IONA

Artix™

Artix WSDLGen Guide

Version 4.2, March 2007

Making Software Work Together™

Updated: April 3, 2007

Contents

List of Figures

Preface

What is Covered in This Book

This book describes how to use the WSDLGen command-line utility to generate code from a WSDL contract. As well as describing the standard WSDLGen code generating templates, the book explains how to develop custom templates, which you can then use to generate Artix applications implemented in either Java or C++.

Who Should Read This Book

This book is aimed primarily at Java developers and C++ developers who are interested in using code generation to accelerate the process of implementing Web service applications.

This book might also be of some interest to build engineers who need to generate Makefiles and Ant build files based on the content of WSDL contracts.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see Using the Artix Library

Using WSDLGen

This chapter explains how to use the standard templates provided with WSDLGen to generate sample applications in C++ and in Java.

In this chapter

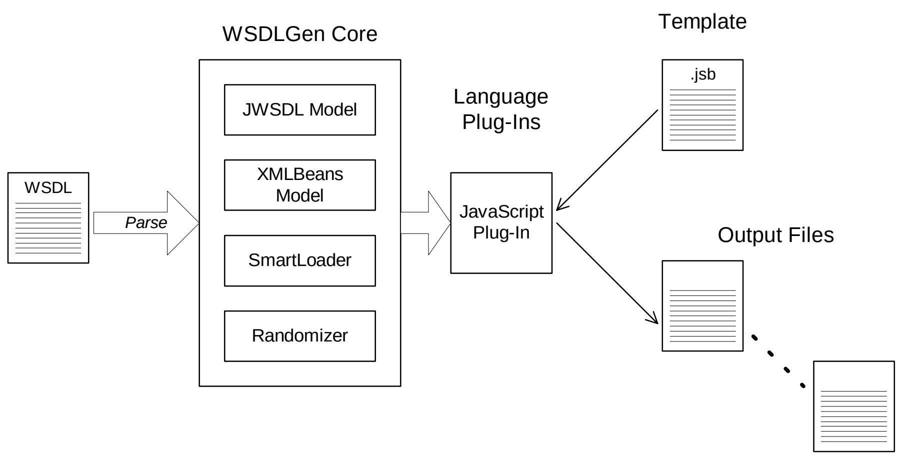
This chapter discusses the following topics:

WSDLGen Architecture

Overview

Figure 1 provides an overview of the WSDLGen code generator architecture.

Figure 1: *WSDLGen Code Generator Architecture*



WSDLGen core

At the heart of the WSDLGen code generator are two object models that represent the parsed contents of the WSDL contract, as follows:

- *JWSDL model*—a model that recognizes the standard elements of a WSDL contract, identifying each type of WSDL element with a Java class.
- *XMLBeans model*—a model that recognizes the elements of an XML schema definition. This model is used to represent the `types` section of a WSDL contract (where the parameter data types are defined).

For more details about the core parsers, see "Parser Overview" on page 50.

The WSDLGen core also provides additional utilities, as follows:

- *SmartLoad utility*—provides the capability to load template files from a well-known location (a search path for SmartLoad can be specified in the WSDLGen configuration file).

- *Randomizer utility*—can be used to generate random parameter data. This is useful for generating sample application code.

Language plug-ins

The WSDLGen architecture has been designed so that it is possible to support additional template languages by adding a plug-in to the core. Currently, WSDLGen supports only the JavaScript language.

JavaScript plug-in

JavaScript (also known as ECMAScript) is an object-based scripting language that has a syntax similar to C or Java. Unlike object-oriented languages, however, JavaScript is not a strongly-typed language.

The JavaScript plug-in enables you to write code-generating templates in the JavaScript language. The choice of JavaScript as the template language has no impact on the choice of generated language: you can use JavaScript templates to generate code in C++, Java or any other language.

Standard templates

WSDLGen provides a standard suite of templates that take a WSDL contract and generate a sample *Artix application* in C++ or Java based on the interfaces defined in the contract.

Custom templates

It is also possible for you to develop your own custom templates. An easy way to get started with developing custom templates is to take one of the standard WSDLGen templates and modify it for your own requirements—see “Developing Basic Templates” on page 21 for details.

Generating Code with the wsdlgen Utility

Syntax of wsdlgen

The `wsdlgen` command-line utility has the following syntax:

```
wsdlgen [-T TemplateFile]* [-java JavaOption]* [-cxx C++Option]*
    [-C ConfigFile] [-D Name=Value]* WSDLFile
```

Where a pair of square brackets, `[]`, denotes an optional part of the syntax and the asterix character, `*`, implies that the preceding option can be repeated 0 or more times.

You must specify the location of a valid WSDL contract file, *`WSDLFile`*. You can also supply the following options:

`-T` *`TemplateFile`*	Specifies the location of a bilingual template file, *`TemplateFile`*, that governs code generation. This option can be repeated, in order to generate code from multiple templates in one invocation.
`-java` *`JavaOption`*	Specifies an option, *`JavaOption`*, for the generation of Artix Java code. The following Java options are supported: `impl`, `server`, `client`, `plugin`, `all`, `ant`.
`-cxx` *`C++Option`*	Specifies an option, *`C++Option`*, for the generation of Artix C++ code. The following C++ options are supported: `impl`, `server`, `client`, `plugin`, `all`, `make`.
`-C` *`ConfigFile`*	Specifies the location of the WSDLGen configuration file, *`ConfigFile`*. If this option is not set, `wsdlgen` reads the default configuration file (located in `%IT_ARTIX_ETC_DIR%/wsdlgen.cfg` on Windows and `$IT_ARTIX_ETC_DIR/wsdlgen.cfg` on UNIX).
`-D` *`Name=Value`*	Specifies the value, *`Value`*, of a JavaScript property, *`Name`*. In particular, the `portType` property can be set in order to specify which WSDL port type you want to generate code for.

Generating code from a specific template (or templates)

You can specify explicitly which templates to run, by invoking the `wsdlgen` utility with the `-T` option. For example, suppose you have a WSDL contract file, `hello_world.wsdl`, and you wish to generate a sample implementation of the `Greeter` port type. You could invoke the `wsdlgen` utility as follows:

```
wsdlgen -D portType=Greeter
    -T templates\cxx\ArtixCxxImplH.jsb
    -T templates\cxx\ArtixCxxImplCxx.jsb
    hello_world.wsdl
```

Variables defined at the command line

The following JavaScript variables can be set at the command line, using the `-D` option of the `wsdlgen` command:

- `portType`—local name of the port type for which code is generated.
- `bindingName`—local name of the binding for which code is generated.
- `serviceName`—local name of the service for which code is generated.
- `portName`—name of the port for which code is generated.

You can set the following combinations of these variables at the command line:

- `serviceName` and `portName`—generate code for the specified service and port.
- `serviceName`—generate code for the specified service and the first port of that service.
- `portType`—generate code for the first service, port, and binding associated with the specified port type.
- `bindingName`—generate code for the first service and port associated with the specified binding.
- *None specified*—generate code for the first service and port in the WSDL contract.

Generating C++ code

When generating C++ code from the standard templates, it is usually simpler to use the `-cxx` switch instead of the `-T` option. The `-cxx` switch enables you to run useful template combinations with a single option. For example, to generate a sample implementation of the `Greeter` port type from the `hello_world.wsdl` file, you could invoke the `wsdlgen` utility as follows:

```
wsdlgen -D portType=Greeter -cxx impl hello_world.wsdl
```

The `-cxx` switch supports the following parameters:

`impl`	For the given *PortType* port type (specified by the `portType` property), generate the files *PortType*`Impl.h` and *PortType*`Impl.cxx` that implement *PortType*. Also, generate stub code and type files for the port type.
`server`	For the given *PortType*, generate a file, *PortType*`ServerSample.cxx`, that implements the `main()` function for a standalone server. Also, generate stub code and type files for the port type.
`client`	For the given *PortType*, generate a file, *PortType*`ClientSample.cxx`, that invokes all of the operations in the *PortType* port type. Also, generate stub code and type files for the port type.
`plugin`	For the given *PortType*, generate all of the files required for a plug-in implementation of the server. The resulting plug-in can then be deployed into an Artix container (see *Developing Artix Applications in C++* for more details).
`all`	Specifying `-cxx all` is equivalent to specifying `-cxx impl -cxx plugin -cxx client`.
`make`	Generate a Makefile for the C++ application. This option must be used in combination with one or more of the following options `-cxx plugin`, `-cxx server`, `-cxx client`, or `-cxx all`.

Generating Java code

When generating Java code from the standard templates, it is usually simpler to use the `-java` switch instead of the `-T` option. The `-java` switch enables you to run useful template combinations with a single option. For example, to generate a sample implementation of the `Greeter` port type from the `hello_world.wsdl` file, you could invoke the `wsdlgen` utility as follows:

```
wsdlgen -D portType=Greeter -java impl hello_world.wsdl
```

The `-java` switch supports the following parameters:

`impl`	For the given *PortType* port type (specified by the `portType` property), generate the files *PortType*`.java` and *PortType*`Impl.java`. Also, generate stub code for the port type.
`server`	For the given *PortType*, generate a file, *PortType*`ServerSample.java`, that implements the `main()` function for a standalone server. Also, generate stub code for the port type.
`client`	For the given *PortType*, generate a file, *PortType*`ClientSample.java`, that invokes all of the operations in the *PortType* port type. Also, generate stub code for the port type.
`plugin`	For the given *PortType*, generate all of the files required for a plug-in implementation of the server. The resulting plug-in can then be deployed into an Artix container (see *Developing Artix Applications in Java* for more details).
`all`	Specifying `-java all` is equivalent to specifying `-java impl -java plugin -java client`.
`ant`	Generate an Apache Ant build file for the Java application.

