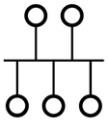

Orbix®

Web Services Programmer's Reference

Version 6.1, December 2003

Making Software Work Together™

Updated: 18-Dec-2003

M 3 1 7 4

Contents

List of Tables vii

List of Figures ix

Preface xi

Chapter 1 Developing Web Service Clients 1
- **Generating Client Code** 3
- **J2SE Client** 4
  - J2SE Client Architecture 5
  - Generating J2SE Client Code 7
  - Using the J2SE Client Demo 8
  - Using the Web Service Interface in Custom Code 10
  - Controlling Client I/O Settings 13
  - Controlling SOAP Message Processing 14
  - Handling Web Service Exceptions 15
- **J2ME Client** 18
  - J2ME Protocol Options 19
  - Generating a J2ME Client 20

Chapter 2 Customizing SOAP Faults 25
- **Controlling SOAP Faults** 26
- **Mapping Exceptions to SOAP Faults** 27

Chapter 3 Adding Handlers 29
- **About Handlers** 32
- **Implementing Handlers** 36
  - Stream Handlers 37
  - Message Handlers 40
  - Invocation Handlers 42
- **Adding Handlers to a Web Service** 43
- **Adding Handlers to a Web Service Client** 45

Chaining Handlers **46**
Writing a Data Content Handler for SOAP Attachments **47**

Chapter 4 Supported Data Types **51**
Mapping from Java to WSDL **52**
Supported Java Objects 53
Primitive Java Types 54
Common Java Classes 55
Java Arrays and Sequences 56
Java Structures 57
Java Exceptions 59
Mapping from CORBA IDL to WSDL **61**
Primitive CORBA IDL Types 62
CORBA IDL Arrays and Sequences 64
CORBA IDL Structures 65
CORBA IDL Enumeration 66
CORBA IDL Unions 67
CORBA Exceptions 68
Mapping from WSDL to Java **71**
Supported Primitive XML Schema Types 72
Supported Derived XML Schema Types 74
Other WSDL Type Mappings 76
Links to the XML Schema Specifications 81

Chapter 5 XAR Properties **83**
<chain> **86**
<chainSequence> **87**
<complexType> **88**
<dependencies> **90**
<endpoint> **91**
<handler> **92**
<include> **93**
<operation> **94**
<param> **95**
<part> **96**
<reference> **98**
<resource> **99**
<schema> **100**

<schemas> 101
<service> 102
<soapproperties> 103
<source> 105

Index 107

List of Tables

Table 1: Command-line Options for a J2SE Client Demo 9
Table 2: J2ME Client Limitations 18
Table 3: SOAPFaultException Constructors 26
Table 4: ServerExceptionHandler Methods 27
Table 5: InputStreamHandler Methods 37
Table 6: OutputStreamHandler Methods 39
Table 7: MessageHandler Methods 40
Table 8: Key Methods of the DataContentHandler Interface 49
Table 9: Supported Java Types and the WSDL Mapping 54
Table 10: Supported Common Java Classes and the WSDL Mapping 55
Table 11: Supported CORBA IDL Types and the WSDL Mapping 62
Table 12: Supported Primitive XML Schema Types and the Java Mapping 72
Table 13: Supported Derived XML Schema Types and the Java Mapping 74

List of Figures

Figure 1: Interaction of J2SE client code with a Web service 5
Figure 2: J2ME Wireless Toolkit GUI 20
Figure 3: Create J2ME Client from WSDL 21
Figure 4: J2ME Wireless Toolkit Phone Simulator 23
Figure 5: Handler interfaces and classes 30
Figure 6: Client-side handlers 32
Figure 7: Message handler chains 33
Figure 8: Server-side handlers 34
Figure 9: Message handlers 35
Figure 10: A SOAP message and a SOAP with attachments message 48

Preface

Audience

This guide is aimed at developers who are developing Web services. Java or other programming experience is assumed.

Updated documentation

The latest updates to the documentation can be found at this URL: http://www.iona.com/docs/.

Additional resources

The IONA knowledge base contains helpful articles, written by IONA experts. You can access the knowledge base at the following location:

http://www.iona.com/support/kb/

The IONA update center contains the latest releases and patches for IONA products:

http://www.iona.com/support/update/

Additional resources

The IONA knowledge base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

The IONA update center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com .

Typographical conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS or Windows command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

CHAPTER 1

Developing Web Service Clients

Clients developed in Web Service Builder provide all the Web service access usually needed. You can also use the generated code as the basis for creating custom applications.

In either case, all low-level programming issues including SOAP, XML, and WSDL technologies are hidden, so you can concentrate on getting Web services working quickly.

Types of Web service clients

Web Service Builder (and equivalent command-line tools) can help you develop several types of client applications:

J2ME Client**:** A lightweight client that runs in the Java 2 Micro Edition (J2ME) environment.

J2SE Client**:** A client that uses the Java 2 Platform, Standard Edition (J2SE) interface. A J2SE client can have either RPC- or document-style interaction with a Web service.s

.NET interoperability

The clients that you generate with Web Service Builder or equivalent command-line tools are standards-compliant. Interoperability is verified against Microsoft's.NET toolkit and MS SOAP. The SOAP client that you build can access Web services that are constructed with Microsoft tools, just like any other Web service.

In this chapter

This chapter contains the following sections:

Generating Client Code	page 3
J2SE Client	page 4
J2ME Client	page 18

Generating Client Code

Client types

After deploying a Web service, you need a way to access it. Web service builder can generate the following client types:

J2SE client: You can generate a J2SE DOM or RPC client. Either client consists of an interface class that is created at compile time, along with an implementation class that is created and instantiated at runtime based on the Java 1.3 proxy scheme.

You can also generate a J2SE client with command-line tools `xmlbus.WSDLToInterface` and `xmlbus.WSDLToJ2SEDemo`

J2ME client: Web Service Builder can generate code for a working J2ME client that can access the Web service. You can compile and run the J2ME client application to access the Web service's methods from devices like a WAP-enabled phone or a palmtop computer.

You can also generate a J2ME client with the command-line tool `xmlbus.WSDLToJ2MEClient`.

Client code sources

In general, client code can be generated from two sources:

- The XAR file for Web service that is created in Web Service Builder
- The WSDL of any Web service.

J2SE Client

Overview

You can build J2SE clients that access a Web service with Web Service Builder or command-line tools. A J2SE client consists of a Web service interface class that is created at compile time, with an implementation proxy class that is created and instantiated at runtime, based on the Java 1.3 proxy scheme.

In this section

This section contains the following topics:

J2SE Client Architecture	page 5
Generating J2SE Client Code	page 7
Using the J2SE Client Demo	page 8
Using the Web Service Interface in Custom Code	page 10
Controlling Client I/O Settings	page 13
Controlling SOAP Message Processing	page 14
Handling Web Service Exceptions	page 15

J2SE Client Architecture

Key features

A J2SE client include the following features:

Web service Java interface: A Java interface that represents the Web service's WSDL information. Client code such as the J2SE client demo or your custom client code calls this proxy code to access the Web service.

J2SE client demo: A simple demonstration client that tests the Web service from the command line. This code calls the Web service Java interface.

WebServiceProxy object: A proxy object created at runtime that implements the Web service Java interface and accesses the Web service. The `WebServiceProxy` object is instantiated in client code such as the J2SE client demo or your custom client code.

How it works

The following figure shows how the various pieces of code interact:

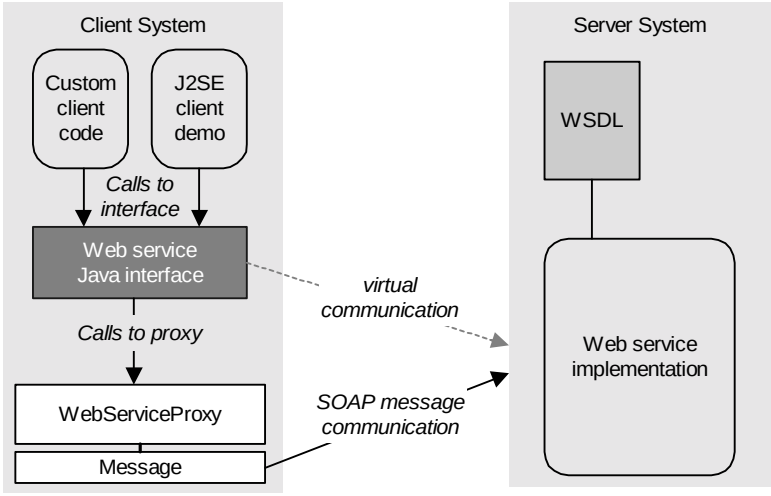

Figure 1: *Interaction of J2SE client code with a Web service*

At runtime, the WebServiceProxy and Message objects are created. The WebServiceProxy implements the interface on the client side. When the client code—J2SE client demo or custom code—calls a method on the Web service interface, the WebServiceProxy object provides the mapping to the methods defined in the WSDL. Finally, the Message object communicates the information from the WebServiceProxy to the actual Web service on the server side.

Note: The generating tool uses the WSDL to create the Web service Java interface and the J2SE client demo.

If you already have the Web service's Java interface class, you can use it directly. For example, if you generate the Web service from your application's interface, you can use the original interface in the client code, provided the methods on the interface correspond to the Web services WSDL.

If you lack interface code for a deployed Web service, use a URL to the Web service's WSDL to generate the Web service Java interface class

Generating J2SE Client Code

Steps

Follow these steps to generate J2SE client code:

1. Use Web Service Builder to generate the Web service Java interface and J2SE client demo (see "Generating Client Code" on page 3). You can also generate the code with the command-line tools `xmlbus.WSDLToInterface` and `xmlbus.WSDLToJ2SEDemo`.

   For this example, use the Finance application provided with your installation.
2. Set the environment for compiling and running J2SE clients by running `itws_clientenv.bat` (Windows) or by sourcing `itws_clientenv` (UNIX) from your installation's `/asp/`*`Version`*`/bin` subdirectory.
3. Compile the Web service interface class for the J2SE client. For example:

   ```
   javac FinanceInterface.java
   ```

Using the J2SE Client Demo

Overview

The J2SE client demo is a ready-made client that accesses the Web service from which it was built. When the tester is invoked, the user specifies the URL of a WSDL and the method call. A J2SE client demo lets you try Web service operations (that use parameters of simple types) and modify some features of the WSDL.

Steps

To use the J2SE client demo, perform the following steps:

1. Set the client environment by running `itws_clientenv.bat` (Windows) or by sourcing `itws_clientenv` (UNIX) from your installation's `/asp/`*Version*`/bin` subdirectory .
2. If you generated a J2SE DOM client, edit the following line of code:

   ```
   org.w3c.dom.Document doc = null;
   ```

   Replace `null;` as follows:

   ```
   com.iona.webservices.util.DOMUtils.createDocumentFromStream
    (new FileInputStream(args[1]));
   ```
3. Compile the J2SE client demo's generated code, including the demonstration client code. For example:

   ```
   javac FinanceInterface.java FinanceProxyDemo.java
   ```
4. If you run the J2SE client demo without arguments, it shows usage options and a list of the available Web service methods. For example:

```
set classpath=.;%classpath%
java FinanceProxyDemo
Syntax is: FinanceProxyDemo [-debug] [-url soapurl] [-wsdl
   wsdllocation] operation [args...]
   operation is one of:
      calculateFutureValue
      showTaxRate
      paymentMortgage
      periodMortgage
      calculateAPR
      calculateRate
      calculateTimeToDoubleUsingRuleOf72
      calculateRateToDoubleUsingRuleOf72
      calculateTimeToDouble
      calculateRateToDouble
```

The command-line options available for a J2SE client demo are described in Table 1.

