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, J2ME, 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 Specification 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#136252/Form ID#011801)
3. JAX-RPC Subset Overview 25
   3.1 Introduction 25
   3.2 Functionality Subset Overview 25
      3.2.1 Stub based Invocation 26
      3.2.2 Operation mode 26
      3.2.3 Data Types 26
      3.2.4 SOAP Fault 27
      3.2.5 Service Endpoint Model 27
      3.2.6 Extensible Type Mapping 27
      3.2.7 JAX-RPC Runtime Services 27
   3.3 J2ME Platform Specific Notes 27

4. JAX-RPC Subset Requirements 29
   4.1 Resources 29
   4.2 Alignment with WS-I Basic Profile 29
   4.3 Protocol Encoding 30
   4.4 Protocol Bindings 31
   4.5 Transport 31
   4.6 Supported Type System 32
   4.7 XML Encoding for SOAP Messages 32
   4.8 JAX-RPC Runtime System 32
   4.9 Default Type Mapping 33
   4.10 Extensible Type Mapping 33
   4.11 Service Endpoint Model 33
   4.12 Service Description 34
   4.13 Service Registration and Discovery 34
   4.14 Java API for XML Binding (JAXB) 34
4.15 Application level Modes of Interaction 34
4.16 Relationship to JAXM and SAAJ 35
4.17 Parameter Passing Semantics 36
4.18 Service Context 36
4.19 SOAP Messages with Attachments 36
4.20 SOAP Message Handler 36
4.21 Literal Mode 36
4.22 Application Portability 37

5. WSDL/XML to Java Mapping 39
5.1 XML Names 39
5.2 XML to Java Type Mapping 39
  5.2.1 Simple Types 40
    5.2.1.1 Mapping for xsd:float and xsd:double 41
  5.2.2 Array 41
  5.2.3 XML Struct and complex Type 42
  5.2.4 JavaBeans API 43
  5.2.5 Enumeration 43
  5.2.6 Simple Types Derived By Restriction 43
  5.2.7 Simple Types Derived Using xsd:list 43
5.3 WSDL to Java Mapping 43
  5.3.1 WSDL Document 43
  5.3.2 Extensibility Elements 44
  5.3.3 WSDL Port Type 44
  5.3.4 WSDL Operation 45
  5.3.5 Holder Classes 46
  5.3.6 WSDL Fault 46
  5.3.7 WSDL Port 47
  5.3.8 WSDL Service 48

6. SOAP Binding 49
B. References  83

C. Change Log  85
Preface

This document is the Java 2 Micro Edition Web Service Specification. The specification has been developed under the Java Community Process (JCP) version 2.1 as Java Specification Request 172 (JSR-172). Further details on this specification in relation to the JCP can be found at the following website:

http://jcp.org/jsr/detail/172.jsp

Expert Group

The authors would like to thank the following individuals and organizations for their contributions to the expert group:

- Chul Jin Ahn, LG Telecom
- Charlton Barreto, Inprise
- Sandeep Chatterjee, Hewlett-Packard
- Guillaume Comeau, Zucotto Wireless,
- Jim D’Ambrosia, BEA Systems
- Priyen Doshi, Insignia Solutions
- Cliff Draper, Sun Microsystems, Inc.
- Jean Finka, Smart Fusion
- Jamie Hayes, Hutchison 3G UK Limited
- Steve Jones, Netdecisions Holdings United
- Alan Kaplan, Panasonic Information and Network Technologies Laboratory
- Satoshi Kikuchi, Fujitsu Limited
- Paul Kline, Motorola
Acknowledgements

The authors would like to thank the following individuals for their help and encouragement during the writing of this specification, Roberto Chinnici, Danny Coward, Mark Hapner, Tim Lindholm, Bonnie Kellett, Roger Pease and Paul Sandoz.

This specification builds on the work of others, specifically JSR-63 Java API for XML Processing and JSR-101 Java API for XML based RPC. The authors would like to thank the respective expert groups and their specification leads.
Typographic Conventions

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. % You have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, when contrasted with on-screen computer output</td>
<td>% su</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new words or terms, words to be emphasized</td>
<td>Read Chapter 6 in the User's Guide. These are called class options. You must be superuser to do this.</td>
</tr>
<tr>
<td></td>
<td>Command-line variable; replace with a real name or value</td>
<td>To delete a file, type rm filename.</td>
</tr>
</tbody>
</table>

Feedback

Please send comments and feedback on this specification to:

jsr-172-comments@sun.com
CHAPTER 1

Goals

1.1 Overall Goal

Provide two new capabilities to the J2ME platform:

- access to remote SOAP / XML based web services
- parsing XML data

There is great interest and activity in the Java community in the use of web services standards and infrastructures to provide the programming model for the next generation of enterprise services. There is considerable interest in the developer community in extending enterprise services out to J2ME clients.

1.2 Main Goals and Deliverables

The main deliverables of the JSR-172 specification are two new, independent, optional packages:

1. an optional package adding XML Parsing support to the platform.

   Structured data sent to mobile devices from existing applications will likely be in the form of XML. In order to avoid including code to process this data in each application, it is desirable to define an optional package that can be included with the platform.

2. Create an optional package to facilitate access to XML based web services from CDC and CLDC based profiles.
This optional package will define an API to allow mobile devices to access remote XML based web services. Where possible it will avoid defining new network protocols and formats and reuse existing standards.

**Note** – J2ME Optional Packages are described in JSR-68, J2ME Platform Specification [1].

---

### 1.3 XML Parsing sub-goals

The XML parsing optional package has the following additional goals:

- Strict subset wherever possible of JSR-063 JAXP 1.2 [2] functionality
- Meet platform size requirements
  
  Ensure that the API fits within the footprint requirements of the target devices.
- Meet platform performance requirements
  
  Ensure that the API can be implemented within the runtime memory and processing requirements for the target devices.

---

### 1.4 Web Services sub-goals

The web services optional package has the following additional goals:

- Subset JAX-RPC 1.1 functionality
- Separate deliverable from XML Parsing optional package

  The web services API optional package should not depend on the XML parsing optional package. It MUST be possible to deliver the web services optional package independent of XML parsing
- Provide access to web services from J2ME; no server capabilities

  This JSR will not define web service endpoints for the target devices. This functionality may be addressed in a future version of the specification.
- Meet platform size requirements
  
  Ensure that the API fits within the footprint requirements of the target devices.
- Meet platform performance requirements
Ensure that the API can be implemented within the runtime memory and processing requirements for the target devices.
JAXP Subset

2.1 Overall Goal

The goal of this optional package is to define a strict subset wherever possible of the XML parsing functionality defined in JSR-063 JAXP 1.2 [2] that can be used on the Java 2 Micro Edition Platform (J2ME) [1].

2.2 Why XML on J2ME?

XML is becoming a standard means for clients to interact with backend servers, their databases and related services. With its platform neutrality and strong industry support, XML is being used by developers to link networked clients with remote enterprise data. An increasing number of these clients are based on the J2ME platform, with a broad selection of mobile phones, PDAs, and other portable devices. As developers utilize these mobile devices more to access remote enterprise data, XML support on the J2ME platform is becoming a requirement.

2.3 Platform Requirements

In order to provide implementations that are useful on the widest possible range of configurations and profiles, this specification is treating the Connected Limited Device Configuration (CLDC) 1.0 [3] as the lowest common denominator platform.
The target size for a complete implementation, including interfaces, is 35Kb. See “Change Log” on page 85 for notes on requirement changes during the development of this specification.

**Note** — size is defined as being the sum of the uncompressed Java classfiles required for both the interface definitions and their implementations.

### 2.4 API Requirements

This optional package is a strict subset of the JAXP 1.2 specification. The following modifications define the functionality subset:

- an implementation **MUST NOT** provide any support for the Simple API for XML Parsing (SAX) 1.0 [4] interfaces
  
  SAX 1.0 has been superseded by SAX 2.0 which is included.

- an implementation **MUST NOT** provide any support for Document Object Model (DOM) Level 1.0 or 2.0
  
  The DOM is generally considered to be too heavy, both in terms of implementation size and runtime memory footprint, to be used on the J2ME platform.