Standard WSDLGen Templates

Overview

WSDLGen provides a variety of standard templates that you can use to generate sample application code directly from a WSDL contract. These templates are located in the `ArtixInstallDir/artix/Version/templates` directory.

C++ templates

Table 1 lists the WSDLGen templates that can be used to generate C++ examples.

Table 1: *WSDLGen Templates for Generating C++ Code*

C++ Template File	Description
`ArtixCxxActivatorCxx.jsb`	Generate the implementation of a *service activator* class (to use in conjunction with a container plug-in). When a service is deployed in an Artix container, the service activator makes it possible to start and stop the service at runtime using the `it_container_admin` utility.
`ArtixCxxActivatorH.jsb`	Generate the header file for the service activator class.
`ArtixCxxClientMain.jsb`	Generate a sample C++ client.
`ArtixCxxDeployDescr.jsb`	Generate an XML deployment descriptor for deploying a plug-in into the Artix container.
`ArtixCxxImplCxx.jsb`	Generate an outline servant implementation for the port type specified by the `portType` property.
`ArtixCxxImplH.jsb`	Generate the header file for the servant implementation.
`ArtixCxxMakefile.jsb`	Generate a sample Makefile.
`ArtixCxxPlugin.jsb`	Generate a sample plug-in implementation (for deploying into an Artix container).
`ArtixCxxPluginScript.jsb`	Generate a script that starts an Artix container process and deploys the plug-in into the container.
`ArtixCxxServerMain.jsb`	Generate a sample server `main()` function (for a standalone application).

Table 1: *WSDLGen Templates for Generating C++ Code*

C++ Template File	Description
ArtixCxxStubTypes.jsb	Generate stub code for specified port type specifed by the portType property.

Java templates

Table 2 lists the WSDLGen templates that can be used to generate Java examples.

Table 2: *WSDLGen Templates for Generating Java Code*

Java Template File	Description
ArtixJavaActivator.jsb	Generate a *service activator* class (to use in conjunction with a container plug-in). When a service is deployed in an Artix container, the service activator makes it possible to start and stop the service at runtime using the it_container_admin utility.
ArtixJavaAntfile.jsb	Generate a sample build.xml file, for use with the Apache Ant build utility.
ArtixJavaClientMain.jsb	Generate a sample Java client.
ArtixJavaDeployDescr.jsb	Generate an XML deployment descriptor for deploying a plug-in into the Artix container.
ArtixJavaImpl.jsb	Generate an outline implementation for the port type specified by the portType property.
ArtixJavaPluginFactory.jsb	Generate a sample plug-in factory implementation (for deploying into an Artix container).
ArtixJavaPlugin.jsb	Generate a sample plug-in implementation (for deploying into an Artix container).
ArtixJavaPluginScript.jsb	Generate a script that starts an Artix container process and deploys the plug-in into the container.
ArtixJavaServerMain.jsb	Generate a server main() function (for deploying the server in standalone mode).
ArtixJavaStubTypes.jsb	Generate stub code and type files for port type specifed by the portType property.

WSDLGen Configuration File

Overview

The `wsdlgen` utility has its own configuration file, which is defined in XML format. This configuration file enables you to customize WSDLGen by:

- Setting JavaScript variables.
- Setting SmartLoader paths.

Default location

The WSDLGen configuration is stored at the following default location:

```
ArtixInstallDir/artix/Version/etc/wsdlgen.cfg
```

Setting JavaScript variables

You can initialize JavaScript variables from the WSDLGen configuration file, as shown in Example 1.

Example 1: *Setting JavaScript Variables in the Configuration File*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdlgen>
    <defines>
        <foo>fooValue</foo>
        <!-- ... -->
    </defines>
    ...
</wsdlgen>
```

Where the `defines` element can contain any number of entries of the form *<VariableName>Value</VariableName>*. Each configuration entry of this form is equivalent to including the following JavaScript code at the top of your template:

```
var VariableName = "Value";
```

Setting SmartLoader paths

You can define a search path for the smart loader utility in the WSDLGen configuration file by adding a sequence of `path` elements inside an enclosing `paths` element, as shown in Example 2.

Example 2: *Setting SmartLoader Paths in the Configuration File*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdlgen>
    <paths>
        <path>/home/fflintstone/.wsdlgen</path>
        <path>/usr/local/templates/wsdlgen</path>
        <!-- ... -->
    </paths>
    ...
</wsdlgen>
```

When searching for scripts included through the smart loader mechanism, WSDLGen searches the directories listed in the `paths` element. For more details about the smart loader utility, see "smartLoader utility" on page 33.

Developing Basic Templates

This chapter provides an introduction to the subject of writing your own templates for generating code in Java and C++.

In this chapter

This chapter discusses the following topics:

Writing Custom Templates

Overview

The simplest approach to take when writing a custom template is to take one of the WSDLGen samples and modify it to your own requirements. This chapter aims to provide you with enough information to understand the sample templates and to use the WSDLGen programming interfaces effectively.

Running a custom template

To generate code using a custom template, specify the template to the `wsdlgen` utility using the `-T` command-line option. For example, to generate code from a `FooBar.wsdl` WSDL contract file using a `CustomTempl.jsb` custom template, you would invoke the `wsdlgen` utility as follows from the command line:

```
wsdlgen -T CustomTempl.jsb FooBar.wsdl
```

For more details about the `wsdlgen` command-line syntax, see "Generating Code with the wsdlgen Utility" on page 12.

Bilingual files

WSDLGen templates are written in a special file format known as a *bilingual file* and identified by the `.jsb` file suffix. The bilingual file format enables you to freely mix the JavaScript language and the target language together in the one file. For details, see "Bilingual Files" on page 24.

Predefined objects

To provide you with convenient access to data and objects derived from the WSDL contract, WSDLGen creates predefined objects in JavaScript. For example, the `wsdlModel` object provides access to a complete parse tree of the WSDL contract (using the JWSDL API).

For details, see "Predefined Objects" on page 29.

Built-in APIs

A few different APIs are provided for writing templates, as follows:

- *WSDLGen API for Artix Java*—utility functions for generating Artix Java code from WSDL.
- *WSDLGen API for Artix C++*—utility functions for generating Artix C++ code from WSDL.
- *WSDLGen randomizer*—a random data generator, used internally by WSDLGen to generate random parameter values.
- *JWSDL API*—a WSDL parser based on the JWSDL standard. See "The JWSDL Parser" on page 59 for details.
- *XMLBeans API*—an XML schema parser. See "The XMLBeans Parser" on page 70 for details.

Bilingual Files

Overview

The basic purpose of a JavaScript template in WSDLGen is to generate code in a *target language* (such as Java or C++). Consequently, if a code generating template was written in pure JavaScript, it would contain a large number of print directives to produce the required target code. In practice, this style of coding quickly leads to templates that are virtually illegible (you might be familiar with this sort of problem in the context of HTML-generating servlet code).

To solve this difficulty, WSDLGen introduces the concept of a *bilingual file* for developing code-generating templates. The basic idea of the bilingual file is that a set of escape sequences enable you to switch back and forth between the generating language and the target language. Example 3 shows a sample outline of such a bilingual file, with one section of the file (enclosed between `[***` and `***]`) expressed in the target language.

Example 3: *Sample Outline of a Bilingual File.*

```
// JavaScript Bilingual File
openOutputFile(PathName)

// Put JavaScript code here...
...

[***
    // Put TargetLanguage code here...
    ...
***]

closeOutputFile()
```

Opening and closing the output file

A bilingual file must be associated with an output destination. You can specify an output file for the generated code by calling the following function in your script (typically, at the start of the template):

```
openOutputFile(PathName)
```

Where *PathName* specifies the path to the generated output file. On UNIX platforms, an alternative form of the `openOutputFile()` function is available, which lets you set file permissions on the output file:

```
openOutputFile(PathName, Permissions)
```

Where *Permissions* is a string value formatted in the same way as a standard `chmod` permission string. For example, the string, `u=rwx,g=rx,o=x`, would give full permissions to the owner, read and execute permissions to the group, and execute permission to all others. For full details of the permission string syntax, enter `man chmod` at the command line.

You can close the output file by calling the following function (typically, at the end of the template):

```
closeOutputFile()
```

The call to `openOutputFile()` establishes an association between the destination file, *PathName*, and the blocks of generated code written in the target language. All of the generated code is sent to the file, *PathName*, specified by the `openOutputFile()` function.

Note: If `openOutputFile()` is not called, the output is directed to standard out by default.

Output text delimiters

Blocks of generated code are delimited by the output text delimiters shown in Table 3

Table 3: *Character Sequences for Delimiting Output Text*

Character Sequence	Description
`[***`	Beginning of a code block written in the target language.
`***]`	End of the code block written in the target language.

Escaping within output text

Within the scope of the output text delimiters, you can escape back to JavaScript using the escape characters shown in Table 4.

Table 4: *Escape Characters Used in Output Text*

Escape Sequence	Description
`$VarName$`	Substitute a JavaScript variable, `VarName`, embedding it in a line of output text—see “Variable escape” on page 26.
`@JavaScript`	Escape to a line of JavaScript—see “Line escape” on page 27.