Table 1: *Command-line Options for a J2SE Client Demo*

Option	Description
`-debug`	Causes the display of SOAP messages when the client tester runs.
`-url` *`soapurl`*	Overrides the URL in the client. This is useful if you want to use a different server implementation of the Web service.

`-wsdl` *`wsdllocation`*	Overrides the location of the WSDL file, so you can specify a different implementation. Use this option if you want to use the client for a different Web service other than the one for which the client was generated.
	Note: You might also need to change the client code for `WebServiceProxy.getProxy()`:
	`Object proxy = WebServiceProxy.getProxy(`
	`  "", //Set to null`
	`  "", //Set to null`
	`  xwarInterface.class,`
	`  wsdlPath,`
	`  debug,`
	`  url,`
	`  userDefinedDataContentHandler`
	`);`

`operation [args...]`	Causes execution of an operation for the Web service with its appropriate arguments.

5. The following example shows how to execute the J2SE client demo with an operation and argument. This example shows the monthly payment on a loan of 100000 with an interest rate of 8% over a period of 30 years.

```
% java FinanceProxyDemo paymentMortgage 100000 8.0 30
% 733.7645738793778
```

Using the Web Service Interface in Custom Code

Overview

With a few simple steps, you can use the Web service Java interface class in custom code to interact with the Web service. The result is that local method calls give your client access to the remote Web service.

Code example

Example 1 is taken from the generated J2SE client demo, which you can use as a guide for your own code.

Example 1: *Custom Client Code*

```
...
import com.iona.webservices.soap.proxy.*;
import com.iona.webservices.client.*;
import com.iona.webservices.client.j2se.*;
import com.iona.webservices.handlers.*;

/**
 * FinanceService
 */
...
Object proxy = WebServiceProxy.getProxy(
    "FinanceService",
    "FinancePort",
    FinanceInterface.class,
    wsdlPath,
    debug,
    url,
    userDefinedDataContentHandler);
FinanceInterface impl = (FinanceInterface)proxy;
...
if ("paymentMortgage".equals(args[0])) {
    double result = impl.paymentMortgage(
        J2SEUtils.parseDouble(args[1]),
        J2SEUtils.parseDouble(args[2]),
        Integer.parseInt(args[3]));
    System.out.println(J2SEUtils.doubleToString(result));
    foundOp = true;
}
...
```

Code explanation

This code executes as follows:

1. Imports the classes required by the client implementation—in this example, the `WebServiceProxy` class and handlers.
2. Calls the `WebServiceProxy` object's `getProxy()`method to bind the interface with the corresponding WSDL, with the following arguments:

`FinanceService`	The name of the Web service.
`FinancePort`	The name of the Web service's endpoint
`FinanceServiceInterface.class`	The interface class for the Web service.
`wsdlPath`	The WSDL file. The default is set to the path used when the code is generated. You can reset this value when running the J2SE client demo using the `-wsdl` option. (See Table 1 on page 9.)
`debug`	An optional `boolean` argument for displaying debugging information. The default is set to `false`. You can reset this value when running the J2SE client demo using the `-debug` option (see Table 1 on page 9).
`url`	An optional `String` argument. The default is set to `null`. You can reset this value when running the J2SE client demo using the `-url` option. (See Table 1 on page 9.)
`userDefinedDataContenHandler`	An optional `HashMap` object.

3. Using the Web service is as simple as making Java method calls. For this example, the mortgage payment is calculated using the three input arguments as input, as follows:

```
impl.paymentMortgage(
    J2SEUtils.parseDouble(args[1]),
    J2SEUtils.parseDouble(args[2]),
    Integer.parseInt(args[3]));
```

Usage guidelines

Keep the following considerations in mind when working with J2SE clients:

- A client side runtime library, `SoapClient.jar`, is required. See the `itws_clientenv` script in "Generating J2SE Client Code" on page 7.
- The reflective nature of the coding presents a minor performance reduction.
- SOAP messages and connections are created at runtime and cannot be modified.
- You should maintain the interface class for each service.
- Some changes to the WSDL require you to regenerate the Web service interface. These include changes to methods, including added or removed methods, changes to the number of parameters to methods, and changes to data types.

Controlling Client I/O Settings

The `com.iona.webservices.soap.client.io.ClientIOSettings` interface provides methods that let you control how a client performs it's IO operations. These include:

- The endpoint URL that the client contacts
- Content handlers that convert MIME streams to objects
- I/O listeners that are useful for debugging.
- Socket layer properties such as timeouts and keepalives.

To obtain a handle to these settings, call:

```
ClientIOSettings io = WebServiceProxy.getClientIOSettings(proxy);
```

Controlling SOAP Message Processing

The `MessageSettings` interface (`com.iona.webservices.soap.client.message.MessageSettings`) provides methods that let you control how a client creates and processes SOAP messages. These include:

- Setting the charset encoding that is used (default is UTF-8).
- Specifying whether to add and validate `xsi:type` attributes.
- Default namespace prefixes.

Many of these settings can help clients interoperate with other servers and improve performance. For example, turning off addition and validation of `xsi:type` attributes can increase performance, but at the expense of validation.

To obtain a handle to these settings, call:

```
MessageSettings ms = WebServiceProxy.getMessageSettings(proxy);
```

Handling Web Service Exceptions

When a Web service returns a SOAP fault, a J2SE client can handle it in two ways:

- The generated ProxyDemo client catches any `RemoteSoapFaultException` that the Web service throws. The exception members—`faultActor`, `faultcode`, `faultString`, and `Detail`—are accessible to the client code, as shown in Example 2.
- Exception handlers that implement the `ClientExceptionHandler` interface can be registered with the ClientChain, as shown in Example 3.

Catching RemoteSoapFaultException

For example, the following ProxyDemo code is generated for a J2SE client:

Example 2: *Catching RemoteSOAPFaultException in a ProxyDemo*

```
...
} catch (RemoteSOAPFaultException sfx) {
          String faultCode = sfx.getFaultCode();
          String faultActor = sfx.getFaultActor();
          String faultString = sfx.getFaultString();
          Detail detail = sfx.getDetail();
          System.err.println("FaultCode: "+faultCode);
          System.err.println("FaultActor: "+faultActor);
          System.err.println("FaultString: "+faultString);
}
...
```

Writing client exception handlers

Catching the RemoteSOAPFaultException can be supplemented or supplanted by one or more exception handlers that you write. These handlers must be registered with the client's handler chain with `addClientExceptionHandler()` (see "Chaining Handlers" on page 46).

The following example shows how you might write a client exception handler for SOAP message faults:

Example 3: *Client Exception Handler for SOAP Faults*

```
import java.io.*;
import java.util.*;
import com.iona.webservices.handlers.*;
import com.iona.webservices.handlers.exception.*;
import com.iona.webservices.handlers.message.*;
import javax.xml.soap.*;
import com.iona.webservices.jaxm.soap.MessageImpl;

public class ExHandler1 implements ClientExceptionHandler {

   public void init(HandlerContext ctx) {
   }

   public void destroy() {
   }

   public void handleException(MessageContext ctx, Throwable th, SOAPMessage fault)
   throws MessageHandlerException {

      try {
         if (fault.getSOAPPart().getEnvelope().getBody().hasFault()) {
            SOAPFault sf = fault.getSOAPPart().getEnvelope().getBody().getFault();
            String fstr = sf.getFaultString();
            Iterator iter = sf.getDetail().getDetailEntries();
            String trace = "";

            if (iter.hasNext()) {
               DetailEntry entry = (DetailEntry)iter.next();
               trace = entry.getValue();
                     //entry.getElementName().getLocalName().startsWith("StackTrace"));
            }
            System.out.println("code="  + sf.getFaultCode() + ", str="
               + sf.getFaultString() + ", actor=" + sf.getFaultActor());
            System.out.println("trace=" + trace);

            if (fstr.startsWith("java_io_FileNotFoundException")) {
               th = new java.io.FileNotFoundException(trace);
               System.out.println("create FileNOtFoundException");
            }
```

Example 3: *Client Exception Handler for SOAP Faults*

```
         } else {
            System.out.println("create SOAPFaultException");
            throw new SOAPFaultException("Invalid msg",
                  "InvalidFaultString",
                  "InvalidFaultActor");
        }

/*
        if (th instanceof FileNotFoundException) {
           String msg = ((FileNotFoundException)th).getMessage();
           if ("no file".equals(msg)) {
              throw new SOAPFaultException("2SFCode", "2SFString", "2SFActor");
           } else {
              throw new SOAPFaultException("Invalid msg " + msg,
                 "InvalidFaultString",
                 "InvalidFaultActor");
           }
        }*/

     } catch (SOAPException ex) {
        throw new MessageHandlerException(ex);
     }

    }
}
```

J2ME Client

Overview

J2ME client run in the Java 2 Micro Edition (J2ME) environment. The generated code consists of:

- A client that can be embedded in any J2ME application
- A sample Mobile Information Device applet (MIDlet) that shows how to use the client.

Functional constraints

J2ME clients are not as full-featured as J2SE clients (see page 4). The following restrictions apply:

Table 2: *J2ME Client Limitations*

Disallowed	Notes
Floating point data types	`float` and `double` data types in the Web service's WSDL are represented as `String` type.
SOAP attachments	
Multi-reference SOAP encoding	Disallowed if a value can be referenced by more than one accessor
Arrays and structures	Document or literal encoding is limited to simple types.
HTTPS support	If not supported by the Mobile Information Device Profile (MIDP) emulator. For example, the JavaSoft emulator does not support HTTPS.

J2SE-specific interfaces	The following interfaces are not supported:
	`ClientChain`
	`ClientSecurity`
	`MessageSettings`
	`ClientIOSettings`

J2ME Protocol Options

A J2ME client can communicate with servers in two ways.

- Streamed HTTP over a raw socket
- Native HTTP provided by the J2ME device

Streamed HTTP

By default, clients try to use streamed HTTP over a raw socket with the J2ME socket protocol handler. This works best for most servers. However, not all J2ME devices support the use of raw sockets. Also, this method does not support HTTPS.

All servers support streamed HTTP.

Native HTTP

Alternatively, clients can communicate through the J2ME device's native HTTP connection support (`HttpConnection`). This built-in HTTP connection support normally chunks the data.

Native HTTP is not supported by the following servers:

- IONA Orbix E2A Application Server
- BEA WebLogic Server

To use native HTTP support, change the protocol portion of the URL in the generated client from `socket` to `http`.

Generating a J2ME Client

The following procedure assumes usage of Sun Microsystem's J2ME Wireless Toolkit (http://java.sun.com/products/j2mewtoolkit/download.html).

Steps

Follow these steps to generate and use a J2ME client demo:

1. Start J2ME Wireless Toolkit:

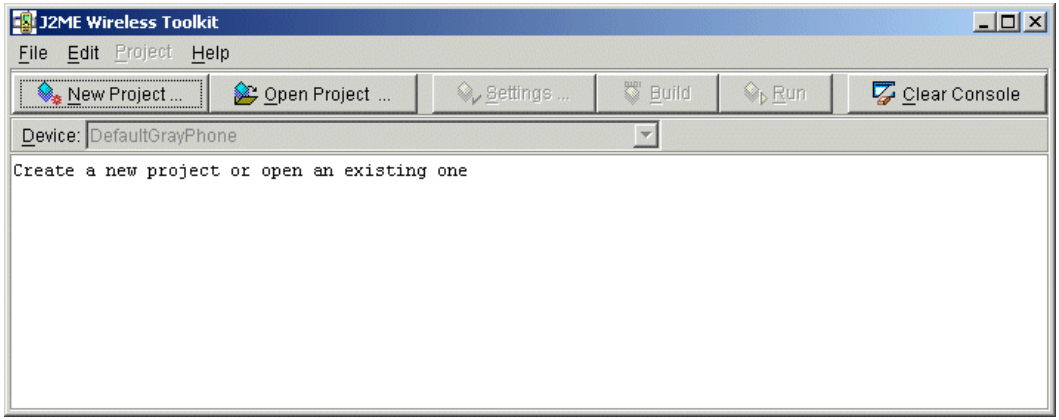

Figure 2: *J2ME Wireless Toolkit GUI*

2. Click **New Project**.
3. Set the following values:

Project Name: A project name that you assign.

