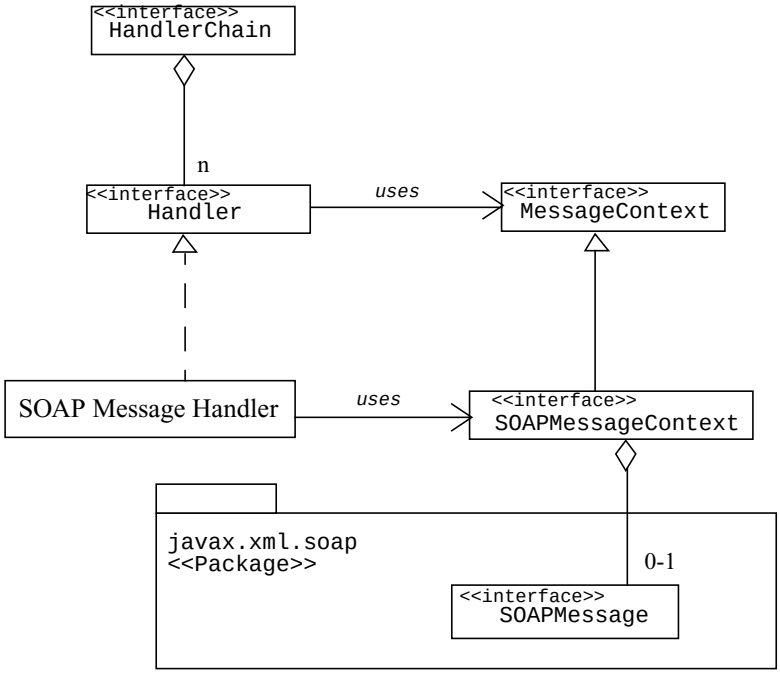
An example of a SOAP message handler is as follows:

A secure stock quote service requires that the SOAP body be encrypted and the SOAP message digitally signed to prevent any unauthorized access or tampering with the RPC requests or responses for this service. A SOAP message handler on the service client side encrypts and digitally signs the SOAP message before communicating the request to the remote service endpoint. On the server side, a SOAP message handler intercepts the SOAP message, performs verification and decryption steps before dispatching the RPC request to the target service endpoint implementation. The same steps are repeated in reverse for the RPC response carried in a SOAP message.

Note that other types of handlers (for example; stream based handlers, post-binding typed handlers) can be developed for SOAP message processing specific to a JAX-RPC runtime system and the corresponding container. However, the JAX-RPC specification specifies APIs for only SOAP message handlers. Any container specific processing of SOAP message is out of scope for this version of the JAX-RPC specification.

12.1 JAX-RPC Handler APIs

The following diagram shows the class diagram for the handler APIs.



12.1.1 Handler

A JAX-RPC handler is required to implement the `javax.xml.rpc.handler.Handler` interface. The following code snippet shows the `Handler` interface.

```
package javax.xml.rpc.handler;
public interface Handler {
    boolean handleRequest(MessageContext context);
    boolean handleResponse(MessageContext context);
    boolean handleFault(MessageContext context);
    // ...
}
```

A `Handler` implementation class is required to provide a default constructor.

The methods `handleRequest` and `handleResponse` perform the actual processing work for a handler. The method `handleRequest` processes the request SOAP message, while the method `handleResponse` processes the response SOAP message. The method `handleFault` performs the SOAP fault processing.

The `MessageContext` parameter provides access to the message context (for example: a SOAP message that carries an RPC request or response) that is processed by a handler. The section 12.2, “Handler Model” specifies more details about the implementation model for a SOAP message handler.

12.1.2 SOAP Message Handler

A SOAP message handler class is required to implement the `javax.xml.rpc.handler.Handler` interface. This handler gets access to the SOAP message (that carries either an RPC request or response in the SOAP Body element) from the `SOAPMessageContext`. Refer to the `MessageContext` and `SOAPMessageContext` interfaces for more details.

12.1.3 GenericHandler

The `javax.xml.rpc.handler.GenericHandler` class is an abstract class that implements the `Handler` interface. Handler developers should typically subclass the `GenericHandler` class unless the `Handler` implementation class needs another class as its superclass.

The `GenericHandler` class is a convenience abstract class that makes writing handlers easy. This class provides default implementations of the lifecycle methods `init` and `destroy` and also different handle methods. A handler developer should only override methods that it needs to specialize as part of the derived `Handler` implementation class.

12.1.4 HandlerChain

The `javax.xml.rpc.handler.HandlerChain` represents an ordered list of handlers. All elements in the `HandlerChain` are of the type `javax.xml.rpc.handler.Handler`.

```
package javax.xml.rpc.handler;
public interface HandlerChain extends java.util.List {
   boolean handleRequest(MessageContext context);
   boolean handleResponse(MessageContext context);
   boolean handleFault(MessageContext context);

   // Lifecycle method
   void init(java.util.Map config);
   void destroy();

   //... Additional methods not shown
}
```

A JAX-RPC runtime system implementation is required to provide the implementation class (or classes) for the `HandlerChain` interface.

An implementation class for the `HandlerChain` interface abstracts the policy and mechanism for the invocation of the registered handlers. The default invocation policy is to invoke handlers in the order of registration with the `HandlerChain` instance. However, a `HandlerChain` implementation class may apply additional handler invocation policies based on the SOAP message processing model and the processed SOAP headers.

In the case of a one-way call, only the `handleRequest` methods are invoked. Neither `handleResponse` nor `handleFault` methods are invoked as part of the processing of the SOAP message `HandlerChain`.

Example

In this example, three `Handler` instances `H1`, `H2` and `H3` are registered (in this order) in a single `HandlerChain` instance that is used for both request and response processing. The default invocation order for these handlers is as follows:

- H1.handleRequest
- H2.handleRequest
- H3.handleRequest
- H3.handleResponse
- H2.handleResponse
- H1.handleResponse

The implementation of the handle methods in a SOAP message handler may alter this invocation order. Refer to the section 12.2, “Handler Model” for more details. A HandlerChain implementation class may also provide implementation specific invocation policy.

12.1.5 HandlerInfo

The HandlerInfo class represents the configuration data for a Handler. A HandlerInfo instance is passed in the Handler.init method to initialize a Handler instance.

The following code snippet shows the HandlerInfo class:

```
package javax.xml.rpc.handler;
public class HandlerInfo implements java.io.Serializable {
   public HandlerInfo() { }
   public HandlerInfo(Class handlerClass, java.util.Map config,
                      QName[] headers) {
              ... }
   public void setHandlerClass(Class handlerClass) { ... }
   public Class getHandlerClass() { ... }
   public void setHandlerConfig(java.util.Map config) { ... }
   public java.util.Map getHandlerConfig() { ... }

   public QName[] getHeaders() { ... }
   public void setHeaders(QName[] headers) { ... }
}
```

Refer to the Javadocs for more details on the HandlerInfo class.

12.1.6 MessageContext

The interface MessageContext abstracts the message context that is processed by a handler in the handleRequest, handleResponse or handleFault method.

```
package javax.xml.rpc.handler;
public interface MessageContext {
   void setProperty(String name, Object value);
   Object getProperty(String name);
   void removeProperty(String name);
   boolean containsProperty(String name);
   java.util.Iterator getPropertyNames();
}
```

The MessageContext interface provides methods to manage a property set. The MessageContext properties enable handlers in a handler chain to share processing related state. For example, a handler may use the setProperty method to set value for a specific property in the message context. One or more other handlers in the handler chain may use the getProperty method to get the value of the same property from the message context.

Please note that there is no specified relationship between the message context properties and either the Stub properties described in section 8.2.2, “Stub Configuration” or the Call properties described in section 8.2.4, “DII Call Interface”.

A `HandlerChain` instance is required to share the same `MessageContext` across `Handler` instances that are invoked during a single request and response or fault processing on a specific service endpoint.

`Handler` instances in a handler chain should not rely on the thread local state to share state between handler instances. Handler instances in a `HandlerChain` should use the `MessageContext` to share any SOAP message processing related state.

12.1.7 SOAPMessageContext

The interface `SOAPMessageContext` provides access to the SOAP message for either RPC request or response. The `javax.xml.soap.SOAPMessage` specifies the standard Java API for the representation of a SOAP 1.1 message with attachments. Refer to the JAXM specification [13] for more details on the `SOAPMessage` interface.

```
package javax.xml.rpc.handler.soap;
public interface SOAPMessageContext extends MessageContext {
   SOAPMessage getMessage();
   void setMessage(SOAPMessage message);
   // ...
}
```

12.2 Handler Model

This section specifies the configuration and processing model for the SOAP message handlers.

12.2.1 Configuration

A JAX-RPC handler may be configured and used on the service client as follows:

- On the service client side, a request handler is invoked before an RPC request is communicated to the target service endpoint.
- On the service client side, a response or fault handler is invoked before an RPC response is returned to the service client from the target service endpoint.

A JAX-RPC handler may configured and used on a service endpoint as follows:

- On the service endpoint side, a request handler is invoked before an RPC request is dispatched to the target service endpoint.
- On the service endpoint side, a response or fault handler is invoked before communication back to the service client from the target service endpoint.

12.2.2 Processing Model

A SOAP message handler is required to process a SOAP message and generate SOAP faults based on the processing model specified in the SOAP [4] specification.

HandlerChain

A HandlerChain is configured to act in the role of one or more SOAP actors, each actor specified using a URI called SOAP actor name. A HandlerChain always acts in the role of a special SOAP actor next. Refer to the SOAP specification for the URI name for this special SOAP actor. The following methods in the HandlerChain interface support configuration of the SOAP actor roles:

```
package javax.xml.rpc.handler;
public interface HandlerChain extends java.util.List {
    // ...
    void setRoles(String[] soapActorNames);
    String[] getRoles();
}
```

A SOAP message Handler instance is associated with SOAP header blocks using the qualified name of the outermost element of each header block. A Handler indicates that it would process specific header blocks through this association. The following method initializes a Handler instance for this association:

```
package javax.xml.rpc.handler;
public interface Handler {
    void init(HandlerInfo config);
    // ...
}
```

A SOAP message Handler instance gets access to the SOAP Actor roles (set for the HandlerChain instance for this SOAP node) through the SOAPMessageContext. getRoles method. A Handler instance uses this information about the SOAP Actor roles to process the SOAP header blocks. Note that the SOAP actor roles cannot be changed during the processing of SOAP message through the HandlerChain.

A HandlerChain (on the SOAP receiver node) performs the following steps during the SOAP message processing. Refer to the SOAP specification [4] for the normative details on the SOAP message processing model.:

- Identify the set of SOAP actor roles in which this HandlerChain (at this SOAP node) is to act
- Identify all header blocks targeted at this node that are mandatory
- If one or more of the header blocks identified in the preceding step are not understood by this node then generate a single SOAP MustUnderstand fault. If such a fault is generated, any further processing is not done. The MustUnderstand fault is propagated to the client in both cases where MustUnderstand fault is generated on either the server side or client side as part of SOAP message processing.
- Process all header blocks targeted at the node by invoking its configured chain of handlers.
- If processing is unsuccessful, exactly one SOAP fault is generated by this HandlerChain either by handlers or JAX-RPC runtime system. This SOAP fault is propagated to the client instead of the response SOAP message.

Handler

A HandlerChain delegates processing of the SOAP message to its configured chain of handlers.

The handleRequest, handleResponse and handleFault methods for a SOAP message handler get access to the SOAPMessage from the SOAPMessageContext. The implementation of these methods can modify the SOAPMessage including the headers and body elements.

The `handleRequest` method performs one of the following steps after performing handler specific processing of the request SOAP message:

- Return `true` to indicate continued processing of the request handler chain. The `HandlerChain` takes the responsibility of invoking the next entity. The next entity may be the next handler in the `HandlerChain` or if this handler is the last handler in the chain, the next entity is the target service endpoint. The mechanism for dispatch or invocation of the target service endpoint depends on whether the request `HandlerChain` is on the client side or service endpoint side.
- Return `false` to indicate blocking of the request handler chain. In this case, further processing of the request handler chain is blocked and the target service endpoint is not dispatched. The JAX-RPC runtime system takes the responsibility of invoking the response handler chain next with the appropriate `SOAPMessageContext`. The `Handler` implementation class has the responsibility of setting the response SOAP message in the `handleRequest` method and perform additional processing in the `handleResponse` method. In the default processing model, the response handler chain starts processing from the same `Handler` instance (that returned `false`) and goes backward in the execution sequence.
- Throw the `javax.xml.rpc.soap.SOAPFaultException` to indicate a SOAP fault. The `Handler` implementation class has the responsibility of setting the SOAP fault in the SOAP message in either `handleRequest` and/or `handleFault` method. If `SOAPFaultException` is thrown by a server-side request handler's `handleRequest` method, the `HandlerChain` terminates the further processing of the request handlers in this handler chain and invokes the `handleFault` method on the `HandlerChain` with the SOAP message context. Next, the `HandlerChain` invokes the `handleFault` method on handlers registered in the handler chain, beginning with the `Handler` instance that threw the exception and going backward in execution. The client-side request handler's `handleRequest` method should not throw the `SOAPFault-Exception`. Refer to the SOAP specification for details on the various SOAP `faultcode` values and corresponding specification.
- Throw the `JAXRPCException` or any other `RuntimeException` for any handler specific runtime error. If `JAXRPCException` is thrown by a `handleRequest` method, the `HandlerChain` terminates the further processing of this handler chain. On the server side, the `HandlerChain` generates a SOAP fault that indicates that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to a runtime error during the processing of the message. Refer to the SOAP specification for details on the various SOAP `faultcode` values. On the client side, the `JAXRPCException` or runtime exception is propagated to the client code as a `RemoteException` or its subtype.

The `handleResponse` method performs the processing of the SOAP response message. It does one of the following steps after performing its handler specific processing of the SOAP message:

- Return `true` to indicate continued processing of the response handler chain. The `HandlerChain` invokes the `handleResponse` method on the next `Handler` in the handler chain.
- Return `false` to indicate blocking of the response handler chain. In this case, no other response handlers in the handler chain are invoked. On the service endpoint side, this may be useful if response handler chooses to issue a response directly without requiring other response handlers to be invoked.
- Throw the `JAXRPCException` or any other `RuntimeException` for any handler specific runtime error. If `JAXRPCException` is thrown by the `handleResponse` method, the `HandlerChain` terminates the further processing of this handler chain. On the server side, the `HandlerChain` generates a SOAP fault that indicates that the message could not be processed for reasons not directly attributable to the contents of

the message itself but rather to a runtime error during the processing of the message. On the client side, the `JAXRPCException` or runtime exception is propagated to the client code as a `RemoteException` or its subtype.