Variable escape

Within the scope of the output text delimiters, you can substitute the value of a JavaScript variable using the dollar sign, `$`, as an escape character. To make the substitution, enclose the JavaScript variable name between two dollar signs, `$VarName$`.

For example, if `intfName` is a JavaScript variable that holds a WSDL port type name, you could declare a Java class to implement this port type using the following fragment of bilingual file.

```
// JavaScript Bilingual File
openOutputFile(PathName)

[***
```

```
public class $intfName$Impl implements java.rmi.Remote {
***]

// More script (not shown)...
...
closeOutputFile()
```

The implementation class name is derived by adding the `Impl` suffix to the porty type name. For example, if generating code for the `Greeter` port type, `$intfName$Impl` would expand to `GreeterImpl`.

Line escape

Within the scope of the output text delimiters, you can escape to a line of JavaScript code by putting the *at* symbol, `@`, at the start of a line (as the first non-whitespace character).

For example, the following bilingual file generates a Java function, `ListInterfaceOps()`, that lists all of the operations in the current WSDL interface.

```
// JavaScript Bilingual File
...
openOutputFile(PathName)

[***
    ...
    public static void ListInterfaceOps() {
        System.out.println("Operation is one of: ");
        @for (var i = 0; i < numOps; i++) {
        System.out.println("  $operations[i].getName()$");
        @}
    }
}
***]

closeOutputFile()
```

Unlike the variable escape mechanism, `$VarName$`, the line escape does *not* produce any output text as a side effect of its execution. While the line enclosing a variable escape sequence, `$VarName$`, is implicitly enclosed in a print statement, the line escaped by the *at* symbol, `@`, is not printed.

Escaping the escape characters

Occasionally, you might need to output the dollar, `$`, and at sign, `@`, character literals inside the scope of an output text block. For this purpose, WSDLGen defines the `$dollar$` and `$at$` variables, which resolve to literal dollar, `$`, and literal *at*, `@`, inside an output text block.

For example, you could insert the `$` and `@` character literals into your output code, as shown in the following example:

```
// JavaScript Bilingual File
...
[***
    cout << "Here is a contrived example," << endl;
    cout << "that shows how to get the $dollar$ and $at$"
         << " literals into your output." << endl;
***]
```

Predefined Objects

Overview

The programming interface provided by WSDLGen includes a number of predefined JavaScript objects. Some of these predefined objects are simple variables (for example, `intfName`, containing the name of the current port type), whilst others provide access to particular APIs (for example, `wsdlModel`, which provides access to the JWSDL parser API).

List of predefined objects

Table 5 shows the list of JavaScript objects predefined by WSDLGen.

Table 5: *Predefined JavaScript Objects*

JavaScript Object	Description
`bindingName`	Local part of the binding name for which code is generated. You can set this variable when you invoke the `wsdlgen` command (see "Variables defined at the command line" on page 13).
`cxxIntfName`	A name derived from `intfName` by replacing any dot characters, ., with underscores, _. For example, `simple.simpleIntf` would become `simple_simpleIntf`.
`cxxNamespace`	The C++ namespace in which to define the generated implementation classes. Its value is derived from the WSDL target namespace.
`cxxServiceName`	A name derived from `serviceName` by replacing any dot characters, ., with underscores, _. For example, `simple.simpleService` would become `simple_simpleService`.
`intfName`	A name derived from the port type name, `portType`, by dropping the `PortType` suffix (if any).
`javaIntfName`	A name derived from `intfName` by removing any dot characters, ., and capitalizing the subsequent letter. For example, `simple.simpleIntf` would become `SimpleSimpleIntf`.

Table 5: *Predefined JavaScript Objects*

JavaScript Object	Description
`javaPackage`	The Java package name in which to define the generated implementation classes. Its value is derived from the WSDL target namespace.
`javaServiceName`	A name derived from `serviceName` by removing any dot characters, `.`, and capitalizing the subsequent letter. For example, `simple.simpleService` would become `SimpleSimpleService`.
`jsModel`	A wrapper for the `wsdlModel` object.
`operations[]`	An array of operation objects, of `javax.wsdl.Operation` type. See "JWSDL Parser Classes" on page 65 for details.
`parametersList`	[An instance of the utility class, `com.iona.wsdlgen.common.ParametersList`. This object enables you to obtain a list of parts and faults for every WSDL operation.]
`portName`	Port name for which code is generated. You can set this variable when you invoke the `wsdlgen` command (see "Variables defined at the command line" on page 13).
`portType`	Local part of the port type name for which code is generated. You can set this variable when you invoke the `wsdlgen` command (see "Variables defined at the command line" on page 13).
`randomizer`	An instance of a WSDLGen utility that generates random numbers. The WSDLGen templates use this object to generate random parameters.
`schemaModel`	An instance of the `org.apache.xmlbeans.SchemaTypeLoader` class, which provides access to an XML schema parser. See "The XMLBeans Parser" on page 70 for details.

Table 5: *Predefined JavaScript Objects*

JavaScript Object	Description
`smartLoader`	An instance of a WSDLGen utility that imports JavaScript or bilingual files from a well-known location. The search path for the smart loader can be specified in the WSDLGen configuration file.
`serviceName`	Local part of the service name for which code is generated. You can set this variable when you invoke the `wsdlgen` command (see "Variables defined at the command line" on page 13).
`tns`	The namespace of the port type, binding, and service elements. Specifically, this variable contains the value of the `targetNamespace` attribute from the `definitions` element in the WSDL contract.
`wsdlModel`	An instance of the `javax.wsdl.Definition` class, which provides access to a JWSDL parser. See "Parsing WSDL" on page 49 for details.
`wsdlFile`	The location of the WSDL contract file.

WSDL and schema models

The following objects represent the roots of the WSDL model and the XML schema model respectively:

- `wsdlModel`
- `schemaModel`

These parser objects provide a complete model of the WSDL elements and XML schema types defined in the WSDL contract. Typically, it is not necessary to use these APIs in a basic template. For more advanced applications, however, see "Parsing WSDL" on page 49 for details about the parser APIs.

operations[] array

An array of operation objects representing all of the operations in the `portType` port type. The operation objects are instances of `javax.wsdl.Operation`, which is part of the JWSDL API.

For example, you can print out the names of all the operations in the portType port type as follows:

```
// JavaScript Bilingual File
...
for (var i=0; i < operations.length; i++) {
[***
    System.out.println("Operation["+i+"] name = "
        + $operations[i].getName()$
    );
***]
}
```

For more details about the javax.wsdl.Operation class, see "JWSDL Parser Classes" on page 65.

---

parametersList object

The parametersList object provides a method, getPartsAndFaults(), that provides access to all of the message parts and faults associated with a particular WSDL operation.

For example, to obtain the parts and faults associated with the ith operation of the current WSDL interface, make the following JavaScript call:

```
var partsAndFaults = parametersList.getPartsAndFaults(
       portType + operations[i].getName()
   )
```

Where the argument to getPartsAndFaults() is a key, consisting of a port type name concatenated with an operation name.

By calling partsAndFaults.parts()[k]—where k lies in the range 0 to partsAndFaults.parts().length—you can obtain a PartHolder object, which holds the following items:

- partsAndFaults.parts()[k].getPart()—returns the javax.wsdl.Part object that represents the current part.
- partsAndFaults.parts()[k].getDirection()—returns one of the following direction flag values: DIRECTION_IN, or DIRECTION_OUT.

By calling `partsAndFaults.faults()[k]`—where `k` lies in the range `0` to `partsAndFaults.faults().length`—you can obtain a `FaultHolder` object, which holds the following items:

- `partsAndFaults.faults()[k].getName()`—returns the fault name.
- `partsAndFaults.faults()[k].getParts()`—returns the array of `javax.wsdl.Part` objects contained in the fault.

---

smartLoader utility

The *smart loader utility* provides a way of including files located relative to a well-known directory (or directories). For example, if you are implementing a custom template, you could include the contents of the file, `CustomUtils/MyUtilities.js`, at the start of your template by calling `smartLoad()` as follows:

```
# JavaScript Bilingual File
smartLoad("CustomUtils/MyUtilities.js");
...
```

Where the included file, `CustomUtils/MyUtilities.js`, is located under one of the directories listed in the `paths` element in the WSDLGen configuration file. Example 4 shows an example of a configuration file that specifies two path directories, with each directory enclosed in a `path` element. The directories are searched in the order in which they appear in the configuration file.