MIDlet Class Name: The name of the MIDlet class to be generated in Web Service Builder, as follows: *project-name*`MIDlet`.For example, in order to create a J2ME client from the project Finance, enter `FinanceMIDlet`.

4. Click **Create Project**. J2ME Wireless Toolkit creates a directory with the project name as follows:

```
j2meTookit-install/apps/project-name
```

For example:

```
C:\WTK104\apps\Finance
```

5. In Web Service Builder, select the desired project and choose **Generate|J2ME Client**. In the Output Directory Selection, specify the J2ME project's source directory as follows:

```
j2meTookit-install/apps/project-name/src
```

For example:

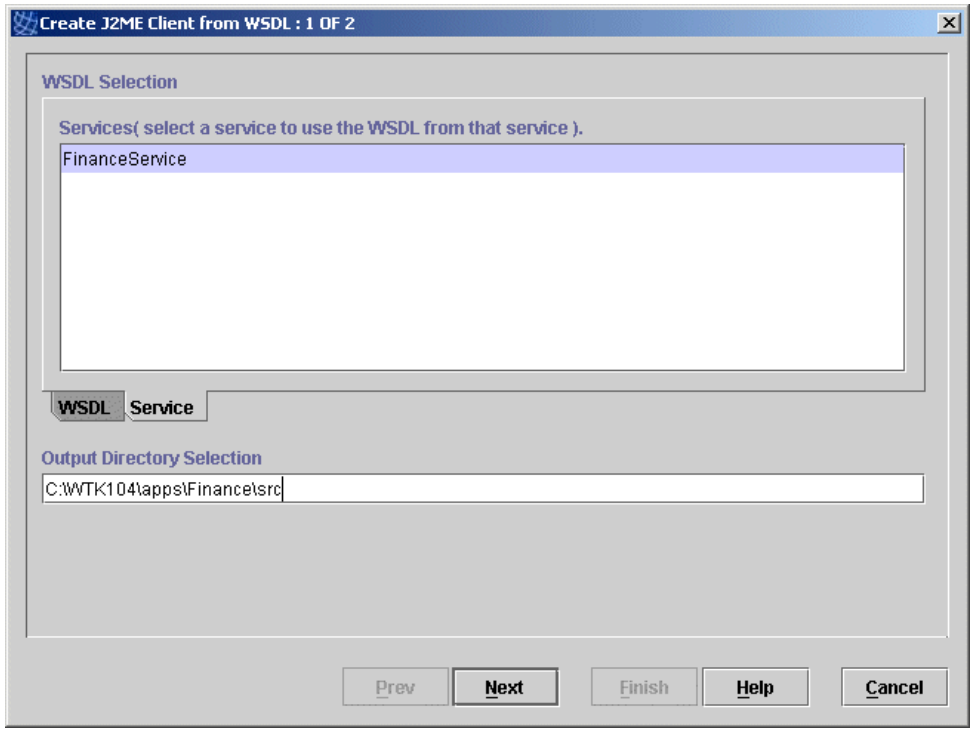

Figure 3: *Create J2ME Client from WSDL*

6. Copy the following files from
   install-root/asp/*Version*/lib/webservices/

   ```
   j2meclient.jar
   kxml.zip
   ```

   Put these files in *j2meTookit-install*/apps/*project-name*/lib.
7. In J2ME Wireless Toolkit:
   - Click **Build**
   - Choose the desired device and click **Run**

J2ME Wireless Toolkit runs the service on the selected device:

Figure 4: *J2ME Wireless Toolkit Phone Simulator*

CHAPTER 2

Customizing SOAP Faults

This chapter shows how to writes code that customizes SOAP faults.

Overview

SOAP faults are messages returned to a client in the case of an error. Normally, the Orbix container returns a SOAP fault whenever a Web service implementation raises an exception. However, the default contents of these SOAP faults might not be appropriate for certain applications.

In this chapter

Orbix provides the following ways for an application to customize the contents of SOAP faults returned to clients:

Controlling SOAP Faults	page 26
Mapping Exceptions to SOAP Faults	page 27

Controlling SOAP Faults

Overview

An application that wants to return a SOAP fault with specific contents can raise a `com.iona.webservices.handlers.message.SOAPFaultException`. When raising this exception, an application can specifically set the `<faultcode>`, `<faultstring>`, `<actor>`, and fault `<details>` that are returned to the application.

Constructors

There are four constructors for this exception:

Table 3: *SOAPFaultException Constructors*

Constructor	Description
`SOAPFaultException( String faultCode, String faultString, String actor, javax.xml.soap.Detail detail )`	Creates a `SOAPFaultException` with specific `faultcode`, `faultstring`, and `actor` tags, and with a detail element represented as a SAAJ `Detail` object. The detail element can be created using the SAAJ APIs provided by Orbix.
`SOAPFaultException( String faultCode, String faultString, String actor )`	Creates a `SOAPFaultException` with specific `faultcode`, `faultstring`, and `actor` tags, but without any detail information.
`SOAPFaultException( String faultCode, String faultString, String actor, Exception ex )`	Creates a `SOAPFaultException` with specific `faultcode`, `faultstring`, and `actor` tags, and whose detail tag contains a stack trace for the provided exception.
`SOAPFaultException( String faultCode, Exception ex )`	Creates a `SOAPFaultException` with a specific `faultcode` tag, whose `faultstring` tag contains the message of the provided exception, and whose detail tag contains a stack trace for the provided exception.

More information on `SOAPFaultException` can be found in the *Web Services JavaDoc*.

Mapping Exceptions to SOAP Faults

Overview

Orbix offers a `ServerExceptionHandler` interface which provides you with flexibility in mapping exceptions raised by the Web service implementation with SOAP faults returned to the client. This section discusses the following topics:

- ServerExceptionHandler interface
- Using custom exception handlers
- Chaining exception handlers

ServerExceptionHandler interface

The `ServerExceptionHandler` interface provides a way to convert exceptions raised during the processing of a message into a specific SOAP response. By writing a `ServerExceptionHandler`, you can customize the way in which server-side exceptions are reported to clients. For example, you might write a `ServerExceptionHandler` to convert an application-specific message (such as `LoginFailed`) into a SOAP fault with a specific `<faultcode>` or `<faultstring>`.

To create a custom server exception handler, you must implement the interface `ServerExceptionHandler` with the following methods

Table 4: *ServerExceptionHandler Methods*

Method	Description
`public void init( HandlerContext context )`	Initializes the handler. This method is called when a server exception handler is first created. This method can be empty.

Table 4: *ServerExceptionHandler Methods*

Method	Description
`public SOAPMessage handleException( MessageContext context, Throwable th, MessageHandlerException mex )`	Called when an exception occurs during the processing of a SOAP message. This method takes three parameters: • The context of the message causing the exception. • The original exception thrown by the Web service implementation. • The `MessageHandlerException` raised during the processing of exceptions. The `ServerExceptionHandler` returns a `SOAPMessage` indicating the response it wants to return to the client, or null to indicate that it does not want to customize the response.
`public void destroy()`	Called to destroy the handler; This method can be empty.

Using custom exception handlers

After you've implemented your custom exception handler, it needs to be placed into the Web service. A custom exception handler is a special type of message handler which is made part of a Web service in three steps:

1. Compile the custom exception handler
2. Insert the custom exception handler into a Web service
3. Add the handler to an endpoint's handler chain

These steps are described in detail in "Adding Handlers to a Web Service Client" on page 45.

Chaining exception handlers

A single Web service can be configured with more than one `ServerExceptionHandler`. This lets you write simpler handlers that process only a single exception, instead of requiring you to handle all possible exceptions with a single `ServerExceptionHandler`. When an exception occurs during the processing of a SOAP message, the container calls the exception handlers in the order in which they are specified in the XAR. The engine stops when one of the handlers returns a non-null value from `handleException`.

CHAPTER 3

Adding Handlers

Web service handlers let you intercept SOAP messages at various points in their life-cycle and customize message processing.

For example, you can use handlers to incorporate compression, encoding, and logging logic into a Web service. With Web Service Builder, you can easily add one or more message handlers to a Web service.

In this chapter

This chapter discusses the following topics:

About Handlers	page 32
Implementing Handlers	page 36
Adding Handlers to a Web Service	page 43
Adding Handlers to a Web Service Client	page 45
Chaining Handlers	page 46
Writing a Data Content Handler for SOAP Attachments	page 47

Message handling API

Figure 5 shows the Java interfaces and classes that are provided for implementing handlers for Web service applications and clients:

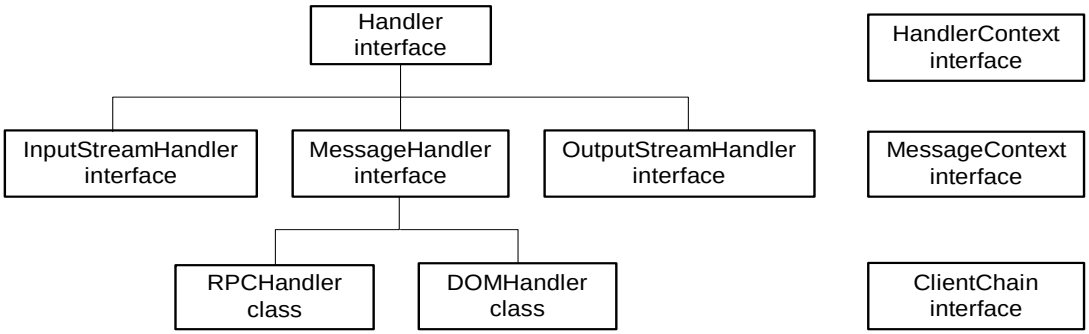

Figure 5: *Handler interfaces and classes*

- Interfaces `InputStreamHandler`, `OutputStreamHandler`, and `MessageHandler` extend the `Handler` interface to provide control and access to various points in the SOAP message life cycle. See "Implementing Handlers" on page 36.
- Classes `RPCHandler` and `DOMHandler` implement the `MessageHandler` interface to provide custom tasks for SOAP messages that are RPC-based and document-based, respectively. See "Adding Handlers to a Web Service" on page 43
- Interface `HandlerContext` initializes handlers, and provides repository information. Interface `MessageContext` provides handlers with information to process a SOAP request received, such as endpoint information.
- Interface `ClientChain` provides a mechanism for adding message and stream handlers to clients. See "Adding Handlers to a Web Service Client" on page 45.

SOAP message manipulation API

An implementation of the `javax.xml.soap` package is provided, for manipulating SOAP message objects in custom Web service applications. See `http://java.sun.com/xml/saaj/index.html` for the complete SAAJ documentation.

About Handlers

Overview

This section includes the following topics:

- Series of handlers process messages
- Handler chains
- Server-side handlers
- Server-side message handlers
- Synchronizing server-side and client-side handlers

Series of handlers process messages

Figure 6 shows how a client's SOAP message passes through a series of handlers for both the output message and the returning input message.

