Java Web Services: Up and Running, 1st Edition

by Martin Kalin

Publisher: **O'Reilly Media, Inc.**
Pub Date: **February 15, 2009**
Print ISBN-13: **978-0-596-52112-7**
Pages: **336**

Overview

With this example-driven book, you get a quick, practical, and thorough introduction to Java's API for XML Web Services (JAX-WS) and the Java API for RESTful Web Services (JAX-RS). *Java Web Services: Up and Running* takes a clear, no-nonsense approach to these technologies by providing you with a mix of architectural overview, complete working code examples, and short yet precise instructions for compiling, deploying, and executing a sample application. You'll not only learn how to write web services from scratch, but also how to integrate existing services into your Java applications. All the source code for the examples is available from the book's companion website. With *Java Web Services: Up and Running*, you will:

- Understand the distinction between SOAP-based and REST-style services
- Focus on the WSDL (Web Service Definition Language) service contract
- Understand the structure of a SOAP message and the distinction between SOAP versions 1.1 and 1.2
- Learn various approaches to delivering a Java-based RESTful web service, and for consuming commercial RESTful services
- Know the security requirements for web services, both SOAP- and REST-based
- Learn how to implement JAX-WS in various application servers

Ideal for students and experienced programmers alike, *Java Web Services: Up and Running* is the concise guide you need to get going on this technology right away.

Copyright

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Mike Loukides

Editor: Julie Steele

Production Editor: Sarah Schneider

Preface

This is a book for programmers interested in developing Java web services and Java clients against web services, whatever the implementation language. The book is a code-driven introduction to *JAX-WS* (Java API for XML-Web Services), the framework of choice for Java web services, whether SOAP-based or REST-style. My approach is to interpret JAX-WS broadly and, therefore, to include leading-edge developments such as the Jersey project for REST-style web services, officially known as *JAX-RS* (Java API for XML-RESTful Web Services).

JAX-WS is bundled into the *Metro Web Services Stack*, or *Metro* for short. Metro is part of core Java, starting with Standard Edition 6 (hereafter, core Java 6). However, the Metro releases outpace the core Java releases. The current Metro release can be downloaded separately from https://wsit.dev.java.net. Metro is also integrated into the Sun application server, GlassFish. Given these options, this book's examples are deployed in four different ways:

Core Java only

> This is the low-fuss approach that makes it easy to get web services and their clients up and running. The only required software is the Java software development kit (SDK), core Java 6 or later. Web services can be deployed easily using the `Endpoint`, `HttpServer`, and `HttpsServer` classes. The early examples take this approach.

Core Java with the current Metro release

> This approach takes advantage of Metro features not yet available in the core Java bundle. In general, each Metro release makes it easier to write web services and clients. The current Metro release also indicates where JAX-WS is moving. The Metro release also can be used with core Java 5 if core Java 6 is not an option.

Standalone Tomcat

> This approach builds on the familiarity among Java programmers with standalone web containers such as Apache Tomcat, which is the

reference implementation. Web services can be deployed using a web container in essentially the same way as are servlets, JavaServer Pages (JSP) scripts, and JavaServer Faces (JSF) scripts. A standalone web container such as Tomcat is also a good way to introduce *container-managed* security for web services.

GlassFish

This approach allows deployed web services to interact naturally with other *enterprise* components such as Java Message Service topics and queues, a *JNDI* (Java Naming and Directory Interface) provider, a backend database system and the `@Entity` instances that mediate between an application and the database system, and an *EJB* (Enterprise Java Bean) container. The EJB container is important *because* a web service can be deployed as a stateless Session EJB, which brings advantages such as container-managed thread safety. GlassFish works seamlessly with Metro, including its advanced features, and with popular *IDE*s (Integrated Development Environment) such as NetBeans and Eclipse.

An appealing feature of JAX-WS is that the API can be separated cleanly from deployment options. One and the same web service can be deployed in different ways to suit different needs. Core Java alone is good for learning, development, and even lightweight deployment. A standalone web container such as Tomcat provides additional support. A Java application server such as GlassFish promotes easy integration of web services with other enterprise technologies.