The `handleFault` method performs the SOAP fault related processing. The JAX-RPC runtime system should invoke the `handleFault` method if a SOAP fault needs to be processed by either client-side or server-side handlers. The `handleFault` method does one of the following steps after performing handler specific processing of the SOAP fault:

- Return `true` to indicate continued processing of the fault handlers in the handler chain. The `HandlerChain` invokes the `handleFault` method on the next `Handler` in the handler chain.
- Return `false` to indicate blocking of the fault processing in the handler chain. In this case, no other handlers in the handler chain are invoked. The JAX-RPC runtime system takes the further responsibility of processing the SOAP message.
- Throw `JAXRPCException` or any other `RuntimeException` for any handler specific runtime error. If `JAXRPCException` is thrown by the `handleFault` method, the `HandlerChain` terminates the further processing of this handler chain. On the server side, the `HandlerChain` generates a SOAP fault that indicates that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to a runtime error during the processing of the message. On the client side, the `JAXRPCException` or runtime exception is propagated to the client code as a `RemoteException` or its subtype.

Please note that when a JAXRPCException or RuntimeException raised on the server is converted to a SOAP fault for the purpose of being transmitted to the client, there are no guarantees that any of the information it contains will be preserved.

Example

The following shows an example of the SOAP fault processing. In this case, the request handler `H2` on the server side throws a `SOAPFaultException` in the `handleRequest` method:

- `H1.handleRequest`
- `H2.handleRequest`->throws `SOAPFaultException`
- `H2.handleFault`
- `H1.handleFault`

Example: SOAP Message Handler

The following shows an example of a SOAP message handler.

```
package com.example;
public class MySOAPMessageHandler
          extends javax.xml.rpc.handler.GenericHandler {
   public MySOAPMessageHandler() { ... }

   public boolean handleRequest(MessageContext context) {
      try {
          SOAPMessageContext smc = (SOAPMessageContext)context;
          SOAPMessage msg = smc.getMessage();
          SOAPPart sp = msg.getSOAPPart();
          SOAPEnvelope se = sp.getEnvelope();
          SOAPHeader sh = se.getHeader();
          // Process one or more header blocks
          // ...
          // Next step based on the processing model for this
```

```
            // handler
         }
         catch(Exception ex) {
            // throw exception
         }
      }
      // Other methods: handleResponse, handleFault init, destroy
}
```

12.3 Configuration

This section specifies Java APIs for the programmatic configuration of SOAP message handlers.

12.3.1 Handler Configuration APIs

A developer performs the programmatic configuration of handlers using the `javax.xml.rpc.handler.HandlerRegistry` interface.

The `Service` interface provides access to the `HandlerRegistry` instance as shown in the following code snippet:

```
package javax.xml.rpc;
public interface Service {
   HandlerRegistry getHandlerRegistry();
   // ... Additional methods
}
```

The method `getHandlerRegistry` returns the `HandlerRegistry` instance for this `Service` instance.

The following code snippet shows the `HandlerRegistry` interface:

```
package javax.xml.rpc.handler;
public interface HandlerRegistry extends java.io.Serializable {
   java.util.List getHandlerChain(QName portName);
   void setHandlerChain(QName portName, java.util.List chain);
   // ...
}
```

A JAX-RPC runtime system is required to provide implementation class for the `HandlerRegistry` interface.

A handler chain is registered per service endpoint, as indicated by the qualified name of a port. The `getHandlerChain` returns the handler chain for the specified service endpoint. The returned handler chain is configured using the `java.util.List` interface. Each element in this list is required to be of the Java type `javax.xml.rpc.handler.HandlerInfo`. The programmatic registration of `Handler` should be performed prior to the runtime creation of an endpoint proxy or `Call` object using the `javax.xml.rpc.Service` methods.

12.3.2 Deployment Model

The JAX-RPC specification does not specify the standard deployment and packaging model for the SOAP message handlers. This model would be defined as part of the J2EE 1.4 specifications [3].

12.4 Handler Lifecycle

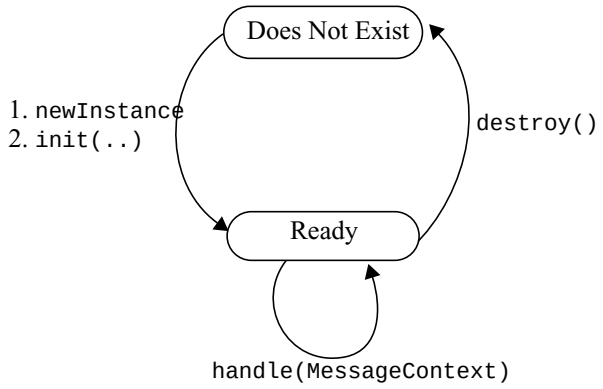
A SOAP message handler is required to be implemented as a stateless instance. A SOAP message handler must not maintain any SOAP message related state in its instance variables across multiple invocations of the `handle` method. In terms of the SOAP message processing functionality, the JAX-RPC runtime system considers all instances of a specific handler class as equivalent. The JAX-RPC runtime system may choose any ready instance of a `Handler` class to invoke the handle methods. This makes the `Handler` instances as capable of pooling by the JAX-RPC runtime system per deployed endpoint component. However, JAX-RPC runtime system is not required to support pooling of `Handler` instances.

A SOAP message handler is required to implement the following lifecycle methods of the `javax.xml.rpc.handler.Handler` interface:

```
package javax.xml.rpc.handler;
public interface Handler {
    void init(HandlerInfo config);
    void destroy();
    // ...
}
```

The JAX-RPC runtime system is required to manage the lifecycle of `Handler` instances by invoking the `init` and `destroy` methods.

The following state transition diagram shows the lifecycle of a `Handler` instance:



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.

After a `Handler` is instantiated, the JAX-RPC runtime system is required to initialize the `Handler` before this `Handler` instance can start processing the SOAP messages. The JAX-RPC runtime system invokes the `init` method to enable the `Handler` instance to initialize itself. The `init` method passes the handler configuration as a `HandlerInfo` instance. The `HandlerInfo` is used to configure the `Handler` (for example: setup access to an external resource or service) during the initialization. The `init` method also associates a `Handler` instance (using the `HandlerInfo`) with zero or more header blocks using the corresponding `QNames`.

In the `init` method, the `Handler` class may get access to any resources (for example; access to an authentication service, logging service) and maintain these as part of its instance variables. Note that these instance variables must not have any state specific to the SOAP message processing performed in the various `handle` method.

Once the `Handler` instance is created and initialized (and is in the `Ready` state), the JAX-RPC runtime system may invoke the different `handle` method multiple times.

The JAX-RPC runtime system is required to invoke the `destroy` method when the runtime system determines that the `Handler` object is no longer needed. For example, the JAX-RPC runtime may remove a `Handler` object when the runtime system (in an managed operational environment) is shutting down or managing memory resources.

The `destroy` method indicates the end of lifecycle for a `Handler` instance. The `Handler` instance releases its resources and performs cleanup in the implementation of the `destroy` method. After successful invocation of the `destroy` method, the `Handler` object is available for the garbage collection.

A `RuntimeException` (other than `SOAPFaultException`) thrown from any method of the `Handler` results in the `destroy` method being invoked and transition to the "`Does Not Exist`" state.

13 JAX-RPC Runtime Services

This chapter specifies requirements for security and session management for the JAX-RPC runtime system implementations.

13.1 Security

The JAX-RPC specification requires that a service client be able to authenticate to the service endpoint. Note that this section specifies requirements for different authentication mechanisms when HTTP or HTTP/S is used as the underlying transport.

13.1.1 HTTP Basic Authentication

The JAX-RPC specification requires support for HTTP Basic Authentication for protocol bindings over the HTTP transport.

The HTTP Basic Authentication uses user name and password for authenticating a service client. The `javax.xml.rpc.Stub` and `javax.xml.rpc.Call` interfaces are required to support "`javax.xml.rpc.security.auth.username`" and "`javax.xml.rpc.security.auth.password`" properties for the HTTP Basic Authentication. For convenience, the properties above can be referenced using resp. the `USERNAME_PROPERTY` and `PASSWORD_PROPERTY` constants defined by the `Stub` and `Call` interfaces.

During invocation of a remote method, HTTP server (considered part of a JAX-RPC server side runtime system) uses user name and password to authenticate the service client. The authentication is performed in a specific security realm.

Example

The following shows an illustrative example of a service client accessing a service endpoint bound to SOAP over HTTP. The user name and password passed in the `getStockQuoteProvider` method are used for authenticating the service client using the HTTP Basic Authentication:

```
StockQuoteService sqs = // ... Get access to the service
StockQuoteProvider sqp = sqs.getStockQuoteProviderPort(
                              "<username>", "<password>");
float quote = sqp.getLastTradePrice("ACME");
```

The following shows the same example using the properties mechanism for the configuration of a stub instance:

```
StockQuoteProvider_Stub sqp = // ... get to the stub;
sqp._setProperty(Stub.USERNAME_PROPERTY, "<username>");
sqp._setProperty(Stub.PASSWORD_PROPERTY, "<password>");
float quote = sqp.getLastTradePrice("ACME");
```

13.1.2 SSL Mutual Authentication

The JAX-RPC specification does not require support for the certificate based mutual authentication using HTTP/S (HTTP over SSL) mechanism.

13.1.3 SOAP Security Extensions

The JAX-RPC specification does not require support for the SOAP Security Extensions for digital signature [16].

13.2 Session Management

The JAX-RPC specification requires that a service client be able to participate in a session with a service endpoint.

In the JAX-RPC 1.1 version, the session management mechanisms require use of HTTP as the transport in the protocol binding. This version of the JAX-RPC specification does not specify (or require) session management using SOAP headers given that there is no standard SOAP header representation for the session information. SOAP based session management may be considered in the future versions of the JAX-RPC specification.

A JAX-RPC runtime system is required to use at least one of the following mechanisms to manage sessions:

- Cookie based mechanism: On the initial method invocation on a service endpoint, the server side JAX-RPC runtime system sends a cookie to the service client to initiate a new session. If service client wants to participate in this session, the client side JAX-RPC runtime system then sends the cookie for each subsequent method invocation on this service endpoint. The cookie associates subsequent method invocations from the service client with the same session.
- URL rewriting involves adding session related identifier to a URL. This rewritten URL is used by the server-side JAX-RPC runtime to associate RPC invocations to the service endpoint with a session. The URL that is rewritten depends on the protocol binding in use.
- SSL session may be used to associate multiple RPC invocations on a service endpoint as part of a single session.

A session (in JAX-RPC) is initiated by the server-side JAX-RPC runtime system. The server-side JAX-RPC runtime system may use `HTTPSession` (defined in the Servlet specification [3]) to implement support for the HTTP session management.

A service client uses the `javax.xml.rpc.session.maintain` property (set using the `Stub` or `Call` interfaces) to indicate whether or not it wants to participate in a session with a service endpoint. By default, this property is `false`, so the client does not participate in a session by default. However, by setting `session.maintain` property to

`true`, the client indicates that it wants to join the session initiated by the server. In the cookie case, the client runtime system accepts the cookie and returns the session tracking information to the server, thereby joining the session.

The client code by setting the `session.maintain` property assumes that it would participate in a session if one is initiated by the server. The actual session management happens transparent to the client code in the client-side runtime system.

Note that JAX-RPC specification does not require session management as part of the interoperability requirements (specified in the chapter 14).

14 Interoperability

The XML based RPC services are typically defined, deployed and used on heterogeneous environments and platforms. Interoperability of various JAX-RPC runtime system implementations with other vendor products (including Microsoft .Net) is an extremely important goal for the JAX-RPC specification.

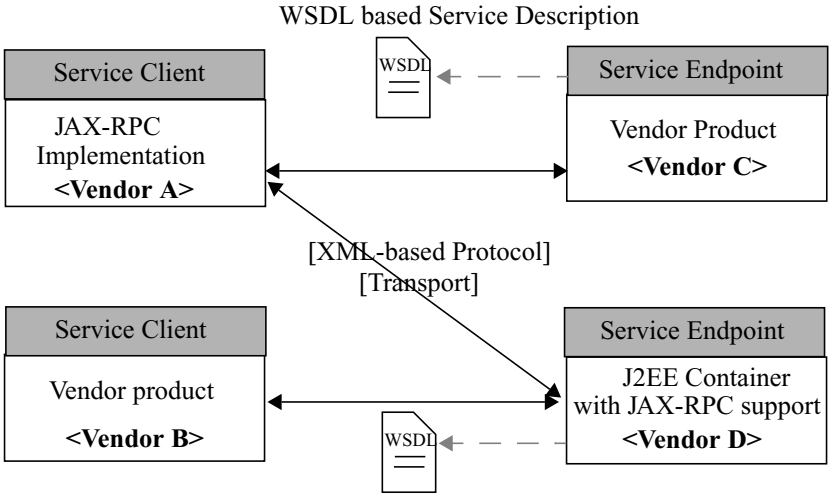
This chapter specifies scenarios and requirements for the interoperability of a vendor's JAX-RPC runtime system implementation with other JAX-RPC implementations and XML based RPC products.

14.1 Interoperability Scenario

The following diagram shows a typical heterogeneous environment for XML based RPC services. This environment is used to describe various interoperability scenarios later in this chapter.

An example service endpoint is defined and deployed separately on two products from vendors C and D. The vendor D has a standard J2EE compatible application server that supports a JAX-RPC runtime system implementation. The vendor C has a vendor specific product that supports XML based RPC using SOAP 1.1 and WSDL 1.1.

Service clients are developed and deployed each on vendor A and B products. The vendor A provides a JAX-RPC runtime system implementation based on either the J2SE platform or a J2EE compatible application server. The vendor B provides an XML based RPC product that supports SOAP 1.1 and WSDL 1.1.



The following table illustrates interoperability scenario matrix based on the above heterogeneous environment. In this matrix, a JAX-RPC implementation includes a JAX-RPC runtime system and deployment tools.

TABLE 14-1 Interoperability Matrix

Service Client Environment	Service Provider Environment	JAX-RPC 1.1 Scope
JAX-RPC implementation based on either J2SE or J2EE compatible container	Vendor specific XML based RPC product with SOAP 1.1 and WSDL 1.1 support	Scenario addressed by the JAX-RPC Interoperability requirements
JAX-RPC implementation based on either J2SE or J2EE compatible container	JAX-RPC implementation based on either J2SE or J2EE compatible container	Scenario addressed by the JAX-RPC Interoperability requirements
Vendor specific XML based RPC product with SOAP 1.1 and WSDL 1.1 support	JAX-RPC implementation based on either J2SE or J2EE compatible container	Scenario addressed by the JAX-RPC Interoperability requirements
Vendor specific XML based RPC product with SOAP 1.1 and WSDL 1.1 support	Vendor specific XML based RPC product with SOAP 1.1 and WSDL 1.1 support	Out of Scope

14.2 Interoperability Goals

The JAX-RPC interoperability requirements address the following goals:

- To support an out-of-box interoperability between various JAX-RPC compatible implementations
- To specify requirements that are testable for interoperability
- To leverage interoperability work done by standards bodies (example: W3C, WS-I) and key interoperability initiatives (including the SOAPBuilders [19] community)
- To specify interoperability requirements without any additional requirements on the JAX-RPC service client and service endpoint model. Service clients and service developers should not be exposed to the interoperability requirements.

14.3 Interoperability Requirements

The following section specifies interoperability requirements for the scenarios identified as within the scope of the JAX-RPC 1.0 specification.

Please note that the JAX-RPC 1.1 specification added several interoperability requirements based on the work done in the Web Service Interoperability Organization. These requirements are detailed in section 14.4, “Interoperability Requirements: WS-I Basic Profile Version 1.0”.