Example 4: *Smart Loader Path in the WSDLGen Configuration File*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdlgen>
    <paths>
        <path>/home/fflintstone/.wsdlgen</path>
        <path>/usr/local/templates/wsdlgen</path>
        <!-- ... -->
    </paths>
    ...
</wsdlgen>
```

Generating Java Code

Overview

This section provides a brief overview of the most important WSDLGen functions for generating Java code. The following topics are described:

- Indentation level.
- Generally useful functions.
- Generating operation calls in a Java consumer.
- Catching fault exceptions.
- Generating operation implementations.

Indentation level

Some of the functions in the WSDLGen API generate multi-line output. To give you some control over the layout of the resulting output, these functions take an integer parameter, `IndentLevel`, that lets you specify the initial level of indentation.

Generally useful functions

Table 6 summarizes the most useful general purpose functions in the WSDLGen API for generating Java code .

Table 6: *General Purpose Functions*

Function	Description
`namespaceToURL(URLString)`	Replace every occurrence of the period character, `.`, with the slash character, `/`, in the string, `URLString`.
`getJavaPackage(NS)`	Maps a WSDL namespace string, `NS`, to a Java package name, using the standard JAX-RPC mapping rule.

Example 5 shows how you might use the `namespaceToURL()` function in a JavaScript bilingual file. In this example, the function is used to generate the path to a sample client implementation.

Example 5: *The namespaceToURL() Function*

```
// JavaScript Bilingual File
openOutputFile(namespaceToURL(javaPackage) + intfName +
    "ClientSample.java")
```

The preceding code fragment reflects the fact that it is conventional for a Java class such as, `com.iona.hello_world_soap_http.GreeterClientSample`, to be stored at the file location, `com/iona/hello_world_soap_http/GreeterClientSample.java`.

Generating operation calls in a Java consumer

Table 7 summarizes the WSDLGen functions that you use to generate a WSDL operation call in the Java language.

Table 7: *Functions for Generating an Operation Call in Java*

Function	Description

`artixJavaOperParamDecl(`
`PortTypeName,`
`Op,`
`IndentLevel`
`)`
	Declare all of the parameters that are required for this operation invocation.

`artixJavaPopulateAllInParts(`
`PortTypeName,`
`Op,`
`IndentLevel`
`)`
	Populate each of the request parameters with random data.

`artixJavaClientServerCall(`
`PortTypeName,`
`Op,`
`IndentLevel`
`)`
	Call the operation, *Op*.

`artixJavaPrintAllOutParts(`
`PortTypeName,`
`Op,`
`IndentLevel`
`)`
	Print out all of the returned parameters.

All of the functions in Table 7 take the same kind of arguments:

- *PortTypeName* is the local name of the port type on which the operation is defined;
- *Op* is a `javax.wsdl.Operation` instance that represents the operation in the WSDL model;

- *IndentLevel* specifies how many levels of indentation are applied to the generated code.

Example 6 shows how to use the preceding functions to generate the operation calls in a Web service client. The code iterates over every operation in the current port type, generating code to declare and initialize the parameters and then call the operation.

Example 6: *Generating Operation Calls in Java*

```
// JavaScript Bilingual File
...
for (var i = 0; i < operations.length; i++) {
[***
   {
       $artixJavaOperParamDecl(portType, operations[i], 3)$

        $artixJavaPopulateAllInParts(portType, operations[i], 3)$

        $artixJavaClientServerCall(portType, operations[i], 3)$

       $artixJavaPrintAllOutParts(portType, operations[i], 4)$
   }
***]
}
```

For example, if the preceding script is run against the `hello_world.wsdl` file, it generates the Java code shown in Example 7.

Example 7: *Generated Operation Calls in Java*

```
// Java
        ...
        {
            java.lang.String theResponse;

            theResponse = impl.sayHi();

            System.out.println("sayHi RECVD:");
            System.out.println(theResponse);
            System.out.println();
        }
        {
            java.lang.String me;
            java.lang.String theResponse;
```

Example 7: *Generated Operation Calls in Java*

```
            me = "Curry";

            theResponse = impl.greetMe(me);

            System.out.println("greetMe RECVD:");
            System.out.println(theResponse);
            System.out.println();
        }
```

Catching fault exceptions

To help you generate the code for catching a fault exception, WSDLGen provides the `parametersList` object, which enables you to obtain a list of faults for any WSDL operation by calling the `parametersList.getPartsAndFaults()` method. For details of how to use the `parametersList` object, see "parametersList object" on page 32.

Example 8 shows an example of how to generate Java code to catch the fault exceptions associated with the operation, `operation[i]`.

Example 8: *Generating Code to Catch a Fault Exception*

```
// JavaScript Bilingual File
...
[***
    ...
    @var partsAndFaults =
   parametersList.getPartsAndFaults(
       portType + operations[i].getName()
   )
    @if (partsAndFaults.faults().length != 0) {
        try {
    @}
            // Code to call i'th operation (not shown)
            ...
    @if (partsAndFaults.faults().length != 0) {
        }
        @var faults =
   artixJavaGetFaultNames(partsAndFaults.faults())
        @for each (fault in faults) {
        catch ( $fault$ ex ) {
            System.out.println("Exception: $fault$ has
   Occurred.");
            ex.printStackTrace();
        }
```

Example 8: *Generating Code to Catch a Fault Exception*

```
        @}
    @}
    ...
***]
```

For example, if you run the preceding script against the `userfault.wsdl` file, you would obtain the Java code shown in Example 9. In Artix, the name of the fault exception class is equal to the name of the corresponding XML schema fault type, with the first letter uppercased.

Example 9: *Generated Java Catch Clause*

```
// Java
...
try {
    // Code to call i'th operation (not shown)
    ...
}
catch ( My_exceptionType_Exception ex ) {
    System.out.println(
        "Exception: My_exceptionType_Exception has Occurred."
    );
    ex.printStackTrace();
}
```

Generating operation implementations

Table 8 summarizes the WSDLGen functions that you use to generate an operation implementation in the Java language.

Table 8: *Functions for Generating a Java Implementation Class*

Function	Description

`artixJavaOperSig(`
   `PortTypeName``,`
   `Op``,`
   `IndentLevel`
`)`
	Return the signature of the operation, *`Op`*, in the port type, *`PortTypeName`*, as it appears in the Java implementation class. If needed, this function also declares the relevant fault exceptions.

Table 8: *Functions for Generating a Java Implementation Class*

Function	Description
`artixJavaPopulateAllOutParts(` *`PortTypeName`*`,` *`Op`*`,` *`IndentLevel`* `)`	Generates code to populate all of the out parameters for the operation, *`Op`*, in the port type, *`PortTypeName`*.

Example 10 shows a fragment of a script that uses the preceding functions to generate a Java implementation class. The script iterates over all of the operations in the current port type, generating an implementing method for each one.

Example 10: *Generating a Java Implementation Class*

```
// JavaScript Bilingual File
...
[***
package $javaPackage$;
@if (isHolderTypesRequired(true) == true) {
import $javaPackage$.holders.*;
@}
/**
 * $javaPackage$.$intfName$Impl
 */
public class $intfName$Impl implements java.rmi.Remote {

@var numOps = operations.length
@for (var i = 0; i < numOps; i++) {

    $artixJavaOperSig(portType, operations[i], 0, "")$ {
        System.out.println("$operations[i].getName()$ invoked");
        $artixJavaPopulateAllOutParts(portType, operations[i],
   2)$
    }

@}
}
***]
```

For example, if the preceding script is run against the `hello_world.wsdl` file, it generates the Java code shown in Example 11.

Example 11: *Generated Java Implementation Class*

```
package com.iona.hello_world_soap_http;

import com.iona.hello_world_soap_http.*;
/**
 * com.iona.hello_world_soap_http.GreeterImpl
 */
public class GreeterImpl implements java.rmi.Remote {


    public java.lang.String sayHi() {
        System.out.println("sayHi invoked");
        java.lang.String theResponse = new java.lang.String();
        theResponse = "Curry";
        return theResponse;
    }


    public java.lang.String greetMe(java.lang.String me) {
        System.out.println("greetMe invoked");
        java.lang.String theResponse = new java.lang.String();
        theResponse = "Wsdlgen";
        return theResponse;
    }

}
```

Generating C++ Code

Overview

This section provides a brief overview of the most important WSDLGen functions for generating C++ code. The following topics are described:

- Generating operation calls in a C++ consumer.
- Functions for generating C++ implementations.
- Generating an implementation header.
- Generating a C++ implementation class.

Generating operation calls in a C++ consumer

Table 9 summarizes the WSDLGen functions that you use to generate a WSDL operation call in the C++ language.

Table 9: *Functions for Generating an Operation Call in C++*

Function	Description

`artixCxxOperParamDecl(`
   `PortTypeName``,`
   `Op``,`
   `IndentLevel`
`)`
	Declare all of the parameters required for this operation invocation.

`artixCxxOperParamPopulate(`
   `PortTypeName``,`
   `Op``,`
   `IndentLevel``,`
   `IgnoreDirection``,`
   `Print`
`)`
	Populate each of the request parameters with random data. The *`IgnoreDirection`* parameter indicates which of the parameters should *not* be initialized. It can take either of the following values:
	• `DIRECTION_OUT`—appropriate for consumer code, or
	• `DIRECTION_IN`—appropriate for implementation code.
	The *`Print`* parameter indicates whether to generate code that prints out the parameters. The value `true` means print and the value `false` means do not print.

`artixCxxOperParamCall(`
   `PortTypeName``,`
   `Op`
`)`
	Generate the parameter list needed for calling the operation, *`Op`*.

Most of the functions in Table 9 take the following arguments:

- *PortTypeName* is the local name of the port type on which the operation is defined;
- *Op* is a `javax.wsdl.Operation` instance that represents the operation in the WSDL model;
- *IndentLevel* specifies how many levels of indentation are applied to the generated code.

Example 12 shows how to use the preceding functions to generate the operation calls in a Web service client. The code iterates over every operation in the current port type, generating code to declare and initialize the parameters and then call the operation.

Example 12: *Generating Operation Calls in C++*

```
// JavaScript Bilingual File
...
for (var i = 0; i < operations.length; i++) {
[***
{
    $artixCxxOperParamDecl(portType, operations[i], 1)$

    $artixCxxOperParamPopulate(portType, operations[i], 1,
   DIRECTION_OUT, false)$


   client->$operations[i].getName()$($artixCxxOperParamCall(port
   Type, operations[i])$);
}