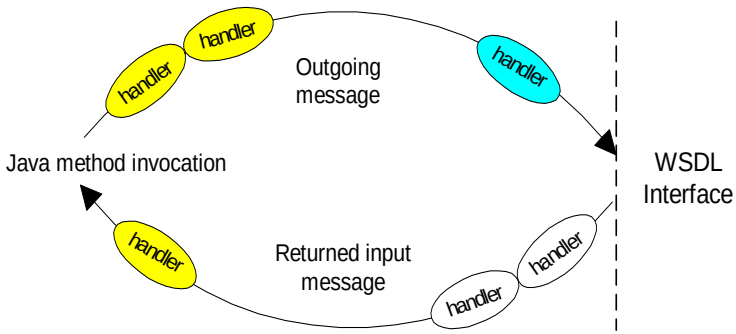

Figure 6: *Client-side handlers*

Handler chains

Handlers can be grouped into chains. Each handler chain addresses a distinct part of the SOAP message lifecycle. Chain types include stream and message handlers:

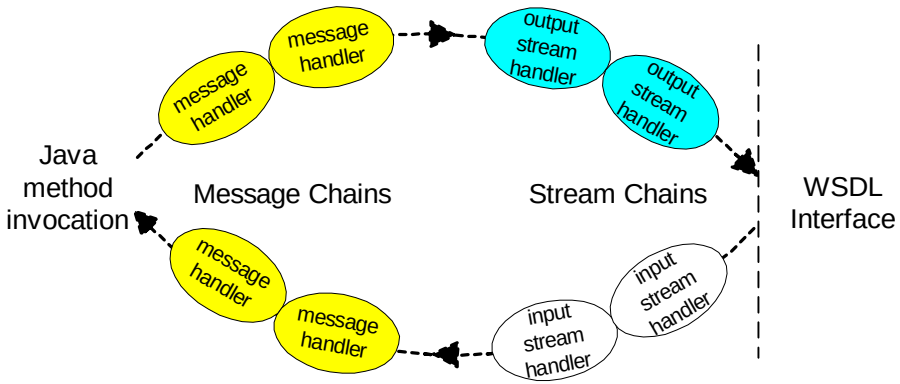

Figure 7: *Message handler chains*

Stream handlers are used to manipulate the raw streamed data of a SOAP message. There are two types of stream handlers:

- Input stream handlers process the message data stream immediately after it arrives off the network—for example, for decryption and decompression.
- Output stream handlers process the message data stream just before it goes out to the network—for example, encryption and compression.

Server-side handlers

In Figure 8, the server has several handler chains that process the incoming message, and others that process it before it returns to the client:

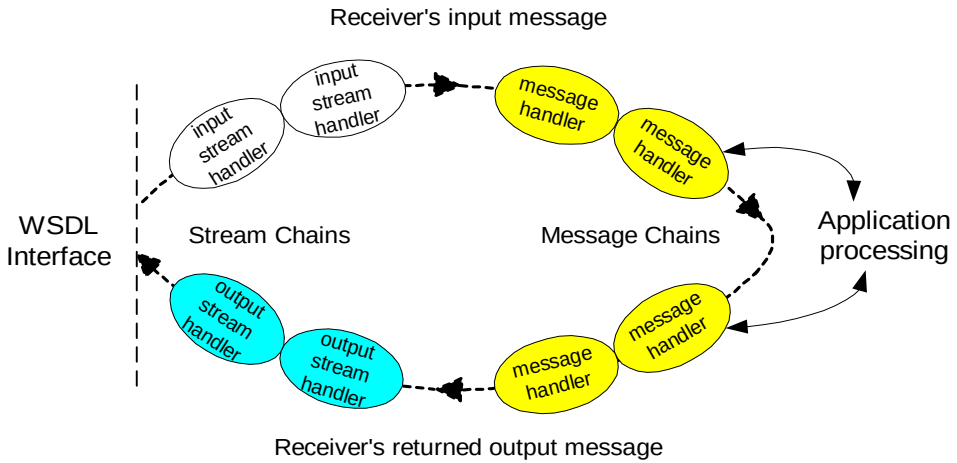

Figure 8: *Server-side handlers*

Server-side message handlers

Server-side message handlers perform application-specific processing with implementations of the `RPCHandler` and `DOMHandler` classes. These handlers access the code for the various types of Web services you have built,

including those based on Java classes, EJBs, CORBA resources, operation flows, Java DOM objects, and schema maps. Other internal handlers deal with issues such as security and routing.

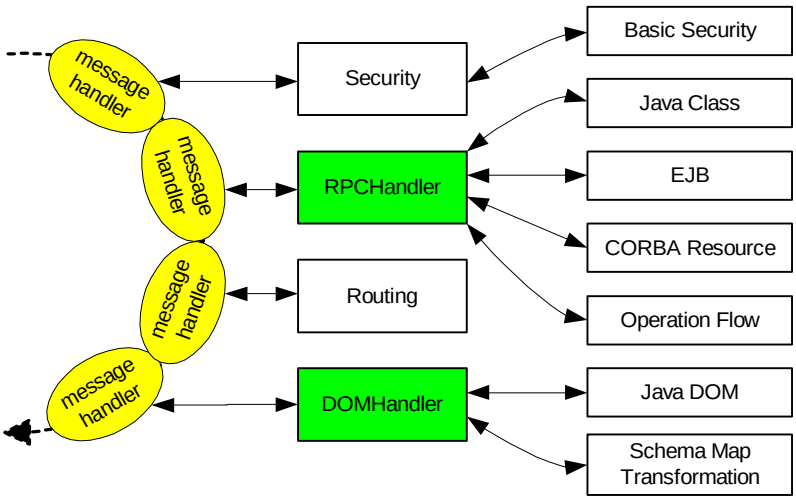

Figure 9: *Message handlers*

Synchronizing server-side and client-side handlers

Because the WSDL does not contain any information about handlers, it is important for client-side and server-side developers to understand and synchronize their respective handlers. For example, a Web service might have stream handlers that decrypt incoming SOAP messages and encrypt the SOAP messages before they are returned to the client. In this case, the client must have corresponding encryption and decryption handlers.

Implementing Handlers

Overview

Provided handler interfaces let you write custom handlers for several points in the SOAP message life-cycle. These include:

1. The raw SOAP message as it comes off the wire, which is handled by the `InputStreamHandler` interface.
2. The message itself, which is handled by the `MessageHandler` interface.
3. The implementation's method invocation which is handled by the `RPCHandler` interface.
4. The raw SOAP message immediately prior to being placed in the wire, which is handled by the `OutputStreamHandler` interface.

In this section

This section discusses the following topics:

Stream Handlers	page 37
Message Handlers	page 40
Invocation Handlers	page 42

Stream Handlers

Overview

Input and output stream handlers are implementations of interfaces `InputStreamHandler` and `OutputStreamHandler`, respectively. Both are extensions of interface `Handler`, in `com.iona.webservices.handlers`. These handlers enable access to the raw bytes of SOAP messages immediately above the network transport layer.

Input stream handlers

Input streams are manipulated via handlers that implement interface `InputStreamHandler`. Input stream handlers let a Web service client or service process raw SOAP messages as they come off the wire. For example, an input stream handler can decompress or decrypt SOAP messages, or log incoming requests.

The life cycle of an input stream handler is managed by the Web services container, which calls the following methods in this order:

- `createStream()`
- `beginRead()`
- `endRead()`

As it receives SOAP messages off the wire, the Web services container calls `read()` on the `InputStream` returned from `createStream()` after it calls `beginRead()`. After the container returns from `read()`, the Web services container calls `endRead()`.

Table 5 shows the methods that an input stream handler implements:

Table 5: *InputStreamHandler Methods*

Method	Description
`init()`	`public void init(HandlerContext context)` Initializes the handler. This method is called when a message handler is first created. This method can be empty.

Table 5: *InputStreamHandler Methods*

Method	Description
`createStream()`	`InputStream createStream(InputStream is, MessageContext context) throws InputStreamHandlerException` Processes the passed input stream and creates a new `InputStream` to hold the processed data. This method returns a reference to the new input stream. The input stream created by this method is used by the Web services container to process the raw SOAP message before converting it into a SAAJ message object. This is the first method the Web services container calls on a registered input stream handler.
`beginRead()`	`public void beginRead(InputStream is, MessageContext context) throws InputStreamHandlerException` The Web services container calls this method before reading the input stream returned by `createStream()`. The input stream passed to `beginRead()` is the input stream returned from `createStream()`. This method can be empty.
`endRead()`	`public void endRead(InputStream is, MessageContext context) throws InputStreamHandlerException` The Web services container calls this method when it returns from reading the input stream. The input stream passed to `endRead()` is the input stream returned from `createStream()`. This method can be empty.
`destroy()`	`public void destroy()` Destroys the handler. This method can be empty.

Output stream handlers

Output stream handlers implement interface `OutputStreamHandler`. Output stream handlers let a Web service client or service process raw SOAP messages just before they are put on the wire. For example, an output stream handler can be used to build a logging facility.

The life cycle of an output stream handler is managed by the Web services container, which calls the following methods in this order:

- `createStream()`
- `beginWrite()`
- `endWrite()`

The Web services container calls `write()`on the `OutputStream` returned from `createStream()` after it calls `beginWrite()`. After it returns from `write()`, the Web services container calls `endWrite()`.

Table 6 shows the methods that an output stream handler implements:

Table 6: *OutputStreamHandler Methods*

Method	Description
`init()`	`public void init(HandlerContext context)` Initializes the handler. This method is called when a message handler is first created. This method can be empty.
`createStream()`	`OutputStream createStream(OutputStream os, MessageContext context) throws OutputStreamHandlerException` Processes the passed output stream and creates a new output stream to hold the processed stream. This method returns a reference to the new output stream. The output stream created by this method is used by the Web services container to process the raw SOAP message before sending it to the network transport layer. This is the first method the Web services container calls on a registered output stream handler.
`beginWrite()`	`public void beginWrite(OutputStream is, MessageContext context) throws OutputStreamHandlerException` The Web services container calls this method prior to writing the output stream returned by `createStream()`. The output stream passed to `beginRead()` is the output stream returned from `createStream()`. This method can be empty.
`endWrite()`	`public void endWrite(OutputStream is, MessageContext context) throws OutputStreamHandlerException` The Web services container calls this method when it returns from writing the output stream. The output stream passed to `endRead()` is the output stream returned from `createStream()`. This method can be empty.
`destroy()`	`public void destroy()` Destroys the handler. This method can be empty.

Message Handlers

Overview

After the SOAP request is processed by a chain of input stream handlers, the Web services container turns the SOAP message into a SAAJ message object. The SAAJ message is an object representation of a SOAP message. This object model is based upon the SAAJ specification.

All the handlers are cached so there is only one instance of handler for all the calls. When you redeploy the Web service, the handlers are reset and reinitialized when they receive the first call.

This section discusses the following topics:

- MessageHandler interface
- MessageHandler methods

MessageHandler interface

The `MessageHandler` interface provides access to the elements of the SOAP message. It uses the SAAJ interfaces to provide access to the object representation of the original SOAP message. Using this interface, you can write message handlers to process specific parts of the SOAP message. For example, you might build a handler to report the information in a message's header element.

MessageHandler methods

To create a custom message handler, you must implement the `MessageHandler` interface. Table 7 shows the methods to implement:

Table 7: *MessageHandler Methods*

Method	Description
`init()`	`public void init(HandlerContext context)` Initializes the handler. This method is called when a message handler is first created. Handlers are created when the associated web service endpoint receives the first call. This method can be empty.

`processMessage()`	`public SOAPMessage processMessage(SOAPMessage message, MessageContext context)`
	`throws MessageHandlerException`
	Processes the SOAP message using the methods provided in the SAAJ API. The method returns the processed message.

Table 7: *MessageHandler Methods*

Method	Description
`destroy()`	`public void destroy()` Destroys the handler. Also, when you undeploy a Web service, `destroy()` is automatically called on each handler. This method can be empty.

Invocation Handlers

Overview

The `RPCHandler` interface is a special case of the `MessageHandler` interface. It provides methods to invoke RPC calls. When using the `RPCHandler` interface, you do not have to worry about disassembling the SAAJ message. Implementations are provided for processing the SOAP message, validating it against the WSDL, and handling the serialization and deserialization.

Note that there can be only one invocation handler in a message handler chain because the SOAP message is consumed with the request.