14.3.1 SOAP based Interoperability

An interoperable JAX-RPC implementation must be able to support SOAP 1.1 with attachments [6] as the underlying protocol. A vendor's implementation of the SOAP 1.1 protocol must be based on the standard SOAP specification [4]. A SOAP based JAX-RPC runtime system implementation must interoperate with other JAX-RPC implementations and vendor specific products that support SOAP 1.1 with attachments.

The JAX-RPC specification does not specify interoperability requirements for any protocol other than the SOAP 1.1 with attachments. However, the JAX-RPC specification allows a JAX-RPC runtime system implementation to support protocol bindings other than SOAP. Note that any non-SOAP based JAX-RPC implementation is not interoperable by definition based on the JAX-RPC interoperability requirements.

14.3.2 SOAP Encoding and XML Schema Support

The JAX-RPC specification requires support for the following representation for remote call and response in a SOAP message:

- Encoded representation using the SOAP 1.1 encoding: The rules and format of serialization for the XML data types are based on the SOAP 1.1 encoding [4]. A JAX-RPC runtime system implementation is required to support the SOAP 1.1 encoding. Interoperability requirements for any other encodings are outside the scope of the JAX-RPC specification.

Refer to the R03 for the XML Schema data types and Java types that an interoperable JAX-RPC runtime system implementation is required to support.

The sections 4.2 and 5.3 specify the standard mapping between the XML data types and Java types. A JAX-RPC implementation is required to support the standard type mapping between XML data and Java types.

14.3.3 Transport

A JAX-RPC runtime system implementation must be able to support HTTP 1.1 as the underlying transport. The required SOAP binding to HTTP 1.1 is specified in the SOAP specification [4].

The interoperability requirements for any other transport are outside the scope of the JAX-RPC specification. Note that a future version of the JAX-RPC specification may consider specifying interoperability requirements for additional transports. This support depends on the standardization (in the SOAP W3C working group) of the SOAP binding framework for different transports.

14.3.4 WSDL Requirements

For each deployed JAX-RPC service endpoint, a JAX-RPC implementation must be able to export an equivalent WSDL 1.1 based service description. The mapping to a WSDL based service description should follow the standard Java to WSDL mapping specified in the section 5.5, "Java to WSDL Mapping".

An exported WSDL based service description must not include any vendor specific extensibility elements. Note that the use of such vendor specific extensibility elements may constrain interoperability.

An interoperable JAX-RPC implementation must be capable of importing a WSDL 1.1 based service description. The mapping of an imported WSDL service description to the equivalent Java representation is required to follow the standard WSDL to Java mapping as specified in the section 4.3, “WSDL to Java Mapping”. Note that the use of a specific service client programming model (whether generated stub or dynamic proxy based) must not impact the interoperability.

14.3.5 Processing of SOAP Headers

An interoperable JAX-RPC implementation must be able to support the standard processing model for SOAP headers as specified in the SOAP specification [5]. Note that the SOAP specification does not define any standard representation for the SOAP headers.

An explicit goal of the JAX-RPC specification is not to define any SOAP header representation for transaction, security or session related information. A goal of the JAX-RPC specification is to leverage work done in other standardization groups for this aspect. An important point to note is that any JAX-RPC specific definition of SOAP headers or session related information is against the design goal of achieving SOAP based interoperability with heterogeneous environments.

14.3.6 Mapping of Remote Exceptions

A remote method in a service endpoint interface is required to throw the standard `java.rmi.RemoteException`. A `RemoteException` or its subclass maps to a fault in the corresponding SOAP message. Refer to the SOAP specification [4] for requirements on the representation of a fault in a SOAP message.

A SOAP fault includes the `faultcode` sub-element. The `faultcode` must be present in a SOAP fault and the `faultcode` value must be a qualified name. A JAX-RPC implementation must map a `java.rmi.RemoteException` or its subclass to a standard `faultcode`. This enables JAX-RPC implementations to interoperate in terms of handling of the remote exceptions.

TABLE 14-2 Mapping between RemoteException and SOAP Faults

SOAP faultcode Value	Error Description	RemoteException Mapping
`soap-env:Server`	Server cannot handle the message because of some temporary condition. Example: out of memory condition	`java.rmi.ServerException`
`soap-env:DataEncodingUnknown`	Parameters are encoded in a data encoding unknown to the server.	`java.rmi.MarshalException`

14.3.7 Security

JAX-RPC specification does not specify requirements for the security interoperability for the JAX-RPC implementations. This will be specified in a future version of the JAX-RPC specification.

14.3.8 Transaction

Transaction interoperability is an optional feature in the JAX-RPC specification. A JAX-RPC implementation is not required to implement support for the transaction context propagation.

14.4 Interoperability Requirements: WS-I Basic Profile Version 1.0

The JAX-RPC 1.1 specification adds several interoperability requirements based on the work done in the Web Service Interoperability Organization.

The WS-I Basic Profile Version 1.0 ([22], henceforth "BP") establishes a set of requirements that instances and consumers of Web Services must satisfy in order to be declared *conformant*.

In the rest of this section, the term "conformant" must be assumed to mean "conformant with the WS-I Basic Profile Version 1.0". All-uppercase words, such as DESCRIPTION, MESSAGE, RECEIVER, etc. are used to refer to terms defined in the Basic Profile specification.

14.4.1 Requirements On Java-to-WSDL Tools

The Java-to-WSDL tools provided by a JAX-RPC implementation must be able to produce a WSDL 1.1 document containing a conformant DESCRIPTION of the service when starting from a JAX-RPC service endpoint interface that uses only types for which chapter 5 ("Java to XML/WSDL Mapping") provides a standard mapping.

The following BP requirements apply to Java-to-WSDL tools and the DESCRIPTIONs they produce under the conditions listed in the previous paragraph:

- (section 3.3) 0002, 0003;
- (section 5.1) 2028, 2029, 2001, 2002, 2003, 2004, 4003, 2005, 2007, 2022, 2023, 4003, 2025, 2026;
- (section 5.2) 2101, 2105, 2110, 2111, 2112;
- (section 5.3) 2201, 2210, 2202, 2203, 2207, 2204, 2208, 2205, 2209, 2206;
- (section 5.4) 2301, 2302, 2303, 2304, 2305, 2306;
- (section 5.5) 2401;
- (section 5.6) 2700, 2701, 2702, 2705, 2706, 2710, 2711, 2716, 2717, 2726, 2718, 2719, 2740, 2741, 2720, 2721, 2754, 2722, 2723;
- (section 5.7) 2800, 2801;
- (section 7.1) 5001.

14.4.2 Requirements on WSDL-to-Java Tools

The WSDL-to-Java tools provided by a JAX-RPC implementation must be able to process WSDL 1.1 documents that contain conformant DESCRIPTIONs and that use any XML Schema feature with the exception of those explicitely excluded in Appendix 18.

The following BP requirements apply to WSDL-to-Java tools as CONSUMERs of a conformant DESCRIPTION under the conditions listed in the previous paragraph.

- (section 5.1) 4002, 2008, 2020, 2021, 2024, 2027;
- (section 5.2) 2114;
- (section 5.6) 2707, 2709, 2713, 2728, 2747, 2748.

14.4.3 Requirements On JAX-RPC Runtime Systems

Since JAX-RPC contains an extensible SOAP Message Handler framework, it is impossible to guarantee that any JAX-RPC client application or endpoint will *always* produce or consume messages which are conformant. Even if, for the purpose of establishing interoperability requirements, we removed message handlers from JAX-RPC, there would still be several other possibilities offered by the platform or by containers to intercept and alter messages, ranging from protocol handlers in J2SE to servlet filters in J2EE.

The intent of this section is therefore to ensure that, provided that neither user-written code nor container-specific functionality that is outside the scope of JAX-RPC alters incoming or outgoing SOAP messages in ways that make them non-conformant or deliberately produces a non conformant SOAP message, JAX-RPC 1.1-based applications will indeed behave in a conformant way and produce and accept conformant messages.

The runtime system of a JAX-RPC 1.1 implementation (henceforth “runtime”) must satisfy the following BP requirements when operating as either a consumer or a provider of an INSTANCE whose DESCRIPTION is conformant. The requirements are grouped in three different categories according to the operations of the runtime that they affect.

- When creating a javax.xml.soap.SOAPMessage starting from a Java method invocation (on the client) or the result of one (on the server) and before passing it to the appropriate handler chain for processing:
  - (section 3.4) 0004, 0005, 0006;
  - (section 4.1) 1000, 1001, 1004, 1031, 1005, 1006, 1007, 1008, 1009, 1011, 1013, 1014;
  - (section 5.2) 2113;
  - (section 5.6) 2712, 2729, 2735, 2737.
- When serializing a javax.xml.soap.SOAPMessage into XML, after the message has been processed by the appropriate handler chain, and sending it over the wire (this includes the degenerate case of a server handling a one-way request):
  - (section 3.4) 0007;
  - (section 4.1) 1012;
  - (section 4.3) 1140, 1141, 1132, 1108, 1109, 1117, 1118, 1124, 1111, 1112, 1116, 1126;
  - (section 5.6) 2714, 2715, 2727, 2744, 2745.

- When receiving a SOAP message over the wire and deserializing it into a javax.xml.soap.SOAPMessage object:
  - (section 4.1) 4001, 1002, 1003, 1016, 1010, 1015, 1017;
  - (section 4.2) 1025, 1027, 1028, 1029, 1030;
  - (section 4.3) 1107, 1119, 1125, 1113, 1114, 1115, 1130;
  - (section 5.6) 2750, 2742, 2743, 2724, 2725, 2739, 2753, 2751, 2752, 2746.

15 Extensible Type Mapping

This chapter specifies APIs for supporting an extensible type mapping framework in a JAX-RPC implementation.

The JAX-RPC specification specifies a standard mapping between XML data types and Java types. The standard type mapping supports a set of XML data types defined in the SOAP 1.1 encoding and XML Schema specification. Refer to the "Appendix: XML Schema Support" for the supported set of XML data types. The standard type mapping also specifies the XML mapping for the set of Java types supported by the JAX-RPC. Refer to the section 4.2, "XML to Java Type Mapping" and section 5.3, "Java to XML Type Mapping" for more details on the standard type mapping.

A JAX-RPC implementation may need to support mapping between XML data types and Java types beyond that addressed by the standard type mapping specification.

The JAX-RPC specification specifies APIs to support an extensible type mapping framework. These APIs enable development of pluggable serializers and deserializers to support an extensible mapping between any Java type and XML data type. The pluggable serializers and deserializers may be packaged as part of a JAX-RPC implementation or may be provided by tools vendors, service developers and service clients.

15.1 Design Goals

The JAX-RPC specification identifies the following design goals for the extensible type mapping framework:

- The type mapping framework should enable the development of pluggable serializers and deserializers using different XML processing mechanisms and representations. Note that the performance of a pluggable serializer or deserializer depends on the choice of an XML processing mechanism and representation. For example, a DOM based deserializer is typically less performance efficient than a SAX or streaming parser based deserializer.
- The type mapping framework should not enforce a specific XML processing mechanism or representation on a JAX-RPC implementation. A JAX-RPC implementation should be allowed to use a performance efficient XML processing mechanism.
- The application programming model for development and pluggability (into the type mapping framework) of serializers and deserializers should be simple. The type mapping framework should enable a JAX-RPC implementation vendor, application developer and tools vendor to develop pluggable serializers and deserializers.

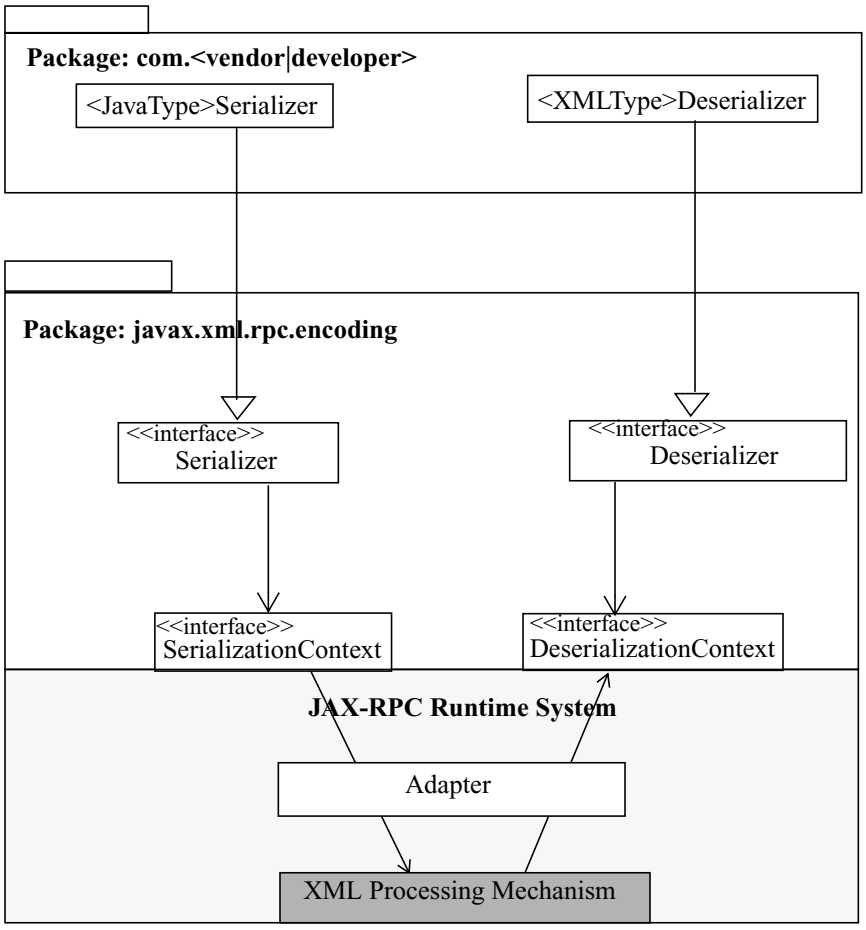
Note that the portability of pluggable serializers and deserializers across various JAX-RPC implementations is not addressed in the JAX-RPC 1.1 version. The reasons are as follows:

- Existing SOAP implementations typically use either SAX based or streaming pull parser for the XML processing. The primary motivation is better performance. It will be difficult (in terms of the time to market for various JAX-RPC implementations) to enforce any standard portable API for XML processing and representation on all JAX-RPC implementations.
- Portability of XML processing mechanism specific serializers and deserializers to a JAX-RPC implementation that uses a different XML processing mechanism is non-trivial to implement. For example, a DOM based serializer may not be easily pluggable on a JAX-RPC implementation that uses a streaming pull parser.

Support for portable serializers and deserializers will be addressed in the next version of the JAX-RPC specification.

15.2 Type Mapping Framework

The `javax.xml.rpc.encoding` package defines a set of Java interfaces/classes for the type mapping framework. The following diagram shows the JAX-RPC type mapping framework.



The `Serializer` and `Deserializer` interfaces enable development of pluggable serializers and deserializers that support extensible mapping between XML data types and Java types.