***]
}
```

For example, if the preceding script is run against the `hello_world.wsdl` file, it generates the C++ code shown in Example 13.

Example 13: *Generated Operation Calls in C++*

```
// Java

{
    IT_Bus::String theResponse;

    client->sayHi(theResponse);
```

Example 13: *Generated Operation Calls in C++*

```
}

{
    IT_Bus::String me;
    IT_Bus::String theResponse;

    me = "Curry";

    client->greetMe(me, theResponse);
}
```

Functions for generating C++ implementations

Table 10 summarizes the WSDLGen functions that you use to generate an implementation class in the C++ language.

Table 10: *Functions for Generating a C++ Implementation*

Function	Description

`artixCxxOperSig(`
   `Prefix``,`
   `PortTypeName``,`
   `Op``,`
   `IndentLevel``,`
   `Trailing`
`)`
	Return the signature of the operation, *`Op`*, in the port type, *`PortTypeName`*. This function can be used in various contexts; that is, either in the header file or the C++ implementation file. The Prefix string—which should be in the format *`CxxNamespace`*`::`*`ClassName`*`::`—allows you to prefix the function signature with the name of the implementation class. The trailing string, *`Trailing`*, is appended to the end of the generated signature.

`artixCxxOperImpl(`
   `PortTypeName``,`
   `Op``,`
   `echoParams`
`)`
	Generates code to populate the out parameters of the operation, *`Op`*, in the port type, *`PortTypeName`*. Normally, the parameters are populated with random values. However, if you specify the *`echoParams`* flag to be `true`, any parameters declared both IN and OUT will echo the incoming value back to the caller.

Generating an implementation header

Example 14 shows a script that uses the preceding functions to generate an implementation header file. The script iterates over all of the operations in the current port type, generating a function declaration for each one.

Example 14: *Generating a C++ Implementation Header*

```
// JavaScript Bilingual File
...
var UpperInfName = intfName.toUpperCase()
[***
#ifndef IT_Bus_$cxxNamespace$_$UpperInfName$IMPL_INCLUDED_
#define IT_Bus_$cxxNamespace$_$UpperInfName$IMPL_INCLUDED_

#include "$intfName$Server.h"

namespace $cxxNamespace$
{
    class $intfName$Impl : public $intfName$Server
    {
      public:

        $intfName$Impl(IT_Bus::Bus_ptr bus);

        virtual ~$intfName$Impl();

        virtual IT_Bus::Servant*
        clone() const;


@var numOps = operations.length
@for (var i = 0; i < numOps; i++) {

    $artixCxxOperSig("", portType, operations[i], 2, ";\n")$
@}

    };
}

#endif
***]
```

For example, if the preceding script is run against the `hello_world.wsdl` file, it generates the C++ header file shown in Example 15.

Example 15: *Generated C++ Implementation Header*

```
#ifndef
    IT_Bus_COM_IONA_HELLO_WORLD_SOAP_HTTP_GREETERIMPL_INCLUDED_
#define
    IT_Bus_COM_IONA_HELLO_WORLD_SOAP_HTTP_GREETERIMPL_INCLUDED_

#include "GreeterServer.h"

using namespace COM_IONA_HELLO_WORLD_SOAP_HTTP;

namespace COM_IONA_HELLO_WORLD_SOAP_HTTP
{
    class GreeterImpl : public GreeterServer
    {
      public:

        GreeterImpl(
            IT_Bus::Bus_ptr bus
        );

        virtual ~GreeterImpl();

        virtual IT_Bus::Servant*
        clone() const;

        void
        sayHi(
            IT_Bus::String& theResponse
        ) IT_THROW_DECL((IT_Bus::Exception));

        void
        greetMe(
            const IT_Bus::String& me,
            IT_Bus::String& theResponse
        ) IT_THROW_DECL((IT_Bus::Exception));

    };
}

#endif //IT_SIMPLE_SERVICE_IMPL_INCLUDED_
```

Generating a C++ implementation class

Example 16 shows a script that uses the functions from Table 10 on page 43 to generate a C++ implementation class. The script iterates over all of the operations in the current port type, generating a member function for each one.

Example 16: *Generating a C++ Implementation Class*

```
// JavaScript Bilingual File
...
[***
#include "$intfName$Impl.h"
#include <it_cal/cal.h>
#include <it_cal/iostream.h>
#include <it_bus/to_string.h>

IT_USING_NAMESPACE_STD
using namespace $cxxNamespace$;
using namespace IT_Bus;

$intfName$Impl::$intfName$Impl(IT_Bus::Bus_ptr bus) :
   $intfName$Server(bus)
{

}

$intfName$Impl::~$intfName$Impl()
{

}

IT_Bus::Servant*
$intfName$Impl::clone() const
{
    return new $intfName$Impl(get_bus());
}

***]

var numOps = operations.length
for (var i = 0; i < numOps; i++) {
[***
$artixCxxOperSig(intfName + "Impl::", portType, operations[i],
   0, "")$
{
    $artixCxxOperImpl(portType, operations[i], true)$
```

Example 16: *Generating a C++ Implementation Class*

```
}

***]

}
```

For example, if the preceding script is run against the `hello_world.wsdl` file, it generates the C++ implementation class shown in Example 17.

Example 17: *Generated C++ Implementation Class*

```
#include "GreeterImpl.h"
#include <it_cal/cal.h>
#include <it_cal/iostream.h>
#include <it_bus/to_string.h>

IT_USING_NAMESPACE_STD
using namespace $cxxNamespace$;
using namespace IT_Bus;

GreeterImpl::GreeterImpl(
    IT_Bus::Bus_ptr bus
) : GreeterServer(bus)
{
    // complete
}

GreeterImpl::~GreeterImpl()
{
   // complete
}

IT_Bus::Servant*
$intfName$Impl::clone() const
{
    return new $intfName$Impl(get_bus());
}

void
GreeterImpl::sayHi(
    IT_Bus::String& theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    theResponse = IT_Bus::String("Curry");
```

Example 17: *Generated C++ Implementation Class*

```
}

void
GreeterImpl::greetMe(
    const IT_Bus::String& me,
    IT_Bus::String& theResponse
) IT_THROW_DECL((IT_Bus::Exception))
{
    theResponse = me;
}
```

Parsing WSDL

This chapter introduces you to the subject of parsing WSDL using the low-level APIs, JWSDL and Apache XMLBeans. The higher-level WSDLGen API is built on top of these basic parsing APIs.

In this chapter

This chapter discusses the following topics:

Parser Overview

Overview

The parsing APIs that underly WSDLGen are taken from the following open source products:

- WSDL4J (reference implementation of the JWSDL standard),
- Apache XMLBeans.

These two parsers provide alternative views of the WSDL contract. The JWSDL model is useful for parsing WSDL artifacts, such as port types, bindings, and services. The XMLBeans model, on the other hand, is an XML schema parser, which is more useful for parsing the XML schema types defined in the WSDL contract.

JWSDL

JWSDL is a Java API for parsing WSDL contracts. This API is being developed under the Java Community Process, JSR 110. A copy of the JWSDL specification and complete Javadoc for the JWSDL API can be downloaded from the following location:

http://jcp.org/en/jsr/detail?id=110

Apache XMLBeans

Apache XMLBeans is an open source API for parsing XML schemas. It is useful for parsing the contents of the `schema` elements in a WSDL contract. The home page for the XMLBeans project is:

http://xmlbeans.apache.org/

The complete Javadoc for XMLBeans v2.2.0 is available at the following location:

http://xmlbeans.apache.org/docs/2.2.0/reference/index.html

Rhino

Rhino is a Java implementation of JavaScript that includes the capability to map Java APIs into JavaScript (the *scripting Java* feature). In the context of WSDLGen, this capability of Rhino is exploited to make both the JWSDL API and the XMLBeans API available in JavaScript (these APIs are originally specified in Java only).

Due to the strong similarity between Java syntax and JavaScript syntax, the mapped APIs are remarkably intuitive to use from within JavaScript. For details about how this mapping works, see:

http://www.mozilla.org/rhino/ScriptingJava.html

Basic Parsing

Overview

This section discusses some basic topics in parsing WSDL contracts. In particular, you need to be aware of how the contract style (document/literal wrapped or RPC/literal) affects how you parse a WSDL port type.

In this section

This section contains the following subsections:

The WSDL and XML Schema Models

Overview

WSDLGen enables JavaScript programs to access the JWSDL API and the XMLBeans API from by defining the following JavaScript objects:

- `wsdlModel`—the root of the JWSDL parser model.
- `schemaModel`—the root of the XMLBeans parser model.

These two objects are pushed into JavaScript using the Rhino Java-to-JavaScript mapping feature.

wsdlModel instance

To access the JWSDL API from within JavaScript, use the `wsdlModel` object, which is an instance of the `javax.wsdl.Definition` class mapped to JavaScript.

The JWSDL `Definition` class represents the top level element of the WSDL contract (see "JWSDL Parser Classes" on page 65). For example, you can use the `wsdlModel` object to obtain a list of all the port types in the contract as follows:

```
// JavaScript

var portTypeMap = wsdlModel.getPortTypes()
var portTypeArr = portTypeMap.values().toArray()