- an implementation **MUST NOT** provide any support for XSLT

- an implementation **MUST** support the SAX 2.0 subset defined in this specification

- an implementation **MUST** provide support for XML namespaces [6]

- an implementation **MUST** provide support for both **UTF-8** and **UTF-16** encodings

### 2.5 Validating Parser Support

An implementation **MAY** support validation of XML documents against a DTD. XML validation is an expensive process in terms of processing power and memory usage and would likely not be supported on most J2ME devices. However, if the platform has the ability to support it, it **MAY** provide a validating parser (due to the limited nature of most J2ME devices, it is expected only one parser will be supported, but it is allowable to support both).
If an implementation provides a validating XML parser, that parser MUST conform to the XML 1.0 Specification [5] per section 5.1 in respect to Validating processors.

2.6 Non-validating Parser Support

Most J2ME platforms will likely provide a non-validating XML parser, as it is much less resource intensive than its validating counterpart.

If an implementation provides a non-validating XML parser, it MUST conform to Section 5.1 of the XML 1.0 Specification [5], which only requires implementations parse the internal DTD subset:

“[non-validating parsers] are required to process all the declarations they read in the internal DTD subset and in any parameter entity that they read, up to the first reference to a parameter entity that they do not read; that is to say, they must use the information in those declarations to normalize attribute values, include the replacement text of internal entities, and supply default attribute values. Except when standalone="yes", they must not process entity declarations or attribute-list declarations encountered after a reference to a parameter entity that is not read, since the entity may have contained overriding declarations.”

Although there is no requirement to parse external entities, the parser MUST notify the application of unparsed entities via the DefaultHandler method skippedEntity.

2.7 Conformance Requirements

In order to achieve a predictable environment for applications, implementations of this specification must meet certain conformance requirements.

Implementations of this specification MUST be conformant to Section 5.1 of the XML 1.0 recommendation. In addition, implementations MUST be conformant to Section 6 of the XML Namespaces recommendation as well as the SAX 2.0 interfaces. See Section 2.4 of this specification for references to the relevant specifications.
2.8 JAXP Subset APIs

There are three packages which comprise the JAXP API subset:
- javax.xml.parsers
- org.xml.sax
- org.xml.sax.helpers

When inspecting the API set, one will quickly notice that much of what exists in the J2SE JAXP API set is missing from the J2ME JAXP API set. The size requirements for the J2ME platform are strict, allowing only approximately 35Kb for a complete JAXP implementation. However, although many of the classes are gone, much of the functionality remains.

2.8.1 javax.xml.parsers

The javax.xml.parsers package contains the classes to obtain and reference a platform's given parser implementation. The package contains four classes:
- SAXParser
- SAXParserFactory
- FactoryConfigurationError
- ParserConfigurationException
2.8.2 org.xml.sax

The org.xml.sax package contains a subset of the SAX 2.0 API classes and interfaces. The interfaces included in this package are:

- Attributes
- Locator

The classes included in this package are:

- InputSource
- SAXException
- SAXNotRecognizedException
- SAXNotSupportedException
- SAXParseException

2.8.3 org.xml.sax.helpers

The org.xml.sax.helpers package contains a class for applications to extend to receive parse events:

- DefaultHandler
JAX-RPC Subset Overview

This chapter gives an overview of the functionality provided by the JAX-RPC Subset.

3.1 Introduction

JAX-RPC is a Java API for interacting with SOAP [7] based web services. This specification defines a subset of the JAX-RPC 1.1 [8] specification that is appropriate for the J2ME platform.

The functionality provided in the subset reflects both the limitations of the platform; memory size and processing power, as well as the limitations of the deployment environment; low bandwidth and high latency.

3.2 Functionality Subset Overview

The following sections provide an overview of the functionality of the JAX-RPC subset. Subsequent chapters expand upon the functionality provided. In all cases the functionality is a subset of that described in JAX-RPC 1.1 specification. Unless otherwise noted the behavior of the JAX-RPC subset runtime is identical to that described in the JAX-RPC 1.1 specification.
3.2.1 Stub based Invocation

Implementations MUST support local stub based (also known as static stubs) invocation via the Stub interface. There is no support in the subset for dynamic proxies or dynamic invocation interface (DII). Support for dynamic proxies and DII will be evaluated for inclusion in future versions of this specification. See section 7.2, 'Client Side APIs'.

3.2.2 Operation mode

Implementations MUST generate stubs that use the document style and literal use (document/literal). See section 6.2, 'Operation Style attribute'.

Note – this does restrict the web services that can be interacted with but greatly simplifies implementation. SOAP 1.2 [9] is moving away from the SOAP encoding [SOAP 1.2 Part 2 Section 4.1].

3.2.3 Data Types

Implementations MUST support WSDL documents referencing the following data types:

- boolean
- byte
- short
- int
- long
- float
- double
- String
- complex types
- arrays of primitive and complex types

See section 5.2.3, 'XML Struct and complex Type' for details on complex types.
3.2.4 SOAP Fault

A SOAP fault MUST be mapped to a service specific Java exception or a RemoteException. The SOAPFaultException is not included in the API subset. See section 5.3.6, ‘WSDL Fault’.

3.2.5 Service Endpoint Model

There is no support for the service endpoint model. The subset only provides support for clients to access web service endpoints. Limited devices are not currently capable of acting as service endpoints. The JAX-RPC subset does not include the javax.xml.rpc.server package.

3.2.6 Extensible Type Mapping

There is no support for extensible type mapping. The subset does not include the javax.xml.rpc.encoding package.

3.2.7 JAX-RPC Runtime Services

Implementations MUST support HTTP Basic Authentication and Session Management as defined by JAX-RPC 1.1 section 13.1.1, “HTTP Basic Authentication” and section 13.2, “Session Management” respectively.

3.3 J2ME Platform Specific Notes

The following classes and interfaces are included in the J2ME Web Services Optional Package to satisfy dependencies of JAX-RPC on the CLDC based platforms:

- java.rmi.Remote
- java.rmi.RemoteException
- java.rmi.MarshalException
- java.rmi.ServerException
JAX-RPC Subset Requirements

The following requirements are based on the requirements for JAX-RPC and have been updated to reflect the subset of JAX-RPC functionality defined by this specification. Additional requirements, specific to the JAX-RPC Subset, are also listed.

4.1 Resources

An implementation of the JAX-RPC subset defined by this specification has the following minimum requirements:

50 KB of RAM
25 KB of ROM

These requirements are in addition to those defined for the base platform.

4.2 Alignment with WS-I Basic Profile

The WS-I Basic Profile [19] (WS-I BP) provides recommendations and clarifications for many specifications referenced by this specification, and its superset - JAX-RPC 1.1. To provide interoperability with other web services implementations, JAX-RPC Subset implementations MUST follow the recommendations of the WS-I BP where they overlap with the functionality defined in this specification.
4.3 Protocol Encoding

In order to provide interoperability with the existing web services infrastructure, implementations of the JAX-RPC subset MUST communicate with web service endpoints using SOAP 1.1 encoded messages using an XML based protocol. In FIGURE 4-1 this requirement corresponds to messages flowing to the web service endpoint through the internet / intranet.

Due to limitations of the wireless environment (processing capacity of the devices, latency and bandwidth limitations of the network) there is no requirement that the JAX-RPC subset runtime on the client device support an XML encoding.

If an implementation of the JAX-RPC runtime does not produce and consume XML based protocol encoded messages this MUST be transparent to the web service. In such cases the messages leaving the wireless carrier’s access network MUST be transformed into SOAP 1.1, XML encoded messages before being sent to the web service.

Conversely, SOAP 1.1, XML encoded messages entering the wireless carrier’s network (the response from a service endpoint) MUST be transformed so they can be understood by the device.
4.4 Protocol Bindings

A goal of the JAX-RPC subset specification is to enable support for multiple protocol bindings that are based on the XML Information Set (Infoset) [12]. 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.

An interoperable JAX-RPC subset implementation MUST support the SOAP 1.1 protocol. As noted in section 4.3 “Protocol Encoding”, a JAX-RPC subset runtime implementation is not required to produce and consume an XML 1.0 document based encoding. As a consequence, a JAX-RPC subset runtime implementation that does not produce and consume XML 1.0 documents MUST use a representation that can be transformed into interoperable SOAP 1.1 messages.

Note – The JAX-RPC subset defined in the specification will consider support for the SOAP 1.2 protocol when it is supported by a future version of the JAX-RPC 1.1 specification.

4.5 Transport

An implementation of the JAX-RPC subset’s runtime system MUST support HTTP 1.1 as the transport for SOAP messages. HTTP binding for the SOAP messages is based on the SOAP 1.1 specification [7].

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 the ability to deliver SOAP messages and has a defined protocol binding for the SOAP 1.1 protocol.

