IONA

Artix™

Developing Artix Applications in Java

Version 2.0, March 2004

Making Software Work Together™

Updated: 12-Apr-2004

M 3 1 9 2

Contents

List of Tables

Preface

What is Covered in this Book

Developing Artix Applications in Java discusses the main aspects of developing transport-independent services and service consumers using Java stub and Java skeleton code generated by Artix. This book covers:

- how to access the Artix bus
- how to use generated data types
- how to create user defined exceptions
- how to access the header information for the transports supported by Artix.

Who Should Read this Book

Developing Artix Applications in Java is intended for Artix Java programmers. In addition to a knowledge of Java, this guide assumes that the reader is familiar with the basics of WSDL and XML schemas. Some knowledge of Artix concepts would be helpful, but is not required.

How to Use this Book

If you are new to using Artix to develop Java applications, Chapter 1 provides an overview of the benefits of using Artix and how Artix generates Java code from an Artix contract.

If you are interested in the basics of writing an Artix-enabled service or service consumer, Chapter 2 describes the basic steps to implement a service, connect to the Artix bus, and create JAX-RPC compliant proxies using Artix-generated code.

If you need help understanding how to work with the classes generated to represent complex data types, Chapter 3 gives detailed description of how all of the XMLSchema data types in an Artix contract are mapped into Java code. It also contains details and examples on using the generated Java code.

If you want to create user-defined exceptions, Chapter 4 explains how to describe a user-defined exception in an Artix contract and how exceptions are mapped into Java code by Artix.

If you want to learn how to develop Java code to use XMLSchema `anyType` elements, Chapter 5 describes how they are mapped into Java and describes the Artix classes that allow you to work with them.

Online Help

While using the Artix Designer you can access contextual online help, providing:

- A description of your current Artix Designer screen
- Detailed step-by-step instructions on how to perform tasks from this screen
- A comprehensive index and glossary
- A full search feature

There are two ways that you can access the Online Help:

- Click the **Help** button on the Artix Designer panel, or
- Select **Contents** from the Help menu

Finding Your Way Around the Artix Library

The Artix library contains several books that provide assistance for any of the tasks you are trying to perform. The remainder of the Artix library is listed here, with a short description of each book.

If you are new to Artix

You may be interested in reading:

- *Getting Started with Artix* - the getting started book describe basic Artix concepts.
- *Artix Tutorial* - this book guides you through programming Artix applications.

To design Artix solutions

You should read one or more of the following:

- *Designing Artix Solutions* - this book provides detailed information about using the Artix Designer to create WSDL-based Artix contracts, Artix stub and skeleton code, and Artix deployment bundles.
- *Designing Artix Solutions from the Command Line* - this book provides detailed information about the WSDL extensions used in Artix contracts, and explains the mappings between data types and Artix bindings.

To develop applications using Artix stub and skeleton code

Depending on your development environment you should read one or more of the following:

- *Developing Artix Applications in C++* - this book discusses the technical aspects of programming applications using the Artix C++ API
- *Developing Artix Applications in Java* - this book discusses the technical aspects of programming applications using the Artix Java API

To manage and configure your Artix solution

You should read *Deploying and Managing Artix Solutions.* It describes how to configure and deploy Artix-enabled systems. It also discusses how to manage them once they are deployed.

If you want to know more about Artix security

You should read the *Artix Security Guide*. It outlines how to enable and configure Artix's security features. It also discusses how to integrate Artix solutions into a secure environment.

Have you got the latest version?

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs. Compare the version details provided there with the last updated date printed on the inside cover of the book you are using (at the bottom of the copyright notice).

Additional Resources for Information

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 `doc-feedback@iona.com`.

The IONA knowledge base contains helpful articles, written by IONA experts, about the Orbix and other products. 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/

Typographical Conventions

This book 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 book uses 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, Windows NT, Windows 95, or Windows 98 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.

Understanding the Artix Java Development Model

The Artix Java development tools generate JAX-RPC compliant Java code from WSDL-based Artix contracts. Using the generated code, you can develop transport-independent applications that take advantage of the Artix bus.

In this chapter

This chapter discusses the following topics:

Separating Transport Details from Application Logic

Overview

One of the main benefits of using Artix to develop applications is that it removes the network protocol details, message transport details, and payload format details from the business of developing application logic. Artix enables developers to write robust applications using standard Java APIs and leaves the nitty-gritty of the messaging mechanics up to the system administrators or system architects.

Unlike CORBA or J2EE, however, Artix does not provide this abstraction from the transport details by limiting the types of messaging system the application can work on. It makes the application capable of using any number of transports and payload formats. In addition, Artix allows applications in the same system to interoperate across multiple messaging protocols.

Dividing the logical and physical

Artix achieves this separation of the logical part of an application from the physical details of how data is passed by describing applications using Web Services Description Language (WSDL) as the basis for Artix contracts. Artix contracts are XML documents that describe applications in two sections:

Logical:

The logical section of an Artix contract defines the abstract data types used by the application, the logical operations exposed by the application, and the messages passed by those operations.

Physical:

The physical section of an Artix contract defines how the messages used by the application are mapped for transport across the network and how the application's port is configured. For example, the physical section of the contract would be where it is made explicit that an application will use SOAP over HTTP to expose its operations.

The Artix bus

The Artix bus is a library that provides the layer of abstraction to liberate the application logic from the transport once the code is generated. The bus reads the transport details from the physical section of the Artix contract, loads the appropriate payload and transport plug-ins, and handles the mapping of the data onto and off the wire.

The bus also provides access to the message headers so you can add payload-specific information to the data if you wish. In addition, it provides access to the transport details to allow dynamic configuration of transports.

Representing Services in Artix Contracts

Overview

Services, which are the operations exposed by an application, are described in the logical section of an Artix contract. When defining a service in an Artix contract, you break it down into three parts: the complex data types used in the messages, the messages used by the operations, and the collection of operations that make up the service.

Data types

Complex data types, such as arrays, structures, and enumerations, are described in an Artix contract using XMLSchema. The descriptions are contained within the WSDL `<types>` element. The data type descriptions represent the logical structure of the data. For example, an array of integers could be described as shown in Example 1.

Example 1: *Array Description*

```
<complexType name="ArrayOfInt">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="item"
             type="xsd:int"/>
  </sequence>
</complexType>
```

The described types are used to define the message parts used by the service.

Messages

In an Artix contract messages represent the data passed to and received from a remote system in the execution of an operation. Messages are described using the `<message>` element and consist of one or more `<part>` elements. Each message part represents an argument in an operation's parameter list or a piece of data returned as part of an exception.

Service

In an Artix contract logical services are described using the `<portType>` element and consist of one or more `<operation>` elements. Each `<operation>` element describes an operation that is to be exposed over the network.

Operations are defined by the messages which are passed to and from the remote system when the operation is invoked. In an Artix contract, each operation is allowed to have one input message, one output message, and any number of fault messages. It does not need to have any of these elements. An input message describes the parameter list passed into the operation. An output message describes the return value, and the output parameters of the operation. A fault message describes an exception that the operation can throw. For example, a Java method with the signature `long myOp(char c1, char c2)`, would be described as shown in Example 2.

Example 2: *Operation Description*

```
<message name="inMessage">
  <part name="c1" type="xsd:char" />
  <part name="c2" type="xsd:char" />
</message>
<message name="outMessage">
  <part name="returnVal" type="xsd:int" />
</message>
<portType name="myService">
  <operation name="myOp">
    <input message="inMessage" name="in" />
    <output message="outMessage" name="out" />
  </operation>
</portType>
```

Mapping from an Artix Contract to Java

Overview

Artix maps the WSDL-based Artix contract description of a service into Java server skeletons and client stubs following the JAX-RPC specification. This allows application developers to implement the service's logic using standard Java and be assured that the service will be interoperable with a wide range of other services.

Ports

For each `<port>` element in an Artix contract, a Java interface that extends `java.rmi.Remote` is generated. The name of the generated interface is taken from the `name` attribute of the `<port>` element. The interface's name will be identical to the `<port>`'s name unless the `<port>`'s name ends in `Port`. In this case, the `Port` will be stripped off the interface's name.

The generated interface will contain each of the operations of the `<portType>` to which the `<port>` element is bound. For example, the contract shown in Example 3 will generate an interface, `sportsCenter`, containing one operation, `update`.

Example 3: *SportsCenter Port*

```
<message name="scoreRequest">
  <part name="teamName" type="xsd:string" />
</message>
<message name="scoreReply">
  <part name="score" type="xsd:int" />
</message>
<portType name="sportsCenterPortType">
  <operation name="update">
    <input message="scoreRequest" name="request" />
    <ouput message="scoreReply" name="reply" />
  </operation>
</portType>
<binding name="scoreBinding" type="tns:sportsCenterPortType">
...
<service name="sportsService">
  <port name="sportsCenterPort" binding="tns:scoreBinding">
...
```

The generated Java interface is shown in Example 4.

Example 4: *SportsCenter Interface*

```
//Java
public interface sportsCenter extends java.rmi.Remote
{
  int update(String teamName)
    throws java.rmi.RemoteException;
}
```

Operations

Every <operation> element in a contract generates a Java method within the interface defined for the <operation> element's <portType>. The generated method's name is taken from the <operation> element's name attribute. <operation> elements with the same name attribute will generate overloaded Java methods in the interface.

All generated Java methods throw a `java.rmi.RemoteException` exception. In addition, all <fault> elements listed as part of the operation create an exception to the generated Java method.

Message parts

The message parts of the operation's <input> and <output> elements are mapped as parameters in the generated method's signature. The order of the mapped parameters can be specified using the <operation> element's `parameterOrder` attribute. If this attribute is used, it must list all of the parts of the input message. The message parts listed in the `parameterOrder` attribute will be placed in the generated method's signature in the order specified. Unlisted message parts will be placed in the method signature according to the order the parts are specified in the <message> elements of the contract. The first unlisted output message part is mapped to the generated method's return type. The parameter names are taken from the <part> element's name attribute. If the `parameterOrder` attribute is not specified, input message parts are listed before output message parts. Message parts that are listed in both the input and output messages are considered `inout` parameters and are listed only according to their position in the input message.

All inout and output message parts, except the part mapped to the return value of the generated method, are passed using Java `Holder` classes. For the XML primitive types, the Java Holder class used is the standard Java `Holder` class, defined in `javax.xml.rpc.holders` package, for the appropriate Java type. For complex types defined in the contract, the code generator will generate the appropriate `Holder` classes. For more information on data type mapping, see "Working with Artix Data Types" on page 23.

For example, the contract fragment shown in Example 5 would result in an operation, `final`, with a return type of `String` and a parameter list that contains two input parameters and three output parameters.

Example 5: *SportsFinal Port*

```
<message name="scoreRequest">
  <part name="team1" type="xsd:string" />
  <part name="team2" type="xsd:string" />
</message>
<message name="scoreReply">
  <part name="winTeam" type="xsd:string" />
  <part name="team1score" type="xsd:int" />
  <part name="team2score" type="xsd:int" />
</message>
<portType name="sportsFinalPortType">
  <operation name="final">
    <input message="scoreRequest" name="request" />
    <ouput message="scoreReply" name="reply" />
  </operation>
</portType>
<binding name="scoreBinding" type="tns:sportsFinalPortType">
...
<service name="sportsService">
  <port name="sportsFinalPort" binding="tns:scoreBinding">
...
```

The generated Java interface is shown in Example 6.

Example 6: *SportsFinal Interface*

```
//Java
public interface sportsFinal extends java.rmi.Remote
{
  String final(String team1, String team2,
               IntHolder team1score, IntHolder team2score)
    throws java.rmi.RemoteException;
}
```

Developing Artix Enabled Clients and Servers

Artix generates stub and skeleton code that provides a developer with a simple model to develop transport-independent applications.

In this chapter

This chapter discusses the following topics:

Generating Stub and Skeleton Code

Overview

The Artix development tools include a utility to generate server skeleton and client stub code from an Artix contract. The generated code is similar to code generated by a CORBA IDL compiler. There are two major differences between CORBA-generated code and Artix-generated code:

- Artix-generated code is not restricted to using IIOP and therefore contains generic code that is compatible with a multitude of transports.
- Artix maps WSDL types to Java using the mapping described in the JAX-RPC specification. The resulting types are very different from those generated by an IDL-to-Java compiler.

Generated files

The Artix code generator produces a number of files from the Artix contract. They are named according to the port name specified when the code was generated. The files include:

*portTypeName***.java** defines the Java interface that both the client and server implement.

*portTypeName***Impl.java** defines the class used to implement the server.

*portTypeName***Server.java** is a simple main class for the server.

In addition to these files, the code generator also creates a class for each named schema type defined in the Artix contract. These files are named according to the type name they are given in the contract and contain the helper functions needed to use the data types. The naming convention for the helper type functions conforms to the JAX-RPC specification. For more information on using these generated data types see “Working with Artix Data Types” on page 23.

Generating code from the command line

You generate code at the command line using the command:

```
wsdltojava [-e service][-t port][-b binding][-i portType]
           [-d output_dir][-p package][-impl][-server][-client]
           [-types][-interface][-sample][-all] artix-contract
```

You must specify the location of a valid Artix contract for the code generator to work. The default behavior of `wsdltojava` is to generate all of the java code needed to develop a client and server. You can also supply the following optional parameters to control the portions of the code generated:

`-e` *`service`*	Specifies the name of the service for which the tool will generate code. The default is to use the first service listed in the contract.
`-t` *`port`*	Specifies the name of the port for which code is generated. The default is to use the first port listed in the service.
`-b` *`binding`*	Specifies the name of the binding to use when generating code. The default is to use the first binding listed in the contract.
`-i` *`portType`*	Specifies the name of the portType for which code will be generated. The default is to use the first portType in the contract.
`-d` *`output_dir`*	Specifies the directory to which the generated code is written. The default is the current working directory.
`-p` *`package`*	Specifies the name of the Java package to use for the generated code.
`-impl`	Generates the skeleton class for implementing the server defined by the contract.
`-server`	Generates a simple main class for the server.
`-client`	Generates only the Java interface and code needed to implement the complex types defined by the contract. This flag is equivalent to specifying `-interface -types`.
`-types`	Generates the code to implement the complex types defined by the contract.
`-interface`	Generates the Java interface for the service.
`-sample`	Generates a sample client that can be used to test your Java server.
`-all`	Generates code for all portTypes in the contract.