Implement invoke()

In order to write a custom invocation handler, you must implement the `invoke()` method, which accepts a set of objects that are the result of the deserialization of the SOAP elements, and returns a set of objects that are the result of some type of invocation.

`invoke()` has two possible signatures:

```
Object invoke(Method method, Object[] params,
              MessageContext context)
abstract Object[] invoke(String methodName, Class[] paramTypes,
                         Object[] objs, MessageContext context)
```

Through the context information and the initialization of the handler, the interface obtains access to XAR-related information. Using this collective information, you can choose to process an RPC call in any fashion you choose, such as with CORBA-oriented code.

Adding Handlers to a Web Service

Overview

After you implement a handler, you insert it into a Web service in the following steps:

1. Compile the handler
2. Insert a handler into a Web service
3. Add a handler to an endpoint's handler chain

Compile the handler

To compile a handler:

1. Ensure the correct classes are in your CLASSPATH. From the installation's `/asp/`*`Version`*`/bin` subdirectory, run (Windows) or source (UNIX) the script `itws_clientenv[.bat]`.
2. Compile the Java file:

   `javac` *`myHandler`*`.java`
3. You can break down the compiled JAR file into classes as follows:

   `jar -cvf` *`myHandler`*`.jar` *`classes`*

Insert a handler into a Web service

Follow these steps to insert a handler into a Web service through Web Service Builder.

1. Start Web Service Builder.
2. From the **Projects** list, select the Web service where you wish to insert the handler.
3. Select the **Handlers** tab on the bottom of the work area.
4. Click **Add**, and enter the handler's name and class name.
5. Select the **Classes** tab, then click **Add a Supporting Class**.
6. Locate the file that contains the class for your handler and include it.
7. Repeat steps **5**-**6** for any classes on which your handler has dependencies.

Add a handler to an endpoint's handler chain

After you add a handler to a Web service, you place it in an endpoint's processing chain:

1. Start Web Service Builder.
2. From the **Projects** list, select the target endpoint.

   Note: Endpoints can only use message handlers that are included by their parent Web service.

3. Select **Handler Sequence**.
4. Select the type of handler you wish to add from the **Types of Handlers** panel.
5. The list of handlers available to the Web service will appear in the **Available Handlers** panel. These handlers are not currently being used by the endpoint. Select the handler you want to add to the endpoint's handler chain and use the right arrow between the **Available Handlers** panel and the **Chained Handlers** panel. The handler should move to the **Chained Handlers** panel.
6. To change the order handlers in the chain are called, select the handler you want to move and use the up and down arrows next to the **Chained Handlers** panel to move it around.

Adding Handlers to a Web Service Client

Overview

Handlers are added to Web service clients with the `ClientChain` interface defined in the package `com.iona.webservices.soap.client.chain`.

`ClientChain` interface

This interface includes the following methods for adding handlers to each point in a message's life-cycle.

- `addClientExceptionHandler()`
- `addInputMessageHandler()`
- `addInputStreamHandler()`
- `addOutputMessageHandler()`
- `addOutputStreamHandler()`

Other methods are also available to determine the size of the message handler chain and to remove handlers from chains, among other things. For details of all methods for the `ClientChain` interface, see *Web Services JavaDoc*.

Chaining Handlers

Overview

Handlers can be chained together to increase flexibility and functionality. A Web service has the following types of handler chains:

- Input stream chain
- Message object chain
- Output stream chain
- SOAP fault exception chain

Handlers in a chain are called in sequence, so the first handler in the chain completes its processing and passes the result to the next handler in the chain. Each handler can be independent of all other handlers. This gives you greater flexibility in developing Web service handlers, and makes handlers reusable.

However, handler independence also requires you to chain handlers together in the correct sequence. For example, if a Web service receives an encrypted request in a compressed file, and its input stream handler chain puts the decryption handler ahead of the decompression handler, the request will fail or produce unpredictable results. Or, if a Web service packages a response to include a number of records field in the SOAP header and the Web services client does not have a handler to process it, the client may not function correctly.

Handler chain on a Web service

Using Web Service Builder, you can easily add handlers to a Web service's endpoint handler chain and reorder them using the **Handler Sequence** tab for an end point. See "Add a handler to an endpoint's handler chain" on page 43 for more information.

Handler chain on a client

Client handler chains are built programatically with interface `ClientChain`, defined in the package `com.iona.webservices.soap.client.chain`. See "Adding Handlers to a Web Service Client" on page 45 or the *Web Services JavaDoc* for more information.

Writing a Data Content Handler for SOAP Attachments

Overview

Your installation provides data content handlers for processing SOAP attachments of several common data types, including plain text, XML, JPEG images, and octet streams. These default content handlers can be supplemented or replaced by custom content handlers. Custom content handlers let you manipulate the way a Web service handles default data or define specific data types for a Web service to process. For example, a Web service that handles purchasing requests might require that a purchase order be in a particular format.

This section discusses the following topics:

- Structure of SOAP messages
- Data content handlers
- Default content handlers
- JAMX API
- DataContentHandler interface
- Registering data content handlers

Structure of SOAP messages

Figure 10 shows the structure of a SOAP 1.1 message and the structure of a SOAP 1.1 message with attachments. SOAP with attachments uses the Multipurpose Internet Mail Extensions (MIME) specification.

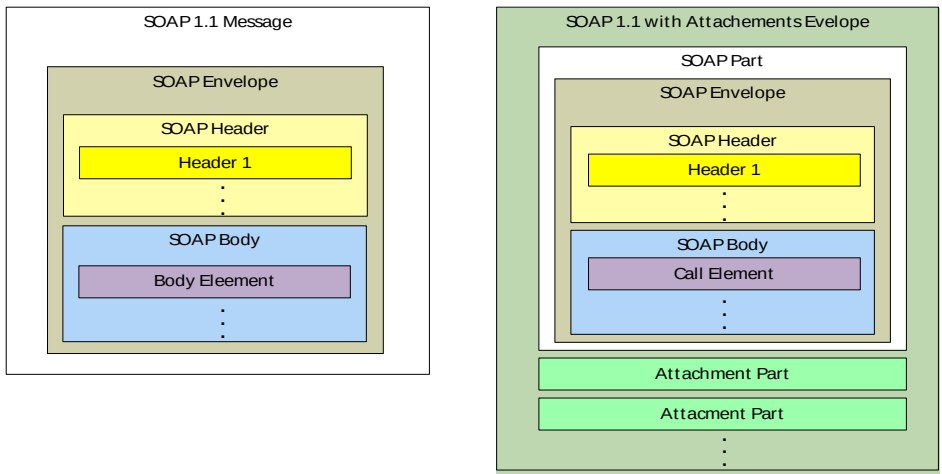

Figure 10: *A SOAP message and a SOAP with attachments message*

Data content handlers

Data content handlers convert raw SOAP attachments into java objects that a Web service, or back-end application server, works with. Each handler corresponds to a particular MIME type and is responsible for converting the raw data stream into the proper java object and converting the java object back into a raw data stream.

Note: As with message handlers, it is critical that both the server and the client are in agreement on the types of objects that will be communicated.

Default content handlers

Default data content handlers are provided for many basic MIME types including, `text/plain`, `text/html`, `image/gif`, and `image/jpeg`. While these are sufficient for simple Web service implementations, a more robust Web service may utilize a custom built purchase order form or another data object model.

JAMX API

Using JAMX with the Java Activation Framework, a set of APIs are exposed that let you build custom data content handlers, and register them with the Web services container.

DataContentHandler interface

To create a data content handler, you must implement the `DataContentHandler` interface defined in `javax.activation`. While the interface contains several operations, only two must be fully implemented in a data content handler:

Table 8: *Key Methods of the DataContentHandler Interface*

Method	Description
`getContent()`	`public Object getContent(DataSource ds) throws IOException` Takes in the raw data and returns the contents as the desired Java object. The returned object will need to be cast into the proper data type.
`writeTo()`	`public void writeTo(Object obj, String mimeType, OutputStream os) throws IOException` Takes a Java Object and writes it to the output stream as raw byte data.

Registering data content handlers

After a data content handler has been developed, it must be compiled and registered with the Web service. To register a data content handler with a Web service using Web Service Builder perform the following steps.

1. Start Web Service Builder.
2. Select the service for which you want to use your handler from the projects list and open its **Content Handlers** tab.
3. On the **Content Handlers** tab, click **Add**.
4. Fill in the name of the handler, the MIME type it filters, and the Java class that implements it. Press **OK**.
5. Select the **Class** tab and add any classes that the content handler requires.

CHAPTER 4

Supported Data Types

Applications that are to be transformed to Web services must use supported method data types. This requirement avoids the generation of invalid code. This chapter shows the data types supported and the type mapping used when mapping between programming languages and WSDL.

In this chapter

This chapter consists of the following sections:

Mapping from Java to WSDL	page 52
Mapping from CORBA IDL to WSDL	page 61
Mapping from WSDL to Java	page 71

Unsupported

Data types that are *not* yet supported include:

- Any class which cannot get or set values.
- Vector, List, and Hashtable types.
- Missing application parts. If the class cannot be loaded, then it cannot be supported.
- CORBA IDL `value` types and object references.

Mapping from Java to WSDL

In this section

This section discusses the following topics:

Supported Java Objects	page 53
Primitive Java Types	page 54
Common Java Classes	page 55
Java Arrays and Sequences	page 56
Java Structures	page 57
Java Exceptions	page 59

Supported Java Objects

Overview

Parameters and return value objects other than simple types require:

- A public, default (no arguments) constructor.
- A `get()` method for all data members.
- A `set()` method for all data members.

JavaBeans

JavaBean type classes (also known as structures) are supported. These data members can be the basic Java types (primitive and common class types), arrays of basic types, or arrays of structures. This means that you can create a complex Java object to serialize over the wire.

Primitive Java Types

Overview

Table 9 shows the Java types for application method parameters and return values supported when creating a Web service. The table also shows the associated WSDL type mapping.

Table 9: *Supported Java Types and the WSDL Mapping*

Java Type	WSDL Type Mapping
`boolean`	`xsd:boolean`
`byte`	`xsd:byte`
`char`	`xsd:string (length=1)`
`char[]`	`Array of xsd:string(length=1)`
`byte[]`	`xsd:base64Binary`
`double`	`xsd:double`
`float`	`xsd:float`
`int`	`xsd:int`
`long`	`xsd:long`
`short`	`xsd:short`

Examples

Examples include the following:

```
public void myMethod(int count){}
public int myMethod(char letter){ return 10; }
public boolean isMyMethod(void){ return true; }
```

Java code containing `char[]` results in WSDL with the following types:

```
<simpleType name="char">
     <restriction base="xsd:string">
         <length value="1"/>
     </restriction>
</simpleType>
<complexType name="ArrayOfchar">
     <complexContent>
         <restriction base="SOAP-ENC:Array">
             <attribute ref="SOAP-ENC:arrayType"
   wsdl:arrayType="xsd1:char[]"/>
         </restriction>
     </complexContent>
</complexType>
```

Common Java Classes

Overview

Table 10 shows the Java classes for application method parameters and return values supported when creating a Web service. The table also shows the associated WSDL type mapping.

Table 10: *Supported Common Java Classes and the WSDL Mapping*

Supported Java Class	WSDL Type Mapping
`java.lang.Boolean`	`xsd:boolean`
`java.lang.Byte`	`xsd:byte`
`java.lang.Character`	`xsd:string (length=1)`
`java.lang.Double`	`xsd:double`
`java.lang.Float`	`xsd:float`
`java.lang.Integer`	`xsd:int`
`java.lang.Long`	`xsd:long`
`java.lang.Short`	`xsd:short`
`java.lang.String`	`xsd:string`
`java.math.BigDecimal`	`xsd:decimal`
`java.math.BigInteger`	`xsd:integer`
`java.util.Calendar`	`xsd:dateTime`
`java.util.Date`	`xsd:dateTime`

java.lang.Object not supported

No direct support is provided for `java.lang.Object` because the actual class of the object must be known. Since `Object` is untyped, there is not sufficient information to build the WSDL at design time and to properly encode and decode the object at runtime. This is an example of missing metadata. The problem affects Java, EJB, and CORBA-based Web services. As a work around, you can manually build a wrapper class, or facade, that uses a concrete type. This wrapper effectively adds the type information that is otherwise missing.