// Iterate over the list of port types
for each (pt in portTypeArr) {
    ... // Do something with the port type, pt.
}
```

schemaModel instance

To access the XMLBeans API from within JavaScript, use the `schemaModel` object, which is an instance of the `org.apache.xmlbeans.SchemaTypeLoader` class mapped to JavaScript.

The XMLBeans `SchemaTypeLoader` class enables you to find the XML schema types and elements defined within the `wsdl:types` element in the WSDL contract (see "XMLBeans Parser Classes" on page 72). For example, you can use the `schemaModel` object to obtain an element named `{http://xml.iona.com/wsdlgen/demo}testParams`, as follows:

```
// JavaScript

var TARG_NAMESPACE = "http://xml.iona.com/wsdlgen/demo"
var elQName = new javax.xml.namespace.QName(TARG_NAMESPACE,
    "testParams")

var el = schemaModel.findElement(elQName)
```

Parsing Document/Literal Wrapped Style

Overview

This subsection describes how to parse a WSDL contract that is written in *document/literal wrapped* style. The document/literal wrapped style is distinguished by the fact that it uses single part messages. Each part is defined to be a sequence type, whose constitutent elements represent operation parameters.

Characteristics of the document/literal wrapped style

A given operation, `OperationName`, must be defined as follows, in order to conform to the document/literal wrapped style of interface:

- *Input message*—the `message` element that represents the operation's input message must obey the following conditions:
    - The message contains just *a single part*.
    - The part references an element (not a type) and the element must be named, `OperationName`.
- *Input element*—the `OperationName` element must be defined as a `sequence` complex type, where each element in the sequence represents a distinct input parameter.
- *Output message*—the `message` element that represents the operation's output message must obey the following conditions:
    - The message contains just *a single part*.
    - The part references an element (not a type) and the element must be named, `OperationNameResponse`.
- *Output element*—the `OperationNameResponse` element must be defined as a `sequence` complex type, where each element in the sequence represents a distinct output parameter.

Sample WSDL contract

Example 18 shows an example of a WSDL contract defining an operation, `testParams`, that conforms to document/literal wrapped style.

Example 18: *Operation Defined in Document/Literal Style*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <wsdl:types>
```

Example 18: *Operation Defined in Document/Literal Style*

```
        <schema targetNamespace="..."
                xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="testParams">
                <complexType>
                    <sequence>
                      <element name="inInt" type="xsd:int"/>
                     <element name="inoutInt" type="xsd:int"/>
                    </sequence>
                </complexType>
            </element>
            <element name="testParamsResponse">
                <complexType>
                    <sequence>
                       <element name="inoutInt" type="xsd:int"/>
                     <element name="outFloat" type="xsd:float"/>
                    </sequence>
                </complexType>
            </element>
        </schema>
    </wsdl:types>
    <message name="testParams">
        <part name="parameters" element="tns:testParams"/>
    </message>
    <message name="testParamsResponse">
        <part name="parameters"
             element="tns:testParamsResponse"/>
    </message>
    <wsdl:portType name="BasePortType">
        <wsdl:operation name="testParams">
            <wsdl:input  message="tns:testParams"
                         name="testParams"/>
            <wsdl:output message="tns:testParamsResponse"
                         name="testParamsResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    ...
</definitions>
```

Parsing RPC/Literal Style

Overview

This subsection describes how to parse a WSDL contract that is written in *RPC/literal* style. The RPC/literal style is distinguished by the fact that it uses multi-part messages (one part for each parameter).

Characteristics of the RPC/literal style

A given operation, `OperationName`, must be defined as follows, in order to conform to the RPC/literal style of interface:

- *Input message*—the `message` element that represents the operation's input message must obey the following conditions:
    - The message can contain multiple parts, where each part represents a distinct input parameter.
    - Each part references a type (not an element).
- *Output message*—the `message` element that represents the operation's output message must obey the following conditions:
    - The message can contain multiple parts, where each part represents a distinct output parameter.
    - Each part references a type (not an element).

Sample WSDL contract

Example 19 shows an example of a WSDL contract defining an operation, `testParams`, that conforms to RPC/literal style.

Example 19: *Operation Defined in RPC/Literal Style*

```
<definitions ...>
    ...
    <message name="testParams">
        <part name="inInt" type="xsd:int"/>
        <part name="inoutInt" type="xsd:int"/>
    </message>
    <message name="testParamsResponse">
        <part name="inoutInt" type="xsd:int"/>
        <part name="outFloat" type="xsd:float"/>
    </message>
    ...
    <portType name="BasePortType">
        <operation name="testParams">
```

Example 19: *Operation Defined in RPC/Literal Style*

```
            <input message="tns:testParams" name="testParams"/>
            <output message="tns:testParamsResponse"
                    name="testParamsResponse"/>
        </operation>
    ...
</definitions>
```

The JWSDL Parser

Overview

This section contains a partial summary of the JWSDL parser API. Only the parts of the API that you would need for generating application code are described here. For a complete description of the API, see JSR 110.

In addition to the JWSDL API, this section also includes a brief description of some Java utilities classes that are extensively used by JWSDL.

In this section

This section contains the following subsections:

Overview of the WSDL Model

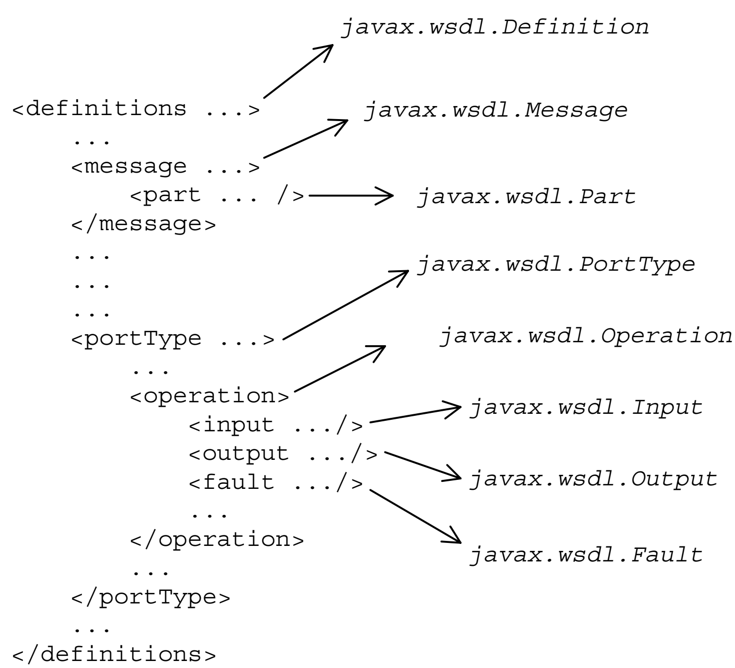
Overview

This section provides a partial overview of the WSDL model supported by the JWSDL parser. We focus here on the subset of the JWSDL API that is useful for generating application code from a WSDL contract. Hence, the discussion omits the API for parsing `wsdl:binding` and `wsdl:service` elements. The API for parsing the `wsdl:portType` element, which is essential for generating application code, is described here.

JWSDL classes required for parsing a port type

Figure 2 provides an overview of the JWSDL classes required for parsing a WSDL port type, showing how each JWSDL class corresponds to an element of the original WSDL contract.

Figure 2: *JWSDL Classes for Parsing a Port Type*

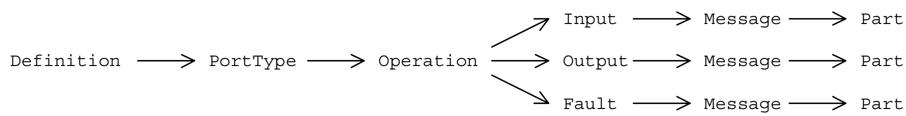


Generally, each JWSDL class is named after the element it represents. Note, however, that the class representing the `definitions` element is called `Definition`, *not* `Definitions`.

Node hierarchy

Each JWSDL class in the nodal hierarchy provides methods to access the WSDL elements it contains or, in some cases, references. Figure 3 shows the most convenient paths you can take to navigate down the node hierarchy when parsing a WSDL port type.

Figure 3: *Navigating the JWSDL Node Hierarchy*



Once you get down as far as a `javax.wsdl.Part` node, you can retrieve the QName of the element (or type) that represents a particular operation argument. To progress further with the parsing, you need to switch to the XMLBeans API, which enables you to parse the XML schema encoding the argument data (see “The XMLBeans Parser” on page 70).

Useful Java Utility Classes

Overview

There are a few Java utility classes that are extensively used by the JWSDL API, as follows:

- `javax.xml.namespace.QName`
- `java.util.Map`
- `java.util.Collection`
- `java.util.Iterator`
- `java.util.List`
- `java.util.ListIterator`

For your convenience, the API for these utility classes is summarized here. This summary does not include all of the methods in these classes, however. For the complete Java API, consult the Javadoc reference on Sun's Web site:

http://java.sun.com/j2se/1.5.0/docs/api/

javax.xml.namespace.QName

The `javax.xml.namespace.QName` class includes the methods shown in Table 11.

Table 11: *Some Methods and Constructors from QName*

Method/Constructor Signature	Description
`QName(String localPart)`	Construct a QName that has no namespace.
`QName(String namespaceURI, String localPart)`	Construct a QName consisting of a namespace URI and a local part.
`QName(String namespaceURI, String localPart, String Prefix)`	Constructor with namespace prefix (the prefix is not very important in the context of WSDL parsing).
`String getLocalPart()`	Get the local part of the QName.
`String getNamespaceURI`	Get the namespace URI of the QName.
`String getPrefix`	Get the prefix (rarely needed).
`String toString()`	Return `"{"+namespaceURI+"}"+localPart`.

java.util.Map

The `java.util.Map<K,V>` class includes the methods shown in Table 12.