Java Package Names

Artix packages

The Artix bus object which provides the transport and payload format independence in Artix is defined in the `com.iona.jbus` package. You will need to import this package and all of its subpackages into all Artix Java applications.

Generated type packages

The generated types are generated into a single package which must be imported for any methods using them. By default, the package name will be mapped from the target namespace of the schema describing the types. The default package name is created following the algorithm specified in the JAXB specification. The mapping algorithm follows four basic steps:

1. The leading `http://` or `urn://` are stripped off the namespace.
2. If the first string in the namespace is a valid internet domain, for example it ends in `.com` or `.gov`, the leading `www.` is stripped off the string, and the two remaining components are flipped.
3. If the final string in the namespace ends with a file extension of the pattern `.xxx` or `.xx`, the extension is stripped.
4. The remaining strings in the namespace are appended to the resulting string and separated by dots.
5. All letters are made lowercase.

For example, the XML namespace `http://www.widgetVendor.com/types/widgetTypes.xsd` would be mapped to the Java package name `com.widgetvendor.types.widgettypes`.

Java packages

Artix applications require a number of standard Java packages. These include:

javax.xml.namespace.QName provides the functionality to work with the XML QNames used to specify services.

javax.xml.rpc.* provides the APIs used to implement Artix Java clients. This package is not needed by server code.

java.io.* provides system input and output through data streams, serialization and the file system.

java.net.* provides the classes need to for communicating over a network. These classes are key to Artix applications that act as Web services.

Developing a Server

Overview

The Artix code generator generates server skeleton code and the implementation shell that serves as the starting point for developing an Artix-enabled server. The skeleton code hides the transport details, allowing you to focus on business logic.

Generating the server implementation class

The Artix code generatition utility, `wsdltojava`, will generate an implementation class for your server when passed the `-impl` command flag.

> **Note:** If your contract specifies any derived types or complex types you will also need to generate the code for supporting those types by specifying the `-types` flag.

Generated code

The implementation class code consists of two files:

PortName.**java** contains the interface the server implements.

*PortName***Impl.java** contains the class definition for the server's implementation class. It also contains empty shells for the methods that implement the operations defined in the contract.

Completing the server implementation

You must provide the logic for the operations specified in the contract that defines the server. To do this you edit the empty methods provided in *`PortName`*`Impl.java`. A generated implementation class for a contract defining a service with two operations, `sayHi` and `greetMe`, would resemble Example 7. Only the code portions highlighted in **`bold`** (in the bodies of the `greetMe()` and `sayHi()` methods) must be inserted by the programmer.

Example 7: *Implementation of the HelloWorld PortType in the Server*

```
// Java
import java.net.*;
import java.rmi.*;
```

Example 7: *Implementation of the HelloWorld PortType in the Server*

```
public class HelloWorldImpl {

    /**
     * greetMe
     *
     * @param: stringParam0 (String)
     * @return: String
     */
   public String greetMe(String stringParam0) {
        System.out.println("HelloWorld.greetMe() called with
   message: "+stringParam0);
        return "Hello Artix User: "+stringParam0;
    }

    /**
     * sayHi
     *
     * @return: String
     */
    public String sayHi() {
        System.out.println("HelloWorld.sayHi() called");
        return "Greetings from the Artix HelloWorld Server";
    }
```

Writing the server main()

The server `main()` of an Artix Java server must do three things before it can service requests:

1. Initialize the Artix bus.
2. Register the server implementation with the Artix bus.
3. Start the Artix bus.

You can use `wsdltojava` to generate a server `main()` with the code to perform these steps by using the `-server` flag. The `main()` shown in Example 10 on page 17 was generated using `wsdltojava`.

Initializing the bus

The Artix bus is initialized using `com.iona.jbus.Bus.init()`. The method has the following signature:

```
static Bus init(String args[]);
```

`init()` takes the `args` parameter passed into the main as a required parameter. Optionally, you can also pass in a second string that specifies the name of the configuration scope from which the bus instance will read its runtime configuration.

This will create a bus instance to host your services, load the Artix configuration information for your application, and load the required plug-ins.

Registering a servant for the server implementation

Before the bus can begin processing requests made on your server, you must register the servant object that implements your server's business logic with the bus. Registering the implementation object's servant with the bus allows the bus to create instances of the implementation object to service requests.

To register your implementation object's servant you create a `com.iona.jbus.Servant` using the path of the WSDL file describing the service interface, an instance of your implementation object, and an instance of an initialized Artix bus. Example 8 shows the code to create a servant for the `HelloWorld` service.

Example 8: *Creating a ServerFactoryBase*

```
//Java
Servant servant =
  new SingleInstanceServant("./HelloWorld.wsdl",
                            new HelloWorldImpl(), bus);
```

After creating the servant, you register it with the bus using the bus' `registerServant()` method. The signature for `registerServant()` is shown in Example 9.

Example 9: *registerServerFactory()*

```
void registerServerFactory(Servant servant
                           QName serviceName,
                           String portName)
throws BusException
```

In addition to the servant, `registerServant()` takes the service's QName as specified in the contract defining the service and the name of the WSDL port the service is instantiating.

Starting the bus

After the bus is initialized and the server implementation is registered with it, the bus is ready to listen for requests and pass them to the server for processing. To start the bus, you use the bus' `run()` method. Once the bus is started, it retains control of the process until it is shut down. The server's `main()` will be blocked until `run()` returns.

Completed server main()

Example 10 shows how the `main()` for a Java Artix server might look.

Example 10: *Server main()*

```
// Java
import com.iona.jbus.*;
import javax.xml.namespace.QName;

public class Server
{
  public static void main(String args[])
  throws Exception
  {
    // Initialize the Artix bus
    Bus bus = Bus.init(args);

    // Register the implementation object factory
    QName name = new QName("http://xmlbus.com/HelloWorld",
                          "HelloWorldService");
    Servant servant =
                  new SingleInstanceServant("./HelloWorld.wsdl",
                                        new HelloWorldImpl());
    bus.registerServant(servant, name, "HelloWorldPort");

    // Start the Bus
    bus.run();
  }
}
```

Developing a Client

Overview

Artix Java clients are implemented using dynamic proxies as described in the JAX-RPC 1.1 specification. The interface used to create the proxy class is defined in the generated file *PortName***.java**. The only Artix-specific code needed by an Artix Java client initializes and shuts down the Artix bus.

Initializing the bus

Client applications initialize the bus in the same manner as server applications, by calling the bus' `init()` method. Client applications, however, do not need to make a call to the bus' `run()` method.

Instantiating a client proxy

Artix Java clients use dynamic proxies, as described in the JAX-RPC specification, to make requests on servers. Dynamic proxies are created using the interface generated from your contract and the `javax.xml.rpc.Service` interface. You need the `QName` of the service for which you are creating the proxy, the `QName` of the endpoint with which the proxy will contact the service, and the URL of the contract defining the service. Once you have these three pieces of information, creating a dynamic proxy requires three steps:

1. Obtain an instance of `javax.xml.rpc.ServiceFactory` to create the service.
2. Use the `ServiceFactory` to create a `Service` instance for the service to which the proxy will connect.
3. Use the `Service` to instantiate the dynamic proxy.

Obtaining a ServiceFactory instance

To obtain an instance of the `ServiceFactory` you call `ServiceFactory.newInstance()`. This returns the `ServiceFactory`. Only one is created per application and the same `ServiceFactory` is returned for each successive call.

Creating a Service instance

A `Service` instance is created from the `ServiceFactory` using `createService()`. `createService()` takes two arguments:

- the URL of the contract defining the service.

- the service's `QName`.

Creating the dynamic proxy

The dynamic proxy is created from the `Service` using `getPort()`. `getPort()` takes two arguments:

- the `QName` of the endpoint with which the proxy contacts the service.
- the name of the generated Java interface in *`PortName`*`.java` with `.class` appended. For example, if the generated interface's name is `HelloWorld`, this argument would be `HelloWorld.class`.

`getPort()` returns an instance of `java.rmi.Remote` that must be cast to the generated interface.

Shutting the bus down

Unlike a server that must shut down the bus from a separate thread, clients do not typically make a call to the bus' `run()` method and can simply call `shutdown()` on the bus before the main thread exits. It is advisable to pass `true` to `shutdown()` to ensure that the bus is fully shutdown before exiting.

Full client code

An Artix Java client developed to access `HelloWorldService` will look similar to Example 11.

Example 11: *Client Code*

```
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;

import javax.xml.namespace.QName;
import javax.xml.rpc.*;

import com.iona.jbus.Bus;

public class HelloWorldClient
{

  public static void main (String args[]) throws Exception
  {
1    Bus bus = Bus.init(args);

2    QName name = new QName("http://iona.com/HelloWorld",
                           "HelloWorldService");
```

Example 11: *Client Code*

```
3       QName portName = new QName("","HelloWorldPort");

4       String wsdlPath = "file:/./HelloWorld.wsdl";
        URL wsdlLocation = new File(wsdlPath).toURL();

5       ServiceFactory factory = ServiceFactory.newInstance();

6       Service service = factory.createService(wsdlLocation, name);

7       HelloWorld impl = (HelloWorld)service.getPort(portName,
                                                  HelloWorld.class);

8       String string_out;

        string_out = impl.sayHi();
        System.out.println(string_out);

9       bus.shutdown(true);

        }
    }
```

The code does the following:

1. The `com.iona.jbus.Bus.init()` function initializes the bus.
2. Creates the service's `QName`.
3. Creates the `QName` of the endpoint with which the proxy will contact the service.
4. Creates the URL of the contract defining the service.
5. The `newInstance()` function returns the `ServiceFactory`.
6. The `createService()` function instantiates the `Service` from which the dynamic proxy is created.
7. The `getPort()` function returns a dynamic proxy to the `HelloWorld` service. `getPort()` returns an instance of `java.rmi.Remote` that must be cast to the interface defining the service.
8. Makes a call on the proxy to request service.
9. Shuts down the bus.

Building an Artix Application

Required jar files

Artix Java applications require that the following Artix jar files are in your class path:

- `it_bus.jar`
- `it_wsdl.jar`
- `it_ws_reflect.jar`
- `ifc.jar`

You also need to ensure that the Artix version of `jaxrpc-api.jar` is used to build your Artix application. The simplest way to make sure the correct version is used is to prepend *`artix_install_dir`*`\artix\2.0\lib` to your class path.

Working with Artix Data Types

Artix maps XMLSchema data types in an Artix contract into Java data types. For primitive types the mapping is a one-to-one mapping to Java primitive types. For complex types, Artix follows the JAX-RPC specification for mapping complex types into Java objects.

In this chapter

This chapter discusses the following topics:

Primitive Types

Overview

Artix follows the JAX-RPC specification for mapping primitive XMLSchema types into Java. In most cases, the mapping from a primitive XMLSchema type is to a primitive Java type. However, some instances require a more complex mapping.

In this section

This section contains the following subsections:

Simple Primitive Type Mapping

Overview

When a message part is described as being of one of the primitive XMLSchema types, the generated parameter's type will be of a corresponding primitive Java type. For example, the message description shown in Example 12 will cause a parameter, `score`, of type `int` to be generated.

Example 12: *Message Description Using a Primitive Type*

```
<message name="scoreResponse">
  <part name="score" type="xsd:int" />
</message>
```

Types derived by restriction

XMLSchema supports the definition of simple types by restricting a primitive type using one of twelve facets. The primitive type from which the new type is defined is called its *base type*. Types defined using restriction of a base type are treated as if the new type were simply of the base type. So a type derived by restricting `xsd:float` would be mapped to a `float` in the generated Java code.

Table of primitive type mappings

The primitive type mappings are shown in Table 1.

Table 1: *Primitive Schema Type to Primitive Java Type Mapping*

Schema Type	Java Type
`xsd:string`	`java.lang.String`
`xsd:int`	`int`
`xsd:insignedInt`	`long`
`xsd:long`	`long`
`xsd:unsignedLong`	`java.math.BigInteger`
`xsd:short`	`short`
`xsd:unsignedShort`	`int`

Table 1: *Primitive Schema Type to Primitive Java Type Mapping*

Schema Type	Java Type
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:integer	java.math.BigInteger
xsd:decimal	java.math.BigDecimal
xsd:dateTime	java.util.Calendar
xsd:QName	javax.xml.namespace.QName
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]

Special Primitive Type Mappings

Overview

Mapping XMLSchema primitives to Java primitives does not work for all possible data descriptions in an Artix contract. Several cases require that an XMLSchema primitive is mapped to the Java primitive's corresponding wrapper type. These cases include:

- an `<element>` with its `nillable` attribute set to `true` as shown in Example 13.

  Example 13: *Nillable Element*

  ```
  <element name="finned" type="xsd:boolean" nillable="true" />
  ```

- an `<element>` with its `minOccurs` attribute set to `0` and its `maxOccurs` attribute set to `1` or its `maxOccurs` attribute not specified as shown in Example 14.

  Example 14: *minOccurs set to Zero*

  ```
  <element name="plane" type="xsd:string" minOccurs="0" />
  ```

- an `<attribute>` with its use attribute set to `optional`, or not specified, and having neither its `default` attribute nor its `fixed` attribute specified as shown in Example 15.

  Example 15: *Optional Attribute Description*

  ```
  <element name="date">
    <complexType>
      <sequence/>
      <attribute name="calType" type="xsd:string"
       use="optional" />
    </complexType>
  </element>
  ```

Mappings

Table 2 shows how primitive XMLSchema types are mapped into Java wrapper classes in these special cases.