Examples

Examples include the following:

```
public void myMethod(Integer count){}
public int countLetters(String essay){ return essay.length();}
public Integer getSize(String s){ return s.length(); }
```

Java Arrays and Sequences

Overview

Arrays and sequences are mapped into the `<complexType>` XML schema type similar to the following:

```
<complexType  name ='ArrayOfstring'>
  <complexContent>
    <restriction base='SOAP-ENC:Array'>
      <attribute ref='SOAP-ENC:arrayType'
   wsdl:arrayType='xsd:string[]'/>
    </restriction>
   </complexContent>
</complexType>
```

Java Structures

Overview

Structures are mapped into the `<all>` XML schema type within the `<complexType>`.

Examples

For example, a structure with three properties (an `int`, a `float`, and a `string`) is mapped to the code shown in Example 4:

Example 4: *WSDL Mapping for a Java Structure*

```
<complexType name="SOAPStruct">
   <all>
      <element name="varInt" type="xsd:int"/>
      <element name="varFloat" type="xsd:float"/>
      <element name="varString" type="xsd:string"/>
    </all>
</complexType>
```

Example 5 shows the Java code that maps to Example 4.

Example 5: *Java Structure Mapping Example*

```
    public class SOAPStruct {
        int m_varInt = 0;
        float m_varFloat = 0.0f;
        String m_varString = "";
        public SOAPStruct() {
        }

        public void setvarInt(int v) {
            m_varInt = v;
        }
        public int getvarInt() {
            return m_varInt;
        }
        public void setvarFloat(float v) {
            m_varFloat = v;
        }
        public float getvarFloat() {
            return m_varFloat;
        }
```

Example 5: *Java Structure Mapping Example*

```
        public void setvarString(String v) {
            m_varString = v;
        }
        public String getvarString() {
            return m_varString;
        }
    }
```

Java Exceptions

Overview

A Java class can declare service-specific exceptions in a method signature. Only checked exceptions are mapped to WSDL faults. A checked exception means it must extend `java.lang.Exception` either directly or indirectly. Unchecked exceptions are runtime exceptions (`java.lang.RuntimeException`) which cannot be mapped to WSDL.

Examples

For example, note the following Java code:

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

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

The checked exception is `InvalidTickerException` because its class extends `java.lang.Exception`. This code results in the WSDL as shown in Example 6:

Example 6: *WSDL Mapping for Java Exceptions*

```
<types>
  <schema ...>
    <!-- Exception definitions -->
    <complexType name="InvalidTickerException">
      <sequence>
  <element name="tickerSymbol" type="xsd:string"/>
      </sequence>
    </complexType>
  </schema>
</types>
<message name="InvalidTickerException">
  <part name="InvalidTickerException"
   type="xsd1:InvalidTickerException"/>
</message>
<portType name="StockQuoteProvider">
  <operation name="getLastTradePrice" ...>
    <input message="tns:getLastTradePrice"/>
    <output message="tns:getLastTradePriceResponse"/>
    <fault name="InvalidTickerException"
   message="tns:InvalidTickerException"/>
  </operation>
</portType>
```

Mapping from CORBA IDL to WSDL

In this section

This section discusses the following topics:

Primitive CORBA IDL Types	page 62
CORBA IDL Arrays and Sequences	page 64
CORBA IDL Structures	page 65
CORBA IDL Enumeration	page 66
CORBA IDL Unions	page 67
CORBA Exceptions	page 68

Primitive CORBA IDL Types

Overview

Table 11 shows the CORBA IDL types for application method parameters and return values that supported when creating a Web service. The table also shows the associated WSDL type mapping.

Table 11: *Supported CORBA IDL Types and the WSDL Mapping*

CORBA IDL Type	WSDL Type Mapping
any	*see note*[a]
boolean	xsd:boolean
char	xsd:string (length=1)
char[]	Array of xsd:string(length=1)
double	xsd:double
fixed	*not supported*
float	xsd:float
long	xsd:int
long double	*not supported*
long long	xsd:long
Object	*not supported*
octet	xsd:byte
short	xsd:short
unsigned long	xsd:unsignedInt
unsigned long long	xsd:unsignedLong
unsigned short	xsd:unsignedShort
string	xsd:string
wchar	xsd:string (length=1)
wstring	xsd:string

a. When you create a Web service that includes CORBA Any data, Web Service Builder asks to indicate the Any's data's type code. This information is used to map the Any to a concrete WSDL type.

char[] example

IDL code containing `char[]` results in WSDL with the following types:

```
<simpleType name="char">
     <restriction base="xsd:string">
         <length value="1"/>
     </restriction>
</simpleType>
```

```
<complexType name="ArrayOfchar">
     <complexContent>
         <restriction base="SOAP-ENC:Array">
             <attribute ref="SOAP-ENC:arrayType"
   wsdl:arrayType="xsd1:char[]"/>
         </restriction>
     </complexContent>
</complexType>
```

CORBA IDL Arrays and Sequences

Overview

Arrays and sequences are mapped into the `<complexType>` XML schema type similar to the following:

```
<complexType  name ='ArrayOfstring'>
  <complexContent>
    <restriction base='SOAP-ENC:Array'>
      <attribute ref='SOAP-ENC:arrayType'
   wsdl:arrayType='xsd:string[]'/>
    </restriction>
   </complexContent>
</complexType>
```

sequence<octet>

A `sequence<octet>` maps to `xsd:base64Binary`.

CORBA IDL Structures

Overview

IDL structures are mapped into the WSDL `<all>` XML schema type within the `<complexType>`.

Example

For example, assume an IDL structure with the following three properties:

```
struct SOAPStruct
{
   long   varInt;
   float  varFloat;
   string varString;
};
```

This IDL structure maps to the WSDL shown in Example 7:

Example 7: *WSDL Mapping for a CORBA Structure*

```
<complexType name="SOAPStruct">
   <all>
      <element name="varInt" type="xsd:int"/>
      <element name="varFloat" type="xsd:float"/>
      <element name="varString" type="xsd:string"/>
    </all>
</complexType>
```

CORBA IDL Enumeration

Overview

IDL enumeration is mapped to an XSchema `<simpleType>` with enumeration restrictions.

Example

For example, assume the following IDL enumeration:

```
enum Beer {
    Wheat, Lambic, Bitter, Stout, Porter
};
```

This IDL enumeration results in the WSDL shown in Example 8.

Example 8: *WSDL Mapping for CORBA IDL Enumeration*

```
<simpleType name="Beer">
    <restriction base="xsd:string">
        <enumeration value="Wheat"/>
        <enumeration value="Lambic"/>
        <enumeration value="Bitter"/>
        <enumeration value="Stout"/>
        <enumeration value="Porter"/>
    </restriction>
</simpleType>
```

CORBA IDL Unions

Overview

IDL unions are mapped to a <choice> complex type with the discriminator mapped to either an attribute for literal endpoints, or to an optional element for encoded endpoints.

Example

For example, assume the following IDL union:

```
union LongUnion switch (long)
{
      case 101: long foo;
      case 102: string bar;
};
```

This IDL union results in the WSDL shown in Example 9.

Example 9: *WSDL Mapping for a CORBA IDL Union*

```
<complexType name="LongUnion">
    <sequence>
       <element maxOccurs="1" minOccurs="0" name="discriminator"
type="xsd:int"/>
        <choice>
            <element name="foo" type="xsd:int"/>
            <element name="bar" type="xsd:string"/>
        </choice>
    </sequence>
</complexType>
```

CORBA Exceptions

Overview

IDL exceptions are mapped in WSDL as constructed types, such as structures. A fault message (`<fault>`) is generated for each exception in a `raises` clause of an IDL operation. Note that in IDL, exceptions can only be used in `raises` clauses and not as operation parameters.

Example

For example, assume the following IDL:

```
// IDL
module Example {
   exception UnknownError {};
   exception BadRecord {
      string why;
   };
   exception RottenApple {
      long numberOfWorms;
   };
   interface SomeInterface {
      long bar(in float pi) raises (BadRecord, UnknownError);
   };
};
```

This code results in the WSDL as shown in Example 10:

Example 10: *WSDL Mapping for CORBA IDL Exceptions*

```
<?xml version="1.0"?>
<definitions name="anExample" ...
  ...
  <!-- Exception definitions -->
    <xsd:complexType name="BadRecord">
      <xsd:sequence>
        <xsd:element name="why" type="xsd:string" maxOccurs="1"
   minOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
```

Example 10: *WSDL Mapping for CORBA IDL Exceptions*

```
    <xsd:complexType name="RottenApple">
      <xsd:sequence>
        <xsd:element name="numberOfWorms" type="xsd:int"
   maxOccurs="1" minOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="UnknownError">
      <xsd:sequence>
      </xsd:sequence>
    </xsd:complexType>

    <simpleType name="completion_status">
        <restriction base="xsd:string">
            <enumeration value="COMPLETED_YES"/>
            <enumeration value="COMPLETED_NO"/>
            <enumeration value="COMPLETED_MAYBE"/>
        </restriction>
    </simpleType>
    <complexType name="SystemException">
        <sequence>
            <element name="completed"
   type="xsd1:completion_status"/>
            <element name="minor" type="xsd:unsignedInt"/>
        </sequence>
    </complexType>
  <!-- Messages related to port: SomeInterface -->
  ...
  <!-- port for Example.SomeInterface -->
  <portType name="SomeInterface">
    <operation name="bar" parameterOrder="_target pi">
      <input message="tns:bar"/>
      <output message="tns:barResponse"/>
      <fault name="BadRecord" message="BadRecord"/>
      <fault name="UnknownError" message="UnknownError"/>
      <fault name="SystemException"
   message="tns:SystemException"/>
    </operation>
  </portType>
</definitions>
```

These fault messages are named after the fully qualified exception name, and consist of a single element, named `exception`, which is of the same type as the mapped complex type corresponding to the exception definition.

Note that when creating a Web service from CORBA IDL, a `SystemException` fault is added to every operation. This is added even if the IDL does not specifically declare a system exception, because CORBA system exceptions are widely used for debugging and conveying other important information.

Mapping from WSDL to Java

Overview

When the Web service tools map from WSDL to Java, the supported types are from the XML Schema specifications of 2001, 2000, and 1999.

In this section

This section discusses the following topics:

Supported Primitive XML Schema Types	page 72
Supported Derived XML Schema Types	page 74
Other WSDL Type Mappings	page 76
Links to the XML Schema Specifications	page 81

Supported Primitive XML Schema Types

Overview

Table 12 shows the primitive XML Schema data types that are supported. Bold indicates supported types. If no Java mapping is shown, the type is not supported. The table includes indicators as to which XML Schema specifications the type applies.

Table 12: *Supported Primitive XML Schema Types and the Java Mapping*

XML Schema Type	Java Mapping	2001	2000	1999
anyURI		X		
base64Binary	**byte[]**	X		
boolean	**boolean**	X	X	X
binary			X	X
date		X		
dateTime	**java.util.Date**	X		
decimal	**java.math.BigDecimal**	X	X	X
double	**double**	X	X	X
duration		X		
ENTITY			X	
float	**float**	X	X	X
gDay		X		
gMonth		X		
gMonthDay		X		
gYear		X		
gYearMonth		X		
hexBinary	**byte[]**	X		

Table 12: *Supported Primitive XML Schema Types and the Java Mapping*

XML Schema Type	Java Mapping	2001	2000	1999
ID			X	
IDREF			X	
NOTATION		X		
Qname		X	X	
recurringInstant				X
string	**java.lang.String**	X	X	X
time		X		
timeInstant	**java.util.Date**			X
timeDuration			X	X
uri				X
uriReference			X	

Supported Derived XML Schema Types

Overview

Table 13 shows the derived XML Schema data types that are supported. Bold indicates supported types. If no Java mapping is shown, the type is not supported. The table includes indicators as to which XML Schema specifications the type applies.