Table 12: *Some Methods from java.util.Map*

Method Signature	Description
`put(K key, V value)`	Add a new entry to the map.
`V get(Object key)`	Use the key to look up a value in the map.
`java.util.Collection<V> values()`	If you want to iterate over all of the values in the map, it is necessary to convert it to a collection first.
`boolean isEmpty()`	True, if the map is empty.
`int size()`	Return the number of entries in the map.

java.util.Collection

The `java.util.Collection<E>` class includes the methods shown in Table 13.

Table 13: *Some Methods from java.util.Collection*

Method Signature	Description
`java.util.Iterator<E> iterator()`	Return an iterator, which can be used to iterate over all members of the collection.

java.util.Iterator

The `java.util.Iterator<E>` class includes the methods shown in Table 14.

Table 14: *Some Methods from java.util.Iterator*

Method Signature	Description
`boolean hasNext()`	True, if a call to `next()` would return another element in the collection.
`E next()`	Return the next element in the collection and increment the iterator index.

java.util.List

The `java.util.List<E>` class includes the methods shown in Table 15.

Table 15: *Some Methods from java.util.List*

Method Signature	Description
`Object[] toArray()`	Convert the list to an array.
`java.util.ListIterator listIterator()`	Return an iterator, which you can use to iterate over all of the list members.
`boolean isEmpty()`	True, if the list is empty.
`int size()`	Return the number of list members.

java.util.ListIterator

The `java.util.ListIterator<E>` class, which is a bidirectional iterator, includes the methods shown in Table 16.

Table 16: *Some Methods from java.util.ListIterator*

Method Signature	Description
`boolean hasNext()`	True, if a call to `next()` would return another list member.
`E next()`	Return the next member of the list and increment the iterator index.
`boolean hasPrevious()`	True, if a call to `previous()` would return another list member.
`E previous()`	Return the previous member of the list and decrement the iterator index.

JWSDL Parser Classes

Overview

This subsection summarizes the JWSDL parser classes that are likely to prove most useful when attempting to parse a port type in the context of generating code.

The following JWSDL classes are summarized here:

- javax.wsdl.Definition
- javax.wsdl.PortType
- javax.wsdl.Operation
- javax.wsdl.Input
- javax.wsdl.Output
- javax.wsdl.Fault
- javax.wsdl.Message
- javax.wsdl.Part

javax.wsdl.Definition

The `javax.wsdl.Definition` class represents a `wsdl:definition` element (top level of a WSDL contract). The most useful methods from the `javax.wsdl.Definition` class are shown in Table 17.

Table 17: *Methods from the javax.wsdl.Definition Class*

Method Signatures	Description
`java.util.Map getPortTypes()`	Get the `portType` elements defined in this `definition` element.

`javax.wsdl.PortType getPortType(`
`    javax.xml.namespace.QName name`
`)`
	Get the `portType` element with the specified name.

`java.util.Map getAllPortTypes()`	Get the `portType` elements defined in this `definition` element and those in any imported `definition` elements down the WSDL tree.
`java.util.Map getImports()`	Get a map of lists containing all the imports defined here.
`java.util.Map getImports(String namespaceURI)`	Get the list of imports for the specified `namespaceURI`.

Table 17: *Methods from the javax.wsdl.Definition Class*

Method Signatures	Description
`java.util.Map getNamespaces()`	Get all namespace associations in this definition.
`String getNamespace(String prefix)`	Get the namespace URI associated with this prefix.
`String getPrefix(String namespaceURI)`	Get a prefix associated with this namespace URI.
`String getTargetNamespace()`	Get the target namespace in which the WSDL elements are defined.

javax.wsdl.PortType

The `javax.wsdl.PortType` class represents a `wsdl:portType` element. The most useful methods from the `javax.wsdl.PortType` class are shown in Table 18.

Table 18: *Methods from the javax.wsdl.PortType Class*

Method Signatures	Description
`java.util.List getOperations()`	Get the operations defined in this port type.

`javax.wsdl.Operation getOperation(`
`String name,`
`String inputName,`
`String outputName`
`)`
	Get the operation with the specified name, `name`. If the operation name is overloaded, you can optionally use the `inputName` (the name of the operation's `input` element) and/or the `outputName` (the name of the operation's `output` element) to disambiguate. Otherwise, set `inputName` and `outputName` to `null`.

`javax.xml.namespace.QName getQName()`	Returns the name of the port type.
`boolean isUndefined()`	True if this port type is not defined.

javax.wsdl.Operation

The `javax.wsdl.Operation` class represents a `wsdl:operation` element. The most useful methods from the `javax.wsdl.Operation` class are shown in Table 19.

Table 19: *Methods from the javax.wsdl.Operation Class*

Method Signatures	Description
`javax.wsdl.Input getInput()`	Get this operation's `input` element.

Table 19: *Methods from the javax.wsdl.Operation Class*

Method Signatures	Description
`javax.wsdl.Output getOutput()`	Get this operation's `output` element.
`java.util.Map getFaults()`	Get this operation's `fault` elements.
`javax.wsdl.Fault getFault(String name)`	Get the `fault` with the specified name.
`String getName()`	Returns the name of the operation.
`boolean isUndefined()`	True if the operation is undefined.

javax.wsdl.Input

The `javax.wsdl.Input` class represents a `wsdl:input` element. The most useful methods from the `javax.wsdl.Input` class are shown in Table 20.

Table 20: *Methods from the javax.wsdl.Input Class*

Method Signatures	Description
`javax.wsdl.Message getMessage()`	Get the input message element.
`String getName()`	Return the name of the `input` element (if any).

javax.wsdl.Output

The `javax.wsdl.Output` class represents a `wsdl:output` element. The most useful methods from the `javax.wsdl.Output` class are shown in Table 21.

Table 21: *Methods from the javax.wsdl.Output Class*

Method Signatures	Description
`javax.wsdl.getMessage()`	Get the output message element.
`String getName()`	Return the name of the `output` element (if any).

javax.wsdl.Fault

The `javax.wsdl.Fault` class represents a `wsdl:fault` element. The most useful methods from the `javax.wsdl.Fault` class are shown in Table 22.

Table 22: *Methods from the javax.wsdl.Fault Class*

Method Signatures	Description
`javax.wsdl.Message getMessage()`	Get the fault message element.
`String getName()`	Return the name of the `fault` element (if any).

javax.wsdl.Message

The `javax.wsdl.Message` class represents a `wsdl:message` element. The most useful methods from the `javax.wsdl.Message` class are shown in Table 23.

Table 23: *Methods from the javax.wsdl.Message Class*

Method Signatures	Description
`java.util.Map getParts()`	Get a map of the message's parts, where the key is a part name and the value is a `javax.wsdl.Part` object.
`javax.wsdl.Part getPart(String name)`	Get the part specified by `name`.
`javax.xml.namespaceQName getQName()`	Get the qualified name of this `message` element.
`boolean isUndefined()`	True if this message element is undefined.

javax.wsdl.Part

The `javax.wsdl.Part` class represents a `wsdl:part` element. The most useful methods from the `javax.wsdl.Part` class are shown in Table 24.

Table 24: *Methods from the javax.wsdl.Part Class*

Method Signatures	Description
`javax.xml.namespace.QName getElementName()`	Get the element node referred to by the part's `element` attribute (if any).
`javax.xml.namespace.QName getTypeName()`	Get the type node referred to by the part's type attribute (if any).

Table 24: *Methods from the javax.wsdl.Part Class*

Method Signatures	Description
`String getName()`	Get the name of the part.

The XMLBeans Parser

Overview

This section contains a partial summary of the XMLBeans parser API, which can be used to parse the parameter data from WSDL operations at runtime. For a complete description of the API, see the XMLBeans 2.2.0 Javadoc.

In this section

This section contains the following subsections:

Overview of the XMLBeans Parser

Overview

This section provides a partial overview of the classes in the XMLBeans parser. The XMLBeans parser actually supports two different kinds of schema model: a static model and a dynamic (runtime) model. The static model is created by generating a set of Java classes that represent the elements of an XML schema. The dynamic model, on the other hand, does not require any Java classes to be generated and can parse any XML schema at runtime.

The section focusses on describing the dynamic (runtime) model.

XMLBeans classes needed to parse XML schema

The following XMLBeans classes are essential for the runtime parsing of XML data:

- `org.apache.xmlbeans.SchemaTypeLoader`—a class that enables you to look up schema types and schema global elements by name.
- `org.apache.xmlbeans.SchemaGlobalElement`—a class that represents elements defined *directly* inside the `xsd:schema` element (in contrast to elements defined at a nested level in the schema, which are known as *local elements*).

  > **Note:** The main difference between a global element and a local element is that a global element can be defined to be a member of a substitution group, whereas a local element cannot. In addition, the elements referenced within a `wsdl:part` element would normally be global elements.

- `org.apache.xmlbeans.SchemaType`—the class that represents a schema type.
- `org.apache.xmlbeans.SchemaProperty`—a class that represents a summary of the elements that *share the same name* within a complex type definition.

  > **Note:** XML schema allows you to define an element with the same name *more than once* inside a complex type declaration.

XMLBeans Parser Classes

Overview

This subsection summarizes the most important XMLBeans parser classes, which you are likely to use while parsing an XML schema type in WSDLGen.