A `Serializer` serializes a Java object to an XML representation based on the type mapping defined between the corresponding Java type and the XML data type. Examples are serializers defined for Java arrays and classes in the standard Java Collection framework. A serializer typically uses the XML schema definition for an XML schema instance to generate the corresponding XML representation.

A `Deserializer` deserializes an XML element or schema instance to a Java object. The deserialization is based on the type mapping defined between an XML data type and the corresponding Java type. A deserializer typically uses the XML schema fragment and the corresponding XML schema instance to map to the corresponding Java object.

The `Serializer` and `Deserializer` are the base interfaces defined to be independent of any XML processing mechanism and XML representation. A JAX-RPC implementation must extend these two interfaces to support development of serializers and deserializers based on a specific XML processing mechanism. Examples of various XML processing mechanisms that may be supported by implementation specific serializers and deserializers are:

- DOM based XML processing
- SAX based XML processing
- Streaming pull parser
- Object based input and output streams

A JAX-RPC runtime system is also required to implement both `SerializationContext` and `DeserializationContext` interfaces. These two interfaces provide XML processing and JAX-RPC runtime system related context to the serializers and deserializers during serialization and deserialization.

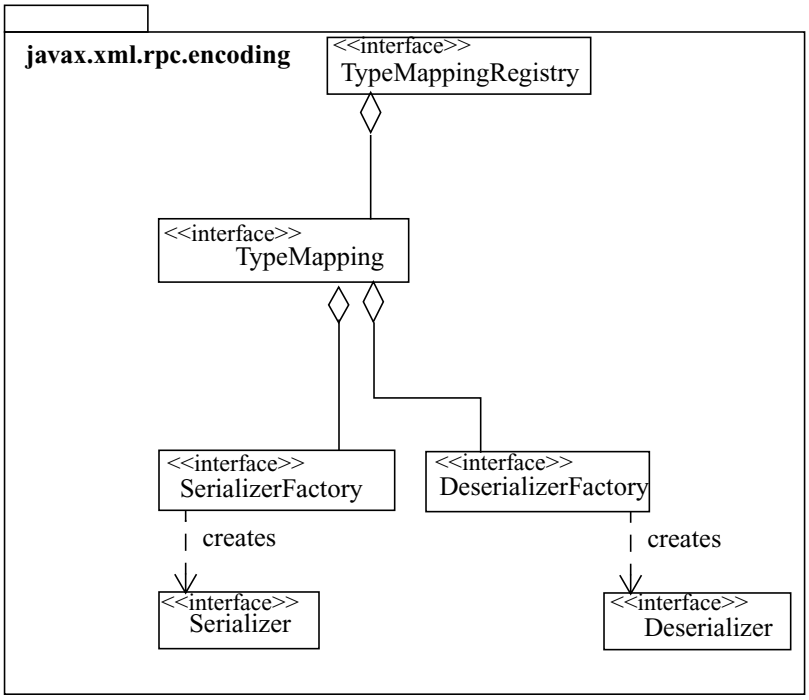
Note that the type mapping framework does not require an implementation of a JAX-RPC runtime system to use a specific XML processing mechanism. For example, a JAX-RPC implementation may use a streaming pull parser for performance efficient XML processing. DOM or SAX based deserializers may still be developed and plugged into the extensible type mapping framework provided by this JAX-RPC implementation. To support such DOM or SAX based deserializers, this JAX-RPC implementation may include adapters from the underlying XML processing mechanism (streaming pull parser in this case) to both SAX and DOM representations.

JAX-RPC specification does not require that a JAX-RPC implementation support pluggablility of serializers and deserializers of multiple XML processing mechanism types. A JAX-RPC implementation may choose to support any number but must support at least one of the XML processing mechanism types for serializers and deserializers.

15.3 API Specification

This section specifies the JAX-RPC APIs for the extensible type mapping framework. The following diagram shows the `javax.xml.rpc.encoding` package:

FIGURE 15-1 Class diagram for the Type Mapping framework



15.3.1 TypeMappingRegistry

The interface `javax.xml.rpc.encoding.TypeMappingRegistry` defines a registry for the type mappings for various encoding styles.

The following code snippet shows the `TypeMappingRegistry` interface:

```
package javax.xml.rpc.encoding;
public interface TypeMappingRegistry extends java.io.Serializable {
   void register(String encodingStyleURI, TypeMapping mapping);
   void registerDefault(TypeMapping mapping);

   TypeMapping getDefaultTypeMapping();
   TypeMapping getTypeMapping(String encodingStyleURI);
   TypeMapping unregisterTypeMapping(String encodingStyleURI);
   boolean removeTypeMapping(TypeMapping);

   TypeMapping createTypeMapping();

   String[] getRegisteredEncodingStyleURIs();
   void clear();
}
```

A `TypeMapping` instance is associated with [1-n] `encodingStyleURIs`. An `encodingStyleURI` is associated with [0-1] `TypeMapping` instance.

The method `register(String, TypeMapping)` registers a `TypeMapping` instance with the `TypeMappingRegistry`. This method replaces any existing registered `TypeMapping` instance for the specified `encodingStyleURI`. The parameter `encodingStyleURI` must be a namespace URI that represent a single encoding style. An example is the SOAP 1.1 encoding represented by the namespace URI `http://schemas.xmlsoap.org/soap/encoding/`. The `""` (zero-length URI) is used to indicate that there are no claims about the encoding style.

The method `unregisterTypeMapping` unregisters a `TypeMapping` instance from the specified `encodingStyleURI`.

The method `registerDefault(TypeMapping)` registers the default `TypeMapping` for all encoding styles supported by the `TypeMappingRegistry`. Successive invocations of the `registerDefault` method replace any existing registered default `TypeMapping` instance. A default `TypeMapping` should include serializers and deserializers that are independent of and usable with any encoding styles.

The JAX-RPC specification does not require that a default `TypeMapping` be registered in the `TypeMappingRegistry`. If the default `TypeMapping` is registered, any other `TypeMapping` instances registered through the `TypeMappingRegistry.register` method (for a set of `encodingStyle` URIs) is required to override the default `TypeMapping`.

The method `getRegisteredEncodingStyleURIs` returns a list of registered `encodingStyle` URIs in this `TypeMappingRegistry` instance.

The method `getTypeMapping` returns the registered `TypeMapping` for the specified `encodingStyleURI`. If there is no registered `TypeMapping` for the specified `encodingStyleURI`, this method returns `null`.

The method `clear` removes all registered `TypeMapping` instances from this `TypeMappingRegistry`.

The method `createTypeMapping` creates an empty `TypeMapping` object. Refer to the `TypeMapping` interface for more details.

15.3.1.1 Configuration of TypeMappingRegistry

In the service client programming model, the `javax.xml.rpc.Service` interface supports the configuration of a `TypeMappingRegistry`.

```
package javax.xml.rpc;
public interface Service {
   TypeMappingRegistry getTypeMappingRegistry();
   // ...
}
```

The method `getTypeMappingRegistry` returns the `TypeMappingRegistry` for this service object. The returned `TypeMappingRegistry` instance is required to be pre-configured for supporting the standard type mapping between XML and Java types as specified in the sections 4.2 and 5.3. This getter method is required to throw the `java.lang.UnsupportedOperationException` if the `Service` class does not support configuration of a `TypeMappingRegistry`.

Dynamic proxies, `Call` objects or instances of generated stub classes created from the service class use the configured `TypeMappingRegistry` to access the extensible type mapping framework. For example, a dynamic proxy uses the configured `TypeMappingRegistry` to perform serialization and deserialization of the Java and XML data types.

Note that a generated stub class may embed the type mapping support in the generated code. In this case, a generated stub class is not required to use the configured type mapping registry.

15.3.2 TypeMapping

The `javax.xml.rpc.encoding.TypeMapping` is the base interface for the representation of type mappings. A `TypeMapping` supports a set of encoding styles. The method `getSupportedEncodings` returns the `encodingStyle` URIs (as `String[]`) supported by a `TypeMapping` instance. The method `setSupportedEncodings` sets the supported `encodingStyle` URIs for a `TypeMapping` instance.

The following code snippet shows the `TypeMapping` interface:

```
package javax.xml.rpc.encoding;
public interface TypeMapping {
   void setSupportedEncodings(String[] encodingStyleURIs);
   String[] getSupportedEncodings();

   boolean isRegistered(Class javaType, QName xmlType);
   void register(Class javaType, QName xmlType,
                      SerializerFactory sf,
                      DeserializerFactory dsf);
   SerializerFactory getSerializer(Class javaType,
                                       QName xmlType);
   DeserializerFactory getDeserializer(Class javaType,
                                          QName xmlType);
   void removeSerializer(Class javaType, QName xmlType);
   void removeDeserializer(Class javaType, QName xmlType);
}
```

The JAX-RPC specification allows serializers and deserializers to be developed independent of any specific encoding style. Such serializers and deserializers are usable across multiple encoding styles. A `TypeMapping` that contains only such serializers and deserializers is required to pass `null` to the `setSupportedEncodings` method and return `null` from the `getSupportedEncodings method.`

For its supported encoding styles, a `TypeMapping` instance maintains a set of tuples of the type {Java type, `SerializerFactory`, `DeserializerFactory`, XML data type}.

The `getSerializer` and `getDeserializer` methods take both `javaType` and `xmlType` parameters. This enables serializers and deserializers to support a flexible mapping between the XML data types and Java types. For example, a deserializer can map a specific XML data type to different Java types based on the application specific configuration.

15.3.3 Serializer

The `javax.xml.rpc.encoding.Serializer` interface defines the base interface for serializers. A `Serializer` serializes a Java object to an XML representation using a specific XML processing mechanism.

A pluggable serializer for a Java type must provide serializer and serializer factory classes that implement the `Serializer` and `SerializerFactory` interfaces respectively.

The JAX-RPC specification requires that the `SerializerFactory` and `Serializer` interfaces be implemented as JavaBeans components. This enables tools to manage and configure the serializers.

The following code snippet shows the base `SerializerFactory` interface:

```
package javax.xml.rpc.encoding;
public interface SerializerFactory extends java.io.Serializable {
   Serializer getSerializerAs(String mechanismType);
   Iterator getSupportedMechanismTypes();
}
```

A `SerializerFactory` is registered with a `TypeMapping` instance as part of the type mapping registry.

The method `getSerializerAs` returns an XML processing mechanism specific serializer. The parameter `mechanismType` identifies the desired XML processing mechanism type. This method throws the `JAXRPCException` if the `SerializerFactory` does not support the desired XML processing mechanism type.

The method `getSupportedMechanismTypes` returns all XML processing mechanism types supported by this `SerializerFactory`.

The following code snippet shows the base `Serializer` interface:

```
package javax.xml.rpc.encoding;
public interface Serializer extends java.io.Serializable {
   String getMechanismType();
}
```

An XML processing mechanism specific serializer extends the base `Serializer` interface. The method `getMechanismType` returns the type of the XML processing mechanism and representation used by this serializer.

The `javax.xml.rpc.encoding.SerializationContext` interface is implemented by the JAX-RPC runtime system in an XML processing mechanism specific manner. A serializer uses the `SerializationContext` interface during the serialization to get the context information related to the XML processing mechanism and to manage information specific to serialization.

The following code snippet shows the `SerializationContext` interface:

```
package javax.xml.rpc.encoding;
public interface SerializationContext {
}
```

15.3.4 Deserializer

A deserializer converts an XML element or schema instance to a Java object. The deserialization is based on the type mapping defined between an XML data type and the corresponding Java type. The `javax.xml.rpc.encoding.Deserializer` defines the base interface for different types of deserializers.

A pluggable deserializer for an XML type must provide deserializer and deserializer factory classes that implement `Deserializer` and `DeserializerFactory` interfaces respectively. Examples of such deserializers are for complex types defined in the XML Schema specification.

JAX-RPC specification requires that the `DeserializerFactory` and `Deserializer` interfaces be implemented as JavaBeans components. This enables tools to manage and configure the deserializers.

The following code snippet shows the `DeserializerFactory` interface:

```
package javax.xml.rpc.encoding;
public interface DeserializerFactory extends java.io.Serializable {
   Deserializer getDeserializerAs(String mechanismType);
   java.util.Iterator getSupportedMechanismTypes();
}
```

A `DeserializerFactory` is registered with a `TypeMapping` instance as part of the type mapping registry.

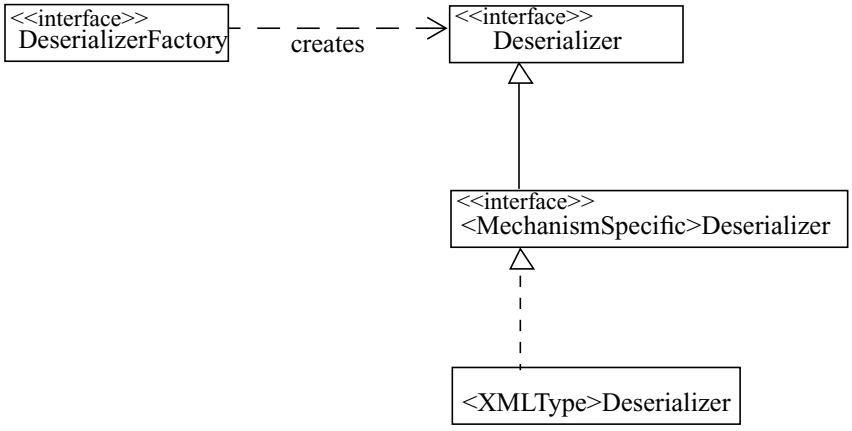
The method `getDeserializerAs` returns an XML processing mechanism specific deserializer. The parameter `mechanismType` identifies the desired XML processing mechanism type. The method `getDeserializerAs` throws the `JAXRPCException` if this `DeserializerFactory` does not support the desired XML processing mechanism type.

The method `getSupportedMechanismTypes` returns all XML processing mechanism types supported by this `DeserializerFactory`.

The following code snippet shows the `Deserializer` interface:

```
javax.xml.rpc.encoding;
public interface Deserializer extends java.io.Serializable {
   String getMechanismType();
}
```

An XML processing mechanism specific deserializer extends the base `Deserializer` interface. The method `getMechanismType` returns the type of the XML processing mechanism supported by this deserializer. Such mechanism specific deserializer may be either DOM based, SAX based, streaming parser based or stream based.



The `javax.xml.rpc.encoding.DeserializationContext` interface is implemented by the JAX-RPC runtime system in an XML processing mechanism specific manner. A deserializer uses this interface to access and maintain context information during the deserialization. The following code snippet shows the `DeserializationContext` interface:

```
package javax.xml.rpc.encoding;
public interface DeserializationContext {
}
```

15.4 Example: Serialization Framework

Refer to the "Appendix: Serialization Framework" for an illustrative implementation of the serialization framework. This API design is based on the JAX-RPC reference implementation.