P.1. Code-Driven Approach

My code examples are short enough to highlight key features of JAX-WS but also realistic enough to show off the production-level capabilities that come with the JAX-WS framework. Each code example is given in full, including all of the `import` statements. My approach is to begin with a relatively sparse example and then to add and modify features. The code samples vary in length from a few statements to several pages of source. The code is deliberately modular. Whenever there is a choice between conciseness and clarity in coding, I try to opt for clarity.

The examples come with instructions for compiling and deploying the web services and for testing the service against sample clients. This approach

presents the choices that JAX-WS makes available to the programmer but also encourages a clear and thorough analysis of the JAX-WS libraries and utilities. My goal is to furnish code samples that can serve as templates for commercial applications.

JAX-WS is a rich API that is explored best in a mix of overview and examples. My aim is to explain key features about the architecture of web services but, above all, to illustrate each major feature with code examples that perform as advertised. Architecture without code is empty; code without architecture is blind. My approach is to integrate the two throughout the book.

Web services are a modern, lightweight approach to *distributed software systems*, that is, systems such as email or the World Wide Web that require different software components to execute on physically distinct devices. The devices can range from large servers through personal desktop machines to handhelds of various types. Distributed systems are complicated because they are made up of networked components. There is nothing more frustrating than a distributed systems example that does not work as claimed because the debugging is tedious. My approach is thus to provide full, working examples together with short but precise instructions for getting the sample application up and running. All of the source code for examples is available from the book's companion site, at http://www.oreilly.com/catalog/9780596521127. My email address is kalin@cdm.depaul.edu. Please let me know if you find any code errors.

P.2. Chapter-by-Chapter Overview

The book has seven chapters, the last of which is quite short. Here is a preview of each chapter:

Chapter 1

This chapter begins with a working definition of *web services*, including the distinction between SOAP-based and REST-style services. This chapter then focuses on the basics of writing, deploying, and consuming SOAP-based services in core Java. There are web service clients written in Perl, Ruby, and Java to underscore the language neutrality of web services. This chapter also introduces Java's SOAP API and covers various ways to inspect web service traffic at the wire level. The chapter elaborates on the relationship between core Java and Metro.

Chapter 2

This chapter focuses on the service contract, which is a *WSDL* (Web Service Definition Language) document in SOAP-based services. This chapter covers the standard issues of web service style (`document` versus `rpc`) and encoding (`literal` versus `encoded`). This chapter also focuses on the popular but unofficial distinction between the *wrapped* and *unwrapped* variations of document style. All of these issues are clarified through examples, including Java clients against Amazon's *E-Commerce* services. This chapter explains how the *wsimport* utility can ease the task of writing Java clients against commercial web services and how the *wsgen* utility figures in the distinction between `document`-style and `rpc`-style web services. The basics of *JAX-B* (Java API for XML-Binding) are also covered. This chapter, like the others, is rich in code examples.

Chapter 3

This chapter introduces SOAP and logical handlers, which give the service-side and client-side programmer direct access to either the entire SOAP message or just its payload. The structure of a SOAP message and the distinction between SOAP 1.1 and SOAP 1.2 are covered. The messaging architecture of a SOAP-based service is discussed. Various code examples illustrate how SOAP messages can be *processed* in support of application logic. This chapter also explains how *transport-level* messages (for instance, the typical HTTP messages that carry SOAP *payloads* in SOAP-based web services) can be accessed and manipulated in *JAX-WS*. This chapter concludes with a section on JAX-WS support for *transporting* binary data, with emphasis on *MTOM* (Message Transmission Optimization *Mechanism*).

Chapter 4

This chapter opens with a technical analysis of what constitutes a REST-style service and moves quickly to code examples. The chapter surveys various approaches to delivering a Java-based RESTful service: `WebServiceProvider`, `HttpServlet`, Jersey Plain Old Java Object (POJO), and restlet among them. The use of a *WADL* (Web Application Definition

Language) document as a service contract is explored through code examples. The *JAX-P* (Java API for XML-Processing) packages, which facilitate XML processing, are also covered. This chapter offers several examples of Java clients against real-world REST-style services, including services hosted by Yahoo!, Amazon, and Tumblr.