Security features available to a JAX-RPC implementation will also depend on the security features provided in the particular J2ME Configuration or Profile used for that implementation. For example the MIDP 2.0 [18] security framework.
The 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 subset runtime system implementation MAY support HTTP/S as the underlying secure transport.

A JAX-RPC subset runtime system implementation MUST additionally conform to the requirements related to “Use of SOAP in HTTP” in the WS-I BP[19], where those requirements overlap with the JAX-RPC Subset specification.

### 4.6 Supported Type System

The JAX-RPC subset specification implementations MUST have support for the following XML type mapping:
- XML types specified in section 5.2 “XML to Java Type Mapping”.

### 4.7 XML Encoding for SOAP Messages

The JAX-RPC subset specification implementations MUST support the literal representation of a SOAP message representing an RPC call or response.

The SOAP 1.1 encoding (also called SOAP 1.1 section 5 encoding) MUST NOT be supported.

A JAX-RPC subset runtime system implementation MUST additionally conform to the requirements related to “XML Representation of SOAP messages” in the WS-I BP[19], where those requirements overlap with the JAX-RPC Subset specification.

### 4.8 JAX-RPC Runtime System

The JAX-RPC Subset runtime system forms the core of a JAX-RPC implementation. JAX-RPC Subset runtime system is a client-side library that provides a set services required for the JAX-RPC Subset runtime mechanisms.
The JAX-RPC Subset specification only defines the J2ME client and does not define any server side requirements.

The JAX-RPC Subset specification requires that a client side JAX-RPC compatible implementation MUST be based on either the J2ME platform Connected, Limited Device Configuration 1.0 (CLDC 1.0) [3] or CLDC 1.1 or Connected Device Configuration 1.0 (CDC 1.0) [10].

The JAX-RPC Subset APIs define the programmatic interface to the JAX-RPC Subset runtime system. Refer to Chapter 7 “JAX-RPC Subset Core APIs” for the specification of the JAX-RPC Subset APIs.

4.9 Default Type Mapping

The JAX-RPC Subset specification specifies the following standard type mapping:

- XML data types to Java types

A JAX-RPC Subset runtime system implementation MUST support this standard type mapping. Refer to section 5.2 “XML to Java Type Mapping” for the specification of the standard type mapping.

4.10 Extensible Type Mapping

The JAX-RPC Subset specification does not provide support for extensible type mapping.

4.11 Service Endpoint Model

The JAX-RPC Subset specification does not specify a standard programming model for a service endpoint running on the J2ME platform.
4.12 Service Description

See Requirement R09, Service Description, of the JAX-RPC 1.1 specification.

A JAX-RPC subset runtime system implementation MUST additionally conform to the requirements related to “Service Description” in the WS-I BP[19], where those requirements overlap with the JAX-RPC Subset specification.

4.13 Service Registration and Discovery

See Requirement R010, Service Registration and Discovery, of the JAX-RPC 1.1 specification.

4.14 Java API for XML Binding (JAXB)

See Requirement R011, Java API for XML Binding (JAXB), of the JAX-RPC 1.1 specification.

4.15 Application level Modes of Interaction

The JAX-RPC subset specification implementations MUST support the following mode of interaction between a client and service endpoint. Note that this interaction mode is visible as part of the JAX-RPC programming model and termed the application level interaction mode.

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 a more primitive implementation specific interaction mode to implement support for the application level interaction mode.

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. This mode is conceptually the same as that used in the client/server model.

![Figure 4-2 Synchronous Request-response Mode](image)

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 [9] for more details on transport message exchange patterns and default HTTP binding.

The JAX-RPC Subset APIs and service client programming model support synchronous request-response mode through the stub based model.

### 4.16 Relationship to JAXM and SAAJ

The JAX-RPC Subset specification does not depend on either the JAXM 1.0 (Java API for XML messaging) specification or the SAAJ (SOAP with Attachments API for Java) API defined by JAXM 1.1.
4.17 Parameter Passing Semantics

See Requirement R014, Parameter Passing Semantics, of the JAX-RPC 1.1 specification.

4.18 Service Context

The JAX-RPC subset specification does not require implementations to provide support for service context.

4.19 SOAP Messages with Attachments

The JAX-RPC Subset specification does not support the use of SOAP messages with attachments as an underlying protocol.

4.20 SOAP Message Handler

The JAX-RPC Subset specification does not support SOAP message handlers.

4.21 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 Subset specification implementations MUST support the literal representation of an RPC request or response in the SOAP body. The encoded representation MUST NOT be supported. Refer to section 6.3 “Literal Representation” for more details.
4.22 Application Portability

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

Portability is achieved by defining a service provider interface (SPI) to the JAX-RPC Subset runtime. The generated implementation of a stub MUST interact with the runtime via this SPI. Refer to Chapter 8 “Runtime Service Provider Interface” for details and requirements.
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

Refer to the JAX-RPC 1.1 specification for illustrative examples

5.1 XML Names

See JAX-RPC 1.1 section 4.1, “XML Names”.

5.2 XML to Java Type Mapping

This section specifies the required standard type mapping of XML data types to the Java types. Implementations MUST support the specified types.

Note that the rules and format of serialization for XML data types are based on the encoding style.
5.2.1 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.

<table>
<thead>
<tr>
<th>Simple type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:string</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:int</td>
<td>int</td>
</tr>
<tr>
<td>xsd:long</td>
<td>long</td>
</tr>
<tr>
<td>xsd:short</td>
<td>short</td>
</tr>
<tr>
<td>xsd:boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>xsd:byte</td>
<td>byte</td>
</tr>
<tr>
<td>xsd:float</td>
<td>java.lang.String or float</td>
</tr>
<tr>
<td>xsd:double</td>
<td>java.lang.String or double</td>
</tr>
<tr>
<td>xsd:QName</td>
<td>javax.xml.namespace.QName</td>
</tr>
<tr>
<td>xsd:base64Binary</td>
<td>byte[]</td>
</tr>
<tr>
<td>xsd:hexBinary</td>
<td>byte[]</td>
</tr>
</tbody>
</table>

1. Support for xsd:base64Binary and xsd:hexBinary in the JAX-RPC Subset specification is only required at the level of SOAP messages as encoding for arrays of bytes.

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

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 table specifies the mapping of the element declarations for the above cases, for the built-in simple XML types.

<table>
<thead>
<tr>
<th>Element Declaration with nillable attribute set to true and the following type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:int</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>xsd:long</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>xsd:short</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>xsd:float</td>
<td>java.lang.String or java.lang.Float</td>
</tr>
<tr>
<td>xsd:double</td>
<td>java.lang.String or java.lang.Double</td>
</tr>
<tr>
<td>xsd:boolean</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>xsd:byte</td>
<td>java.lang.Byte</td>
</tr>
</tbody>
</table>

### 5.2.1.1 Mapping for `xsd:float` and `xsd:double`

The CLDC 1.0 configuration does not provide the float and double native types. In order to support `xsd:float` and `xsd:double` on CLDC 1.0, by default, implementations MUST generate code that maps these types to `java.lang.String`.

To support configurations and platforms (CLDC 1.1 and CDC) that do provide native support for float and double, stub generator implementations MUST also be able to generate code that maps these types to the appropriate native Java type.

How stub generators support these two modes of operation is implementation defined.

**Note** – stubs generated to use the native mapping to float and double cannot be loaded by CDLC 1.0 virtual machine implementations. Therefore, developers of J2ME applications targeting both CDLC 1.0 and CDLC 1.1 should use the default mapping to `java.lang.String`.

### 5.2.2 Array

An XML array is mapped to a Java array with the operator `[]`. JAX-RPC Subset specification implementations MUST support the following XML array definition:
- An array derived from any element where the maxOccurs attribute is a non-negative integer greater than 1 or “unbounded”.

Implementations MUST NOT support array definitions using soapenc:Array or wsdl:arrayType.

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 creation time rather than when the array is declared.

See section “5.2.3 soapenc:Array” of WS-I Basic Profile for examples of WSDL array declarations.

5.2.3 XML Struct and complex Type

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

- The xsd:complexType with sequences of simple type and complex types. Refer to the xsd:sequence[14]

An XML struct maps to a JavaBeans class with the same name as the type of the XML struct. The mapped JavaBeans class provides getter and setter methods for each property mapped from the member elements 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 in the XML struct. Refer to section 5.1 “XML Names”, for the mapping of XML names to Java identifiers.