16 Futures

The following features would be considered in the future versions of the JAX-RPC specification. The technical details of these features would be resolved in the expert group of the corresponding JSR. The following is a brief overview of these features:

- Support for SOAP 1.2 [5]
- Extended support for customization and toolability aspects of JAX-RPC value types. This would enable tools to customize serialization semantics and representation of the JAX-RPC value types.
- Extended support for XML schema types and Java types
- Design of portable serializers and deserializers. This will enable custom serializers to be portable across JAX-RPC implementations.
- Enhanced support for security. For example, mutual authentication using HTTP/S
- Design of portable stubs.
- Support for WSDL 1.2. This depends on the schedule of the W3C Web Services Description Working Group [21].
- Alignment with the JAXB [14] specification.

17 References

[1] Java Language Specification: http://java.sun.com/docs/books/jls/

[2] J2SE: http://java.sun.com/j2se/

[3] J2EE Specifications: http://java.sun.com/j2ee/

[4] W3C Note: SOAP 1.1: http://www.w3.org/TR/SOAP/

[5] W3C: SOAP 1.2: http://www.w3.org/TR/2002/CR-soap12-part1-20021219/ and http://www.w3.org/TR/2002/CR-soap12-part2-20021219/

[6] W3 Note: SOAP Messages with Attachments: http://www.w3.org/TR/SOAP-attachments

[7] Web Services Description Language (WSDL) 1.1: http://www.w3.org/TR/wsdl

[8] W3C Recommendation “XML Schema Part 1: Structures”: http://www.w3.org/TR/xmlschema-1/

[9] W3C Recommendation “XML Schema Part 2: Datatypes”: http://www.w3.org/TR/xmlschema-2/

[10] JSR 109: Implementing Enterprise Web Services: http://jcp.org/jsr/detail?id=109

[11] JSR for J2EE 1.4: http://jcp.org/jsr/detail?id=151

[12] JSR 110: Java APIs for WSDL http://jcp.org/jsr/detail/110.jsp

[13] JAXM specification: http://java.sun.com/xml/jaxm/

[14] JAXB specification: http://java.sun.com/xml/jaxb/

[15] RMI specification: http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html

[16] SOAP Security Extensions: Digital Signature http://www.w3.org/TR/SOAP-dsig/

[17] RMI-IIOP Specification: http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/

[18] JavaBeans Activation Framework: http://java.sun.com/products/javabeans/glasgow/jaf.html

[19] SOAPBuilders Interoperability Lab

[20] XML Information Set: http://www.w3.org/TR/2001/REC-xml-infoset-20011024/

[21] W3C Web Services Description Working Group: http://www.w3.org/2002/ws/desc

[22] WS-I Basic Profile Version 1.0: http://www.ws-i.org/Profiles/Basic/2003-08/BasicProfile-1.0a.html

18 Appendix: XML Schema Support

This chapter specifies the JAX-RPC support for the XML schema data types. This chapter uses the XML Schema specifications, Part 1 [8] and Part 2 [9], and SOAP 1.1 encoding specification [4].

Note that the XML data types listed in this section are not exhaustive. Refer to the XML Schema specification for the normative specification and examples of the XML Schema language. Any XML data types not listed in this section are not required to be supported by a JAX-RPC implementation.

The following definitions are used to indicate the JAX-RPC support:

- Required: A JAX-RPC implementation is required to support the Java mapping of an XML data type by using one or more of the following options:
  - The standard Java mapping of the XML data types as specified in the section 4.2, “XML to Java Type Mapping”
  - Java binding for the XML data types using the JAXB APIs (Java APIs for XML data binding) [14] or an implementation specific Java binding for the XML data types.
  - Pluggable serializer and deserializer framework specified in the chapter 15
- Optional: A JAX-RPC implementation is not required to support the Java mapping of a specific XML data type. A service using this XML data types may not be supported and interoperate with a JAX-RPC implementation.

WSDL to Java mapping tools must be able to process WSDL documents that use any of the features listed below, including those marked as optional. Tools should map XML types that use any of the optional features to javax.xml.soap.SOAPElement when used in literal parts (see section 6.4.1). Additionally, tools must be able to process schemas that use the `xsd:include` and `xsd:import` constructs.

The following XML Schema features are not required to be supported and WSDL to Java mapping tools are allowed to reject documents that use them: `xsd:redefine`, `xsd:notation`, substitution groups.

The examples in the following table use the following namespaces:

```
xmlns:xsd=”http://www.w3.org/2001/XMLSchema”
xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xmlns:soapenc=”http://schemas.xmlsoap.org/soap/encoding/”
```

```
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
Built-in datatypes specified in XML Schema Part 2: Datatypes specification:	xsd:string xsd:integer xsd:int xsd:long xsd:short xsd:decimal xsd:float xsd:double xsd:boolean xsd:byte xsd:QName xsd:dateTime xsd:base64Binary xsd:hexBinary	Required
Built-in datatypes specified in XML Schema Part 2: Datatypes specification:	Remaining simple types, with the exception of xsd:NOTATION and xsd:ENTITY.	Required
Built-in datatypes specified in XML Schema Part 2: Datatypes specification:	xsd:ENTITY xsd:NOTATION xsd:IDREF	Optional
Enumeration	<xsd:element name="EyeColor" type="EyeColorType" /> <xsd:simpleType name="EyeColorType" > <xsd:restriction base="xsd:string" > <xsd:enumeration value="Green" /> <xsd:enumeration value="Blue" /> <xsd:enumeration value="Brown" /> </xsd:restriction> </xsd:simpleType>	Required
Array of Bytes	<picture xsi:type="soapenc:base64"> aG93IG5vDyBicm73biBjb3cNCg== </picture>	Required support for: • `soapenc:base64` • `xsd:base64Binary` • `xsd:hexBinary`

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
SOAP Encoding: Struct	Refer to the <xsd:complexType>	
<xsd:complexType> with elements of both simple and complex types Refer <xsd:all> and <xsd:sequence> for additional requirements	<xsd:complexType name="USAddress" > <xsd:sequence> <xsd:element name="name" type="xsd:string"/> <xsd:element name="street" type="xsd:string"/> <xsd:element name="city" type="xsd:string"/> <xsd:element name="state" type="xsd:string"/> <xsd:element name="zip" type="xsd:decimal"/> </xsd:sequence> <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/> </xsd:complexType>	Required support for • multi-reference • single-reference • external reference Refer to the SOAP 1.1 specification for more details
<xsd:attribute> in <xsd:complexType>	See <xsd:attribute> in the previous example	Required
Occurence Constraints for xsd:element: • minOccurs • maxOccurs • default • fixed	<xsd:complexType name="PurchaseOrderType"> <xsd:sequence> <xsd:element name="shipTo" type="USAddress"/> <xsd:element name="billTo" type="USAddress"/> <xsd:element **ref**="comment" **minOccurs**="0"/> <xsd:element name="items" type="Items"/> </xsd:sequence> <xsd:attribute name="orderDate" type="xsd:date"/> </xsd:complexType>	Refer to the section section 4.2.3, “XML Struct and Complex Type” for support of occurance constraints.

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
<xsd:ref> attribute for reference to global elements	<xsd:element **ref**="comment" minOccurs="0"/>	Required
Derivation of new simple types by restriction of an existing simple type	<xsd:simpleType name="myInteger"> <xsd:restriction base="xsd:integer"> <xsd:minInclusive value="10000"/> <xsd:maxInclusive value="99999"/> </xsd:restriction> </xsd:simpleType>	Required
Facets used with restriction element	Refer XML Schema Part 2: Datatypes for details on the facets	Required.
List Type <xsd:list>	<xsd:simpleType name="listOfMyIntType"> <xsd:list itemType="myInteger"/> </xsd:simpleType> <listOfMyInt>20003 15037 95977 95945</listOfMyInt>	Required
Union Type <xsd:union>	<xsd:simpleType name="zipUnion"> <xsd:union memberTypes="USState listOfMyIntType"/> </xsd:simpleType>	Optional

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
SOAP Encoding Array defined using the type `soapenc:arrayType` in the schema instance Refer to the SOAP specification for more details	<xsd:element name="myFavoriteNumbers" type="soapenc:Array"/> *Schema instance:* <myFavoriteNumbers soapenc:arrayType="xsd:int[2]" > <number>3</number> <number>4</number> </myFavoriteNumbers>	Required support: • Array with complex types • Multi-dimension arrays • Single-reference or multi-reference values • Sparse arrays • Partially transmitted arrays Optional support: • Nested arrays • Array with instances of any subtype of the specified arrayType

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
SOAP Encoding Array derived from `soapenc:Array` by restriction using the `wsdl:arrayType` attribute	<complexType name="ArrayOfInteger"> <complexContent> <restriction base="soapenc:Array"> <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:int[]"/> </restriction> </complexContent> </complexType>	Required
SOAP Encoding Array derived from `soapenc:Array` by restriction	<xsd:simpleType name="phoneNumberType" > <xsd:restriction base="xsd:string" /> </xs:simpleType> <xsd:complexType name="ArrayOfPhoneNumbersType" > <xsd:complexContent> <xsd:restriction base="soapenc:Array" > <xsd:sequence> <xsd:element name="phoneNumber" type="phoneNumberType" maxOccurs="unbounded" /> </xsd:sequence> </xsd:restriction> </xsd:complexContent> </xsd:complexType> </xsd:schema>	Required

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
Derivation of a complex type from a simple Type	<xsd:element name="internationalPrice"> <xsd:complexType> <xsd:simpleContent> <xsd:extension base="xsd:decimal"> <xsd:attribute name="currency" type="xsd:string"/> </xsd:extension> </xsd:simpleContent> </xsd:complexType> </xsd:element>	Required
<xsd:anyType>	<xsd:element name="anything" type="xsd:anyType"/>	Optional
<xsd:sequence>	<xsd:complexType name="PurchaseOrderType"> <xsd:sequence> <xsd:element name="shipTo" type="USAddress"/> <xsd:element name="billTo" type="USAddress"/> <xsd:element name="items" type="Items"/> </xsd:squence> </xsd:complexType>	Required
<xsd:choice>	<xsd:complexType name="PurchaseOrderType"> <xsd:sequence> <xsd:choice> <xsd:group ref="shipAndBill"/> <xsd:element name="singleUSAddress" type="USAddress"/> </xsd:choice> <xsd:element name="items" type="Items"/> </xsd:sequence>	Optional

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
<xsd:group>	<**xsd:group** name="shipAndBill"> <xsd:sequence> <xsd:element name="shipTo" type="USAddress"/> <xsd:element name="billTo" type="USAddress"/> </xsd:sequence> </xsd:group>	Optional
<xsd:all>	<xsd:complexType name="PurchaseOrderType"> <xsd:all> <xsd:element name="shipTo" type="USAddress"/> <xsd:element name="billTo" type="USAddress"/> <xsd:element name="items" type="Items"/> </xsd:all> </xsd:complexType>	Required
<xsd:nil> and **<xsd:nillable>** attribute	<xsd:element name="shipDate" type="xsd:date" nillable="true"/> <shipDate xsi:nil="true"></ shipDate>	Required

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
Derivation of complex Types by extension	<complexType name="**Address**"> <sequence> <element name="name" type="string"/> <element name="street" type="string"/> <element name="city" type="string"/> </sequence> </complexType> <complexType name="**USAddress**"> <complexContent> <extension base="Address"> <sequence> <element name="state" type="USState"/> <element name="zip" type="positiveInteger"/> </sequence> </extension> </complexContent> </complexType> <complexType name="**UKAddress**"> <complexContent> <extension base="Address"> <sequence> <element name="postcode" type="UKPostcode"/> </sequence> </extension> </complexContent> </complexType>	Required

TABLE 18-1 XML Schema support in JAX-RPC specification

XML Schema	Example of XML Schema fragment and/or XML Schema instance	Support in JAX-RPC 1.1
Derivation of complex types by restriction	<complexType name="ConfirmedItems"> <complexContent> <restriction base="Items"> <sequence> <!-- item element is different than in Items --> <element name="item" minOccurs="1" maxOccurs="unbounded"> <!-- remainder of definition is same as Items --> <complexType> <sequence> <element name="productName" type="string"/> <element name="quantity" type="positiveInteger":> </sequence> </complexType> </element> </sequence> </restriction> </complexContent> </complexType>	Optional
Abstract Types	<complexType name="Vehicle" **abstract**="true"/> <complexType name="Car"> <complexContent> <extension base="target:Vehicle"/> </complexContent> </complexType> <complexType name="Plane"> <complexContent> <extension base="target:Vehicle"/> </complexContent> </complexType>	Optional
Creation and use of derived types: • final attribute • fixed attribute • block attribute	<complexType name="Address" final="restriction"> <sequence> <element name="name" type="string"/> <element name="street" type="string"/> <element name="city" type="string"/> </sequence> </complexType>	Optional

19 Appendix: Serialization Framework

This chapter describes the serialization framework that is part of the JAX-RPC Reference Implementation (JAX-RPC RI). Note that this design is non-prescriptive and serves as an illustration of how the base type mapping framework can be extended in an implementation specific manner.

The serialization framework includes the interfaces, classes, and algorithms used by the JAX-RPC RI to serialize and deserialize SOAP messages using SOAP 1.1 Section 5 encoding. It is used to serialize and deserialize all Java types that are subtypes of `java.lang.Object`, including JavaBeans, structures, arrays, boxed primitives, and any other kind of object. Primitive Java types (i.e., int, float, etc.) are handled by a similar but much simpler serialization framework that is only briefly described here.

The SOAP 1.1 encoding allows arbitrary graphs of Java objects to be serialized as XML. The most interesting case is when an object graph contains multiple references to the same object. In this case, the `HREF` mechanism of the SOAP encoding may be used to preserve object identity, thus making it possible to restore an object graph exactly as it was before serialization. By far, the main complicating factor in serializing and deserializing object graphs using SOAP encoding is ensuring that `HREFs` are handled correctly. As will become clear, most of the JAX-RPC RI serialization framework is motivated by the need to correctly process `HREFs`.

This chapter is divided into two main sections:

- Serialization section describes how the framework serializes a graph of Java objects as XML using SOAP encoding.
- Deserialization section describes how the framework deserializes SOAP-encoded object graphs into corresponding Java object graphs.

19.1 Serialization

A serializer is a class that knows how to encode instances of a particular Java type as XML using SOAP encoding. When a client of the serialization framework (e.g., a stub) needs to serialize a Java object, it obtains a reference to a serializer for the object's type and invokes the `serialize` method on it, passing as parameters the object to be serialized, an `XMLWriter`, and a `SOAPSerializationContext`. The `serialize` method creates the XML representation of the Java object by using the passed-in `XMLWriter`, which provides methods for writing XML elements to a stream.

Serializing a particular object consists of opening a new XML element that represents the object, and then creating sub-elements for each of the object's members. Each member is serialized by invoking the `serialize` method on an appropriate serializer,

passing in the same `XMLWriter` and `SOAPSerializationContext` that was used to serialize the parent object. After serializing each of its members, the `serialize` method closes the XML element for the object and returns.

For example, suppose that we want to serialize an instance of a Java class named `Address`. The code to do this would be:

```
Address address = new Address("1234 Riverside Drive",
                              "Gainesville",
                              "Georgia", "30506");
XMLWriter writer = new XMLWriter(...);
SOAPSerializationContext serContext =
          new SOAPSerializationContext(...);
AddressSerializer addressSer = new AddressSerializer(...);
addressSer.serialize(address, new QName(null, "home-address"),
                     writer, serContext);
serContext.serializeTrailingBlocks(writer);
```

The XML emitted by this code might look like this:

```
<home-address href="#ID-1"/>
<tns:Address xsi:type="tns:Address" id="ID-1">
    <street href="#ID-2"/>
    <city href="#ID-3"/>
    <state href="#ID-4"/>
    <zip href="#ID-5"/>
</tns:Address>
<soapenc:string xsi:type="xsd:string" id="ID-2">1234 Riverside Drive
</soapenc:string>
<soapenc:string xsi:type="xsd:string" id="ID-3">Gainesville
</soapenc:string>
<soapenc:string xsi:type="xsd:string" id="ID-4">Georgia
</soapenc:string>
<soapenc:string xsi:type="xsd:string" id="ID-5">30506
</soapenc:string>
```

19.1.1 Serializers

A serializer is a Java class with the following characteristics:

- A serializer must implement the `SOAPSerializer` interface
- A serializer must be stateless. This allows the JAX-RPC runtime to create and reuse a single serializer instance for each XML Schema data type in the system, thus minimizing the number of serializer objects that are created. Alternatively, a serializer may store immutable state that is initialized when the serializer instance is constructed.
- A serializer must be multi-thread safe, thus allowing the JAX-RPC runtime to use a single serializer instance across multiple threads simultaneously.

19.1.2 SOAPSerializationContext

The `SOAPSerializationContext` class contains information that must be shared among and passed between the serializers that participate in the serialization of a graph of Java objects. As an object graph is being serialized, the same `SOAPSerializationContext` instance is passed to each serializer that participates. Each method in the `SOAPSerializer` interface accepts a parameter of type `SOAPSerializationContext` which is used to propagate the context between the various serializers.

The primary (and currently only) purpose of the SOAPSerializationContext is to ensure correct processing of HREFs. A SOAPSerializationContext contains a map from object references to SOAPSerializationState objects. The first time a new object is encountered in the serialization process, a SOAPSerializationState object is created for that object and added to the map. The SOAPSerializationState for an object stores a unique ID assigned to that object and a reference to the serializer for that object. The class SOAPSerializationState is defined as follows:

```
package com.sun.xml.rpc.encoding.soap;
public class SOAPSerializationState {
    // ...
    public SOAPSerializationState(Object obj, String id,
                                  SOAPSerializer serializer) {...}
    public Object getObject() {...}
    public String getID() {...}
    public SOAPSerializer getSerializer() {...}
}
```

Associating a unique ID with each object allows the serialization framework to preserve object identity in the SOAP encoded object graph by always using the same ID for an object as it encodes href and id attributes. Associating a serializer with each object allows the serialization framework to know which serializers to call as it serializes the independent elements that represent multiple-reference objects (i.e., the trailing blocks).

With an understanding of SOAPSerializationState, we are prepared to look at the definition of SOAPSerializationContext:

```
package com.sun.xml.rpc.encoding.soap;
public class SOAPSerializationContext
          implements javax.xml.rpc.encoding.SerializationContext {
    // ...
    public SOAPSerializationContext()
                throws SerializationException {...}

    public SOAPSerializationContext(String prefix)
                                  throws SerializationException {...}

    public SOAPSerializationState registerObject(Object obj,
                                         SOAPSerializer serializer)
                                  throws SerializationException {...}

    public SOAPSerializationState lookupObject(Object obj)
                                  throws SerializationException {...}

    public void serializeTrailingBlocks(XMLWriter writer)
                                  throws SerializationException {...}
}
```

The registerObject method is called by serializers to add an object to the serialization context. References to the object and its serializer are passed as parameters to registerObject, which generates a unique ID for the object and adds a SOAPSerializationState for the object to its internal map. If the object has already been registered, registerObject simply returns the existing SOAPSerializationState for the object.

The lookupObject method is used by serializers to determine if an object already exists in the serialization context, and, if so, to gain access to the SOAPSerializationState for the object. If the object is unknown to the serialization context, lookupObject returns null.

The `serializeTrailingBlocks` method is called at the end of the serialization process to serialize independent elements (or trailing blocks) for each of the multi-reference objects encountered during the serialization process.

19.1.3 SOAPSerializer Interface

All serializers implement the `SOAPSerializer` interface. The definition of the `SOAPSerializer` interface is shown below:

```
package com.sun.xml.rpc.encoding.soap;
public interface SOAPSerializer
          extends javax.xml.rpc.encoding.Serializer {

    // ...
   void serializeReference(Object obj, QName name,
                           XMLWriter writer,
                           SOAPSerializationContext context)
                  throws SerializationException;

   void serializeInstance(Object obj, QName name,
                           boolean isMultiRef,
                           XMLWriter writer,
                           SOAPSerializationContext context)
                  throws SerializationException;

   void serialize(Object obj, QName name,
                      XMLWriter writer,
                      SOAPSerializationContext context)
                  throws SerializationException;
}
```

The `serializeReference` method serializes a `HREF` to the passed-in object. The `obj` parameter is a reference to the object that is to be serialized. The `name` parameter is the name to be used for the XML element that represents the object. The `writer` parameter is the `XMLWriter` that should be used to write the XML representation of the object. The `context` parameter is the `SOAPSerializationContext` to be used during serialization of the object. As an example, the following code would be used to serialize an `HREF` to an object (as opposed to serializing the object itself):

```
   Address address = new Address("1234 Riverside Drive",
                           "Gainesville",
                           "Georgia", "30506");
   XMLWriter writer = new XMLWriter(...);
   SOAPSerializationContext serContext =
                        new SOAPSerializationContext(...);
   AddressSerializer addressSer = new AddressSerializer(...);
   addressSer.serializeReference(address,
                           new QName(null, "home-address"),
                           writer, serContext);
   serContext.serializeTrailingBlocks(writer);
```

The XML emitted by this code might look like this:

```
<home-address href="#ID-1"/>
```

Pseudo-code for a serializer's `serializeReference` method is shown below:

```
public void serializeReference(Object obj, QName name,
                            XMLWriter writer,
                            SOAPSerializationContext context)
                            throws SerializationContext {
    if (obj == null) {
```

```
        create a new element with the specified name
        set the element's xsi:null attribute to "true"
      set the element's xsi:type attribute to the appropriate XSD type
        close the element
    }
    else {
        create a new element with the specified name
      SOAPSerializtionState state= context.registerObject(obj,this);
        set the element's href attribute to state.getID()
        close the element
    }
}
```

The `serializeInstance` method serializes the object instance itself (not an reference to it). The `obj` parameter is a reference to the object that is to be serialized. The `name` parameter is the name to be used for the XML element that represents the object. The `isMultiRef` parameter indicates whether the object is a single-reference or a multiple-reference instance (multiple-reference instances have an id attribute, single-reference instances do not). The `writer` parameter is the `XMLWriter` that should be used to write the XML representation of the object. The `context` parameter is the `SOAPSerializationContext` to be used during serialization of the object. As an example, the following code would be used to serialize a multiple-reference object:

```
Address address = new Address("1234 Riverside Drive",
                              "Gainesville",
                              "Georgia", "30506");
XMLWriter writer = new XMLWriter(...);
SOAPSerializationContext serContext =
                      new SOAPSerializationContext(...);
AddressSerializer addressSer = new AddressSerializer(...);
addressSer.serializeInstance(address,
                          new QName(null, "home-address"),
                          true, writer, serContext);
serContext.serializeTrailingBlocks(writer);
```

The XML emitted by this code might look like this:

```
<home-address href="#ID-1"/>
<tns:Address xsi:type="tns:Address" id="ID-1">
    <street href="#ID-2"/>
    <city href="#ID-3"/>
    <state href="#ID-4"/>
    <zip href="#ID-5"/>
</tns:Address>
<soapenc:string xsi:type="xsd:string" id="ID-2">1234 Riverside Drive
</soapenc:string>
<soapenc:string xsi:type="xsd:string" id="ID-3">Gainesville
</soapenc:string>
<soapenc:string xsi:type="xsd:string" id="ID-4">Georgia
</soapenc:string>
<soapenc:string xsi:type="xsd:string" id="ID-5">30506
</soapenc:string>
```

Pseudo-code for a serializer's serializeInstance method is shown below:

```
public void serializeInstance(Object obj, QName name,
                              boolean isMultiRef,
                              XMLWriter writer,
                              SOAPSerializationContext context)
                              throws SerializationContext {
    if (obj == null) {
        create a new element with the specified name
        set the element's xsi:null attribute to "true"
      set the element's xsi:type attribute to the appropriate XSD type
```

```
        close the element
    }
    else {
        create a new element with the specified name
      set the element's xsi:type attribute to the appropriate XSD type
        if (isMultiRef) {
            SOAPSerializtionState state =
                        context.registerObject(obj, this);
            set the element's id attribute to state.getID()
        }
        for each member M of this object {
           create a sub-element for M by calling the serialize method
                on the serializer for M's type
        }
        close the element
    }
}
```

The `serialize` method of a serializer performs "default" serialization of the passed-in object. Each serializer is allowed to decide whether its data type should be serialized as single-reference or multiple-reference instances by default. If a serializer chooses single-reference to be the default, then calling `serialize` will have the same effect as calling `serializeInstance` with the `isMultiRef` parameter set to false. Alternatively, if a serializer chooses multiple-reference as its default serialization mode, then calling `serialize` will have the same effect as calling `serializeReference`. Of course, the code invoking a serializer may choose to explicitly call `serializeReference` or `serializeInstance` if it knows exactly how it wants to serialize the object. However, in most cases it is best to let the serializer for a particular data type decide how instances of that data type are serialized by default.

Pseudo-code for a serializer's `serialize` method is shown below:

```
public void serialize(Object obj, QName name,
                      XMLWriter writer,
                      SOAPSerializationContext context)
                      throws SerializationContext {
    if (multiple-reference is the default serialization mode) {
        serializeReference(obj, name, writer, context);
    }
    else {
        serializeInstance(obj, name, false, writer, context);
    }
}
```

Serializing Trailing Blocks

When a graph of Java objects is serialized, references between objects are normally serialized as `HREFs` by calling the `serializeReference` method (perhaps indirectly through `serialize`). As the serialization process proceeds, the `SOAPSerializationContext` creates a `SOAPSerializationState` for each object that is referenced. In addition to the object's unique ID, the `SOAPSerializationState` stores a reference to the object's serializer, which is determined when the first reference to the object is serialized. In addition to storing a map from object references to `SOAPSerializationStates`, the `SOAPSerializationContext` also stores a list of objects that have been referenced during serialization but have yet to be serialized themselves (i.e., a list of multiple-reference objects). After explicitly invoking the serializers for each root of an object graph, it is necessary to call the `serializeTrailingBlocks` method on the `SOAPSerializationContext` in order to finish serializing all of the objects that have been referenced but not yet serialized. The `serializeTrailingBlocks` method proceeds by serializing each multiple-reference

object in the list as a top-level independent element with its id attribute set to the appropriate value. Of course, each multiple-reference object may in turn contain references to other objects, which means that the list of objects that need to be serialized can grow even as `serializeTrailingBlocks` proceeds; it simply continues until the list is empty, at which point the object graph has been completely serialized.

19.1.4 Primitive Serializers

The serialization framework as described thus far can only handle Java types that are subtypes of java.lang.Object. Of course, primitive types play a very important role and must also be serialized. The JAX-RPC RI also includes serializers for all of the primitive Java data types such as int, float, etc. These primitive serializers are much simpler than the object serializers because they don't have to deal with `HREF`s or nulls. Each primitive serializer has a single method of the following form:

```
public void serialize(TYPE value, QName name,
                      XMLWriter writer)
              throws SerializationException;
```

where `TYPE` is a primitive type such as int, float, etc. The serialize method simply writes the XML representation for the primitive value to the `XMLWriter`.

19.2 Deserialization

A deserializer is a class that knows how to recreate instances of a particular Java type that have been serialized using SOAP encoding. When a client of the serialization framework (e.g., a tie) needs to deserialize an object, it obtains a reference to a deserializer for the object's type and invokes the `deserialize` method on it, passing as parameters the `XMLReader` that contains the SOAP encoded XML representation of the object and a `SOAPDeserializationContext`. The `deserialize` method reconstructs the object from the XML representation and returns a reference to the object as its result. Deserializing an object involves the following steps:

1. Open the XML element that represents the object
2. Recursively deserialize each of the object's members which are encoded as sub-elements
3. Create a new instance of the Java type, initializing it with the deserialized members
4. Return the new object as the result of `deserialize`

Each member is deserialized by invoking the `deserialize` method on an appropriate deserializer, passing in the same `XMLReader` and `SOAPDeserializationContext` that was used to deserialize the parent object.

The deserialization problem is made significantly more complex by the `HREF` mechanism supported by SOAP encoding. Ideally, whenever the `deserialize` method is called, the deserializer would be able to completely instantiate and initialize the object being deserialized and return it as the result of `deserialize`. Unfortunately, if the SOAP encoded representation of the object contains `HREF`s to other objects, `deserialize` will not be able to fully reconstitute the object if member objects to which it refers have not been deserialized yet. Only when all member objects have been deserialized will it be possible to finish deserializing the parent object, and this condition might not be satisfied until much later in the deserialization process. For this reason, the serialization

framework allows the deserialize method to only partially deserialize an object, and then register listeners that allow it to be notified later when its incomplete member objects become available, thus allowing it to complete deserialization of the parent object. The details of these mechanisms are provided in the following sections.

19.2.1 Deserializers

A deserializer is a Java class with the following characteristics:

- A deserializer must implement the SOAPDeserializer interface
- A deserializer must be stateless. This allows the JAX-RPC runtime to create and reuse a single deserializer instance for each XML Schema data type in the system, thus minimizing the number of deserializer objects that are created. Alternatively, a deserializer may store immutable state that is initialized when the deserializer instance is constructed.
- A deserializer must be multi-thread safe, thus allowing the JAX-RPC runtime to use a single deserializer instance across multiple threads simultaneously.

All deserializers implement the SOAPDeserializer interface. The definition of SOAPDeserializer is shown below:

```
package com.sun.xml.rpc.encoding.soap;
public interface SOAPDeserializer
          extends javax.xml.rpc.encoding.Deserializer {
   Object deserialize(QName name, XMLReader reader,
                  SOAPDeserializationContext context)
                           throws DeserializationException;