Table 2: *Primitive Schema Type to Java Wrapper Class Mapping*

Schema Type	Java Type
`xsd:int`	`java.lang.Integer`
`xsd:long`	`java.lang.Long`
`xsd:short`	`java.lang.Short`
`xsd:float`	`java.lang.Float`
`xsd:double`	`java.lang.Double`
`xsd:boolean`	`java.lang.Boolean`
`xsd:byte`	`java.lang.Byte`

Unsupported Primitive Types

List of unsupported primitive types

The following XMLSchema primitive types are currently not supported by Artix Java:

```
xsd:duration
xsd:time
xsd:date
xsd:gYearMonth
xsd:gYear
xsd:gMonthDay
xsd:gDay
xsd:gMonth
xsd:anyURI
xsd:nonPositiveInteger
xsd:nonNegativeInteger
xsd:negativeInteger
xsd:positiveInteger
xsd:ENTITY
xsd:NOTATION
xsd:IDREF
soapenc:base64
```

Using XMLSchema Simple Types

Overview

XMLSchema allows you to create simple types by deriving a new type from another primitive type or simple type. Simple types are described in the `<types>` section of an Artix contract using a `<simpleType>` element.

The new types are described by restricting the *base type* with one or more of a number of facets. These facets limit the possible valid values that can be stored in the new type. For example, you could define a simple type, `SSN`, which is a string of exactly 9 characters. Each of the primitive XMLSchema types has their own set of optional facets. Artix does not enforce the use of all the possible facets. However, to ensure interoperability, your service should enforce any restrictions described in the contract.

Describing a simple type in XMLSchema

Example 16 shows the syntax for describing a simple type.

Example 16: *Simple Type Syntax*

```
<simpleType name="typeName">
  <restriction base="baseType">
    <facet value="value"/>
    <facet value="value"/>
    ...
  </restriction>
</simpleType>
```

The type description is enclosed in a `<simpleType>` element and identified by the value of the `name` attribute. The base type from which the new simple type is being defined is specified by the `base` attribute of the `<restriction>` element. Each facet element is specified within the `<restriction>` element. The available facets and their valid setting depends on the base type. For example, `xsd:string` has six facets including:

- `length`
- `minLength`
- `maxLength`
- `pattern`
- `whitespace`

Example 17 shows an example of a simple type, `SSN`, which represents a social security number. The resulting type will be a string of the form `xxx-xx-xxxx`. `<SSN>032-43-9876<SSN>` is a valid value, but `<SSN>032439876</SSN>` is not valid.

Example 17: *SSN Simple Type Description*

```
<simpleType name="SSN">
  <restriction base="xsd:string">
    <pattern value="\d{3}-\d{2}-\d{4}" />
  </restriction>
</simpleType>
```

Mapping simple types to Java

Artix maps simple types to the type of the simple type's base type. So any message using the simple type `SSN`, shown in Example 17, would be mapped to a `String` because the base type of `SSN` is `xsd:string`. For example, the contract fragment shown in Example 18 would result in a Java method, `creditInfo()`, which took a parameter, `socNum`, of `String`.

Example 18: *Credit Request with Simple Types*

```
<message name="creditRequest">
  <part name="socNum" type="SSN" />
</message>
...
<portType name="creditAgent">
  <operation name="creditInfo">
    <input message="tns:creditRequest" name="credRec" />
    <output message="tns:creditReport" name="credRep" />
  </operation>
</portType>
```

Because this mapping does not place any restrictions on the values placed a variable that is mapped from a simple type and Artix does not enforce all facets, you must ensure that your application logic enforces the restrictions described in the contract for maximum interoperability.

Unenforced facets

Artix does not enforce the following facets:

- `length`
- `minLength`
- `maxLength`

- `pattern`
- `enumeration`
- `whiteSpace`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`
- `totalDigits`
- `fractionDigits`

Enforced facets

Artix enforces the following facets:

- `enumeration`

For more information on the enumeration facet, read "Enumerations" on page 63.

Using XMLSchema Complex Types

Overview

Complex types are described in the `<types>` section of an Artix contract. Typically, they are described in XMLSchema using a `<complexType>` element. In contrast to simple types, complex types can contain multiple elements and carry attributes.

Complex types are generated into Java objects according to the mapping specified in the JAX-RPC specification. Each generated object has a default constructor, methods for setting and getting values from the object, and a method for stringifying the object.

In this section

This section contains the following subsections:

Sequence and All Complex Types

Overview

Complex types often describe basic structures that contain a number of fields or elements. XMLSchema provides two mechanisms for describing a structure. One method is to describe the structure inside of a <sequence> element. The other is to describe the structure inside of an `<all>` element. Both methods of describing a structure result in the same generated Java classes.

The difference between using a <sequence> and an `<all>` is in how the elements of the structure are passed on the wire. When a structure is described using a <sequence>, the elements are passed on the wire in the exact order they are specified in the contract. When the structure is described using an `<all>`, the elements of the structure can be passed on the wire in any order.

> **Note:** If neither <sequence>, `<all>`, nor `<choice>` is used to specify how the elements of the complex type are to be transmitted, the default is <sequence>.

Mapping to Java

A complex type described with <sequence> or `<all>` is mapped to a Java class whose name is derived from the `name` attribute of the `<complexType>` element in the contract from which the type is generated. As specified in the JAX-RPC specification, the generated class has a getter and setter method for each element described in the type. The individual elements of the complex type are mapped to private variables within the generated class.

The generated setter methods are named by prepending `set` onto the name of the element as given in the contract. They take a single parameter of the type of the element and have no return value. For example, if a complex type contained the element shown in Example 19, the generated setter method would have the signature `void setName(String val)`.

Example 19: *Element Name Description*

```
<complexType name="Address">
  <all>
    <element name="Name" type="xsd:string" />
    ...
  </all>
</complexType>
```

The generated getter methods are named by prepending get onto the name of the element as given in the contract. They take no parameters and return the value of the specified element. For example, the generated getter method for the element described in Example 19 would the signature like `String getName()`.

Note: If the name of the element begins with a lowercase letter, the getter and setter methods will capitalize the first letter of the element name before prepending `get` or `set`.

In addition to the getter and setter methods, Artix also generates a `toString()` method for each complex type. The `toString()` method returns a string containing a labeled list of the values for each element in the class.

The maxOccurs attribute

Any elements whose `maxOccurs` attribute is set to a value greater than one or set to `unbounded`, results in the generation of a Java array to contain the value of the element. For example, the element described in Example 20 would result in the generation of a private variable, `observedSpeed`,of type `float[]`.

Example 20: *Element with MaxOccurs Greater than One*

```
<complexType name="drugTestResults">
  <sequence>
    <element name="observedSpeed" type="xsd:float"
             maxOccurs="unbounded"/>
...
  </sequence>
</complexType>
```

The getter and setter methods for `observedSpeed` are shown in Example 21.

Example 21: *observedSpeed Getter and Setter Methods*

```
// Java
public class drugTestResults
{
  private float[] observedSpeed;
...
  void setObservedSpeed(float[] val);
  float[] getObservedSpeed();
...
}
```

Example

Suppose you had a contract with the complex type, `monsterStats`, shown in Example 22.

Example 22: *monsterStats Description*

```
<complexType name="monsterStats">
  <all>
    <element name="name" type="xsd:string" />
    <element name="weight" type="xsd:long" />
    <element name="origin" type="xsd:string" />
    <element name="strength" type="xsd:float" />
    <element name="specialAttack" type="xsd:string"
             maxOccurs="3" />
  </all>
</complexType>
```

The Java class generated to support `monsterStats` would be similar to Example 23.

Example 23: *monsterStats Java Class*

```
// Java
public class monsterStats
{
  public static final String TARGET_NAMESPACE =
   "http://monsterBootCamp.com/types/monsterTypes";

  private String name;
  private long weight;
  private String origin;
  private float strength;
  private String[] specialAttack;

  public void setName(String val)
  {
    name=val;
  }
  public String getName()
  {
    return name;
  }
```

Example 23: *monsterStats Java Class*

```
public void setWeight(long val)
{
  weight=val;
}
public long getWeight()
{
  return weight;
}

public void setOrigin(String val)
{
  origin=val;
}
String getOrigin()
{
  return origin;
}

public void setStrength(float val)
{
  strength=val;
}
public float getStrength()
{
  return strength;
}

public void setSpecialAttack(String[] val)
{
  specialAttack=val;
}
public String[] getSpecialAttack()
{
  return specialAttack;
}
```

Example 23: *monsterStats Java Class*

```
   public String toString()
   {
          StringBuffer buffer = new StringBuffer();
          if (name != null) {
              buffer.append("name: "+name+"\n");
          }
          if (weight != null) {
              buffer.append("weight: "+weight+"\n");
          }
          if (origin != null) {
              buffer.append("origin: "+origin+"\n");
          }
          if (strength != null) {
              buffer.append("strength: "+strength+"\n");
          }
          if (specialAttack != null) {
              buffer.append("specialAttack: "+specialAttack+"\n");
          }
          return buffer.toString();
     }
}
```

Choice Complex Types

Overview

XMLSchema allows you to describe a complex type that may contain any one of a number of elements using a `<choice>` element as part of the complex type description. When elements are contained within a `<choice>` element, only one of the elements will be transmitted across the wire. XMLSchema does not require that a discrimintator is specified as part of complex type using a `<choice>` element and how to determine which element is valid is left to the implementation.

Mapping to Java

Like complex types described with a `<sequence>` element or an `<all>` element, complex types described with a `<choice>` element are mapped to a Java class with getter and setter methods for each possible element inside the `<choice>` element. In addition, the generated Java class for a `<choice>` complex type includes an additional element, `_discriminator`, to hold the *discriminator* and a method for each element to determine if it is the current valid value for the choice. For each element in the choice, a method `isSet`*`elem_name`*`()` is generated. If the element is the currently valid value, its `isSet` method returns `true`. If not, the method returns `false`.

The discriminator is set in each of the complex type elements' setter methods. This means that while any of the elements in the Java object representing the complex type may contain valid data, the discriminator points to the last element whose value was set. As stated in the Web services specification only the element to which the discriminator is set will be placed on the wire by a server. For Artix developers this has two implications:

1. Artix servers will only write out the value for the last element set on an object representing a `<choice>` complex type.
2. When Artix clients receive an object representing a `<choice>` complex type, only the element pointed to by the discriminator will contain valid data.

Example

Suppose you had a contract with the complex type, `terrainReport`, shown in Example 24.

Example 24: *terrainReport Description*

```
<complexType name="terrainReport">
  <choice>
    <element name="water" type="xsd:float" />
    <element name="pier" type="xsd:short" />
    <element name="street" type="xsd:long" />
  </choice>
</complexType>
```

The Java class generated to represent `terrainReport` would be similar to Example 25.

Example 25: *terrainReport Java Class*

```
// Java
public class TerrainReport
{
  public static final String TARGET_NAMESPACE =
   "http://GlobeStrollers.com";

  private String __discriminator;

  private float water;
  private short pier;
  private long street;
```

Example 25: *terrainReport Java Class*

```
public void setWater(float _v)
{
  this.water=_v;
  _discriminator="water"’
}
public float getWater()
{
  return water;
}
public boolean isSetWater()
{
  if(__discriminator != null &&
     __discriminator.equals("water")) {
  return true;
  }

  return false;
}

public void setPier(short _v)
{
  this.pier=_v;
  _discriminator="pier";
}
public short getPier()
{
  return pier;
}
public boolean isSetPier()
{
  if(__discriminator != null &&
     __discriminator.equals("pier")) {
  return true;
  }

  return false;
}
```

Example 25: *terrainReport Java Class*

```
  public void setStreet(long _v)
  {
    this.street=_v;
    _discriminator="street";
  }
  public long getStreet()
  {
    return street;
  }
  public boolean isSetStreet()
  {
    if(__discriminator != null &&
       __discriminator.equals("street")) {
    return true;
    }

    return false;
  }

  public void _setToNoMember()
  {
    __discriminator = null;
  }

  public String toString()
  {
        StringBuffer buffer = new StringBuffer();
        if (water != null) {
            buffer.append("water: "+water+"\n");
        }
        if (pier != null) {
            buffer.append("pier: "+pier+"\n");
        }
        if (street != null) {
            buffer.append("street: "+street+"\n");
        }
        return buffer.toString();
    }
}
```

Attributes

Overview

Artix supports the use of <attribute> declarations within the scope of a <complexType> definition. When defining structures for an XML document <attribute> declarations provide a means of adding information to be specified within the tag, not the value that the tag contains. In other words, when describing the XML element <value currency="euro">410<\value> in XMLSchema currency would be described using an <attribute> declaration as shown in Example 26.