Table 13: *Supported Derived XML Schema Types and the Java Mapping*

XML Schema Type	Java Mapping	2001	2000	1999
byte	**byte**	X	X	
CDATA			X	
century			X	
date	**java.util.Date**		X	X
ENTITIES		X	X	X
ENTITY		X		X
ID		X		X
IDREF		X		X
IDREFS		X	X	X
int	**int**	X	X	
integer	**java.math.BigInteger**	X	X	X
language		X	X	X
long	**long**	X	X	
month			X	
Name		X	X	X
NCName		X	X	X
negativeInteger	**java.math.BigInteger**	X	X	

Table 13: *Supported Derived XML Schema Types and the Java Mapping*

XML Schema Type	Java Mapping	2001	2000	1999
NMTOKEN		X	X	X
NMTOKENS		X	X	X
nonNegativeInteger	**java.math.BigInteger**	X	X	X
nonPositiveInteger	**java.math.BigInteger**	X	X	X
normalizedString	**java.lang.String**	X		
positiveInteger	**java.math.BigInteger**	X	X	X
NOTATION			X	X
QName				X
recurringDate			X	
recurringDay			X	
short	**short**	X	X	
time			X	X
timeInstant	**java.util.Date**			X
timePeriod			X	
token		X	X	
unsignedByte	**short**	X	X	
unsignedInt	**long**	X	X	
unsignedLong	**java.math.BigInteger**	X	X	
unsignedShort	**int**	X	X	
year			X	

Other WSDL Type Mappings

In this section

This section describes the WSDL to Java mapping that is used for the following WSDL types:

- <choice>
- <enumeration>
- <fault>

<choice>

The mapping for the <choice> WSDL type is a class as shown in the following examples.

Note: CORBA IDL union is mapped to the <choice> WSDL type. See "CORBA IDL Unions" on page 67.

Assume the following WSDL <ComplexType> with the <choice> element:

```
<complexType name="LongUnion">
    <sequence>
       <element maxOccurs="1" minOccurs="0" name="discriminator"
type="xsd:int"/>
        <choice>
            <element name="foo" type="xsd:int"/>
            <element name="bar" type="xsd:string"/>
        </choice>
    </sequence>
</complexType>
```

This maps to the following Java class:

```
public class LongUnion {

    public static final String XMLBUS_VERSION = ...;

    public static final String TARGET_NAMESPACE =
   "http://xmlbus.com/CORBAApp/xsd";

    private String __discriminator;
```

```
    public Integer discriminator;
    private int foo;
    private String bar;

    public int getfoo() {
        return foo;
    }

    public void setfoo(int _v) {
        this.foo = _v;
        __discriminator = "foo";
    }

    public String getbar() {
        return bar;
    }

    public void setbar(String _v) {
        this.bar = _v;
        __discriminator = "bar";
    }

    public void setToNoMember() {
        __discriminator = null;
    }

    public String _getDiscriminator() {
        return __discriminator;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("discriminator:
   "+discriminator.toString()+"\n");
        buffer.append("foo: "+Integer.toString(foo)+"\n");
        buffer.append("bar: "+bar+"\n");
        return buffer.toString();
    }
}
```

<enumeration>

The mapping for the <enumeration> WSDL type matches the JAX-RPC mapping for a schema enumeration. For example, assume the following WSDL:

```
<simpleType name="Beer">
    <restriction base="xsd:string">
        <enumeration value="Wheat"/>
        <enumeration value="Lambic"/>
        <enumeration value="Bitter"/>
        <enumeration value="Stout"/>
        <enumeration value="Porter"/>
    </restriction>
</simpleType>
```

This maps to the following Java class:

```
public class Beer {

    public static final String XMLBUS_VERSION = ...;

    public static final String TARGET_NAMESPACE =
   "http://xmlbus.com/CORBAApp/xsd";

    private final String _val;

    public static final String _Wheat = "Wheat";
    public static final Beer Wheat = new Beer(_Wheat);

    public static final String _Lambic = "Lambic";
    public static final Beer Lambic = new Beer(_Lambic);

    public static final String _Bitter = "Bitter";
    public static final Beer Bitter = new Beer(_Bitter);

    public static final String _Stout = "Stout";
    public static final Beer Stout = new Beer(_Stout);

    public static final String _Porter = "Porter";
    public static final Beer Porter = new Beer(_Porter);

    protected Beer(String value) {
        _val = value;
    }
```

```
    public String getValue() {
        return _val;
    };

    public static Beer fromValue(String value) {
        if (value.equals("Wheat")) {
            return Wheat;
        }
        if (value.equals("Lambic")) {
            return Lambic;
        }
        if (value.equals("Bitter")) {
            return Bitter;
        }
        if (value.equals("Stout")) {
            return Stout;
        }
        if (value.equals("Porter")) {
            return Porter;
        }
        throw new IllegalArgumentException("Invalid enumeration
   value: "+value);
    };

    public String toString() {
        return ""+_val;
    }
}
```

<fault>

The WSDL `<fault>` element specifies the abstract message format for error messages that might be output as a result of a remote operation. According to the WSDL specification, a fault message must have a single part.

A `<fault>` is mapped to one of the following:

- A `java.rmi.RemoteException` or its subclass
- A service-specific Java exception
- A `javax.xml.rpc.soap.SOAPFaultException`

Service-Specific Exceptions

A service-specific Java exception extends the class `java.lang.Exception` directly or indirectly. The single message part in the WSDL `<message>` (which is referenced from the `<fault>` element) can be a simple XML type or an `xsd:complexType` type.

Example

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

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

This maps to the following Java interface shown in Example 11. Note that `getLastTradePrice()` throws the `InvalidTickerException` based on the mapping of the corresponding `<fault>`:

Example 11: *WSDL <fault> Element Mapped to Java Exception*

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

Links to the XML Schema Specifications

2001 XML Schema	The 2001 XML Schema specification is located at http://www.w3.org/TR/xmlschema-2/. The Schema's URL is located at http://www.w3c.org/2001/XMLSchema.
2000 XML Schema	The 2000 XML Schema specification is located at http://www.w3.org/TR/2000/CR-xmlschema-2-20001024/. The Schema's URL is located at http://www.w3c.org/2000/10/XMLSchema.
1999 XML Schema	The 1999 XML Schema specification is located at http://www.w3.org/TR/1999/WD-xmlschema-2-19991217/. The Schema's URL is located at http://www.w3c.org/1999/XMLSchema.

XAR Properties

XARs contain an XML document that describes the properties of the XAR and the Web services it encapsulates.

Overview

The file `properties.xml` is a sample XAR properties document. Each element in the document specifies certain properties of the XAR and its contents. Using these elements, you can reconstruct the WSDL for all of the services encapsulated by the XAR.

XAR hierarchy

The following example shows the hierarchy of a XARs elements.

```
<xar>
    <dependencies>
        <include>...
        <reference>...
        <resource>...
    </dependencies>
    <service>
        <schemas>
            <schema>
        </schemas>
        <dependencies>
           <resource>
        <soapproperties>
            <targetnamespace>...
            <schemanamespace>...
        </soapproperties>
        <handler>
```

```
        <endpoint>
            <soapproperties>
                <style>...
                <transport>...
            </soapproperties>
            <source>
                <param>...
                ...
            </source>
            <chainSequence>
                <chain>
            </chainSequence>
           <operation>
                <soapproperties>
                    <soapaction>
                    <input>
                        <encodingstyle>...
                        <use>...
                    </input>
                    <output>
                        <encodingstyle>...
                        <use>...
                    </output>
                    <style>...
                </soapproperties>
                 <method>...
                 <display>...
                 <part>
                    <type>
                    <wsdltype>...
                    <mimetype>...
                    <attachable>...
                </part>
                ...
            </operation>
            ...
        </endpoint>
        ...
    </service>
    ...
</xar>
```

Top-level XAR elements

The following example shows the top-level elements of `properties.xml`:

```
1  <xar application="MyApplication">
2     <dependencies>
         ...
      </dependencies>
3     <service name="MyApplicationService">
        ...
      </service>
     ...
      <service name="Service2">
        ...
      </service>
      ...
</xar>
```

1. \<xar> is the top level element of the properties.xml file. It takes one attribute, `application`, which contains the string entered for the **XAR Application Name** in Web Service Builder.
2. <dependencies> lists all the classes that the web services contained in the XAR are dependent on. It contains two sub-elements: `<include>` and `<reference>`.
3. <service> describes a Web service. It has sub-elements describing its endpoints, operations, and SOAP messages. It has one attribute, `name`, which specifies the Web service's name. `properties.xml` has one <service> element for each Web service encapsulated by the XAR.

<chain>

Lists the handlers for each stage in the SOAP lifecycle.

Contained in

```
<xar>
   <service>
      <endpoint>
         <chainSequence>
            <chain>
```

Attributes

`handlerSequence` Lists the handlers in the chain. The handler names used must match the name attribute specified in one of the service level `<handler>` elements. The handlers are listed in the order they are executed.

`type` Specifies at what stage in the SOAP lifecycle the chain is for. The valid values consist of the following:

- `InputStreamHandler`
- `OutputStreamHandler`
- `MessageHandler`

<chainSequence>

Lists the message handlers used by the Web service.

Contained in

```
<xar>
   <service>
      <endpoint>
         <chainSequence>
```

Contains

Up to three <chain> elements, one for each point in the SOAP message lifecycle.

<complexType>

Describes a complex datatype or an array.

Contained in

```
<xar>
   <service>
      <schemas>
         <schema>
            <complexType>
```

Attributes

`name` is the fully qualified name of the datatype.

Contains

`<complexContent>` If the datatype being described is an array, the `<complexType>` element contains a `<complexContent>` element, which in turn contains a `<restriction>` element. The `<restriction>` element takes one attribute, `base`, which specifies the SOAP encoding type for the array. The `<restriction>` element encapsulates an `<attribute>` element. The `<attribute>` element takes two attributes:

- `ref` - Specifies the SOAP encoding type for the array elements.
- `wsdl:arrayType` - Specifies the XSchema type for the array elements.

`<all>` If the datatype being describes is a structure, the `<complexType>` element encapsulates an `<all>` element. The `<all>` element contains one `<element>` element for each component of the structure being described. The `<element>` element takes two attributes:

- `name` - Specifies the name given to the compenent.
- `type` - Specifies the XSchema datatype of the component.

Examples

The following code sample shows a <complexType> element describing an array:

```
<complexType name="ArrayOfstring">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
   wsdl:arrayType="xsd:string[]" />
    </restriction>
  </complexContent>
</complexType>
```

The following code sample shows a <complexType> element describing a structure:

```
<complexType name="LineItem">
  <all>
    <element name="SupplierName" type="xsd:string" />
    <element name="UnitPrice" type="xsd:float" />
    <element name="TotalPrice" type="xsd:float" />
    <element name="Quantity" type="xsd:float" />
    <element name="ProductName" type="xsd:string" />
  </all>
</complexType>
```

<dependencies>

Specifies which classes are included in the XAR's CLASSPATH and which files are directly included.

A XAR can either include a reference to a Java class by having it listed in its CLASSPATH, or it can directly include a copy of the class.

Contained in

```
<xar>
   <dependencies>
   <service>
      <dependencies>
```

This element appears in two places in `properties.xml`:

- The <xar> element includes a <dependencies> element which specifies the java classes that all of the Web services encapsulated in the XAR have access to.
- Each <service> element also includes a <dependencies> element which specifies Java classes that only the specific service can access.