}
```

The deserialize method is called to deserialize an object. The name parameter is the expected name of the XML element that contains the object. If the actual element name does not match the expected element name, deserialize will throw an exception. The reader parameter contains the SOAP encoded XML data that is to be deserialized. The context parameter contains information that must be shared among and passed between the deserializers that participate in the deserialization of an object graph. As an object graph is deserialized, the same SOAPDeserializationContext instance is passed to each deserializer that participates. If all goes well, deserialize will return a reference to a completely deserialized Java object. As explained in the next section, this might not always be the case.

19.2.2 SOAPDeserializationContext

The SOAPDeserializationContext class stores information that is necessary to properly reconstruct object graphs that make use of the SOAP encoding HREF mechanism. Specifically, it stores a SOAPDeserializationState object for each object encountered during the deserialization process that cannot be completely deserialized by its deserializer's deserialize method. Ideally, when the deserialize method is done, it returns a reference to the object that it was asked to deserialize. However, there are two scenarios in which deserialize is unable to completely deserialize the object:

- The deserializer has been asked to deserialize an HREF to the object, and the object itself has not been completely deserialized yet
- The object being deserialized contains references (either directly or indirectly) to other objects which have not yet been completely deserialized

In these cases, instead of returning a reference to the object itself, it returns a reference to the SOAPDeserializationState for the object. The SOAPDeserializationState returned by deserialize will have been registered with the SOAPDeserialization Context so that it can be accessed by other deserializers that participate in deserialization of the object graph.

SOAPDeserializationState stores several pieces of information that track the progress of an object as it is being deserialized:

- A reference to the object, which is null if it has not yet been instantiated
- The current state of the object: None, Created, Initialized, Complete (more on this later)
- Information about other objects that this object depends on
- Information about other objects that depend on this object
- A reference to the deserializer for this object.
- A reference to the SOAPInstanceBuilder for this object (more on this later)

A partial definition of SOAPDeserializtionState is shown below:

```
package com.sun.xml.rpc.encoding.soap;
public class SOAPDeserializationState {
    // ...
    public SOAPDeserializationState(){...}
    public void setInstance(Object instance){...}
    public Object getInstance(){...}
    public boolean isComplete(){...}
    public void doneReading(){...}
    public void setDeserializer(SOAPDeserializer deserializer)
                    throws DeserializationException {...}
    public void registerListener(
                               SOAPDeserializationState parentState,
                               int memberIndex){...}
    public void setBuilder(SOAPInstanceBuilder builder){...}
    // ...
}
```

The setInstance method is called by a deserializer when the object that is being deserialized is instantiated (i.e., when "new" is called). The object need not have been completely deserialized in order to call setInstance; it only needs to have been constructed. The getInstance method can be used by deserializers to obtain a reference to the object during the deserialization process.

The isComplete method can be called to determine whether or not the object has been completely deserialized yet.

The doneReading method is called by an object's deserializer to notify the serialization framework that it is finished deserializing the object's data members. This does not imply that the object is entirely complete because it may still have unresolved dependencies on other objects. It does mean, however, that the XML element representing the object has been completely processed.

The setDeserializer method is called to tell the serialization framework which deserializer to use for this object. This information is required by the framework when it comes time to deserialize the trailing blocks that represent multiple-reference objects.

The registerListener method is used by the deserializer of a parent object to declare a dependency on a member object that has not yet been completely deserialized (and is thus unavailable), and to request that it be notified as the deserialization of the member object progresses. Specifically, the deserializer for the parent object calls registerListener on the SOAPDeserializationState for the member object, passing

as parameters a reference to the parent's SOAPDeserializationState and an index that uniquely identifies the member within the parent. This call sets up the necessary callback relationship that allows the parent deserializer to be notified when the member object has been completely deserialized, thus allowing it to complete serialization of the parent object.

The setBuilder method is described later in this section.

With this understanding of SOAPDeserializationState, we are ready to look at the definition of SOAPDeserializationContext:

```
public class SOAPDeserializationContext
       implements javax.xml.rpc.encoding.DeserializationContext {
    // ...
    public SOAPDeserializationContext()
                    throws DeserializationException {...}
    public SOAPDeserializationState getStateFor(String id) {...}

    public void deserializeRemainingElements(XMLReader reader)
                    throws DeserializationException {...}
}
```

The getStateFor method is used to obtain a reference to the SOAPDeserialization State for the object with a particular ID. The first time the state for a particular ID is requested, the context creates a new SOAPDeserializationState for that object and adds it to the internal state map.

The deserializeRemainingElements method is called at the end of the deserialization process to deserialize the independent elements (or trailing blocks) that represent the multiple-reference objects.

19.2.3 The deserialize Method

The workhorse of the deserialization process is the deserialize method. The deserializer for each Java type implements deserialize in a way that is appropriate for that type. For example, the deserializer for a JavaBean would deserialize each of the bean's properties, set these properties on a new instance of the bean, and return the new bean instance as the result of the deserialize method.

As has been mentioned previously, the ideal case occurs when deserialize is able to instantiate and completely initialize the object, thus allowing it to return the object itself as the result. However, in those cases where the object cannot be completely deserialized, the deserialize method returns the SOAPDeserializationState for the object instead of the object itself. This allows whoever called deserialize (usually the deserializer of a parent object), to register for events that will allow it to eventually complete its deserialization. Therefore, it is the responsibility of the code that calls deserialize to check the return value to determine whether it's the object itself or the SOAPDeserializationState that represents the incompletely deserialized object (this check is performed using Java's instanceof operator).

When the deserialize method completes, the object being deserialized can be in one of four states. The first state occurs when the deserializer was able to completely deserialize the object, in which case the object is referred to as being COMPLETE. In the other three states, the object has dependencies on at least one member object that is not yet completely deserialized, thus preventing the complete deserialization of the parent object. The first incomplete state occurs when at least one of the incomplete member objects is required as a constructor parameter for the parent object, thus preventing the deserializer from instantiating the parent object until the member is complete; this state

is referred to as NONE. The second incomplete state occurs when the deserializer is able to instantiate the object by calling "new", but it is not able to completely initialize the other members of the object due to one or more incomplete member objects; this state is referred to as CREATED. The third incomplete state occurs when the deserializer is able to instantiate the object and initialize all of its members, but at least one of the object's members is still not entirely complete (although it has been instantiated); this state is referred to as INITIALIZED. If the object is COMPLETE, deserialize returns the actual object as its return value. If the object is NONE, CREATED, or INITIALIZED, deserialize returns the object's SOAPDeserializationState as its return value.

Pseudo-code for a deserializer's deserialize method is shown below:

```
public Object deserialize(QName name, XMLReader reader,
                      SOAPDeserializationContext context)
             throws DeserializationException {

    if (the element's name doesn't match the expected name) {
        throw exception
    }

    if (the element is an HREF) {
        String href = the value of the element's href attribute
        SOAPDeserializationState state = context.getStateFor(href);
        state.setDeserializer(this);

        if (state.isComplete()) {
            return state.getInstance();
        }
        else {
            return state;
        }
    }

    if (the element has an xsi:type attribute) {
        if (the xsi:type attribute doesn't have the expected value) {
            throw exception
        }
    }

   if (the element has an xsi:null attribute && xsi:null == "true") {
        if (the element has an id attribute) {
            String id = the value of the element's id attribute
           SOAPDeserializationState state = context.getStateFor(id);
            state.setDeserializer(this);
            state.setInstance(null);
            state.doneReading();
        }
        return null;
    }

    Object[] members = new Object[number of members];
    SOAPDeserializationState state = null;
    boolean isComplete = true;

    if (the element has an id attribute) {
        String id = the value of the element's id attribute
        state = context.getStateFor(id);
        state.setDeserializer(this);
    }

    for each member of this object {
```

```
            SOAPDeserializer memberDeser = the deserializer for the member
             QName memberName = the expected QNAME of member's sub-element
             XMLReader memberReader =
                      XMLReader for the member's sub-element
             Object member[i] = memberDeser.deserialize(memberName,
                                                        memberReader,
                                                        context);

             if (member[i] instanceof SOAPDeserializationState) {
                 isComplete = false;

                 if (state == null) {
                 // it's a single-reference object (i.e., no id attribute)
                    state = new SOAPDeserializationState();
                 }

                 SOAPDeserializationState memberState =
                     (SOAPDeserializationState)member[i];

                 memberState.registerListener(state, i);
             }
         }

         if (isComplete) {
             instantiate and completely initialize a new instance

             if (state != null) {
                 state.setInstance(instance);
                 state.doneReading();
             }

             return instance;
         }
         else {
             if (all constructor arguments are available) {
                 instantiate a new instance
                 initialize the new instance as completely as possible
                 state.setInstance(instance);
             }

            SOAPBuilder builder = a new builder initialized with whatever
                                  state will be needed later to finish
                                  deserializing the object
             state.setBuilder(builder);
             state.doneReading();

             return state;
         }
     }
```

This pseudo-code demonstrates the logic of the `deserialize` method, but it does not impose a particular implementation on deserializers. Although the algorithm looks complicated, it is actually quite simple to write deserializers for many Java types, especially those that have no-argument constructors (e.g., structures and arrays). It is also possible to write generic deserializers that can handle entire classes of types such as arrays and JavaBeans. In addition, much of the code for serializers can be placed in library base classes, greatly simplifying development of new serializers.

19.2.4 Instance Builders

When a deserializer is unable to completely deserialize an object, it must do two things:

1. Register the `SOAPDeserializationState` for the parent object as a listener with the `SOAPDeserializationState` of the incomplete member object. This allows the deserializer to be notified of progress made in the deserialization of the member object.

2. The deserializer needs to remember any state that it accumulated during `deserialize` that it will need later to finish deserializing the parent object when the member object is complete. For example, it might need to remember the values of some members that were completely deserialized during the call to `deserialize` and will be needed later to complete initialization of the object.

The problem with (2) is that we have assumed that deserializers are stateless and are thus incapable of storing state that will be needed later to complete deserialization of the object when its members are complete. Instance builders are the mechanism used to overcome the limitations on stateless deserializers. In effect, instance builders allow deserializers to become stateful when they need to. When a deserializer is only able to partially deserialize an object, it creates an instance builder and stores a reference to it in the object's `SOAPDeserializationState`. The instance builder is initialized to store any state that the deserializer will need later to complete the task of deserializing the object once its members have been deserialized.

In addition to storing state, instance builders are actually responsible for completing deserialization of the parent object (thus the name "instance builder"). The serialization framework will notify the instance builder when deserialization of its members has progressed to the point that it should be able to construct the parent object and complete its initialization. Therefore, the instance builder is really an extension of the stateless deserializer that originally began deserializing of the parent object.

An instance builder is a Java class that implements the `SOAPInstanceBuilder` interface. The definition of this interface is:

```
package com.sun.xml.rpc.encoding.soap;
public interface SOAPInstanceBuilder {
    int memberGateType(int memberIndex);
    void setMember(int index, Object memberValue);
    void construct();
    void initialize();
    void setInstance(Object instance);
    Object getInstance();
}
```

The `memberGateType` method is called by the serialization framework to query the instance builder about the types of dependencies it has on incomplete member objects. The value returned by this method for a particular member tells the framework what the instance builder needs that member for (construction or initialization), and how complete the member must be before the instance builder can make use of it (created, initialized, or complete). This information allows the framework to determine when deserialization has progressed far enough that the instance builder should be able to instantiate or initialize the object.

The `setMember` method is called by the framework to notify the instance builder whenever one of its member objects has been instantiated (it still might not be entirely complete, but at least it's been instantiated).

The `construct` method is called by the framework when all members needed for construction are ready, thus allowing the instance builder to instantiate the object.

The `initialize` method is called by the framework when all members needed to complete initialization of the object are ready.

The `setInstance` method is called to provide the instance builder with a reference to the object, which is useful when the object was instantiated by the deserializer before the instance builder was created (e.g., during `deserialize`).

The `getInstance` method is called by the framework to get a reference to the object after it has been instantiated by the instance builder (e.g., after a call to `initialize`).

19.2.5 Deserializing Trailing Blocks

In order to completely deserialize an object graph, it is necessary to explicitly invoke the deserializer for each root of the graph in the same order that the roots were originally serialized. After doing this, it is also necessary to call the `deserializeRemainingElements` method on the `SOAPDeserializationContext` to complete deserialization of the trailing blocks that represent multiple-reference objects.

For example, the code required to deserialize this SOAP encoded object graph:

```
<home-address href="#ID-1"/>
<tns:Address xsi:type="tns:Address" id="ID-1">
    <street href="#ID-2"/>
    <city href="#ID-3"/>
    <state href="#ID-4"/>
    <zip href="#ID-5"/>
</tns:Address>
<xsd:string xsi:type="xsd:string" id="ID-2">1234 Riverside Drive</
xsd:string>
<xsd:string xsi:type="xsd:string" id="ID-3">Gainesville</xsd:string>
<xsd:string xsi:type="xsd:string" id="ID-4">Georgia</xsd:string>
<xsd:string xsi:type="xsd:string" id="ID-5">30506</xsd:string>
```

would be as follows:

```
XMLReader reader = new XMLReader(...);
SOAPDeserializationContext deserContext =
                    new SOAPDeserializationContext(...);
AddressDeserializer addressDeser = new AddressDeserializer(...);
Object obj = addressDeser.deserialize(
                                  new QName(null, "home-address"),
                                  reader, deserContext);
deserContext.deserializeRemainingElements(reader);
Address address = null;
if (obj instanceof SOAPDeserializationState) {
address = (Address)(((SOAPDeserializationState)obj).getInstance());
}
else {
address = (Address)obj;
}
```

19.2.6 Primitive Deserializers

Just as for serialization, deserialization of primitive Java types are handled by a separate framework because they are not subtypes of `java.lang.Object`. The JAX-RPC RI also includes deserializers for all of the primitive Java data types such as int, float, etc. These

primitive deserializers are much simpler than the object deserializers because they don't have to deal with HREFs or nulls. Each primitive deserializer has a single method of the following form:

```
public TYPE deserialize(QName name, XMLReader reader)
          throws DeserializationException;
```

where TYPE is a primitive type such as int, float, etc.

19.3 XMLWriter

The XMLWriter interface is used to write XML documents. The following code snippet shows the XMLWriter interface with a brief description of the methods. Note that this interface is specific to the JAX-RPC reference implementation.

```
package com.sun.xml.rpc.streaming;
public interface XMLWriter {
   // The following methods write the start tag for an element
   void startElement(QName name) throws XMLWriterException;
   void startElement(QName name, String prefix)
                                      throws XMLWriterException;
   void startElement(String localName)
                                   throws XMLWriterException;
   void startElement(String localName, String uri)
                                   throws XMLWriterException;
   void startElement(String localName, String uri,
                          String prefix)
                                   throws XMLWriterException;

   // The following methods write an attribute of the current element
   void writeAttribute(QName name, String value)
                                   throws XMLWriterException;
   void writeAttribute(String localName, String value)
                                   throws XMLWriterException;
   void writeAttribute(String localName, String uri,
                       String value) throws XMLWriterException;

   // Write a namespace declaration of the current element.
   void writeNamespaceDeclaration(String prefix, String uri)
                                   throws XMLWriterException;
   // Write character data within an element.
   void writeChars(String chars) throws XMLWriterException;
   void writeChars(CDATA chars) throws XMLWriterException;

   // Write a comment within an element.
   void writeComment(String comment)
                                   throws XMLWriterException;

   // Write the end tag for the current element.
   void endElement() throws XMLWriterException;

   // Return the prefix factory in use by this writer.
   PrefixFactory getPrefixFactory();

   // Set the prefix factory to be used by this writer.
   void setPrefixFactory(PrefixFactory factory);

   // Return the URI for a given prefix. If the prefix is undeclared,
   // return null.
   String getURI(String prefix) throws XMLWriterException;
```

```
    // Return a prefix for the given URI. If no prefix for the given
    // URI is in scope, return null
    String getPrefix(String uri) throws XMLWriterException;

    // Flush the writer and its underlying stream.
    void flush() throws XMLWriterException;

    // Close the writer and its underlying stream.
    void close() throws XMLWriterException;
}
```

19.4 XMLReader

The `XMLReader` interface provides a high-level streaming parser interface for reading XML documents. The following code snippet shows the `XMLReader` interface with a brief description of the methods. Note that this interface is specific to the JAX-RPC reference implementation.