Example 26: *XMLSchema for value*

```
<element name="value">
  <complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="units" type="xsd:string"
                       use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

When describing data types for use in developing application logic, however, attributes are treated as elements of a structure. For each <attribute> declaration contained within a complex type description, an element is generated in the class for the attribute along with the appropriate getter and setter methods. The application code must respect the use attribute of the attribute, but the generated Java code does not enforce this behavior.

Describing an attribute in XMLSchema

An XMLSchema <attribute> declaration has two required attributes. The name attribute identifies the attribute. The use attribute specifies if the attribute is required, optional, or prohibited..

An <attribute> declaration also has two optional attributes. The type attribute specifies the type of value the attribute can take. It is used when the attribute takes a value of a primitive type or of a type that is predefined in the contract. If the type attribute is omitted from the <attribute> declaration, the format of the data value must be described as part of the

<attribute> declaration. Example 27 shows an <attribute> declaration for an attribute, catagory, that can take the values autobiography, non-fiction, or fiction.

Example 27: *Attribute with an In-Line Data Description*

```
<attribute name="category" use="required">
  <simpleType>
    <restriction base="xsd:string">
      <enumeration value="autobiography"/>
      <enumeration value="non-fiction"/>
      <enumeration value="fiction"/>
    </restriction>
  </simpleType>
</attribute>
```

Example 28 shows an alternate description of the catagory attribute using the type attribute.

Example 28: *Category Attribute Using the type Attribute*

```
<simpleType name="catagoryType">
  <restriction base="xsd:string">
    <enumeration value="autobiography"/>
    <enumeration value="non-fiction"/>
    <enumeration value="fiction"/>
  </restriction>
</simpleType>
<complexType name="attributed">
...
  <attribute name="category" type="catagoryType" use="required">
</complexType>
```

The default/fixed attribute can be used when the use attribute is set to optional. When the default attribute is given, the value of the generated element is defaulted to the value specified. When the fixed attribute is given, the value of the generated element is set to the value specified and cannot be changed. In the generated Java class, using the fixed attribute results in the generated element not having a setter method.

Example mapping to Java

Suppose you had a contract with the complex type, `terrainReport`, shown in Example 29.

Example 29: *techDoc Description*

```
<complexType name="techDoc">
  <all>
    <element name="product" type="xsd:string" />
    <element name="version" type="xsd:short" />
  <all>
  <attribute name="usefullness" type="xsd:float" use="optional"
             default="0.01" />
</complexType>
```

The Java class generated to represent `terrainReport` would be similar to Example 30.

Example 30: *techDoc Java Class*

```
// Java
public class TechDoc
{
  public static final String TARGET_NAMESPACE =
   "http://www.docUSA.org/usability";

  private String product;
  private short version;
  private Float usefullness = new Float(0.01);

  public void setProduct(String val)
  {
    product=val;
  }
  public String getProdcut()
  {
    return product;
  }
```

Example 30: *techDoc Java Class*

```
  public void setVersion(short val)
  {
    version=val;
  }
  public short getVersion()
  {
    return version;
  }

  public void setUsefullness(Float val)
  {
    usefullness=val;
  }
  public Float getUsefullness()
  {
    return usefullness;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();

    if (prudcut != null) {
      buffer.append("product: "+product+"\n");
    }
    if (version != null) {
      buffer.append("version: "+version+"\n");
    }
    if (usefullness != null) {
      buffer.append("usefullness: "+usefullness+"\n");
    }
  return buffer.toString();
  }
}
```

Nesting Complex Types

Overview

XMLSchema allows you to create complex types that contain elements of a complex type through a process called nesting. There are two ways of nesting complex types:

- Nesting with Named Types
- Nesting with Anonymous Types

Nesting with Named Types

When you nest with a named type your element declaration is the same as when the element was of a primitive type. The name of the complex type that describes the element's data is placed in the element's `type` attribute as shown in Example 31.

Example 31: *Nesting with a Named Type*

```
<complexType name="tweetyBird">
  <sequence>
    <element name="caged" type="xsd:boolean" />
    <element name="granny_proximity" type="xsd:int" />
  </sequence>
</complexType>
<complexType name="sylvesterState">
  <sequence>
    <element name="hunger" type="xsd:int" />
    <element name="food" type="tweetyBird" />
  </sequence>
</complexType>
```

The complex type `sylvesterState` includes an element, `food`, of type `tweetyBird`. The advantage of using named types is that `tweetyBird` can be reused as either a standalone complex type or nested in another complex type description.

Nesting with Anonymous Types

When you nest with an anonymous type, the element declaration for the nested complex type does not have a `type` attribute. Instead, the element's type description is provided as part of the element's declaration. Example 32 shows a description of `sylvesterState` using an anonymous type.

Example 32: *Nesting with an Anonymous Type*

```
<complexType name="sylvesterState">
  <sequence>
    <element name="hunger" type="xsd:int" />
    <element name="food">
      <complexType>
        <sequence>
          <element name="caged" type="xsd:boolean" />
          <element name="granny_proximity" type="xsd:int" />
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

In this example, the `food` element of `sylvesterState` still contains a `caged` sub-element and a `granny_proximity` sub-element. However, the complex type used to describe `food` cannot be re-used.

Mapping to Java

When a complex type containing nested complex types is mapped to Java, each complex type that is nested creates a generated class to represent it. The generated class for the top level complex type will have elements whose elements are instances of the class generated to represent their type. For example, the `sylvesterState` complex type, two Java classes will be generated. One to represent the type of the `food` element and one to represent `sylvesterState`.

The name of the classes generated to support the nested complex types depends on the style of nesting used. For named nested complex types, the generated class takes its name from the `name` attribute of the complex type used to describe it. So the nested type in Example 31 on page 48 would result in a class called `TweetyBird` and the `food` element of `SylvesterState` would be an instance of `TweetyBird`.

When you use anonymous nested complex types Artix names the class generated to represent the nested class by appending `_type` to the name of the parent complex type's `name` attribute. If that does not produce a unique name, Artix will append _*n*, where *n* is an incrementing whole number, to the name until it finds a unique name for the generated class. For example, the nested type in Example 32 on page 49 would generate a class, `SylvesterState_type`, to represent the type of the `food` element in `SylvesterState`. If there were another complex type whose name was `SylvesterState_type` in the contract from which the code was generated, Artix would name the class generated to support the `food` element `SylvesterState_type_1`.

Example using nested types

If you had an application using the complex type shown in Example 31 on page 48 your application would include two classes to support it, `TweetyBird` and `SylvesterState`.

Example 33 shows the generated Java class for `tweetyBird`.

Example 33: *TweetyBird Class*

```
//Java
public class TweetyBird
{
  public static final String TARGET_NAMESPACE =
   "http://toonville.org/foodstuffs";

  private boolean caged;
  private int granny_proximity;

  public boolean isCaged()
  {
    return caged;
  }

  public void setCaged(boolean val)
  {
    caged=val;
  }
```

Example 33: *TweetyBird Class*

```
  public int getGranny_proximity()
  {
    return granny_proximity;
  }

  public void setGranny_proximity(int val)
  {
    granny_proximity=val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();

    if (caged != null) {
      buffer.append("caged: "+caged+"\n");
    }
    if (granny_proximity != null) {
      buffer.append("granny_proximity: "+granny_proximity+"\n");
    }
    return buffer.toString();
  }
}
```

The generated class for `sylvesterState`, shown in Example 34, has one element, `food`, that is an instance of `TweetyBird`.

Example 34: *SylvesterState Class*

```
//Java
public class SylvesterState
{
  public static final String TARGET_NAMESPACE =
   "http://toonville.org/cats";

  private int hunger;
  private TweetyBird food;
```

Example 34: *SylvesterState Class*

```
  public int getHunger()
  {
    return hunger;
  }

  public void setHunger(int val)
  {
    hunger=val;
  }

  public TweetyBird getFood()
  {
    return food;
  }

  public void setFood(TweetyBird val)
  {
    food=val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();

    if (caged != null) {
      buffer.append("hunger: "+hunger+"\n");
    }
    if (granny_proximity != null) {
      buffer.append("food: "+food+"\n");
    }
    return buffer.toString();
  }
}
```

When you set the value of `SylvesterState.food`, you must pass a valid `TweetyBird` object to `setFood()`. Also, when you get the value of `SylvesterState.food`, you are returned a `TweetyBird` object which has its own getter and setter methods. Example 35 shows an example of using the nested type `sylvesterState` in using the generated Java classes.

Example 35: *Working with Nested Complex Types*

```
// Java
```

Example 35: *Working with Nested Complex Types*

```
1 SylvesterState hunter = new SylvesterState();
hunter.setHunger(25);

2 TweetyBird prey = new TweetyBird();
prey.setCaged(false);
prey.setGranny_proximity(0);

3 hunter.setFood(pery);

4 System.out.println("The cat is this hungry:
    "+hunter.getHunger());
System.out.println("The food is caged:
    "+hunter.getFood().isCaged());

5 TweetyBird outPrey = hunter.getFood();
System.out.println("Granny is this many feet away:
    "+outPrey.getGranny_proximity());
```

The code in Example 35 does the following:

1. Instantiates a new `SylvesterState` object and sets its `hunger` element to 25.
2. Instantiates a new `TweetyBird` object and sets its values.
3. Sets the `food` element on `hunter`.
4. Prints out the value of the `hunger` element and the value of the `food` element's `caged` element.
5. Gets the `food` element, assigns it to `outPrey` then prints out the `granny_proximity` element.

Deriving a Complex Type from a Simple Type

Overview

Artix supports derivation of a complex type from a simple type. A simple type has, by definition, neither sub-elements nor attributes. Hence, one of the main reasons for deriving a complex type from a simple type is to add attributes to the simple type.

Example 36 shows an example of a complex type, `internationalPrice`, derived by extension from the `xsd:decimal` simple type to include a currency attribute.

Example 36: *Deriving a Complex Type from a Simple Type by Extension*

```
<complexType name="internationalPrice">
    <simpleContent>
        <extension base="xsd:decimal">
            <attribute name="currency" type="xsd:string"/>
        </extension>
    </simpleContent>
</complexType>
```

The `<simpleContent>` tag indicates that the new type does not contain any sub-elements and the `<extension>` element defines the derivation by extension from `xsd:decimal`.

Java mapping

A complex type derived from a simple type is mapped to a Java class. The class will contain an element, `value`, of the simple type from which the complex type is derived. The class will also have a `get_value()` and a `set_value()` method. In addition, the generated class will have an element, and the associated getter and setter methods, for each attribute that extends the simple type.

Example 37 shows the generated Java class representing `internationalPrice` class generated from Example 36.

Example 37: *internationalPrice Java Class*

```
//Java
public class InternationalPrice
{
  public static final String TARGET_NAMESPACE =
   "http://moneyTree.com";


  private String currency;
  private java.math.BigDecimal _value;

  public String getCurrency()
  {
    return currency;
  }

  public void setCurrency(String val)
  {
    currency = val;
  }

  public java.math.BigDecimal get_value()
  {
    return _value;
  }

  public void set_value(java.math.BigDecimal val)
  {
    _value = val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();
    if (currency != null) {
      buffer.append("currency: "+currency+"\n");
    }
    if (_value != null) {
      buffer.append("_value: "+_value+"\n");
    }
    return buffer.toString();
  }
}
```

The value of the currency attribute, which is added by extension, can be accessed and modified using the `getCurrency()` and `setCurrency()` methods. The simple type value (that is, the value enclosed between the `<internationalPrice>` and `</internationalPrice>` tags) can be accessed and modified by the `get_value()` and `set_value()` methods.

Occurrence Constraints

Overview

XMLSchema allows you to specify the minimum and maximum number of times that an element in a complex type can occur. You specify these occurrence constraints on an element by setting the element's `minOccurs` and `maxOccurs` attributes. The `minOccurs` attribute specifies the minimum number of times the element must occur. The `maxOccurs` attribute specifies the upper limit for how many times the element can occur. For example, if an element, `lives`, were to occur at least twice and no more than nine times in a complex type it would be described as shown in Example 38.

Example 38: *Occurrence Constraints Setting*

```
<complexType name="houseCat">
  <all>
    <element name="name" type="xsd:string" />
    <element name="lives" type="xsd:short" minOccurs="2"
             maxOccurs="9" />
  </all>
</complexType>
```

Given the description in Example 38, a valid `houseCat` element would have a single `name` and at least two `lives`. However, a valid `houseCat` element could not have more than nine `lives`.

Note: When a sequence schema contains a *single* element definition and this element defines occurrence constraints, it is treated as an array. See "SOAP Arrays" on page 60.

Mapping to Java

When a complex type contains an element with its `maxOccurs` attribute set to a value greater than one, the element is mapped to an array of the corresponding Java type. Because XMLSchema requires that the `maxOccurs` attribute of an element is set to a value equal to or greater than the value of the element's `minOccurs`, the code generator will generate a warning if the `minOccurs` attribute is set without a `maxOccurs` attribute. So all valid elements with an occurrence constraint will be mapped into an array.

For example, the complex type, `houseCat`, shown in Example 38 will be mapped to the Java class `HouseCat` shown in Example 39.

Example 39: *HouseCat Java Class*

```
// Java
public class HouseCat
{
  private String name;
  private short[] lives;

  public void setName(String val)
  {
    name=val;
  {
  public String getName()
  {
   return name;
  }

  public void setLives(short[] val)
  {
    lives=val;
  {
  public short[] getLives()
  {
   return lives;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();
    if (name != null)
    {
      buffer.append("name: "+name+"\n");
    }
    if (lives != null)
    {
      buffer.append("lives: "+lives+"\n");
    }
    return buffer.toString();
  }
}
```

The generated code does not force you to obey the min. and max occurrence rules from the contract, but your application code should be sure the obey the contract rules. Attempting to send too few or too many occurrences of an element across the wire will create unpredictable results.

SOAP Arrays

Overview

SOAP encoded arrays support the definition of multi-dimensional arrays, sparse arrays, and partially transmitted arrays. They are mapped directly to Java arrays of the base type used to define the array.

Syntax of a SOAP Array

SOAP arrays can be described by deriving from the `SOAP-ENC:Array` base type using the `wsdl:arrayType`. The syntax for this is shown in Example 40.

Example 40: *Syntax for a SOAP Array derived using wsdl:arrayType*

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="ElementType<ArrayBounds>"/>
    </restriction>
  </complexContent>
</complexType>
```

Using this syntax, *`TypeName`* specifies the name of the newly-defined array type. *`ElementType`* specifies the type of the elements in the array. *`<ArrayBounds>`* specifies the number of dimensions in the array. To specify a single dimension array you would use `[]`; to specify a two-dimensional array you would use either `[][]` or `[,]`.

You can also describe a SOAP Array using a simple element as described in the SOAP 1.1 specification. The syntax for this is shown in Example 41.

Example 41: *Syntax for a SOAP Array derived using an Element*

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <sequence>
        <element name="ElementName" type="ElementType"
                 maxOccurs="unbounded"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

When using this syntax, the element's `maxOccurs` attribute must always be set to unbounded.

Java mapping

SOAP arrays, like basic arrays, are mapped to Java arrays and do not cause a new class to be generated to represent them. Instead, any message that was specified in the Artix contract as being of type *`ArrayType`* or any element of another complex type that was of type `ArrayType` in the Artix contract would be mapped to an array of the appropriate type.

For example, the SOAP Array, `SOAPStrings`, shown in Example 42 defines a one-dimensional array of strings. The `wsdl:arrayType` attribute specifies the type of the array elements, `xsd:string`, and the number of dimensions, `[]` implying one dimension.

Example 42: *Definition of a SOAP Array*

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

Any message of type `SOAPStrings` and any complex type element of type `SOAPStrings` would be mapped to `String[]`. So the contract fragment shown in Example 43, would result in the generation a Java method `celebWasher()` that took a parameter, `badLang`, of type `String[]`.

Example 43: *Operation Using an Array*

```
...
<message name="badLang" type="SOAPStrings" />
<portType name="censor">
  <operation name="celebWasher">
    <input message="badLang" name="badLang" />
  </operation>
</portType>
...
```

Multi-dimensional arrays

Multi-dimensional arrays are also mapped to a Java array of the appropriate type. In the case of a multi-dimensional array, the generated Java array would have the same dimensions as the SOAP array. For example, if `SOAPStrings` were mapped to a two-dimensional array, as shown in Example 44, the mapping of `celebWasher()` would take a parameter, `badLang`, of type `String[][]`.

Example 44: *Definition of a two-dimensional SOAP Array*

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

Sparse and partially transmitted arrays

Sparse and partially transmitted arrays are simply special cases of how an array is populated. A sparse array is an array where not all of the elements are set. For example, if you had an array, `intArray[]`, of 10 integers and only filled in `intArray[1]`, `intArray[6]`, and `intArray[9]`, it would be considered a sparse array.

A partially transmitted array is an array where only a certain range of elements are set. For example, if you had a two dimensional array, `hotMatrix[`*x*`][`*y*`]`, and only set put values in elements where $9 > x > 5$ and $4 > y > 0$, it would be considered a partially transmitted array.

Artix handles both of these cases automatically for you. However, due to differences between Web service implementations, an Artix Java client may receive a fully allocated array with only a few elements containing valid data.

Enumerations

Overview

In XMLSchema, enumerations are described by derivation of a simple type using the syntax shown in Example 45.

Example 45: *Syntax for an Enumeration*

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value" />
    <enumeration value="Case2Value" />
    ...
    <enumeration value="CaseNValue" />
  </restriction>
</simpleType>
```

EnumName specifies the name of the enumeration type. *EnumType* specifies the type of the case values. *CaseNValue*, where *N* is any number one or greater, specifies the value for each specific case of the enumeration. An enumerated type can have any number of case values, but because it is derived from a simple type, only one of the case values is valid at a time.