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 (see section 5.2.4 “JavaBeans API”) class. The Java type of the array is mapped from the type attribute of the XML element. Refer to the JAX-RPC specification for examples.

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

See JAX-RPC 1.1 section 4.2.3, “XML Struct and Complex Type” for examples.
5.2.4 JavaBeans API

Not all J2ME profiles and configurations provide support for the JavaBeans APIs. All references made to functionality provided by the JavaBeans API are referring to the functionality provided by the Java 2 Standard Edition (J2SE).

JAX-RPC Subset runtime implementations are not required to provide support for JavaBeans APIs. The use of the JavaBeans APIs in the JAX-RPC Subset is restricted to development time tasks, for example, static stub generation.

5.2.5 Enumeration

The JAX-RPC Subset does not provide support for XML enumerations.

5.2.6 Simple Types Derived By Restriction

The JAX-RPC Subset does not provide support for deriving simple types by restriction.

5.2.7 Simple Types Derived Using xsd:list

The JAX-RPC Subset does not provide support for deriving simple types using xsd:list.

5.3 WSDL to Java Mapping

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

5.3.1 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 MUST 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 [15].

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.

5.3.2 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 Subset specification specifies mapping of the extensibility elements for SOAP. Refer to the Chapter 6 “SOAP Binding”. However, the JAX-RPC specification does not address mapping of any vendor specific extensibility elements. A JAX-RPC Subset implementation MAY support mapping of WSDL extensibility elements at the cost of interoperability and application portability.

5.3.3 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 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 section 5.1 “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 section 5.3.4 “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.

A method MAY also throw service specific exceptions based on the mapping of a WSDL faults. Refer to section 5.3.6 “WSDL Fault” for more details.

5.3.4 WSDL Operation

A wsdl:operation defined in a wsdl:portType maps to a Java method on the mapped Java service endpoint interface. The JAX-RPC subset mapping of a wsdl:operation to a Java method MUST include the use of the wsdl:binding element. Refer to 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 section 5.1 “XML Names” for the mapping of XML names to Java identifiers.

In line with the WS-I Basic Profile, the JAX-RPC Subset does not support the overloading of operation names within a wsdl:portType. Consequently wsdl:operation names MUST be unique within a wsdl:portType.

The JAX-RPC subset specification implementations MUST support the mapping of operations with request-response. The standard Java mapping of operations defined with other transmission primitives (notification, solicit-response, one-way) is considered out of scope in the JAX-RPC subset 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 section 5.3.6 “WSDL Fault” for more details on the Java mapping of WSDL faults.

Parameter Passing Modes

The JAX-RPC subset 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 WS-I Basic Profile makes the following requirements for the parameterOrder:

- The order of the parts in a message MUST be the definitive order of the part elements on the wire
- The parameterOrder may be used as a hint to indicate the return value and method signatures
- If the parameterOrder attribute is present, it MUST omit at most one part from the output message
- If no part is omitted there is no return value

The JAX-RPC specification specifies the following rules for the in 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.

5.3.5 **Holder Classes**

The JAX-RPC Subset does not include Holder classes (see JAX-RPC 1.1 section 4.3.5, “Holder Classes”). Holders are not required as the WS-I Basic Profile limits the SOAP body for document/literal requests to at most one message part. See Chapter 6 “SOAP Binding” for details.

5.3.6 **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.

In the JAX-RPC Subset a `wsdl:fault` MUST be mapped to either a `java.rmi.RemoteException` (or its subclass), or a service specific Java exception (described later in this section). Refer to section 6.4 “SOAP Fault” for more details on the Java mapping of a WSDL fault based on the SOAP binding.

Refer to JAX-RPC 1.1 section 14.3.6, “Mapping of Remote Exceptions” for the mapping between the standard SOAP faults [9] 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.

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`.

See JAX-RPC 1.1 section 4.3.6, “WSDL Fault” for examples.

---

### 5.3.7 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 Subset service client programming model, a service endpoint (defined using `wsdl:port`) is accessed using an instance of a generated stub class.
5.3.8 WSDL Service

The JAX-RPC Subset does not provide a Java mapping for \textit{wsdl:service}. \textit{wsdl:port} elements are only accessible through an instance of a generated Stub class.
CHAPTER 6

SOAP Binding

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

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 Subset specification implementations MUST support the following case (termed operation mode) for an operation with the SOAP binding. Later sections of this chapter specify more details:

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

Refer to the WSDL 1.1 [17] specification for more details on the document operation style.

A JAX-RPC Subset implementation MUST use the above operation mode for the mapping of a WSDL based service description to the corresponding Java representation.

A JAX-RPC Subset implementation MUST additionally conform to the requirements related to “SOAP Binding” in the WS-I BP[19], where those requirements overlap with the JAX-RPC Subset specification.
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 Subset programming model, document style operations are mapped to the corresponding remote methods on a service endpoint interface. The rpc operation style is not supported.

A JAX-RPC subset client side implementation MUST support the use of services that follow the JAX-RPC (refer to the beginning of this section) specified document 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 Literal Representation

JAX-RPC Subset specification implementations MUST use the document style of operation, with the literal use. Note that the literal use is defined on the `soap:body` element in the WSDL.

6.3.1 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. Refer to the section 5.2 “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 composer 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. See section 6.4.3 of the
JAX-RPC 1.1 specification for an example.

6.4 SOAP Fault

This section specifies the mapping of 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 use. According to the WSDL 1.1 specification, the soap:fault element
MUST contain only a single message part.

The JAX-RPC Subset does not provide a SOAPFaultException class. A SOAP fault
is mapped to either a service specific exception class or a
java.rmi.RemoteException.
JAX-RPC Subset Core APIs

This chapter specifies JAX-RPC Subset APIs that support the JAX-RPC runtime mechanisms. These APIs are packaged in the javax.xml.rpc package.

7.1 Server side APIs

The JAX-RPC Subset specification does not define any server side APIs. Refer to the JAX-RPC 1.1 specification for details on the server side APIs.

7.2 Client Side APIs

The JAX-RPC Subset specifies the following client side APIs:
- The javax.xml.rpc.Stub interface
- The javax.xml.rpc.JAXRPCException class
- The javax.xml.rpc.NamespaceConstants class

A JAX-RPC Subset runtime system implementation MUST implement the above APIs.

7.2.1 Generated Stub Class

A WSDL to Java mapping tool generates a stub class during the import of a service described in a WSDL document.
All generated stub classes are required to implement the javax.xml.rpc.Stub interface. An instance of a stub class represents a stub instance for the target service endpoint.

A generated stub class is required to implement a service endpoint interface. The name of a generated stub class is <BindingName>_Stub. 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 Subset specification requires that a generated stub class be binding and transport neutral. A stub class should be bound to a Service Provider Interface, see Chapter 8 “Runtime Service Provider Interface for details.

7.2.2 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.

7.3 J2ME based Service Client Programming Model

A J2ME based service client uses generated stub classes to access a service. The following steps are required for a JAX-RPC Subset client to interact with a web service endpoint:
- generate a stub from the WSDL description of the service
- instantiate an instance of the stub
- invoke methods on the stub corresponding to the service endpoints’ wsdl:operation implementation
- package the stub with the J2ME client application
7.3.1 Stub Generation

The code for a stub is generated at development time. The implementation of the Stub interface MUST extend javax.xml.rpc.Stub. The J2ME developer uses a tool that reads the WSDL description of the service that the client will access and generates the appropriate Java code. See Chapter 5 “WSDL/XML to Java Mapping” for details.

Code generated by the stub generator uses the JAX-RPC Subset runtime SPI to invoke service operations. A stub MUST only use the SPI to interact with the runtime. See Chapter 8 “Runtime Service Provider Interface”.

7.3.2 Stub Instantiation

A J2ME client application instantiates an instance of the stub as a ‘proxy’ for the service from which it was created. The client program simply creates a new instance of the stub class generated by the stub generator:

```
StockQuoteService_Stub stub = new StockQuoteService_Stub();
```

CODE EXAMPLE 7-1 creating an instance of a generated stub.