The following XMLBeans classes are summarized here:

- `org.apache.xmlbeans.`SchemaTypeLoader
- `org.apache.xmlbeans.`SchemaGlobalElement
- `org.apache.xmlbeans.`SchemaType
- `org.apache.xmlbeans.`SchemaProperties

SchemaTypeLoader

The `org.apache.xmlbeans.SchemaTypeLoader` class is used to find specific nodes in the XMLBeans parse tree. In particular, you can use it to find element nodes and type nodes. The most useful methods from the `SchemaTypeLoader` class are shown in Table 25.

Table 25: *Methods from the SchemaTypeLoader Class*

Method Signature	Description

`SchemaGlobalElement findElement(`
`    javax.xml.namespace.QName name`
`)`
	Returns the global element definition with the given name, or null if none.

`SchemaType findType(`
`    javax.xml.namespace.QName name`
`)`
	Returns the type with the given name, or null if none.

SchemaGlobalElement

The `org.apache.xmlbeans.SchemaGlobalElement` class represents an element node in the XMLBeans parse tree. The most useful methods from the `SchemaGlobalElement` class are shown in Table 26.

Table 26: *Methods from the SchemaGlobalElement Class*

Method Signature	Description
`javax.xml.namespace.QName getName()`	Returns the form-unqualified-or-qualified name.
`SchemaType getType()`	Returns the type.

Table 26: *Methods from the SchemaGlobalElement Class*

Method Signature	Description
`java.math.BigInteger getMinOccurs()`	Returns the `minOccurs` value for this particle.
`java.math.BigInteger getMaxOccurs()`	Returns the `maxOccurs` value for this particle, or null if it is unbounded.
`boolean isNillable()`	True if nillable; always false for attributes.
`String getSourceName()`	The name of the source file in which this component was defined (if known).

SchemaType

The `org.apache.xmlbeans.SchemaType` class represents a type node in the XMLBeans parse tree. The most useful methods from the `SchemaType` class are shown in Table 27.

Table 27: *Methods from the SchemaType Class*

Method Signature	Description
`SchemaStringEnumEntry enumEntryForString( String s )`	Returns the string enum entry corresponding to the given enumerated string, or null if there is no match or this type is not a string enumeration.
`StringEnumAbstractBase enumForInt(int i)`	Returns the string enum value corresponding to the given enumerated string, or null if there is no match or this type is not a string enumeration.
`StringEnumAbstractBase enumForString(String s)`	Returns the string enum value corresponding to the given enumerated string, or null if there is no match or this type is not a string enumeration.
`SchemaType[] getAnonymousTypes()`	The array of inner (anonymous) types defined within this type.
`int getAnonymousUnionMemberOrdinal()`	For anonymous types defined inside a union only: gets the integer indicating the declaration order of this type within the outer union type, or zero if this is not applicable.
`SchemaAttributeModel getAttributeModel()`	Returns the attribute model for this complex type (with simple or complex content).

Table 27: *Methods from the SchemaType Class*

Method Signature	Description
`SchemaProperty[] getAttributeProperties()`	Returns all the SchemaProperties corresponding to attributes.
`SchemaProperty getAttributeProperty( QName attrName )`	Returns a SchemaProperty corresponding to an attribute within this complex type by looking up the attribute name.
`SchemaType getAttributeType( QName eltName, SchemaTypeLoader wildcardTypeLoader )`	Returns the type of an attribute based on the attribute name and the type system within which (wildcard) names are resolved.
`QName getAttributeTypeAttributeName()`	Returns the attribute qname if this is a attribute type, or null otherwise.
`SchemaType getBaseEnumType()`	If this is a string enumeration, returns the most basic base schema type that this enuemration is based on.
`SchemaType getBaseType()`	Returns base restriction or extension type.
`SchemaType getContentBasedOnType()`	For complex types with simple content returns the base type for this type's content.
`SchemaParticle getContentModel()`	Returns the complex content model for this complex type (with complex content).
`int getContentType()`	Returns `EMPTY_CONTENT`, `SIMPLE_CONTENT`, `ELEMENT_CONTENT`, or `MIXED_CONTENT` for complex types.
`int getDecimalSize()`	For atomic numeric restrictions of decimal only: the numeric size category.
`int getDerivationType()`	Returns an integer for the derivation type, either `DT_EXTENSION`, `DT_RESTRICTION`, `DT_NOT_DERIVED`.
`SchemaProperty[] getDerivedProperties()`	Returns the SchemaProperties defined by this complex type, exclusive of the base type (if any).
`SchemaProperty[] getElementProperties()`	Returns all the SchemaProperties corresponding to elements.

Table 27: *Methods from the SchemaType Class*

Method Signature	Description
`SchemaProperty getElementProperty( QName eltName )`	Returns a SchemaProperty corresponding to an element within this complex type by looking up the element name.
`SchemaType getElementType( QName eltName, QName xsiType, SchemaTypeLoader wildcardTypeLoader )`	Returns the type of a child element based on the element name and an xsi:type attribute (and the type system within which names are resolved).
`XmlAnySimpleType[] getEnumerationValues()`	Returns the array of valid objects from the enumeration facet, null if no enumeration defined.
`SchemaType getListItemType()`	For list types only: get the item type.
`QName getName()`	The name used to describe the type in the schema.
`SchemaType getPrimitiveType()`	For atomic types only: get the primitive type underlying this one.
`SchemaProperty[] getProperties()`	For atomic types only: get the primitive type underlying this one.
`int getSimpleVariety()`	Returns whether the simple type is `ATOMIC`, `UNION`, or `LIST`.
`SchemaStringEnumEntry[] getStringEnumEntries()`	Returns the array of `SchemaStringEnumEntries` for this type: this array includes information about the java constant names used for each string enum entry.
`SchemaTypeSystem getTypeSystem()`	Returns the `SchemaTypeLoader` in which this type was defined.
`SchemaType getUnionCommonBaseType()`	For union types only: get the most specific common base type of the constituent member types.
`SchemaType[] getUnionConstituentTypes()`	For union types only: get the constituent member types.
`SchemaType[] getUnionMemberTypes()`	For union types only: get the shallow member types.
`SchemaType[] getUnionSubTypes()`	For union types only: gets the full tree of member types.

Table 27: *Methods from the SchemaType Class*

Method Signature	Description
`boolean hasAllContent()`	True if the complex content model for this complex type is an `all` group.
`boolean hasAttributeWildcards()`	True if this type permits wildcard attributes.
`boolean hasElementWildcards()`	True if this type permits element wildcards.
`boolean hasPatternFacet()`	True if there are regular expression pattern facets.
`boolean hasStringEnumValues()`	True if this is a string enum where an integer is assigned to each enumerated value.
`boolean isAnonymousType()`	True if the Xsd type is anonymous (i.e., not top-level).
`boolean isAttributeType()`	True if this is a attribute type.
`boolean isBounded()`	True if bounded.
`boolean isBuiltinType()`	True for any of the 40+ built-in types.
`boolean isNoType()`	True for the type object that represents a the absence of a determined type.
`boolean isNumeric()`	True if numeric.
`boolean isPrimitiveType()`	True for any of the 20 primitive types (plus `anySimpleType`).
`boolean isSimpleType()`	True for the anySimpleType and any restrictions/unions/lists.
`boolean isURType()`	True for `anyType` and `anySimpleType`.
`boolean matchPatternFacet(String s)`	True if the given string matches the pattern facets.
`int ordered()`	True if ordered.
`QNameSet qnameSetForWildcardAttributes()`	Returns a `QNameSet` of attributes that may exist in wildcard buchets and are not explicitly defined in this schema type.
`QNameSet qnameSetForWildcardElements()`	Returns a `QNameSet` of elements that may exist in wildcard buchets and are not explicitly defined in this schema type.

SchemaProperties

The `org.apache.xmlbeans.SchemaProperties` class represents a summary of the element definitions that *share the same name* within a complex type definition. Rather than having to look up the properties for all of the different element fields that have the same name, it is usually simpler to obtain the relevant `SchemaProperties` object. The `SchemaProperties` object attempts to unify the properties of the same-name elements in a consistent manner.

The most useful methods from the `SchemaProperties` class are shown in Table 27.

Table 28: *Methods from the SchemaProperties Class*

Method Signature	Description
`SchemaType getContainerType()`	The type within which this property appears.
`String getDefaultText()`	Returns the default or fixed value, if it is consistent.
`XmlAnySimpleType getDefaultValue()`	Returns the default or fixed value as a strongly-typed value, if it is consistent.
`BigInteger getMaxOccurs()`	Returns a summarized maximum occurrence number.
`BigInteger getMinOccurs()`	Returns a summarized minimum occurrence number.
`QName getName()`	The name of this element or attribute.
`SchemaType getType()`	The schema type for the property.
`int hasDefault()`	Returns `NEVER`, `VARIABLE`, or `CONSISTENTLY` defaulted, depending on the defaults present in the elements in this property.
`int hasFixed()`	Returns `NEVER`, `VARIABLE`, or `CONSISTENTLY` fixed, depending on the fixed constraints present in the elements in this property.
`int hasNillable()`	Returns `NEVER`, `VARIABLE`, or `CONSISTENTLY` nillable, depending on the nillability of the elements in this property.
`boolean isAttribute()`	True for attributes.
`boolean isReadOnly()`	True for read-only properties.

Index

Q

R

S

T

V

W

X