For example, an XML document with an element defined by the enumeration `widgetSize`, shown in Example 46, would be valid if it were `<widgetSize>big</widgetSize>`, but not if it were `<widgetSize>big,mungo</widgetSize>`.

Example 46: *widgetSize Enumeration*

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
    <enumeration value="gargantuan"/>
  </restriction>
</simpleType>
```

Mapping to a Java class

Artix maps enumerations to a Java class whose name is taken from the schema type's `name` attribute. So Artix would generate a class, `WidgetSize`, to represent the `widgetSize` enumeration.

Note: If the enumeration is an anonymous type nested inside of a complex type, the naming of the generated Java class follows the same pattern as laid out in "Nesting with Anonymous Types" on page 49.

The generated class contains two static public data members for each possible case value. One, *`_CaseNValue`*, holds the data value of the enumeration instance. The other, *`CaseNValue`*, holds an instance of the class associated with the data value. The generated class also contains four public methods:

fromValue() returns the representative static instance of the class based on the value specified. The specified value must be of the enumeration's type and be a valid value for the enumeration. If an invalid value is specified an exception is thrown.

fromString() returns the representative static instance of the class based on a string value. The value inside the string must be a valid value for the enumeration or an exception will be thrown.

getValue() returns the value for the class instance on which it is called.

toString() returns a stringified representation of the class instance on which it is called.

For example Artix would generate the class, `WidgetSize`, shown in Example 47, to represent the enumeration, `widgetSize`, shown in Example 46 on page 63.

Example 47: *WidgetSize Class*

```
// Java
public class WidgetSize
{
  public static final String TARGET_NAMESPACE =
    "http://widgetVendor.com/types/widgetTypes";
```

Example 47: *WidgetSize Class*

```
private final String _val;

public static final String _big = "big";
public static final WidgetSize big = new WidgetSize(_big);

public static final String _large = "large";
public static final WidgetSize large = new WidgetSize(_large);

public static final String _mungo = "mungo";
public static final WidgetSize mungo = new WidgetSize(_mungo);

public static final String _gargantuan = "gargantuan";
public static final WidgetSize gargantuan = new
 WidgetSize(_gargantuan);

protected WidgetSize(String value)
{
  _val = value;
}

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

Example 47: *WidgetSize Class*

```
  public static WidgetSize fromValue(String value)
  {
    if (value.equals("big"))
    {
      return big;
    }
    if (value.equals("large"))
    {
      return large;
    }
    if (value.equals("mungo"))
    {
      return mungo;
    }
    if (value.equals("gargantuan"))
    {
      return gargantuan;
    }
    throw new IllegalArgumentException("Invalid enumeration
   value: "+value);
    };

  public static WidgetSize fromString(String value)
  {
    if (value.equals("big"))
    {
      return big;
    }
    if (value.equals("large"))
    {
      return large;
    }
    if (value.equals("mungo"))
    {
      return mungo;
    }
    if (value.equals("gargantuan"))
    {
      return gargantuan;
    }
    throw new IllegalArgumentException("Invalid enumeration
   value: "+value);
    };
```

Example 47: *WidgetSize Class*

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

Working with enumerations in Java

Unlike the classes generated to represent complex types, the Java classes generated to represent enumerations do not need to be specifically instantiated, nor do they provide setter methods. Instead, you use the `fromValue()` or `fromString()` methods on the class to get a reference to one of the static members of the enumeration. Once you have the reference to your desired member, you use the `getValue()` method on that member to determine the value for the member.

If you were working with the `widgetSize` enumeration, shown in Example 46 on page 63, to build an ordering system, you would need a way to enter the size of the widget you wanted to order and then store that choice as part of the order. Example 48 shows a simple text entry method for getting the proper member of the enumeration using `fromValue()`,

Example 48: *Using fromValue() to Get a Member of an Enumeration*

```
// Java
temp = new String();
WidgetSize ordered_size;

// Get the type of widgets to order
System.out.println("What size widgets do you want?");
System.out.println("Big");
System.out.println("Large");
System.out.println("Mungo");
System.out.println("Gargantuan");
temp = inputBuffer.readLine();

ordered_size = WidgetSize.fromValue(temp);
```

Because the value used to define the cases of the enumeration is a string, `fromValue()` takes a `String` and returns the member based on the value of the string. In this example, `fromString()` is interchangeable with `fromValue()`. However, if the value of the enumeration were integers, `fromValue()` would take an `int`.

To print the bill you will need to display the size of the widgets ordered. To get the value of the ordered widgets, you could use the `getValue()` method to retrieve the value of the enumeration or you could use the `toString()` method to return the value as a `String`. Example 49 uses `getValue()` to return the value of the enumeration retrieved in Example 48 on page 67

Example 49: *Using getValue()*

```
// Java
String sizeVal = ordered_size.getValue();
System.out.println("You ordered "+sizeVal+" sized widgets.");
```

Deriving Types Using <complexContent>

Overview

Using XMLSchema, you can derive new complex types by extending other complex types using the `<complexContent>` element. When generating the Java class to represent the derived complex type, Artix extends the base type's class. In this way, the Artix-generated Java preserves the inheritance hierarchy intended in the XMLSchema.

Schema syntax

You derive complex types from other complex types by using the `<complexContent>` element and the `<extension>` element. The `<complexContent>` element specifies that the included data description includes more than one field. The `<extension>` element, which is part of the `<complexContent>` definition, specifies the base type being extended to create the new type. The base type is specified by the `<extension>` element's `base` attribute.

Within the `<extension>` element, you define the additional fields that make up the new type. All elements that are allowed in a complex type description are allowable as part of the new type's definition. For example, you could add an anonymous enumeration to the new type, or you could use the `<choice>` element to specify that only one of the new fields is to be valid at a time.

Example 50 shows an XMLSchema fragment that defines two complex types, `widgetOrderInfo` and `widgetOrderBillInfo`. `widgetOrderBillInfo` is derived by extending `widgetOrderInfo` to include two new fields, `orderNumber` and `amtDue`.

Example 50: *Deriving a Complex Type by Extension*

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:decimal"/>
    <element name="order_date" type="xsd:dateTime"/>
    <element name="type" type="xsd1:widgetSize"/>
    <element name="shippingAddress" type="xsd1:Address"/>
  </sequence>
  <attribute name="rush" type="xsd:QName" use="optional" />
</complexType>
```

Example 50: *Deriving a Complex Type by Extension*

```
<complexType name="widgetOrderBillInfo">
  <complexContent>
    <extension base="xsd1:widgetOrderInfo">
      <sequence>
        <element name="amtDue" type="xsd:boolean"/>
        <element name="orderNumber" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Generated Java code

As with all complex types defined in a contract, Artix generates a class to represent complex types derived by extension. When the complex type is derived by extension, the generated class extends the base class generated to support the base complex type in the contract.

For example, the schema in Example 50 on page 69 would result in the generation of two Java classes, `WidgetOrderInfo` and `WidgetBillOrderInfo`. `WidgetOrderBillInfo` would extend `WidgetOrderInfo` because `widgetOrderBillInfo` is derived by extension from `widgetOrderInfo`. Example 51 shows the generated class for `widgetOrderBillInfo`.

Example 51: *WidgetOrderBillInfo*

```
// Java
public class WidgetOrderBillInfo extends WidgetOrderInfo
{
  public static final String TARGET_NAMESPACE =
   "http://widgetVendor.com/types/widgetTypes";

  private boolean amtDue;
  private String orderNumber;

  public boolean isAmtDue()
  {
    return amtDue;
  }
```

Example 51: *WidgetOrderBillInfo*

```
  public void setAmtDue(boolean val)
  {
    this.amtDue = val;
  }

  public String getOrderNumber()
  {
    return orderNumber;
  }

  public void setOrderNumber(String val)
  {
    this.orderNumber = val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer(super.toString());
    buffer.append("amtDue: "+amtDue+"\n");
    if (orderNumber != null)
    {
      buffer.append("orderNumber: "+orderNumber+"\n");
    }
    return buffer.toString();
  }
}
```

Holder Classes

Overview

WSDL allows you to describe operations that have multiple output parameters and operations that have in/out parameters. Because Java does not support pass-by-reference, as C++ does, the JAX-RPC 1.1 specification prescribes the use of holder classes as a mechanism to support output and in/out parameters in Java. The holder classes for the Java primitives, and their associated wrapper classes, are packaged in `javax.xml.rpc.holders`. The names of the holder classes start with a capital letter and end with the `Holder` postfix. The name of the holder class for `long` is `LongHolder`. For primitive wrapper classes, `Wrapper` is placed after the class name and before `Holder`. For example, the holder class for `Long` is `LongWrapperHolder`.

For complex types, Artix generates holder classes to represent the complex type when needed. The generated holder classes follows the same naming convention as the primitive holder classes and implement the `javax.xml.rpc.holders.Holder` interface. For example, the holder class for a complex type, `hand`, would be `HandHolder`.

All holder classes provide the following:

- A public field named `value` of the mapped Java type. For example, a `HandHolder` would have a `value` field of type `Hand`.
- A constructor that sets `value` to a default.
- A constructor that sets `value` to the value of the passed in parameter.

Working with holder classes

A holder class is used in the generated Java code when an operation described in your Artix contract either has an output message with multiple parts or when an operation's input message and output message share a part. For a part to be shared it must have the same name and type in both messages. Example 52 shows an example of an operation that would require holder classes in the generated Java code.

Example 52: *Multiple Output Parts*

```
<message name="incomingPackage">
  <part name="ID" type="xsd:long" />
</message>
```

Example 52: *Multiple Output Parts*

```
<message name="outgoingPackage">
  <part name="rerouted" type="xsd:boolean" />
  <part name="destination" type="xsd:string" />
</message>
<portType name="portal">
  <operation name="router">
    <input message="tns:incomingPackage" name="recieved" />
    <output message="tns:outgoingPackage" name="shipped" />
  </operation>
</portType>
```

Artix will use holder classes for the parameters of the Java method generated to implement the operation, `router`, because the output message has multiple parts. Example 53 shows the resulting Java method signature.

Example 53: *Interface Using Holders*

```
//Java
import java.net.*;
import java.rmi.*;

public interface portal extends java.rmi.Remote
{
  public boolean router(long ID,
                 javax.xml.rpc.holders.StringHolder destination)
  throws RemoteException;
}
```

The first part of the `outgoingPackage` message, `rerouted`, is mapped to a boolean return value because it is the first part in the output message. However, the second output message part, `destination`, is mapped to a holder class because it has to be mapped into the method's parameter list.

An example of an application that implements the `portal` interface might be one that determines if a package has reached its final destination. The `router` method would check to see if it need to be forwarded to a new destination and reset the destination appropriately. Example 54 shows how a server might implement the router method.

Example 54: *Portal Implementation*

```
//Java
import java.net.*;
import java.rmi.*;

// The methods boolean belongsHere() and
// String getFinalDestination() are left
// for the reader to implement.

public class portalImpl
{
  public boolean router(long ID,
                javax.xml.rpc.holders.StringHolder destination)
  {
    if(belongsHere(ID))
    {
      return false;
    }

    destination.value = getFinalDestination(ID);
    return true;
  }

}
```

Example 55 shows a client calling `router()` on a portal service.

Example 55: *Client Calling router()*

```
//Java
StringHolder destination = new StringHolder();
long ID = 1232;
boolean continuing;
```

Example 55: *Client Calling router()*

```
// proxy portalClient obtained earlier
continuing = portalClient.router(ID, destination);

if (continuing)
{
  System.out.println("Package "+ID+" is going to
   "+destination.value);
}
```

Creating User-Defined Exceptions

Artix supports the definition of user-defined exceptions using the WSDL <fault> element. When mapped to Java, the <fault> element is mapped to a throwable exception on the associated Java method.

In this chapter

This chapter discusses the following topics:

Describing User-defined Exceptions in an Artix Contract

Overview

Artix allows you to create user-defined exceptions that your service can propagate back to any client using it. As with any information that is exchanged between a service and client in Artix, the exception must be described in the Artix contract. Describing a user-defined exception in an Artix contract involves the following:

- Describing the message that the exception will transmit.
- Associating the exception message to a specific operation.
- Describing how the exception message is bound to the payload format used by the service.

This section will deal with the first two tasks involved in describing a user-defined exception. The fourth task, describing the binding of the exception to a payload format, is beyond the scope of this book. For information on binding messages to specific payload formats in an Artix contract read *Designing Artix Applications from the Command Line* or *Designing Artix Applications*.