7.3.3 Stub Operations

A J2ME client application uses an instance of a stub to:

- set properties of the stub, for example, a user name and password for HTTP Basic authentication, or setting an endpoint address
- invoke operations at the service endpoint

The following example shows how a client application sets properties on a Stub instance:

```
stub._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
    "http://quotes-r-us.com:8080:/quoteservice/");
```

CODE EXAMPLE 7-2 Setting an endpoint address property of a Stub instance

The client application uses the stub to invoke an operation, in this example the ‘getMostActive’ operation is provided by the service:

```
StockQuote[] mostActive = stub.getMostActive();
```

CODE EXAMPLE 7-3 Invoking an operation on a service endpoint via Stub instance
7.3.4 Packaging

A J2ME client application that makes use of a stub to invoke operations on a service endpoint MUST include the generated stub, along with any other generated artifacts, in its deployment bundle.
CHAPTER 8

Runtime Service Provider Interface

This chapter describes the service provider interface (SPI) of the JAX-RPC subset runtime. See the JavaDoc for the javax.microedition.xml.rpc package for a complete description of the classes and interfaces that make up the SPI.

8.1 Overview

The JAX-RPC subset runtime SPI is used by generated stubs to execute RPC calls, it is not intended to be used by application developers. The SPI only supports the functionality described in the JAX-RPC subset, for example, there is no support for DII and dynamic proxies. The SPI is defined by the Type, Element, ComplexType and Operation classes.

FIGURE 8-1  JAX-RPC Subset Runtime SPI classes
The Type, Element and ComplexType classes are used by the stub to describe the input parameters and return type of an RPC, to the runtime. An object graph of these classes represents a description of the serialization of the values in a complex type.

8.2 Invoking an RPC

A stub uses the SPI to perform the following steps:
- set properties that are needed to invoke an RPC
- create an object graph describing the input parameters
- create an object graph describing the return parameters
- create an Operation object representing an invocation of an RPC
- encode the input parameter values
- invoke the RPC service endpoint
- decode return values from the RPC invocation

The following sections describe these steps in more detail.

8.2.1 Setting Properties

The setProperty method of the Operation class is called by a generated stub to set property values for an RPC invocation. The stub is responsible for converting the property value into the appropriate string format.

The following table describes the property names that MUST be supported:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stub.ENDPOINT_ADDRESS_PROPERTY</td>
<td>Target service endpoint address</td>
</tr>
<tr>
<td>Stub.PASSWORD_PROPERTY</td>
<td>Password for authentication</td>
</tr>
<tr>
<td>Stub.USERNAME_PROPERTY</td>
<td>Username for authentication</td>
</tr>
<tr>
<td>Stub.SESSION_MAINTAIN_PROPERTY</td>
<td>Indicate whether or not a client wants to participate in a session with a service endpoint</td>
</tr>
<tr>
<td>Operation.SOAPACTION_URI_PROPERTY</td>
<td>URI to use for HTTP SOAPAction header</td>
</tr>
</tbody>
</table>
An IllegalArgumentException MUST be thrown by the runtime if a supplied property is not supported.

The WS-I Basic Profile requires that implementations set the SOAPAction HTTP header. The value MUST be a quoted string and be the empty string ("") if the WSDL description for an operation does not include a soapAction.

### 8.2.2 Input Parameters and Return Types

A stub uses the `Type`, `Element` and `ComplexType` classes to describe the parameters and return type of an operation to the runtime. `Type` is type safe enumeration of the allowable parameter data-types. The `Element` class is used to describe additional properties of a parameter. The properties mirror those that can be defined in XML Schema, and referenced in the WSDL description of an element, `type`, `name`, `minOccurs`, `maxOccurs` and `nillable`. Complex types, consisting of a sequence of elements are described to the runtime using the `ComplexType` class. As with the a XML Schema complexType a `ComplexType` contains a sequence of `Elements`.

The following example shows the WSDL, object tree and Java class representations of the objects `EmployeeInfo` and `Name`. An `EmployeeInfo` object contains the employee’s name, an emergency contact name, both of type `Name`, and employee id number, of type integer:

```xml
<xsd:complexType name="NameType">
    <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string"/>
        <xsd:element name="lastName" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="EmployeeInfoType">
    <xsd:sequence>
        <xsd:element name="empName" type="xsd1:NameType"/>
        <xsd:element name="emergencyContact" type="xsd1:NameType"/>
        <xsd:element name="empID" type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="EmployeeInfo" type="xsd1:EmployeeInfoType"/>
```

**CODE EXAMPLE 8-1**  WSDL description of `EmployeeInfo` and `Name` types

```java
ComplexType nameType = new ComplexType();
```

Chapter 8  Runtime Service Provider Interface  8-59
nameType.elements = new Element[2];
nameType.elements[0] = new Element(
    new QName(tns, "firstName"),
    Type.STRING);
nameType.elements[1] = new Element(
    new QName(tns, "lastName"),
    Type.STRING);

ComplexType employeeInfoType = new ComplexType();
employeeInfoType.elements = new Element[3];
employeeInfoType.elements[0] = new Element(
    new QName(tns, "empName"),
    nameType);
employeeInfoType.elements[1] = new Element(
    new QName(tns, "emergencyContact"),
    nameType);
employeeInfoType.elements[2] = new Element(
    new QName(tns, "empID"),
    Type.INT);

Element EmployeeInfo = new Element(
    new QName(tns, "EmployeeInfo"),
    employeeInfoType);

CODE EXAMPLE 8-2  description of EmployeeInfo and Name types passed by a stub to JAX-RPC subset runtime

class Name {
    String firstName;
    String lastName;
}
class EmployeeInfo {
    Name empName;
    Name emergencyContact;
    int empID;
}

CODE EXAMPLE 8-3  EmployeeInfo and Name Java classes

See Appendix A “SPI Use Cases” for further examples.
For each element referenced in the WSDL an equivalent `Element` object is created. The `contentType` of an `Element` MUST be set to, either the corresponding primitive type (enumerated in the class `Type`) or, in the case of a nested complex type, to an instance of `ComplexType` describing the complex type. A `ComplexType` contains an array of `Element` objects, one for each element of the WSDL sequence. The `Elements` appear in the array in the same order as they are defined in the XML Schema sequence referenced in the WSDL.

A `ComplexType` representing an empty sequence contains an empty array of `Element` objects. The `Operation` method `invoke` MUST throw a `JAXRPCException` if the `elements` field of a `ComplexType` is `null`.

8.2.3 Creating an Operation Instance

The `Operation` class is used by a `Stub` instance to describe an RPC invocation to the runtime. The `Operation` static method `newInstance` returns an `Operation` object for a specific `wsdl:operation`. In addition to a `QName` object containing the qualified name of the operation, `Element` objects describing the input parameters and return type of the operation are passed to the runtime via `newInstance`.

```java
Operation op = Operation.newInstance(
    new QName(tns, "promoteEmployee"),
    employee,
    retVal);
```

**CODE EXAMPLE 8-4** Creating an instance of `Operation` for the `wsdl:operation` 'promoteEmployee'.

The name of an operation is the same as an element defined in the type section of the WSDL and referenced as a message part. From R2712 of WS-I BP:

A document-literal binding MUST be represented on the wire as a MESSAGE with a `soap:Body` whose child element is an instance of the global element declaration referenced by the corresponding `wsdl:message` part.

See Appendix A “SPI Use Cases” for further examples.

8.2.4 Encoding Input Parameters

The input parameter values are passed to the runtime encoded in a array of `Object`. Each element in the array is either the corresponding Java object wrapper for a primitive type (`Boolean`, `Short`, `Integer`, etc) or an array of `Object`, representing the fields of a complex type.
The mapping for each of the supported types, and arrays of those types, is shown in TABLE 8-2.

**TABLE 8-2**  mapping of Element types to encoded types

<table>
<thead>
<tr>
<th>Element Type</th>
<th>isNull</th>
<th>isArray</th>
<th>Encoded type</th>
<th>isNull</th>
<th>isArray</th>
<th>Encoded type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type.BOOLEAN</td>
<td>false</td>
<td>false</td>
<td>Boolean</td>
<td>true</td>
<td>false</td>
<td>boolean[]</td>
</tr>
<tr>
<td>Type.BYTE</td>
<td>false</td>
<td>false</td>
<td>Byte</td>
<td>true</td>
<td>false</td>
<td>byte[]</td>
</tr>
<tr>
<td>Type.SHORT</td>
<td>false</td>
<td>false</td>
<td>Short</td>
<td>true</td>
<td>false</td>
<td>short[]</td>
</tr>
<tr>
<td>Type.INT</td>
<td>false</td>
<td>false</td>
<td>Integer</td>
<td>true</td>
<td>false</td>
<td>int[]</td>
</tr>
<tr>
<td>Type.LONG</td>
<td>false</td>
<td>false</td>
<td>Long</td>
<td>true</td>
<td>false</td>
<td>long[]</td>
</tr>
<tr>
<td>Type.FLOAT</td>
<td>false</td>
<td>true</td>
<td>Float</td>
<td>true</td>
<td>false</td>
<td>float[]</td>
</tr>
<tr>
<td>Type.DOUBLE</td>
<td>false</td>
<td>true</td>
<td>Double</td>
<td>true</td>
<td>false</td>
<td>double[]</td>
</tr>
<tr>
<td>Type.STRING</td>
<td>false</td>
<td>true</td>
<td>String</td>
<td>true</td>
<td>false</td>
<td>Object[]</td>
</tr>
<tr>
<td>ComplexType</td>
<td>false</td>
<td>true</td>
<td>Object</td>
<td>true</td>
<td>false</td>
<td>Object[]</td>
</tr>
</tbody>
</table>

1. implementations on platforms without native float and double support map Type.FLOAT and Type.DOUBLE.
2. see Section 5.2.1.1 “Mapping for xsd:float and xsd:double”.
3. the encoded type is an Object array, each element of the array is a String.