```
package com.sun.xml.rpc.streaming;
public interface XMLReader {
    // The initial state of a XMLReader.
    public static final int INITIAL = 0;

     // The state denoting the start tag of an element.
    public static final int START = 1;

    // The state denoting the end tag of an element.
    public static final int END = 2;

    // The state denoting the character content of an element.
    public static final int CHARS = 3;

    // The state denoting a processing instruction.
    public static final int PI = 4;

   // The state denoting that the end of the document has been reached.
    public static final int EOF = 5;

    /* Return the next state of the XMLReader. The return value is
     * one of: START, END, CHARS, PI, EOF.
    **/
    int next() throws XMLReaderException;

    /* Return the next state of the XMLReader. The return value is
     * one of: START, END, CHARS, EOF.
    **/
    int nextContent() throws XMLReaderException;

    /* Return the next state of the XMLReader. The return value is
     * one of: START, END, EOF
    **/
    int nextElementContent() throws XMLReaderException;

    // Return the current state of the XMLReader.
    int getState();

    /* Return the current qualified name. Meaningful only when
     * the state is one of: START, END.
```

```
    **/
    QName getName();

    /* Return the current URI. Meaningful only when the state is
     * one of: START, END.
    **/
    String getURI();

    /* Return the current local name. Meaningful only when the
     * state is one of: START, END, PI.
    **/
    String getLocalName();

    /* Return the current attribute list. Meaningful only when
     * the state is one of: START. The returned Attributes object
     * belong to the XMLReader and is only guaranteed to be valid
     * until the next method is called directly or indirectly
    **/
    Attributes getAttributes();

    /*  Return the current value. Meaningful only when the state
     * is one of: CHARS, PI.
    **/
    String getValue();

    // Return the current element ID.
    int getElementId();

    // Return the current line number
    int getLineNumber();

   /** Return the URI for the given prefix. If there is no namespace
     * declaration in scope for the given prefix, return null.
    **/
    String getURI(String prefix);

    /**
      * Return an iterator on all prefixes in scope.
    **/
    Iterator getPrefixes();

    /* Records the current element.
     * The XMLReader must be positioned on the start tag of the
     * element. The returned reader will play back all events
     * starting with the start tag of the element and ending with
     * its end tag.
    **/
    XMLReader recordElement() throws XMLReaderException;

    // Skip to the end tag of the element with the current element ID.
    void skipElement() throws XMLReaderException;

     // Skip to the end tag for the element with the given element ID
    void skipElement(int elementId) throws XMLReaderException;

    // Close the XMLReader.
    void close() throws XMLReaderException;
}
```

The `next` method is used to read events from the XML document. Each time it is called, the `next` method returns the new state of the reader. Possible states are: `INITIAL` (the initial state), `START` (denoting the start tag of an element), `END` (denoting the end tag of an element), `CHARS` (denoting the character content of an element), `PI` (denoting a processing instruction) and `EOF` (denoting the end of the document).

An `XMLReader` is always namespace-aware and keeps track of the namespace declaration which are in scope at any time during streaming. The `getURI` method can be used to find the URI associated to a given prefix in the current scope.

20 Appendix: Mapping of XML Names

Note that this section has been taken from the JAXB (Java APIs for XML data binding) [14] specification. There are some minor changes and clarifications.

XML schema languages use XML names. The character set supported in the XML name is much larger than the character set defined for valid identifiers for Java class, interface and method.

This chapter defines recommended mapping of XML names to Java identifiers in a way that adheres to the standard Java API design guidelines, generates identifiers that retain mapping from the source schema and is unlikely to result in collisions.

20.1 Mapping

Java identifiers are based on the following conventions:

- Class and interface names always begin with an upper-case letter. The remaining characters are either digits, lower-case letters or upper-case letters. Upper-case letters within a multi-word name serve to identify the start of each non-initial word or sometimes to stand for acronyms. These names may contain an '_' underscore.
- Method names always begin with a lower-case letter, and otherwise are exactly like class and interface names.
- Constant names are entirely in upper case, with each pair of words separated by the underscore character (`'_', \u005F, LOW LINE`).

XML names are much richer than Java identifiers: They may include not only the standard Java identifier characters but also various punctuation and special characters that are not permitted in Java identifiers. Like most Java identifiers, most XML names are in practice composed of more than one natural-language word. Non-initial words within an XML name typically start with an upper-case letter followed by a lower-case letter (as in Java) or are prefixed by punctuation characters (not usual in Java and for most punctuation characters are in fact illegal).

In order to map an arbitrary XML name into a Java class, method, or constant identifier, the XML name is first broken into a word list. For the purpose of constructing word lists from XML names, the following definitions are used:

A `punctuation character` (`punct`) is one of the following:

- `Hyphen ('-', \u002D, HYPHEN-MINUS)`
- `Period ('.', \u002E, FULL STOP)`
- `Colon (':', \u003A, COLON)`
- `Dot ('•', \u00B7, MIDDLE DOT)`
- `\u0387, GREEK ANO TELEIA`

- \u06DD, ARABIC END OF AYAH
- \u06DE, ARABIC START OF RUB EL HIZB

These are all legal characters in XML names.

- A letter is a character for which the method Character.isLetter returns true; identifying a letter based on the Unicode standard. Every letter is a legal Java identifier character, both initial and non-initial.
- A digit is a character for which the method Character.isDigit returns true, identifying a digit according to the Unicode Standard. Every digit is a legal non-initial Java identifier character.
- A mark is a character that is in none of the previous categories but for which the method Character.isJavaIdentifierPart returns true. This category includes numeric letters, combining marks, non-spacing marks, and ignorable control characters.

Every XML name character falls into one of the above categories. Letters are further divided into sub-categories:

- An upper-case letter is a letter for which the method Character.isUpperCase returns true
- A lower-case letter is a letter for which the method Character.isLowerCase returns true
- All other letters are uncased.

An XML name is split into a word list by removing any leading and trailing punctuation characters and then searching for word breaks. A word break is defined by three regular expressions: A prefix, a separator, and a suffix. The prefix matches part of the word that precedes the break, the separator is not part of any word, and the suffix matches part of the word that follows the break. The word breaks are defined as:

TABLE 20-1

Prefix	Separator	Suffix	Example
[^punct]	punct+	[^punct]	foo|--|bar
digit		[^digit]	foo22|bar
[^digit]		digit	foo|22
lower		[^lower]	foo|Bar
upper		upper lower	FOO|Bar
letter		[^letter]	Foo\\u2160
[^letter]		letter	\u2160|Foo

The character \u2160 is ROMAN NUMERAL ONE, a numeric letter.

After splitting, if a word begins with a lower-case character then its first character is converted to an upper case. The final result is a word list in which each word is one of the following:

- A string of upper- and lower-case letters, the first character of which is upper case
- A string of digits
- A string of uncased letters and marks

Given an XML name in a word-list form, Java identifiers are mapped as follows:

- A class or interface identifier is constructed by concatenating the words in the list
- A method identifier is constructed by concatenating the words in the list.
- A prefix verb (example: get, set, is) is prepended to the result or if no prefix is required, the first character is converted to lower case.

This mapping does not change an XML name that is already a legal and conventional Java class, method, or constant identifier, except perhaps to add an initial verb in the case of a property access method.

TABLE 20-2 Illustrative Examples

XML Name	Class Name	Method Name
mixedCaseName	MixedCaseName	mixedCaseName
name-with-dashes	NameWithDashes	nameWithDashes
name_with_underscore	Name_with_underscore	name_with_underscore
other_punct•chars	Other_punctChars	other_punctChars
Answer42	Answer42	answer42

21 Appendix: Change Log

21.1 Changes for the JAX-RPC 1.1 Maintenance Release

- Section 8.2.5: Added ServiceFactory.loadService(Class) method and a requirement on implementations so that applications can use it reliably and not depend on any particular implementation being used (modulo packaging/configuration, that is).
- Section 14.4: Updated this section to align it with the WS-I BP 1.0 Board Approval Draft. Detailed changes are: added R2028, R2029, R4003, R2754 to 14.4.1; added R2114 to 14.4.2; added R1141, R1132, R2250 to 14.4.3; moved R1125, R1113, R1114, R1115, R1130 from the second to the third group in 14.4.3.
- Section 4.2.4: Added “the enclosing xsd:attribute” as a basis for creating a name for the class corresponding to an anonymous enumeration type. Added a new clause requiring implementations to support, in addition to the default mapping, a JAXB-compatible mapping that treats an anonymous enumeration as a derivation by restriction. Also added an example for the anonymous case.
- Section 4.2.6: Changed the mapping of simple types derived using xsd:list to use arrays instead of java.util.List objects.
- Section 4.2.1: Modified the provision about “element declarations with nillable attribute set to true” to cover two more cases: (a) element declarations with minOccurs=”0” and maxOccurs=”1” or absent, and (b) attribute declarations with use=”optional” or absent and not specifying a default or fixed value.
- Section 4.2.3: Changed the name of the property corresponding to the content of a complex type defined using xsd:simpleContent to “_value”. Previously it was “value”.
- Section 4.2.3: Changed the name of the property corresponding to a wildcard to “_any”. Previously it was “any”.
- Section 4.2.3: Changed the mapping of wildcards (xsd:any) to javax.xml.soap.SOAPElement. Previously they were mapped to java.lang.Object. Also added an example.
- Section 14.3: Clarified that the interoperability requirements inherited from JAX-RPC 1.0 have been extended in this revision to include new ones motivated by the WS-I Basic Profile 1.0.
- Section 3.R01 and Chapter 14: Changed all references to “an interoperable JAX-RPC implementation” to “a JAX-RPC implementation”. Rationale: interoperability requirements are, quite simply, requirements that *all* JAX-RPC implementations must satisfy.
- Section 4.2.1: Removed note on javax.xml.namespace.QName being “a placeholder”.

- Section 4.2.1: Added table 4-2 to clarify the mapping of the remaining built-in XSD simple types
- Appendix 18: Specified that tools must be able to handle the xsd:import and xsd:include schema constructs.
- Section 4.3.12: Added name collision rules for Java enumeration classes.
- Section 4.3.12: Added name collision rules for Java classes derived from an xsd:element.
- Section 5.4.1: Added mapping of value types to complex types that use the xsd:sequence compositor. Fixed the example accordingly since it wasn't valid according to the XML Schema specification.
- Section 5.3.3: Deprecated the mapping of Byte[] to xsd:base64Binary.
- Section 8.2.5: Added the two new loadService() methods to javax.xml.rpc.ServiceFactory.
- Section 14.4 (new): Added WS-I BP 1.0 interoperability requirements.
- Section 14.3.6: Removed mapping to/from tentative SOAP 1.2 fault codes.
- Section 14.3.2: Removed literal representation from 1.0 interop requirements section.
- Section 12.1.4: Clarified that for one-way operations only the handleRequest methods are called.
- Section 10.1.2: Specified that, in the servlet-based case, implementations must return a HTTP response code before dispatching a one-way request to a servlet endpoint instance.
- Section 5.5.5: Specified which Java methods can be mapped to one-way operations.
- Section 8.2.4.1: Added a requirement that, in the case of SOAP/HTTP, invokeOneWay block until an HTTP response code has been received from the server or an error occurs.
- Section 5.5.5: Clarified that one-way operations cannot have wsdl:output and wsdl:fault elements.
- Section 6.6 (new): Specified mapping for header faults.
- Section 4.3.6: Specified mapping for literal faults (whose message part uses the "element" attribute instead of "type").
- Section 6.4: Added requirement to support rpc/literal.
- Section 6.4.3: Modified the example to make the alternative mappings clearer.
- Section 6.4.1: Restated the requirement (already present in 1.0) to be able to unwrap document/literal calls as its own form of mapping.
- Section 6.2: Removed the description of "wrappers" since it conflicts with the Basic Profile.
- Javadocs only: Added prefix field, getPrefix method and ternary constructor to the javax.xml.namespace.QName class.
- Appendix 18: Specified some schema constructs as truly optional, meaning that JAX-RPC tools don't have to handle them gracefully.
- Appendix 18: Updated the schema support matrix to mark the newly required features as required.
- Section 6.4.1: Added the ability to map any literal part to SOAPElement even in case it has a standard mapping in JAX-RPC, so that external data binding frameworks can be plugged onto JAX-RPC more easily.
- Sections 5.1.3 and 5.3.2: Added javax.xml.namespace.QName and java.net.URI to the standard Java classes for which a predefined mapping exists.

- Sections 4.2.3 and 4.2.4: Specified that for anonymous types the name to be used for the corresponding Java type must be derived from “the nearest enclosing xsd:element, xsd:simpleType or xsd:complexType”.
- Section 4.2.3: Specified mapping of xsd:any in complex type definitions to java.lang.Object (or an array thereof if maxOccurs is greater than 1).
- Section 4.2.3: Specified that attributes on complex types must be mapped to properties of the resulting JavaBean class.
- Section 4.2.3: Added mapping for complex types defined using xsd:simpleContent and xsd:extension together.
- Section 4.2.6 (new): Specified mapping for simple types defined using xsd:list.
- Section 4.2.5 (new): Specified mapping for simple types derived by restriction.
- Section 5.3.2: Added mapping for the java.net.URI.
- Section 4.2.1: Added mapping for the remaining built-in XML Schema simple types for which JAXB 1.0 defines a special mapping. Also, added a clause to explicitely excluded xsd:NOTATION, xsd:ENTITY and xsd:IDREF.
- Section 4.2.1: Added mapping for the xsd:anyURI type to both java.net.URI (available in J2SE 1.4) and java.lang.String (for compatibility with previous J2SE versions).
- Section 13.1.1: Clarified that constants defined on the Call and Stub interfaces can be used in lieu of the corresponding strings for the “javax.xml.rpc.security.auth.[username|password]” strings.
- Section 10.1.3: Added isUserInRole method to the ServletEndpointContext interface.
- Section 7.5: Specified that for image/gif attachments only decoding is required.
- Sections 4.2.3 and 5.3.4: Clarified that an element with an attribute of the form maxOccurs=”1” should not result in a array-valued property of a value type.
- Section 4.2.3: Clarified that boolean property in value types should follow the JavaBeans conventions, hence use “is” instead of “get” as a prefix.
- Section 4.3.11: Clarified name mapping for the get<name_of_wsdl:port> methods.
- Section 3 (R013): Required support for SAAJ 1.2.
- Removed reference to the TCK in section 3 (R05).
- Fixed the indentation of several XML fragments.
- Added a reference to the WS-I Basic Profile 1.0.
- Updated some obsolete references (JAXB, SOAP 1.2, etc.).
- Updated the release number to 1.1 throughout the document.

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054, U.S.A.
650 960-1300

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 650 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
650 960-1300
Intercontinental Sales: 650 688-9000