Describing the exception message

Messages to be passed in a user-defined exception are described in the same manner as the messages used as input or output messages for an operation. The message is described using the `<message>` element. There are no restrictions on the data types that can be passed as part of an exception message or on the number of parts the message can contain.

> **Note:** When using SOAP as your payload format, you are restricted to using only a single part in your exception messages.

Example 56 shows a message description in an Artix contract.

Example 56: *Message Description*

```
<message name="notEnoughInventory">
  <part name="numInventory" type="xsd:int" />
</message
```

For more information on describing a message in an Artix contract, read *Designing Artix Solutions with Artix Designer* or *Designing Artix Solutions from the Command Line*.

Associating the exception to an operation

Once you have described the message that will be transmitted for your user-defined exception, you need to associate it with an operation in the contract. To do this you add a `<fault>` element to the operation's description. A `<fault>` element takes the same attributes as the `<input>` and `<output>` elements. The `message` attribute specifies the `<message>` element describing the data passed by the exception. The `name` attribute specifies the name by which the exception will be referenced in the binding section of the contract.

Example 57 shows an operation description that uses the message described in Example 56 on page 78 as a user-defined exception.

Example 57: *Operation with a User-defined Exception*

```
<operation name="getWidgets">
  <input message="tns:widgetSizeMessage" name="size" />
  <output message="tns:widgetCostMessage" name="cost" />
  <fault message="tns:notEnoughInventory" name="notEnough" />
</operation>
```

The operation described in Example 57, `getWidgets`, takes one argument denoting the size of the widgets to get from inventory and returns a message stating the cost of the widgets. If the operation cannot get enough widgets, it throws an exception, containing the number of available widgets, back to the client.

How Artix Generates Java User-defined Exceptions

Overview

As specified in the JAX-RPC specification, fault messages describing a user-defined exception in an Artix contract are mapped to a Java exception class by the Artix code generator. The generated class extends the Java `Exception` class so that it can be thrown. It will have one private data member of the type specified in the contract's message part to represent each part of the message, a creation method that allows you to specify the values of each data member, and the associated getter and setter methods for each data member. In addition, the generated class will have a `toString()` method.

The naming scheme for the generated exception class follows that for the generated classes to represent a complex type. The name of the class will be taken from the `name` attribute of the exception's message description and will always start with a capital letter.

Example

Example 58 shows the generated exception class for the fault message in Example 56 on page 78.

Example 58: *Generated Java Class*

```
//Java
import java.util.*;

public class NotEnoughInventory extends Exception
{
  public static final String TARGET_NAMESPACE =
   "http://widgetVendor.com/widgetOrderForm";

  private int numInventory;

  public NotEnoughInventory(int numInventory)
  {
    super();
    this.numInventory = numInventory;
  }
```

Example 58: *Generated Java Class*

```
  public int getNumInventory()
  {
    return numInventory;
  }

  public void setNumInventory(int val)
  {
    numInventory = val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer(super.toString());
    if (size != null)
    {
      buffer.append("numInventory: "+numInventory+"\n");
    }
    return buffer.toString();
  }
}
```

The `TARGET_NAMESPACE` member of the class is the target namespace specified for the Artix contract. It will be the same for all classes generated from a particular contract.

Working with User-defined Exceptions in Artix Applications

Overview

Because Artix generates a standard Java exception class for user-defined exceptions, they are handled like any non-Artix exception in a Java application. The implementation of the service can instantiate and throw Artix user-defined exceptions if they encounter the need. The client invoking the service, as long as it is a JAX-RPC compliant Java web service client or an Artix C++ client, will catch Artix user-defined exceptions like any other exception and inspect the contents using the standard methods.

Example

Example 59 shows how a server implementing the `getWidgets` operation, shown in Example 57 on page 79, might instantiate and throw a `NotEnoughInventory` exception.

Example 59: *Throwing a User-defined Exception*

```
//Java
...
// checkInventory() is left for the reader to implement
// size and numOrdered are parameters passed into the operation
if (numOrdered > checkInventory(size))
{
  throw NotEnoughInventory(checkInventory(size));
}
```

Example 60 shows how a client might catch and report the exception thrown by the server.

Example 60: *Catching a User-defined Exception*

```
// Java
...
try
{
  long cost = getWidgets(size, numOrdered);
}
```

Example 60: *Catching a User-defined Exception*

```
catch(NotEnoughInventory nei)
{
  // get the value stored in the exception
  int numInventory = nei.getNumInventory();
  System.out.println("The factory only has "+numInventory+
                     " widgets of size "+size+".");
}
```

Working with XMLSchema anyTypes

The XMLSchema anyType allows you to place a value of any valid XMLSchema primitive or named complex type into a message. This flexibility, however, adds some complexity to your applications.

In this chapter

This chapter discusses the following topics:

Introduction to Working with XMLSchema anyTypes

XMLSchema anyType

The XMLSchema `anyType` is the root type for all XMLSchema types. All of the primitives are derivatives of this type as are all user defined complex types. As a result, elements defined as being `anyType` can contain data in the form of any of the XMLSchema primitives as well as any complex type defined in a schema document.

Artix and anyType

In Artix, an `anyType` can assume the value of any complex type defined within the `<types>` section of the Artix contract that describes the interface and bindings used by an application. An `anyType` can also assume the value of any XMLSchema primitive. For example, if your contract defines the complex types `joeFriday`, `samSpade`, and `mikeHammer`, an `anyType` used as a message part in an operation can assume the value of an element of type `samSpade` or an element of type `xsd:int`. However, it could not assume the value of an element of type `aceVentura` because `aceVentura` was not defined in the contract.

Artix binding support

Artix supports the use of messages containing parts of `anyType` using payload formats that have a corresponding native construct such as the CORBA `any`. Currently Artix allows using `anyType` with the following payload formats:

- SOAP
- Pure XML
- CORBA

Using anyType in Java

When working with interfaces that use `anyType` parts in it messages, you need to do a few extra things in developing your application. First, you must register the generated type factory class with either the client proxy or the servant depending on which you are developing. Registering the generated type factory with a client proxy is discussed in "Registering Type Factories with a Client Proxy” on page 89. Registering the generated type factory with a servant is discussed in “Registering Type Factories with a Servant” on page 92.

When using data stored in an `anyType`, you can also query the object to determine its actual type before inspecting the data. Retrieving data from an `anyType` is discussed in “Retrieving Data from anyTypes” on page 97.

Java packages for anyType support

When using `anyType` data and the type factories you must import the following:

- `com.iona.webservices.reflect.types.AnyType`
- `com.iona.webservices.reflect.types.TypeFactory`

Registering Type Factories

Overview

When generating Java code, Artix automatically generates a type factory for all user-defined types for contracts that contain an `anyType`. This type factory provides the functionality needed to allow an `anyType` to assume the data of any of the complex types defined in the contract from which the type factory was generated.

You can generate and register more than one type factory per application if you have multiple XMLSchema documents defining types. In the case when you register multiple type factories with an application, the `anyTypes` used in the application can assume the data of any complex type for which the type factories were generated. For example, if you generated a type factory for a schema type defining the data types `larry`, `moe`, and `curly` and you generated a separate type factory from a contract defining the complex type `shemp`, the `anyTypes` used in your application could represent either `larry`, `moe`, `curly`, or `shemp` as long as you registered both type factories with the application.

In this section

This section discusses the following topics:

Registering Type Factories with a Client Proxy

Overview

Type factories are registered with an Artix client proxy using the `Stub` object's `_setProperty()` operation. The client proxy is a child of the `Stub` object so you can simply cast the client proxy to a `Stub` object.

`_setProperty()` takes an array of the type factory's base class. You will need to populate this array with instances of all the type factories you are registering with the client proxy.

Procedure

To register type factories with an Artix Java client proxy complete the following steps:

1. Create the client proxy as described in "Instantiating a client proxy" on page 18.
2. Cast the instantiated client proxy to a `Stub` as shown in Example 61.

Example 61: *Casting a Client Proxy to a Stub*

```
//Java
import javax.xml.rpc.*;

// client proxy, client, created earlier
Stub clientStub = (Stub) client;
```

3. Instantiate the type factories you wish to register with the client proxy as shown in "Instantiating a type factory" on page 89.
4. Create the `TypeFacotry` array used to register the type factories as shown in "Creating a TypeFactory array" on page 90.
5. Register the type factories using `_setProperty()` on the `Stub` object as shown in "Registering the type factories" on page 90.

Instantiating a type factory

When the Artix Java code generator encounters an `anyType` in a contract, it automatically generates a type factory for all of the complex types defined in the contract. The type factory class is named by postfixing `TypeFactory` onto the port type's name. For example if you generated Java code for a port type named `packageDepot`, the generated type factory class would be `packageDepotTypeFactory`.

You instantiate a type factory in the same manner as a typical Java object. Its constructor takes no arguments. Example 62 shows the code to instantiate the type factory for `packageDepot`.

Example 62: *Instantiating a TypeFactory*

```
//Java
packageDepotTypeFactory factory = new packageDepotTypeFactory();
```

Creating a TypeFactory array

The method for registering type factories with the client proxy takes an array of the base type factory class. This class, `com.iona.webservices.reflect.types.TypeFactory`, is the class from which all generated type factories inherit. You can instantiate and populate an array of `TypeFactory` objects using standard Java methods. Example 63 shows code for creating the type factory array to register the `packageDepotTypeFactory` instantiated in Example 62 on page 90.

Example 63: *Creating a TypeFactory Array*

```
//Java
import com.iona.webservices.reflect.types.*;

...
// type factory factory created earlier
TypeFactory[] factArray = new TypeFactory[]{factory};
```

Registering the type factories

You register type factories with the client proxy using the `Stub` object's `_setProperty()` method. The property name for setting Artix type factories is `artix_java_type_factory`. The property's value is the array of `TypeFactory` objects containing all of the type factories you wish to register. Example 64 shows code registering type factories using `_setProperty()`.

Example 64: *Registering Type Factories with _setProperty()*

```
//Java

...
// Stub clientStub and TypeFactory[] factArray obtained above
clientStub._setProperty("artix_java_type_factory", factArray);
```

Determining if the property is set

The client proxy stub provides a method, `_getProperty()` that will return the value of the `artix_java_type_factory` property. You can use this method to determine if the property is already set or to see what type factories are registered with the client proxy. Example 65 shows a code for determining if the type factories have been registered.

Example 65: *Using _getProperty() to See if Type Factories are Registered*

```
// Java
import javax.xml.rpc.*;
import com.iona.webservices.reflect.types.*;

TypeFactory[] setFactory =
              clientStub._getProperty("artix_java_type_factory");
```

Example

Example 66 shows an example of registering two type factories, `packageDepotTypeFactory` and `widgetsTypeFactory`, with a client proxy.

Example 66: *Registering TypeFactories on a Client Proxy*

```
//Java
import javax.xml.rpc.*;
import com.iona.webservices.reflect.types.*;
...
// Start the bus and create the Artix client proxy
1 Stub proxyStub = (Stub) clientProxy;
2 packageDepotTypeFactory fact1 = new packageDepotTypeFactory();
  widgetsTypeFactory fact2 = new widgetsTypeFactory();
3 TypeFactory[] factArray = new TypeFactory[]{fact1, fact2};
4 proxyStub._setProperty("artix_java_type_factory", factArray);
```

The code in Example 66 does the following:

1. Cast the client proxy to a `Stub`.
2. Instantiate the type factories that will be registered.
3. Create and populate an array of `TypeFactory` objects containing the type factories to register.
4. Register the type factories by setting `artix_java_type_factory` using `_setProperty()`.

Registering Type Factories with a Servant

Overview

Type factories are registered with an Artix servant using the servant's `registerTypeFactory()` method. Like the `_setProperty()` method used to register type factories with Artix client proxies, `registerTypeFactory()` takes an array of the type factory base class.

Procedure

To register type factories with an Artix Java servant complete the following steps:

1. Create the servant and register it with the Artix bus as described in "Developing a Server" on page 14.
2. Instantiate the type factories you wish to register with the client proxy as shown in "Instantiating a type factory" on page 92.
3. Create the `TypeFactory` array used to register the type factories as shown in "Creating a TypeFactory array" on page 93.
4. Register the type factories using `resgisterTypeFactory()` on the servant as shown in "Registering the type factories" on page 93.

Instantiating a type factory

When the Artix Java code generator encounters an `anyType` in a contract, it automatically generates a type factory for all of the complex types defined in the contract. The type factory class is named postfixing `TypeFactory` onto the port type's name. For example if you generated Java code for a port type named `packageDepot`, the generated type factory class would be `packageDepotTypeFactory`.

You instantiate a type factory in the same manner as a typical Java object. Its constructor takes no arguments. Example 67 shows the code to instantiate the type factory for `packageDepot`.

Example 67: *Instantiating a TypeFactory*

```
//Java
packageDepotTypeFactory factory = new packageDepotTypeFactory();
```

Creating a TypeFactory array

registerTypeFactory() takes an array of the base type factory class. This class, com.iona.webservices.reflect.types.TypeFactory, is the class from which all generated type factories inherit. You can instantiate and populate an array of TypeFactory objects using standard Java methods. Example 68 shows code for creating the type factory array to register the packageDepotTypeFactory instantiated in Example 67 on page 92.

Example 68: *Creating a TypeFactory Array*

```
//Java
import com.iona.webservices.reflect.types.*;

...
// type factory factory created earlier
TypeFactory[] factArray = new TypeFactory[]{factory};
```

Registering the type factories

You register type factories with the servant using its registerTypeFactory() method with the newly created array of type factories. Example 69 shows code registering type factories with a servant.

Example 69: *Registering Type Factories with _setProperty()*

```
//Java

...
// Servant servant and TypeFactory[] factArray obtained above
servant.registerTypeFactory(factArray);
```

Determining if type factories are registered

You can get a hash table of the type factories registered with a servant using getTypeFactoryMap(). The returned hash table, of type HashMap, contans the QName for the registered type factories and a TypeFactory array containing all of the registered type factories. shows code for returning the hash table of registered type factories.