The following example shows the encoding of the EmployeeInfo type described in CODE EXAMPLE 8-1.

```java
Object[] empName = new Object[] {"John", "Doe"};
Object[] emergencyContact = new Object[] {"Jane", "Doe"};
Object[] employee = new Object[] {empName, emergencyContact, new Integer(empID)};
```

**CODE EXAMPLE 8-5**  Encoding input parameters

Nested in the Object array employee are two Object arrays and an Integer, containing the employee ID, empID. The two nested Object arrays contain the String values for the employee name, empName, and emergency contact name, emergencyContact, respectively.

```java
Object[] name = new Object[] {"John", "Doe"};
Object[] name2 = new Object[] {"Skip", "Barber"};
Object[] emp = new Object[] {name, name2, new Integer(420)};
```
Object[] emp2 = new Object[] {name2, name, new Integer(44)};
Object[] empArray = new Object[] {emp, emp2};
Object[] employee = new Object[] {empArray}

**CODE EXAMPLE 8-6** encoding for an array of Employee objects

Parameters which are arrays are encoded such that the array itself is a single element in the object array. In the above example, CODE EXAMPLE 8-6, the last line shows the values of an array of employee objects wrapped in a single element array. See Section A.2.2 “boolean addGroups(String []), Section A.2.5 “Employee[] getEmployees(Name [])” and Section A.2.6 “boolean scheduleMtg(Employee [])” for further examples.

The encoding of parameters that are null or the empty array detailed in **TABLE 8-3**.

---

**TABLE 8-3** Encoding of null and empty array parameters

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>nillable = “false”</th>
<th></th>
<th>nillable = “true”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>maxOccurs = “1” minOccurs = “0”</td>
<td>maxOccurs &gt; “1” minOccurs = “0”</td>
<td>maxOccurs = “1” minOccurs = “0”</td>
</tr>
<tr>
<td>null</td>
<td>omit element</td>
<td>invalid^2</td>
<td>xs: nil “true”</td>
</tr>
<tr>
<td>Empty Array</td>
<td>invalid^1</td>
<td>invalid^3</td>
<td>invalid^1</td>
</tr>
</tbody>
</table>

1. parameter cannot be an array, maxOccurs = “1”  
2. parameter cannot be null and minOccurs is non-zero. However, if nillable=“true” then individual array elements may be null.  
3. parameter must have non-zero number of array elements as minOccurs is non-zero

An omitted element, where maxOccurs = “1”, is simply not included in the SOAP body. An omitted array element, where maxOccurs is greater than 1, or unbounded, is encoded in the SOAP body as an empty array body.

For the cArrayType element defined in the WSDL shown in CODE EXAMPLE 8-7, the SOAP body of an operation cCall, taking a cArray as a parameter, is shown in CODE EXAMPLE 8-8.

```xml
<element name="cArray" type="tns:cArrayType"/>
<complexType name="cArrayType">
    <sequence>
        <element name="c" type="string" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
</complexType>
```
The operation cCall has the following Java signature:

void cCall(String cArray[])

Calling cCall with a null cArray, or an empty array parameter results in the following SOAP body:

```xml
<tns:cCall>
  <tns:cArray/>
</tns:cCall>
```

8.2.5 Executing an RPC

A stub executes an operation using the invoke method passing it an Object array containing the parameter values for this call, or null if the operation does not take any parameters.

The following example shows an operation that takes the employee type as a parameter.

```java
try {
    Object out = op.invoke(employee);
    catch (JAXRPCException jre) {
        ...
    }
}
```

The method invoke returns the return value of the RPC, to the stub. See Section 8.2.6 “Decoding Return Values” for details.

8.2.6 Decoding Return Values

The return values are in an Object array, encoded in the same way as input parameters, see Section 8.2.4 “Encoding Input Parameters”. The stub decodes this Object array and extracts the values to pass back to the calling application. See Appendix A “SPI Use Cases” for examples.
Optional elements (where the minOccurs attribute is zero) may not appear in SOAP messages of responses. The corresponding Java field MUST be set to it's default initialized value. See the Java Language Specification [15] for details. Java fields of array type MUST be initialized to be an empty array.

8.2.7 Error Handling

As specified in Section 5.3.6 “WSDL Fault”, a SOAP fault is either mapped to a subclass of java.rmi.RemoteException, or a service specific exception.

java.rmi.RemoteException

In the case of a SOAP fault that maps to java.rmi.RemoteException, or a sub-class, a JAXRPCException is thrown by the Operation method invoke. The getLinkedCause method of JAXRPCException is used by the stub to retrieve the generated exception. The stub MUST then re-throw the linked exception (the cause) to the application.

Service Specific Exceptions

If the WSDL description of a service endpoint includes operations with custom faults (wsdl:fault) the generated stub MUST provide an implementation of the FaultDetailHandler interface. The implementation of this interface is used by the runtime to retrieve a description of the detail portion of the SOAP fault. When the stub creates an instance of the Operation class, it passes a reference to the FaultDetailHandler. If no custom faults are declared the Stub is not required to implement the FaultDetailHandler interface and is not required to supply a FaultDetailHandler when creating an instance of Operation.

If the SOAP fault does not map to java.rmi.RemoteException, or a sub-class, and the stub has passed a FaultDetailHandler object reference to the runtime, the runtime MUST call the FaultDetailHandler method handleFault passing the QName of the child element of the SOAP fault detail element. See Section 6.4 “SOAP Fault”. The method handleFault returns an Element, or sub-class object, describing the SOAP fault detail.

Using the description of the SOAP fault detail, the runtime encodes the SOAP fault detail in an Object array as described in Section 8.2.4 “Encoding Input Parameters”. The Object array, and the associated QName, are passed back to the stub in a FaultDetailException object linked, as the cause, in a JAXRPCException object. The stub decodes the Object array, as described in Section 8.2.6 “Decoding Return Values”, and throws the service specific exception to the application.
If there is no mapping for the SOAP fault detail element, `handleFault` will return null. The runtime throws a `JAXRPCException` with the child element of the soap detail element as the message.
Futures

The following items will be considered for future versions or maintenance releases of this specification:
- align with future versions of the JAX-RPC specification
- align with future versions of the WS-I Basic Profile
- align with SOAP 1.2
APPENDIX A

SPI Use Cases

The example in the appendix shows how parameters are passed from the Stub to the SPI runtime. The example includes simple types, arrays of simple types, complex types, arrays of complex types, and various combinations of all four. For reference the WSDL for the examples is included.