Contains

<include> <reference> <resource>

Examples

```
<dependencies>
  <include>C:\jdk1.3.1\jre\lib\rt.jar</include>
  <reference>C:\jdk1.3.1\lib\tools.jar</reference>
</dependencies>
```

<endpoint>

Describes a Web service endpoint. There is one `endpoint` element for each endpoint in the Web service.

Contained in

```
<xar>
<service>
      <endpoint>
```

Attributes

`name` is the port name entered into Web Service Builder when the service was created.

Contains

`<soapproperties>`	Specifies the style of the SOAP message and the transport used to send and receive SOAP messages.
`<source>`	Specifies any parameters that the Web service need to run. This can include command line parameters.
`<chainSequence>`	Lists the message handlers used by the Web service.
`<operation>`	Describes an endpoint operation There is one description for each operation the endpoint supports.

Examples

```
<endpoint name="MyApplicationPort">
  <soapproperties>
    <style>rpc</style>
    <transport>http://schemas.xmlsoap.org/soap/http</transport>
  </soapproperties>
  <source>
  ...
  </source>
  <chainSequence>
  ...
  </chainSequence>
  <operation name="toString">
  ...
  </operation>
  <operation name="parseShort">
  ...
  </operation>
...
</endpoint>
```

<handler>

Listed for each message handler the Web service can use.

Contained in

```
<xar>
<service>
      <handler>
```

Attributes

`class` Specifies the fully qualified name of the Java class which implements the handler.

`name` Specifies the name of the handler. This value can be any string.

<include>

Specifies the classes that are directly included in the XAR. Multiple files are listed in the same element and separated by semicolons.

Contained in

```
<xar>
   <dependencies>
      <include>
   <service>
      <dependencies>
         <include>
```

<operation>

Describes the interface to the implementation the Web service is using.

Contained in

```
<xar>
   <service>
      <endpoint>
         <operation>
```

Attributes

`name` identifies the operation.

Contains

Information about the data elements passed to and from the method and the method's signature, stored in the following sub-elements:

`<soapproperties>`	Specifies how the incoming and outgoing SOAP messages will be formatted.
`<method>`	Specifies the fully qualified signature of the method implementing the Web service operation.
`<display>`	Specifies the name that is displayed in Web Service Builder.
`<part>`	Describes the data representation of input and output parameters to the operation. There is one `<part>` element for each parameter to the operation and one for the return value.

Examples

```
<operation name="parseShort">
  <soapproperties>
  ...
  </soapproperties>
  <method>parseShort</method>
  <display>public static short parseShort(java.lang.String)
   throws java.lang.NumberFormatException</display>
  <part name="param0" type="in">
  ...
  </part>
  <part name="return" type="out">
  ...
  </part>
</operation>
```

<param>

Specifies a parameter required by a Web service. The value of the element is passed to the Web service as the value of the parameter named.

Contained in

```
<xar>
   <service>
      <endpoint>
         <source>
            <param>
```

Attributes

`name` identifies the parameter.

<part>

The data of the Web service operation that is passed in as parameters and that which is passed out as a return value is described in a <part> element.

Contained in

```
<xar>
   <service>
      <endpoint>
         <operation>
            <part>
```

Attributes

name	Specifies the name of the parameter that appears in Web Service Builder and is derived from the method implementing the Web service operation.
type	Specifies the type of parameter. Valid values consist of the following: • in The in values are passed by value and cannot be changed by the operation. • out The out value represent the return value of the method implementing the Web service operation.

Contains

<type>	Specifies the datatype of the parameter. It take a single attribute, class, which specifies the fully qualified class name that implements the datatype.
<wsdltype>	Specifies the XSchema type that represents the data.
<mimetype>	Specifies the MIME type that represents the data. This information is used to determine which Data Content Handler will be used to decode the data.
<attachable>	Specifies if the data can be made a SOAP attachment. Valid values are true and false.

`<mandatoryAttachment>` Specifies if the data must be passed as an attachment. Valid values are `true` and `false`.

Examples

```
<part name="param0" type="in">
  <type class="java.lang.String" />
  <wsdltype>xsd:base64Binary</wsdltype>
  <mimetype>text/plain</mimetype>
  <attachable>true</attachable>
</part>
```

<reference>

Specifies the entries to include in the CLASSPATH. The entries are valid file names for the system the classes are stored on. Separate entries are placed in the same element and separated by semicolons.

Contained in

```
<xar>
 <dependencies>
      <reference>
   <service>
      <dependencies>
         <reference>
```

<resource>

XARs can have included within them resources such as classes, zip files, archive files, image files, and any other file needed by the XAR's Web service implementations. The resource file details are maintained at the XAR level. If a service is going to use a resource, the service level refers to the named resource at the XAR level.

Contained in

```
<xar>
   <dependencies>
      <resource>
   <service>
      <dependencies>
         <resource>
```

Contains

The `<resource>` element contains the following elements at the XAR level under the `<dependencies>` element:

`<description>`	A text description of the resource.
`<type>`	The type of resource stored. Resources can be almost any kind of file needed by the service, but they are typically the following types: • archive • class • image • properties • schema map • miscellaneous For details of the resources a specific XAR contains, see also the specific XAR file of the properties.xml file you are viewing.
`<path>`	The original load path of the resource when available.

Attributes

`name` at the XAR level names the resources. At the service level, `name` refers to a resource name defined at the XAR level.

<schema>

Specifies the XML namespaces used to define the data used by the Web service.

Note: The attributes for this element should not be edited.

Contained in

```
<xar>
   <service>
      <schemas>
         <schema>
```

Contains

The <schema> element encapsulates a number for <complexType> elements. There is one <complexType> element for each complex datatype or array used by the Web service.

<schemas>

Describes the representations of any arrays and complex datatypes used by the Web service.

Contained in

```
<xar>
   <service>
      <schemas>
```

Contains

A single <schema> element.

<service>

Describes a Web service so that its WSDL can be recreated. It has a single attribute, `name`, that specifies the Web service's name. This is the **Service Name** entered into Web Service Builder when the service was created.

Contained in

```
<xar>
   <service>
```

Contains

`<schemas>`	Specifies the XML schemas representing arrays and complex datatypes used by the Web service.
`<dependencies>`	Lists all the classes that implement the Web service.
`<soapproperties>`	Specifies the namespaces entered into Web Service Builder for **Schema Namespace** and **Target Namespace**.
`<handler>`	Specifies the message handlers that the Web service can use to process SOAP messages.
`<endpoint>`	Describes an endpoint in the Web service.

Examples

```
<service name="MyApplicationService">
  <schemas>
  ...
  </schemas>
  <dependencies />
  <soapproperties>
  ...
  </soapproperties>
  <handler
   class="com.iona.webservices.handlers.message.invocation.rpc.J
   avaHandler" name="default" />
  <endpoint name="MyApplicationPort">
  ...
  </endpoint>
</service>
```

<soapproperties>

Specifies SOAP properties. The properties depend on the element in which the <soapproperties> element is in.

Contained in

```
<xar>
    <service>
        <soapproperties>
        <endpoint>
            <soapproperties>
            <operation>
               <soapproperties>
```

Within <operation>, the <soapproperties> element specifies the encoding method for the operation's incoming and outgoing SOAP messages. The messages can be either encoded or literal and use either RPC or Document styles.

Contains

Within <operation>, <soapproperties> uses the following sub-elements to describe the SOAP encoding style to use:

<soapaction>

<input> Specifies how the incoming SOAP message will be encoded. The <input> element takes two sub-elements:

- <encodingstyle>
  Specifies the XSchema namespace to decode the message.
- <use>
  Specifies what encoding method the message is in. Valid values are encoded or literal.

<output> Specifies how the outgoing SOAP message will be encoded. It takes the same sub-elements as <input>.

<style> Specifies the encoding style to use. Valid values consist of the following:

- rpc
- doc

Examples

The code sample below shows an example of a <soapprotperties> element within an <operation> element:

```
<soapproperties>
  <soapaction />
  <input>
    <encodingstyle>
      http://schemas.xmlsoap.org/soap/encoding/
    </encodingstyle>
    <use>encoded</use>
  </input>
  <output>
    <encodingstyle>
      http://schemas.xmlsoap.org/soap/encoding/
    </encodingstyle>
    <use>encoded</use>
  </output>
  <style>rpc</style>
</soapproperties>
```

See also

<service> <endpoint> <operation>

<source>

Specifies parameters that the Web service needs to run. These include command line parameters, class names, and archive or executable locations, among other things. The parameters listed depend on the type of Web service being implemented. For example, a Web service implementing a CORBA object will have its `IOR` and `ORBinit` parameters listed as parameters.

Contained in

```
<xar>
   <service>
      <endpoint>
         <source>
```

Examples

The following code shows a <source> element for a Web service that implements an EJB:

```
<source>
  <param name="class">java.lang.Short</param>
  <param name="classarchive">none</param>
  <param name="classsource">classpath</param>
  <param name="applicationserver">none</param>
  <param name="jndiname">none</param>
</source>
```

Each parameter needed by the Web service is listed under the <source> element in a <param> element.

Index

Numerics
1999-2001 XML Schema Specification 81

A
anyURI 72
API interfaces
  ClientChain 45
  DataContentHandler 49
  MessageHandler 40
  RPCHandler 42

B
base64Binary 72
binary 72
boolean 54, 72
byte 54, 74

C
CDATA 74
century 74
char 54
CLASSPATH 43, 90
ClientChain 45
Client Code
  J2SE client 3
clients 1
CORBA IDL types supported 62

D
DataContentHandler 49
date 72, 74
dateTime 72
debug option for J2SE client 9
decimal 72
derived XML Schema types supported 74
double 54, 72
duration 72

E
ENTITIES 74
ENTITY 72, 74

F
float 54, 72

G
gDay 72
generating
  J2SE Client interface 7
getContent() 49
getProxy() 11
gMonth 72
gMonthDay 72
gYear 72
gYearMonth 72

H
Handlers
  Chaining 46
  Invocations 42
  Messages 40
hexBinary 72

I
ID 73, 74
IDREF 73, 74
IDREFS 74
int 54, 74
integer 74

J
J2ME Client
  using 18
J2SE Client 3
  in custom code 10
  using 4
J2SE Client coding with getProxy() 11
J2SE Client interface, generating 7
J2SE Client tester 8
java.lang 55
java.math 55
java.util 55
Java to WSDL mapping 51

Java types supported 54
javax.activation 49

L
language 74
long 54, 74

M
mapping between Java and WSDL 51
mapping from WSDL to Java 71
MessageHandler 40
Message Object 40
month 74

N
Name 74
NCName 74
negativeInteger 74
NMTOKEN 75
NMTOKENS 75
nonNegativeInteger 75
nonPositiveInteger 75
normalizedString 75
NOTATION 73, 75

P
positiveInteger 75
primitive XML Schema types 72

Q
QName 75
Qname 73

R
recurringDate 75
recurringDay 75
recurringInstant 73
RPCHandler 42

S
schema specifications 71
Schema specifications, links to 81
short 54, 75
string 73
supported derived XML Schema types 74
supported primitive XML Schema types 72

T
time 73, 75
timeDuration 73
timeInstant 73, 75
timePeriod 75
token 75
type mapping between Java and WSDL 51
Types of clients 1

U
unsignedByte 75
unsignedInt 75
unsignedLong 75
unsignedShort 75
uri 73
uriReference 73
url option for J2SE client 9

W
Web Service
  clients 1
  using 1
Web Service Clients
  Adding a Handler 45
writeTo() 49
wsdl option for J2SE client 9
WSDL to Java mapping 51, 71

X
XML
  Schema specifications, links to 81
xsd 54, 55

Y
year 75