Example 70: *Getting Hash Table of Registered Type Factories*

```
//Java
HashMap factMap=servant.getTypeFactoryMap();
```

Example

Example 66 shows an example of registering two type factories, `packageDepotTypeFactory` and `widgetsTypeFactory`, with a client proxy.

Example 71: *Registering TypeFactories with a Servant*

```
//Java
import com.iona.webservices.reflect.types.*;
...
// Start the bus and create the Artix servant
1 packageDepotTypeFactory fact1 = new packageDepotTypeFactory();
widgetsTypeFactory fact2 = new widgetsTypeFactory();
2 TypeFactory[] factArray = new TypeFactory[]{fact1, fact2};
3 servant.registerTypeFactory(factArray);
```

The code in Example 71 does the following:

1. Instantiate the type factories that will be registered.
2. Create and populate an array of `TypeFactory` objects containing the type factories to register.
3. Register the type factories.

Setting anyType Values

Overview

In Artix Java `xsd:anyType` is mapped to `com.iona.webservices.reflect.types.AnyType`. This class provides a number of methods for setting the value of an `AnyType` object. There are setter methods for each of the supported primitive types. In addition, there is an overloaded setter method for storing complex types in an `AnyType`. This method allows you to specify the `QName` for the schema type definition of the content along with the data or you can simply supply the data and Artix will attempt to determine the data's schema type when the object is transmitted.

Setting primitive data

The Artix `AnyType` class provides methods for storing primitive data in an `anyType`. The setter methods for the primitive types are listed in Table 3. These methods automatically set the data type identifier to the appropriate schema type when they store the data.

Table 3: *anyType Setter Methods for Primitive Types*

Method	Java Type	XMLSchema Type
`setBoolean()`	`boolean`	`boolean`
`setByte()`	`byte`	`byte`
`setShort()`	`short`	`short`
`setInt()`	`int`	`int`
`setLong()`	`long`	`long`
`setFloat()`	`float`	`float`
`setDouble()`	`double`	`double`
`setString()`	`string`	`string`
`setShort()`	`short`	`short`
`setUByte()`	`short`	`ubyte`
`setUShort()`	`int`	`ushort`

Table 3: *anyType Setter Methods for Primitive Types*

Method	Java Type	XMLSchema Type
`setUInt()`	`long`	`uint`
`setULong()`	`BigInteger`	`ulong`
`setDecimal()`	`BigDecimal`	`decimal`

Setting complex type data

You set complex data into any `AnyType` using the `setType()` method. `setType()` can be used two ways. The first is to provide the `QName` of the XMLSchema type describing the data to store in the `AnyType` along with the data. Using this method makes it easier to query the contents of `anyType` objects that were created in the current application space because Artix does not set the type identifier until it sends the `anyType` across the wire. Example 72 shows code for storing a `widgetSize` in an `anyType`.

Example 72: *Storing Complex Data and Specifying its Type*

```
//Java
widgetSize size = widgetSize.big;
QName qn = new QName("http://widgetVendor.com/types/",
                     "widgetSize");
AnyType aT =new AnyType();
aT.setType(qn, size);
```

The other way is to simply provide the data value to store in the `AnyType` and Artix will determine the XMLSchema type describing the data. From the receiving end this method for storing data in an `anyType` is equivalent to the first method because Artix identifies the contents schema type when it transmits the data. However, the application that store the value will have no way to determine the data type once the value is stored until it is used as part of a remote invocation. Example 73 shows code for storing a `widgetSize` in an `anyType` without providing its `QName`.

Example 73: *Storing Complex Data without a QName*

```
// Java
widgetSize size = widgetSize.big;
AnyType aT =new AnyType();
aT.setType(size);
```

Retrieving Data from anyTypes

Overview

Because an `anyType` can assume the values of a number of different data types, it is beneficial to be able to determine the type of the data stored in an `anyType` before trying to use it. If you knew the value's type you could cast the value into the proper Java type and work with it using standard Java methods.

Artix's Java implementation of `anyType` provides a mechanism for querying the object to determine the schema type of its value. The type identifier is either set when the value is stored in the `anyType` or if the type is not specified when the value is set, Artix sets it when the data is transported through the bus.

You can also use the standard Java `getClass()` method on the Java `Object` returned from `AnyType.getObject()` to get the Java class of the data stored in the `anyType`.

Determining the type of an anyType

The Artix Java `AnyType` provides a method, `getSchemaTypeName()`, that returns the `QName` of the schema type of the data stored in the `anyType`. Example 74 gets the schema type of an `anyType` and prints it out to the console.

Example 74: *Using getSchemaTypeName()*

```
// Java
import com.iona.webservices.relect.types.*;

AnyType blackBox;

// Client proxy, proxy, instantiated previously
blackBox = proxy.newBox();
QName schemaType = blackBox.getSchemaTypeName();
System.out.println("The type for blackBox is defined in "
                   +schemaType.getNamespaceURI());
System.out.println("blackBox is of type: "
                   +schemaType.getLocalPart());
```

The data stored in an Artix `AnyType` is a stored as a standard Java `Object`, so when the data is extracted you can use the standard `getClass()` method on the returned `Object` to determine its Java type.

Extracting primitive types from an anyType

The Artix `AnyType` provides specific methods for extracting primitive types. lists the getter methods for the supported primitive types and the local part of the schema type name returned by `getSchemaType()`. All of the primitive types have `http://www.w3.org/2001/XMLSchema` as their namespace URI.

Table 4: *Methods for Extracting Primitives from AnyType*

Method	Java Type	Schema Type Name
`getBoolean()`	`boolean`	`boolean`
`getByte()`	`byte`	`byte`
`getShort()`	`short`	`short`
`getInt()`	`int`	`int`
`getLong()`	`long`	`long`
`getFloat()`	`float`	`float`
`getDouble()`	`double`	`double`
`getString()`	`String`	`string`
`getUByte()`	`short`	`unsignedByte`
`getUShort()`	`int`	`unsignedShort`
`getUInt()`	`long`	`unsignedInt`
`getULong()`	`BigInteger`	`unsignedLong`
`getDecimal()`	`BigDecimal`	`decimal`

Extracting complex data from an anyType

The Artix `AnyType` provides a generic method, `getType()`, that can be used to extract complex data. `getType()` returns the data store in the `anyType` as a Java Object that you can then cast to the proper Java type. Example 75 shows an example of retrieving a `widgetSize` from an `anyType`.

Example 75: *Extracting a Complex Type from an anyType*

```
// Java
AnyType any;

// Client proxy, proxy, instantiated earlier
any = proxy.returnWidget();
widgetSize size = (widget)any.getObject();
```

Example

If you had an application that processed orders for computers. It may be that your ordering system could receive orders for laptops and destops. Because the laptops and desktops are configured differently you've decided that the orders will be sent using `anyType` elements that the client then processes. You defined the types, `laptopOrder` and `desktopOrder`, in the namespace `http://myAssemblyLine.com/systemTypes`. Example 76 shows code for receiving the order from the server, querying the returned `AnyType` to see what type of order it is, and then extracting the order from the `AnyType`.

Example 76: *Working with anyTypes*

```
// Java
import javax.xml.namespace.QName;
import com.iona.webservices.reflect.types.*;

AnyType anyOrder;
1
// Client proxy, proxy, instantiated earlier
anyOrder = proxy.getSystemOrder();
2
// Get the schema type of the returned order
QName orderType = anyOrder.getSchemaType();
```

Example 76: *Working with anyTypes*

```
3  if (!(orderType.getNamespaceURI().equals(
         "http://myAssemblyLine.com/systemTypes"))
   {
     // handle the fact that the schema type is from the wrong
     // namespace.
   }

4  if (orderType.getLocalPart().equals("laptopOrder"))
   {
     LapTopOrder order = (LapTopOrder)anyOrder.getType();
     buildLaptop(order);
   }
5  if (orderType.getLocalPart().equals("desktopOrder"))
   {
     DeskTopOrder order = (DeskTopOrder)anyOrder.getType();
     buildDesktop(order);
   }
```

The code in Example 76 on page 99 does the following:

1. Populate `anyOrder`.
2. Query `anyOrder` for its schema type information.
3. Check the namespace of the returned type to ensure it correct.
4. Check if `anyOrder` is a `laptopOrder`. If so, cast `anyOrder` into a `laptopOrder`.
5. Check if `anyOrder` is a `desktopOrder`. If so, cast `anyOrder` into a `desktopOrder`.

CHAPTER 6

Artix IDL to Java Mapping

This chapter describes how Artix maps IDL to Java; that is, the mapping that arises by converting IDL to WSDL (using the IDL-to-WSDL compiler) and then WSDL to Java (using the WSDL-to-Java compiler).

In this chapter

This chapter discusses the following topics:

Introduction to IDL Mapping

Overview

This chapter gives an overview of the Artix IDL-to-Java mapping. Mapping IDL to Java in Artix is performed as a two step process, as follows:

1. Map the IDL to WSDL using the Artix IDL compiler. For example, you could map a file, `SampleIDL.idl`, to a WSDL contract, `SampleIDL.wsdl`, using the following command:

   ```
   idl -wsdl SampleIDL.idl
   ```

2. Map the generated WSDL contract to Java using the WSDL-to-Java compiler. For example, you could generate Java stub code from the `SampleIDL.wsdl` file using the following command:

   ```
   wsdltojava SampleIDL.wsdl
   ```

For a detailed discussion of these command-line utilities, see the *Artix Command Line Reference Guide*.

Alternative Java mappings

If you are already familiar with CORBA technology, you will know that there is an existing standard for mapping IDL to Java directly, which is defined by the Object Management Group (OMG). Hence, two alternatives exist for mapping IDL to Java, as follows:

- Artix IDL-to-Java mapping—this is a two stage mapping, consisting of IDL-to-WSDL and WSDL-to-Java. It is an IONA-proprietary mapping.
- CORBA IDL-to-Java mapping—as specified in the OMG Java Language Mapping document (http://www.omg.org). This mapping is used, for example, by the IONA's Orbix.

These alternative approaches are illustrated in Figure 1.

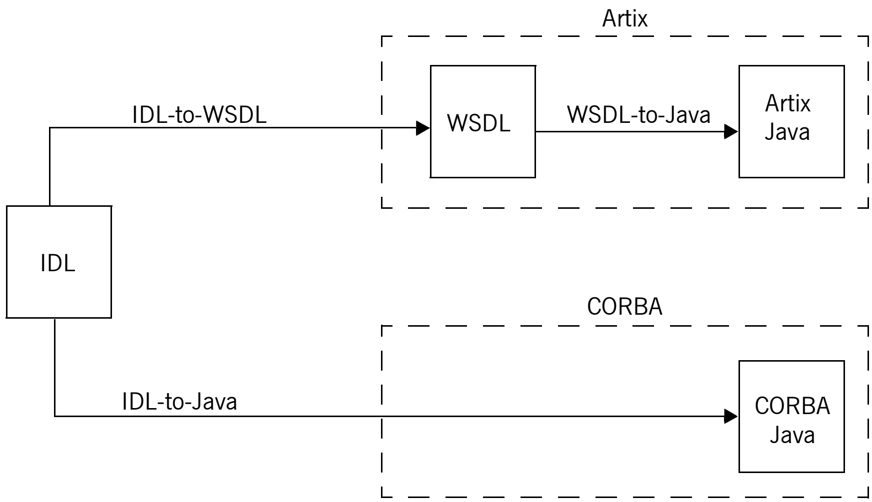

Figure 1: *Artix and CORBA Alternatives for IDL to Java Mapping*

The advantage of using the Artix IDL-to-Java mapping in an application is that it removes the CORBA dependency from your source code. For example, a server that implements an IDL interface using the Artix IDL-to-Java mapping can also interoperate with other Web service protocols, such as SOAP over HTTP.

Unsupported IDL types

The following IDL types are not supported by the Artix Java mapping:

- `long double`.
- Value types.
- Boxed values.
- Abstract interfaces.
- forward-declared interfaces.
- Object.

IDL Basic Type Mapping

Overview

Table 5 shows how IDL basic types are mapped to WSDL and then to Java.

Table 5: *Artix Mapping of IDL Basic Types to Java*

IDL Type	WSDL Schema Type	Java
`boolean`	`xsd:boolean`	`boolean`
`char`	`xsd:byte`	`byte`
`string`	`xsd:string`	`java.lang.String`
`wchar`	`xsd:string`	`java.lang.String`
`wstring`	`xsd:string`	`java.lang.String`
`short`	`xsd:short`	`short`
`long`	`xsd:int`	`int`
`long long`	`xsd:long`	`long`
`unsigned short`	`xsd:unsignedShort`	`int`
`unsigned long`	`xsd:unsignedInt`	`long`
`unsigned long long`	`xsd:unsignedLong`	`java.math.BigInteger`
`float`	`xsd:float`	`float`
`double`	`xsd:double`	`double`
`octet`	`xsd:unsignedByte`	`IT_Bus::UByte`
`fixed`	`xsd:decimal`	`java.math.BigDecimal`

Mapping for string

The IDL-to-WSDL mapping for strings is ambiguous, because the `string`, `wchar`, and `wstring` IDL types all map to the same type, `xsd:string`. This ambiguity can be resolved, however, because the generated WSDL records the original IDL type in the CORBA binding description (that is, within the scope of the `<wsdl:binding> </wsdl:binding>` tags). Hence, whenever an `xsd:string` is sent over a CORBA binding, it is automatically converted back to the original IDL type (`string`, `wchar`, or `wstring`).