A.1 Sample WSDL

The following WSDL is for an endpoint named EmployeeDatabase. The endpoint provides the following services:

- getEmployeeCount - input parameter is a department id (integer) and the return is the number of employees in the department (integer)
- addGroups - input parameter is a set of department names (String []) and the return is a success code (boolean)
- isManager - input parameter is a name (Name type) and the return is a boolean indicating if the named employee is a manager (boolean)
- promoteEmployee - input parameter is an employee (Employee type) and returns a success code (boolean)
- getEmployees - input parameter is an set of employee names (Name [] type) and the return is a set of employee data (Employee [] type)
- scheduleMeeting - input parameter is a set of employees (Employee [] type) and the return is a success code (boolean)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions
targetNamespace="http://www.sun.com/JSR172UseCases"
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
name="JSR172_AppendixA"/>
```
<xsd:complexType name="NameType">
  <xsd:sequence>
    <xsd:element name="firstName" type="xsd:string"/>
    <xsd:element name="lastName" type="xsd:string"/>  
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="NameArrayType">
  <xsd:sequence>
    <xsd:element name="Name" maxOccurs="unbounded" minOccurs="1" nillable="true" type="xsd1:NameType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="EmployeeType">
  <xsd:sequence>
    <xsd:element name="empName" type="xsd1:NameType"/>
    <xsd:element name="empID" type="xsd:int"/>  
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="EmployeeArrayType">
  <xsd:sequence>
    <xsd:element name="Employee" maxOccurs="unbounded" minOccurs="1" nillable="true" type="xsd1:EmployeeType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="EmployeeArray" type="xsd1:EmployeeArrayType"/>

<xsd:element name="getEmployeeCount" type="xsd:int"/>

<xsd:element name="addGroups">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="group" maxOccurs="unbounded" minOccurs="1" nillable="true" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<message name="scheduleMtgRes">
  <part name="scheduleRes" element="xsd:RetVal"/>
</message>

<portType name="EmployeeDBPort">
  <operation name="getEmployeeCount">
    <input message="tns:getEmployeeCountReq"/>
    <output message="tns:getEmployeeCountRes"/>
  </operation>
  <operation name="addGroups">
    <input message="tns:addGroupsReq"/>
    <output message="tns:addGroupsRes"/>
  </operation>
  <operation name="isManager">
    <input message="tns:isManagerReq"/>
    <output message="tns:isManagerRes"/>
  </operation>
  <operation name="promoteEmployee">
    <input message="tns:promoteEmployeeReq"/>
    <output message="tns:promoteEmployeeRes"/>
  </operation>
  <operation name="getEmployees">
    <input message="tns:getEmployeesReq"/>
    <output message="tns:getEmployeesRes"/>
  </operation>
  <operation name="scheduleMtg">
    <input message="tns:scheduleMtgReq"/>
    <output message="tns:scheduleMtgRes"/>
  </operation>
</portType>

<binding name="EmployeeDBBinding" type="tns:EmployeeDBPort">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getEmployeeCount">
    <input name="getEmployeeCountReq">
      <soap:body use="literal"/>
    </input>
    <output name="getEmployeeCountRes">
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="addGroups">
    <input name="addGroupsReq">
      <soap:body use="literal"/>
    </input>
  </operation>
</binding>
<output name="addGroupsRes">
  <soap:body use="literal"/>
</output>
</operation>

<operation name="isManager">
  <soap:operation soapAction="http://www.sun.com/JSR172UseCases/isManager"/>
  <input name="isManagerReq">
    <soap:body use="literal"/>
  </input>
  <output name="isManagerRes">
    <soap:body use="literal"/>
  </output>
</operation>

<operation name="promoteEmployee">
  <input name="promoteEmployeeReq">
    <soap:body use="literal"/>
  </input>
  <output name="promoteEmployeeRes">
    <soap:body use="literal"/>
  </output>
</operation>

<operation name="getEmployees">
  <input name="getEmployeesReq">
    <soap:body use="literal"/>
  </input>
  <output name="getEmployeesRes">
    <soap:body use="literal"/>
  </output>
</operation>

<operation name="scheduleMtg">
  <input name="scheduleMtgReq">
    <soap:body use="literal"/>
  </input>
  <output name="scheduleMtgRes">
    <soap:body use="literal"/>
  </output>
</operation>
</binding>

<service name="EmployeeDatabase">
  <port binding="tns:EmployeeDBBinding" name="EmployeeDBPort">
    <soap:address
      location="http://localhost:8000/ccx/JSR172_AppendixA"/>
  </port>
</service>
</definitions>
A.2 Use Cases

The following code fragments show the stub code that is required to describe an operation to the runtime, how the input values are encoded, and how the return value is decoded.

In all cases the stub builds a description of the input parameters and return type using the Type, Element and ComplexType classes. This description and the operation name are used to create an instance of the Operation. When the operation is invoked the input parameter values are passed to the runtime encoded in an Object. The return value is also an Object which the stub then decodes.

Note – The alias “xsd1” has been used in place of the actual namespace, “http://www.sun.com/JSR172_AppendixA”.

Although not shown in the following code fragments, the stub code would also need to call the Operation method setProperty to set the endpoint address, SOAPAction header, etc.

```
operation.setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, 
    "http://localhost:8000/ccx/JSR172_AppendixA");
operation.setProperty(Operation.SOAPACTION_URI_PROPERTY, 
    "http://www.sun.com/JSR172UseCases/getEmployeeCount");
```

**CODE EXAMPLE A-1** setting Operation properties

A.2.1 int getEmployeeCount(integer)

```
Element getEmployeeCount = new Element( 
    new QName(xsd1, "getEmployeeCount"), 
    Type.INT);

Element empCount = new Element( 
    new QName(xsd1, "EmpCount"), 
    Type.INT);

Operation op = Operation.newInstance( 
    new QName(xsd1, 
```
"getEmployeeCount"),
getEmployeeCount,
empCount);

Object o = op.invoke(new Integer(57));
int retVal = ((Integer)o).intValue();

A.2.2 boolean addGroups(String [])

Element group = new Element(
    new QName(xsd1, "group"),
    Type.STRING,
    1,  // minOccurs
    Element.UNBOUNDED,  // maxOccurs unbounded
    true);  // nillable

ComplexType groupArray = new ComplexType();
groupArray.elements = new Element[1];
groupArray.elements[0] = group;

Element addGroups = new Element(
    new QName(xsd1, "addGroups"),
    groupArray);

Element retVal = new Element(
    new QName(xsd1, "RetVal"),
    Type.BOOLEAN);

Operation op = Operation.newInstance(
    new QName(xsd1, "addGroups"),
    addGroups,
    retVal);

Object[] groups = new Object[] {"Group1", "Group2", "Group3"};
Object[] groupArray = new Object[groups];
Object o = op.invoke(groupArray);

boolean success = ((Boolean)o).booleanValue();

A.2.3 boolean isManager(Name)

ComplexType nameType = new ComplexType();
nameType.elements = new Element[2];
nameType.elements[0] = new Element(
    new QName(xsd1, "firstName"),
    Type.STRING);
nameType.elements[1] = new Element(
    new QName(xsd1, "lastName"),
    Type.STRING);

Element isManager = new Element(
    new QName(xsd1, "isManager"),
    nameType);

Element retVal = new Element(
    new QName(xsd1, "RetVal"),
    Type.BOOLEAN);

Operation op = Operation.newInstance(
    new QName(xsd1, "isManager"),
    isManager,
    retVal);

Object[] name = new Object[] {"Guy", "Isamanjer"};

Object o = op.invoke(name);

boolean RetVal = ((Boolean)o).booleanValue();
### A.2.4 boolean promoteEmployee(Employee)

The Employee type is a Complex type made up of another complex type plus a simple type:

```java
ComplexType nameType = new ComplexType();
nameType.elements = new Element[2];
nameType.elements[0] = new Element(
    new QName(xsd1, "firstName"),
    Type.STRING);
nnameType.elements[1] = new Element(
    new QName(xsd1, "lastName"),
    Type.STRING);
ComplexType employeeType = new ComplexType();
employeeType.elements = new Element[2];
employeeType.elements[0] = new Element(
    new QName(xsd1, "empName"),
    nameType);
employeeType.elements[1] = new Element(
    new QName(xsd1, "empID"),
    Type.INT);
Element promoteEmployee = new Element(
    new QName(xsd1, "promoteEmployee"),
    employeeType);
Element retVal = new Element(
    new QName(xsd1, "RetVal"),
    Type.BOOLEAN);
Operation op = Operation.newInstance(
    new QName(xsd1, "promoteEmployee"),
    promoteEmployee,
    retVal);
Object[] name = new Object[] {"Guy", "Isdoinagreatjob"};
```
Object[] employee = new Object[] {name, new Integer(54)};

Object o = op.invoke(employee);

boolean success = ((Boolean)o).booleanValue();

A.2.5 Employee[] getEmployees(Name [])

ComplexType nameType = new ComplexType();
nameType.elements = new Element[2];
nameType.elements[0] = new Element(
    new QName(xsd1, "firstName"),
    Type.STRING);
nameType.elements[1] = new Element(
    new QName(xsd1, "lastName"),
    Type.STRING);

ComplexType nameArrayType = new ComplexType();
nameArrayType.elements = new Element[1];
nameArrayType.elements[0] = new Element(
    new QName(xsd1, "Name"),
    nameType,
    1, // minOccurs
    Element.UNBOUNDED, // maxOccurs unbounded
    true); // nillable

ComplexType employeeType = new ComplexType();
employeeType.elements = new Element[2];
employeeType.elements[0] = new Element(
    new QName(xsd1, "empName"),
    nameType);
employeeType.elements[1] = new Element(
    new QName(xsd1, "empID"),
    Type.INT);

ComplexType employeeArrayType = new ComplexType();
employeeArrayType.elements = new Element[1];

employeeArrayType.elements[0] = new Element(
    new QName(xsd1, "Employee"),
    employeeType,
    1, // minOccurs
    Element.UNBOUNDED, // maxOccurs unbounded
    true); // nillable

Element getEmployees = new Element(
    new QName(xsd1, "getEmployees"),
    nameArrayType);

Element retVal = new Element(
    new QName(xsd1, "EmployeeArray"),
    employeeArrayType);

Operation op = Operation.newInstance(
    new QName(xsd1, "getEmployees"),
    getEmployees,
    retVal);

Object[] name = new Object[] {"John", "Doe"};
Object[] name2 = new Object[] {"Skip", "Barber"};
Object[] nameArray = new Object[] {name, name2};
Object[] names = new Object[] {nameArray};

Object o = op.invoke(names);

Object[] employees = (Object[]) o;
Object[] empSet = (Object[]) employees[0];
Employee[] result = new Employee[empSet.length];
for (int i = 0; i < empSet.length; i++) {
    Object[] employee = (Object[]) empSet[i];
    Object[] empName = (Object[]) employee[0];
    Name n = new Name();
    n.setFirstName(empName[0]);
n.setLastName(empName[1]);
Employee e = new Employee();
e.setName(n);
e.setID(((Integer)employee[1]).intValue());
result[i] = e;
}

A.2.6 boolean scheduleMtg(Employee [])

ComplexType nameType = new ComplexType();
nameType.elements = new Element[2];
nameType.elements[0] = new Element(
    new QName(xsd1, "firstName"),
    Type.STRING);
nameType.elements[1] = new Element(
    new QName(xsd1, "lastName"),
    Type.STRING);

ComplexType employeeType = new ComplexType();
employeeType.elements = new Element[2];
employeeType.elements[0] = new Element(
    new QName(xsd1, "empName"),
    nameType);
employeeType.elements[1] = new Element(
    new QName(xsd1, "empID"),
    Type.INT);

ComplexType employeeArrayType = new ComplexType();
employeeArrayType.elements = new Element[1];
employeeArrayType.elements[0] = new Element(
    new QName(xsd1, "Employee"),
    employeeType,
    1,   // minOccurs
    Element.UNBOUNDED,  // maxOccurs unbounded
    true);  // nillable
Element scheduleMtg = new Element(
    new QName(xsd1, "scheduleMtg"),
    employeeArrayType);

Element retVal = new Element(
    new QName(xsd1, "RetVal"),
    Type.BOOLEAN);

Operation op = Operation.newInstance(
    new QName(xsd1, "scheduleMtg"),
    scheduleMtg,
    retVal);

Object[] name = new Object[]{"John", "Doe"};
Object[] emp = new Object[]{name, new Integer(420)};
Object[] name2 = new Object[]{"Skip", "Barber"};
Object[] emp2 = new Object[]{name2, new Integer(44)};
Object[] empArray = new Object[]{emp, emp2};
Object[] employees = new Object[]{empArray}

Object o = op.invoke(employees);

boolean success = ((Boolean)o).booleanValue();
References


[17] Web Services Description Language (WSDL) 1.1: http://www.w3.org/TR/wsdl

[18] Mobile Information Device Profile (MIDP) 2.0 http://jcp.org/jsr/detail/118.jsp

## Change Log

<table>
<thead>
<tr>
<th>Version</th>
<th>Change description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expert Draft v0.1</td>
<td>Initial revision</td>
</tr>
<tr>
<td>Expert Draft v0.2</td>
<td>• included external references</td>
</tr>
<tr>
<td></td>
<td>• expanded J2ME service client programming model</td>
</tr>
<tr>
<td></td>
<td>• clarified references to the JavaBeans API</td>
</tr>
<tr>
<td></td>
<td>• clarified Java Serialization Semantics</td>
</tr>
<tr>
<td></td>
<td>• included diagrams</td>
</tr>
<tr>
<td></td>
<td>• editorial updates</td>
</tr>
<tr>
<td>Community Review v0.3</td>
<td>• attempted to clarify protocol and encoding requirements</td>
</tr>
<tr>
<td></td>
<td>• moved section on JavaBeans closer to the first reference to the API</td>
</tr>
<tr>
<td></td>
<td>• editorial updates</td>
</tr>
<tr>
<td>Public Review Draft v0.4</td>
<td>• removed Java to WSDL mapping section</td>
</tr>
<tr>
<td></td>
<td>• added requirement to support complex types</td>
</tr>
<tr>
<td></td>
<td>• rewrote Runtime chapter to match updated SPI</td>
</tr>
<tr>
<td></td>
<td>• clarifications and editorial updates</td>
</tr>
<tr>
<td>Public Review Draft v0.5</td>
<td>• added requirement to align with WS-I BP [19] where its requirements overlap the JAX-RPC subset</td>
</tr>
<tr>
<td></td>
<td>• updated WSDL/XML to Java Mapping and SOAP Binding chapters to reflect WS-I BP requirements / simplifications</td>
</tr>
<tr>
<td>Public Review Draft v0.6</td>
<td>• further SPI simplifications</td>
</tr>
<tr>
<td></td>
<td>• included additional references to WS-I BP [19] to identify better where requirements overlap</td>
</tr>
<tr>
<td></td>
<td>• added SPI use cases appendix</td>
</tr>
<tr>
<td></td>
<td>• added list of futures</td>
</tr>
<tr>
<td></td>
<td>• clarifications and editorial updates</td>
</tr>
</tbody>
</table>
### TABLE C-1

<table>
<thead>
<tr>
<th>Version</th>
<th>Change description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public review Draft v0.7</td>
<td>• more consistent use of RFC2119 (<a href="http://www.ietf.org/rfc/rfc2119.txt">http://www.ietf.org/rfc/rfc2119.txt</a>)&lt;br&gt;• clarifications and editorial updates</td>
</tr>
<tr>
<td>Proposed Final Draft v0.8</td>
<td>JAXP:&lt;br&gt;• removed requirement disallowing processing of DTDs&lt;br&gt;• pulled in <code>InputSource</code> satisfy <code>DefaultHandler</code> method <code>resolveEntity</code>&lt;br&gt;  JAX-RPC:&lt;br&gt;• updated section 7.2.1 to disallow implementation dependent naming of stubs&lt;br&gt;• added requirement for implementations to set the <code>soapAction</code> HTTP header&lt;br&gt;• reworked SPI to remove type map, now defined in terms of class hierarchy&lt;br&gt;• new SPI class and exception for handling service specific exceptions&lt;br&gt;• added reference WSDL to Appendix A&lt;br&gt;• updated type mapping requirements to include float and double&lt;br&gt;• updated SPI to support mapping of custom faults&lt;br&gt;• clarifications and editorial updates</td>
</tr>
<tr>
<td>Proposed Final Draft v0.9</td>
<td>JAXP:&lt;br&gt;• added <code>setFeature</code> and <code>getFeature</code> to the <code>SAXParserFactory</code>&lt;br&gt;• Feedback received during the public review of this specification indicated that by defining a generic XML parsing API, that could only handle a subset of XML 1.0, J2ME devices would be unable to process several widely used XML document formats. In order to address this situation, and to avoid the fragmentation that would be introduced by making support for the subset optional, the expert group opted to increase the footprint requirement to accommodate the minimum XML 1.0 requirements.&lt;br&gt;  JAX-RPC:&lt;br&gt;• removed <code>Iterator</code> and the <code>Stub</code> method <code>_getPropertyNames</code>&lt;br&gt;• minor clarifications and editorial updates</td>
</tr>
<tr>
<td>Proposed Final Draft 2 v0.10</td>
<td>JAX-RPC:&lt;br&gt;• added Table 8-2 to clarify SPI type mappings&lt;br&gt;• added Table 8-3 to clarify SPI handling of <code>null</code> and empty arrays&lt;br&gt;• updated to reference the JAX-RPC 1.1 specification&lt;br&gt;• minor clarifications and editorial updates</td>
</tr>
</tbody>
</table>
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
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