IDL Complex Type Mapping

Overview

This section describes how the following IDL data types are mapped to WSDL and then to Java:

- enum type
- struct type
- union type
- sequence types
- array types
- exception types
- typedef of a simple type
- typedef of a complex type

enum type

Consider the following definition of an IDL enum type, `SampleTypes::Shape`:

```
// IDL
module SampleTypes {
    enum Shape { Square, Circle, Triangle };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::Shape` enum to a WSDL restricted simple type, `SampleTypes.Shape`, as follows:

```
<xsd:simpleType name="SampleTypes.Shape">
   <xsd:restriction base="xsd:string">
       <xsd:enumeration value="Square"/>
       <xsd:enumeration value="Circle"/>
       <xsd:enumeration value="Triangle"/>
   </xsd:restriction>
</xsd:simpleType>
```

The WSDL-to-Java compiler maps the `SampleTypes.Shape` type to a Java class, `SampleTypesShape`, as shown in Example 77.

Example 77: *Java Enumeration*

```
// Java
public class SampleTypeShape
{
  ...

  private final String _val;

  public static final String _Square = "Square";
  public static final SampleTypeShape Square = new SampleTypeShape(_Square);

  public static final String _Circle = "Circle";
  public static final SampleTypeShape Circle = new SampleTypeShape(_Circle);

  public static final String _Triangle = "Triangle";
  public static final SampleTypeShape Triangle = new SampleTypeShape(_Triangle);

  protected SampleTypeShape(String value)
  {
    _val = value;
  }

  public String getValue()
  {
    return _val;
  };

  public static SampleTypeShape fromValue(String value)
  {
        if (value.equals(_Square)) {
            return Square;
        }
        if (value.equals(_Circle)) {
            return Circle;
        }
        if (value.equals(_Triangle)) {
            return Triangle;
        }
        throw new IllegalArgumentException("Invalid enumeration value: "+value);
    };
```

Example 77: *Java Enumeration*

```
    public static SampleTypeShape fromString(String value) {
        if (value.equals("Square")) {
            return Square;
        }
        if (value.equals("Circle")) {
            return Circle;
        }
        if (value.equals("Triangle")) {
            return Triangle;
        }
        throw new IllegalArgumentException("Invalid enumeration value: "+value);
    };

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

}
```

The value of the enumeration type can be accessed using the `getValue()` member function.

Programming with the Enumeration Type

For details of how to use the enumeration type, see "Enumerations" on page 63.

---

union type

Consider the following definition of an IDL union type, `SampleTypes::Poly`:

```
// IDL
module SampleTypes {
    union Poly switch(short) {
        case 1: short theShort;
        case 2: string theString;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::Poly` union to an XML schema choice complex type, `SampleTypes.Poly`, as follows:

```
<xsd:complexType name="SampleTypes.Poly">
    <xsd:choice>
        <xsd:element name="theShort" type="xsd:short"/>
        <xsd:element name="theString" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

The WSDL-to-Java compiler maps the `SampleTypes.Poly` type to a Java class, `SampleTypesPoly`, as shown in Example 78.

Example 78: *Java Union*

```
// Java
public class SampleTypesPoly {

...

    private String __discriminator;

    private short theShort;
    private String theString;

    public short getTheShort() {
        return theShort;
    }

    public void setTheShort(short _v) {
        this.theShort = _v;
        __discriminator = "theShort";
    }

    public boolean isSetTheShort() {
        if(__discriminator != null &&
           __discriminator.equals("theShort")) {
            return true;
        }

        return false;
    }
```

Example 78: *Java Union*

```
    public String getTheString() {
        return theString;
    }

    public void setTheString(String _v) {
        this.theString = _v;
        __discriminator = "theString";
    }

    public boolean isSetTheString() {
        if(__discriminator != null &&
           __discriminator.equals("theString")) {
            return true;
        }

        return false;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("theShort: "+theShort+"\n");
        if (theString != null) {
            buffer.append("theString: "+theString+"\n");
        }
        return buffer.toString();
    }
}
```

The value of the union can be modified and accessed using the `get`*`UnionMember`*`()` and `set`*`UnionMember`*`()` pairs of functions.

Programming with the Union Type

For details of how to use the union type, see “Choice Complex Types” on page 40.

struct type

Consider the following definition of an IDL struct type, `SampleTypes::SampleStruct`:

```
// IDL
module SampleTypes {
    struct SampleStruct {
        string theString;
        long theLong;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStruct` struct to an XML schema sequence complex type, `SampleTypes.SampleStruct`, as follows:

```
<xsd:complexType name="SampleTypes.SampleStruct">
    <xsd:sequence>
        <xsd:element name="theString" type="xsd:string"/>
        <xsd:element name="theLong" type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-Java compiler maps the `SampleTypes.SampleStruct` type to a Java class, `SampleTypesSampleStruct`, as shown in Example 79.

Example 79: *Java Struct*

```
//Java
public class SampleTypesSampleStruct {

...
    private String theString;
    private int theLong;

    public String getTheString() {
        return theString;
    }

    public void setTheString(String val) {
        this.theString = val;
    }
```

Example 79: *Java Struct*

```
    public int getTheLong() {
        return theLong;
    }

    public void setTheLong(int val) {
        this.theLong = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (theString != null) {
            buffer.append("theString: "+theString+"\n");
        }
        buffer.append("theLong: "+theLong+"\n");
        return buffer.toString();
    }
}
```

The members of the struct can be accessed and modified using the `get`*`StructMember`*`()` and `set`*`StructMember`*`()` pairs of functions.

Programming with the Struct Type

For details of how to use the struct type, see “Sequence and All Complex Types” on page 34.

---

sequence types

Consider the following definition of an IDL sequence type, `SampleTypes::SeqOfStruct`:

```
// IDL
module SampleTypes {
    typedef sequence< SampleStruct > SeqOfStruct;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SeqOfStruct` sequence to a WSDL sequence type with occurrence constraints, `SampleTypes.SeqOfStruct`, as follows:

```
<xsd:complexType name="SampleTypes.SeqOfStruct">
    <xsd:sequence>
        <xsd:element name="item"
            type="xsd1:SampleTypes.SampleStruct"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-Java compiler maps the `SampleTypes.SeqOfStruct` type to a Java class, `SampleTypesSeqOfStruct`, as shown in Example 80.

Example 80: *Java Sequence*

```
// Java
public class SampleTypesSeqOfStruct {

    private SampleTypesSampleStruct[] item;

    public SampleTypesSampleStruct[] getItem() {
        return item;
    }

    public void setItem(SampleTypesSampleStruct[] val) {
        this.item = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (item != null) {
            buffer.append("item: "+Arrays.asList(item).toString()+"\n");
        }
        return buffer.toString();
    }
}
```

Programming with Sequence Types

For details of how to use sequence types, see “Sequence and All Complex Types” on page 34 .

array types

Consider the following definition of an IDL union type, `SampleTypes::ArrOfStruct`:

```
// IDL
module SampleTypes {
     typedef SampleStruct ArrOfStruct[10];
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::ArrOfStruct` array to a WSDL sequence type with occurrence constraints, `SampleTypes.ArrOfStruct`, as follows:

```
<xsd:complexType name="SampleTypes.ArrOfStruct">
    <xsd:sequence>
        <xsd:element name="item"
            type="xsd1:SampleTypes.SampleStruct"
            minOccurs="10" maxOccurs="10"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-C++ compiler maps the `SampleTypes.ArrOfStruct` type to a C++ class, `SampleTypesArrOfStruct`, as shown in Example 81.

Example 81: *Java Array*

```
//Java
public class SampleTypesArrOfStruct {

    private SampleTypesSampleStruct[] item;

    public SampleTypesSampleStruct[] getItem() {
        return item;
    }

    public void setItem(SampleTypesSampleStruct[] val) {
        this.item = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (item != null) {
            buffer.append("item: "+Arrays.asList(item).toString()+"\n");
        }
        return buffer.toString();
    }
}
```

Programming with Array Types

For details of how to use array types, see “Sequence and All Complex Types” on page 34 ..

---

exception types

Consider the following definition of an IDL exception type, `SampleTypes::GenericException`:

```
// IDL
module SampleTypes {
    exception GenericExc {
        string reason;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::GenericExc` exception to a WSDL sequence type, `SampleTypes.GenericExc`, and to a WSDL fault message, `_exception.SampleTypes.GenericExc`, as follows:

```
<xsd:complexType name="SampleTypes.GenericExc">
    <xsd:sequence>
        <xsd:element name="reason" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
...
<xsd:element name="SampleTypes.GenericExc"
    type="xsd1:SampleTypes.GenericExc"/>
...
<message name="_exception.SampleTypes.GenericExc">
    <part name="exception"
   element="xsd1:SampleTypes.GenericExc"/>
</message>
```

The WSDL-to-Java compiler maps the `SampleTypes.GenericExc` type to the Java class, `SampleTypesGenericExc`, as shown in Example 82.

Example 82: *SampleTypeGenericExc*

```
public class SampleTypesGenericExc {

    private String reason;

    public String getReason() {
        return reason;
    }

    public void setReason(String val) {
        this.reason = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (reason != null) {
            buffer.append("reason: "+reason+"\n");
        }
        return buffer.toString();
    }
}
```

In addition, the WSDL-to-Java compiler creates a class to map the message, `_exception.SampleTypes.GenericExc`, to a Java exception as shown in Example 83.

Example 83: *Java Excpetion*

```
public class SampleTypesGenericExcException extends Exception {
    private String reason;

    public SampleTypesGenericExcException(String reason) {
        super();
        this.reason = reason;
    }

    public SampleTypesGenericExcException() {
        super();
    }

    public String getReason() {
        return reason;
    }

    public void setReason(String val) {
        this.reason = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer(super.toString());
        if (reason != null) {
            buffer.append("reason: "+reason+"\n");
        }
        return buffer.toString();
    }
}
```

Programming with Exceptions in Artix

For an example of how to use WSDL fault exceptions, see "Creating User-Defined Exceptions" on page 77.

typedef of a simple type

Consider the following IDL typedef that defines an alias of a `float`, `SampleTypes::FloatAlias`:

```
// IDL
module SampleTypes {
    typedef float FloatAlias;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::FloatAlias` typedef directory to the type, `xsd:float`. The WSDL-to-Java compiler then maps the `xsd:float` type directly to the `float` type.

typedef of a complex type

Consider the following IDL typedef that defines an alias of a `struct`, `SampleTypes::SampleStructAlias`:

```
// IDL
module SampleTypes {
    typedef SampleStruct SampleStructAlias;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStructAlias` typedef directly to the plain, unaliased `SampleTypes.SampleStruct` type.

The WSDL-to-Java compiler then maps the `SampleTypes.SampleStruct` WSDL type directly to the `SampleTypesSampleStruct` Java type. The Java mapping uses the original, unaliased type.

Note: The typedef of an IDL sequence or an IDL array is treated as a special case, with a specific Java class being generated to represent the sequence or array type.

IDL Module and Interface Mapping

Overview

This section describes the Artix C++ mapping for the following IDL constructs:

- Module mapping
- Interface mapping
- Operation mapping
- Attribute mapping

Module mapping

An IDL identifier appearing within the scope of an IDL module, `ModuleName::Identifier`, maps to a Java identifier of the form `ModuleNameIdentifier`. That is, the IDL scoping operator, `::`, is removed in Java.

Although IDL modules do *not* map to packages under the Artix Java mapping, it is possible nevertheless to put generated Java code into a package using the `-p` switch to the WSDL-to-Java compiler (see "Generating Stub and Skeleton Code" on page 10). For example, if you pass a package name, `TEST`, to the WSDL-to-Java `-p` switch, the `ModuleName::Identifier` IDL identifier would map to `TEST.ModuleNameIdentifier`.

Interface mapping

An IDL interface, `InterfaceName`, maps to a Java class of the same name, `InterfaceName`. If the interface is defined in the scope of a module, that is `ModuleName::InterfaceName`, the interface maps to the `ModuleNameInterfaceName` Java class.

If an IDL data type, `TypeName`, is defined within the scope of an IDL interface, that is `ModuleName::InterfaceName::TypeName`, the type maps to the `ModuleNameInterfaceNameTypeName` Java class.

Operation mapping

Example 84 shows two IDL operations defined within the `SampleTypes::Foo` interface. The first operation is a regular IDL operation, `test_op()`, and the second operation is a oneway operation, `test_oneway()`.

Example 84: *Example IDL Operations*

```
// IDL
module SampleTypes {

    interface Foo {
        string test_op(
            in long inLong,
            inout long inoutLong,
            out long   outLong
        );

        oneway void test_oneway(in string in_str);
    };
};
```

The operations from the preceding IDL, Example 84 on page 120, map to Java as shown in Example 85.

Example 85: *Mapping of CORBA Operations to Java*

```
//Java
1 public class FooImpl {
   public String test_op(
                       int inLong,
                      javax.xml.rpc.holders.IntHolder inoutLong,
                      javax.xml.rpc.holders.IntHolder outLong) {
      ...
    }

2    public void test_oneway(String in_str) {
      ...
    }
}
```

The preceding Java operation signatures can be explained as follows:

1. The Java mapping of an IDL operation retains a similar signiture to its IDL definition.

   The order of parameters in the Java function signature, `test_op()`, is determined as follows:

   - First, the `in` parameters appear in the same order as in IDL.
   - Next, the and `inout` parametersappear in the same order as in IDL..
   - Finally, the out parameters appear in the same order as in IDL.

2. The Java mapping of an IDL `oneway` operation is straightforward, because a `oneway` operation can have only `in` parameters and a `void` return type.

Attribute mapping

Example 86 shows two IDL attributes defined within the `SampleTypes::Foo` interface. The first attribute is readable and writable, `str_attr`, and the second attribute is readonly, `bool_attr`.

Example 86: *Example IDL Attributes*

```
// IDL
module SampleTypes {
    ...
    interface Foo {
        ...
        attribute string                str_attr;
        readonly attribute boolean      bool_attr;
    };
};
```

The attributes from the preceding IDL, Example 86 on page 121, map to Java as shown in Example 87.

Example 87: *Mapping IDL Attributes to Java*

```
// Java
public class FooImpl {
1   public String _get_str_attr() {
        // User code goes in here.
        return "";
    }
```

Example 87: *Mapping IDL Attributes to Java*

```
    public void _set_str_attr(String _arg) {
        // User code goes in here.
    }
2   public boolean _get_bool_attr() {
        // User code goes in here.
        return false;
    }
```

The preceding C++ attribute signatures can be explained as follows:

1. A normal IDL attribute, `AttributeName`, maps to a pair of accessor and modifier functions in Java, `_get_AttributeName()`, `_set_`*AttributeName*`()`.
2. An IDL readonly attribute, `AttributeName`, maps to a single accessor function in Java, `_get_AttributeName()`.

Index