Chapter 5

This chapter begins with an overview of security requirements for real-world web services, SOAP-based and REST-style. The overview covers central topics such as mutual challenge and message confidentiality, users-roles security, and *WS-Security*. Code examples clarify transport-level security, particularly under HTTPS. Container-managed security is introduced with examples deployed in the standalone Tomcat web container. The security material introduced in this chapter is expanded in the next chapter.

Chapter 6

This chapter starts with a survey of what comes with a Java Application Server (JAS): an EJB container, a messaging system, a naming service, an integrated *database* system, and so on. This chapter has a variety of code examples: a *SOAP-based* service implemented as a stateless Session EJB, `WebService` and `WebServiceProvider` instances deployed through embedded Tomcat, a web service deployed together with a traditional website application, a web service integrated with *JMS* (Java Message Service), a web service that uses an `@Entity` to read and write from the Java DB database system included in GlassFish, and a WS-Security *application* under GlassFish.

Chapter 7

This is a very short chapter that looks at the controversy surrounding SOAP-based and REST-style web services. My aim is to endorse both approaches, either of which is superior to what came before. This chapter traces modern web services from DCE/RPC in the early 1990s through CORBA and DCOM up to the Java EE and .NET frameworks. This

chapter explains why either approach to web services is better than the distributed-object architecture that once dominated in distributed software systems.

P.3. Freedom of Choice: The Tools/IDE Issue

Java programmers have a wide choice of productivity tools such as Ant and Maven for scripting and IDEs such as Eclipse, NetBeans, and IntelliJ IDEA. Scripting tools and IDEs increase productivity by hiding grimy details. In a production environment, such tools and IDEs are the sensible way to go. In a learning environment, however, the goal is to understand the grimy details so that this understanding can be brought to good use during the inevitable bouts of debugging and application maintenance. Accordingly, my book is neutral with respect to scripting tools and IDEs. Please feel free to use whichever tools and IDE suit your needs. My how-to segments go over code compilation, deployment, and execution at the command line so that details such as classpath inclusions and compilation/execution flags are clear. Nothing in any example depends on a particular scripting tool or IDE.

P.4. Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, filenames, file extensions, and emphasis.

Constant width

Used for program listings as well as within paragraphs to refer to program elements such as variable or method names, data types, environment variables, statements, and keywords.

`Constant width bold`

Used within program listings to highlight particularly interesting sections and in paragraphs to clarify acronyms.

This icon signifies a tip, suggestion, or general note.

P.5. Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Java Web Services: Up and Running*, by Martin Kalin. Copyright 2009 Martin Kalin, 978-0-596-52112-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

P.6. Safari® Books Online

NOTE

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at http://safari.oreilly.com.

P.7. How to Contact Us

Please address comments and questions concerning this book to the

publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596521127/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the *O'Reilly* Network, see our website at:

http://www.oreilly.com/

P.8. Acknowledgments

Christian A. Kenyeres, Greg Ostravich, Igor Polevoy, and Ken Yu were kind enough to review this book and to offer insightful suggestions for its improvement. They made the book better than it otherwise would have been. I thank them heartily for the time and effort that they invested in this project. The remaining shortcomings are mine alone, of course.

I'd also like to thank Mike Loukides, my first contact at O'Reilly Media, for his role in shepherding my initial proposal through the process that led to its acceptance. Julie Steele, my editor, has provided invaluable support and the book would not be without her help. My thanks go as well to the many behind-the-scenes people at O'Reilly Media who worked on this project.

This book is dedicated to Janet.

Chapter 1. Java Web Services Quickstart

1.1. What Are Web Services?

Although the term *web service* has various, imprecise, and evolving meanings, a glance at some features typical of web services will be enough to get us into coding a web service and a client, also known as a consumer or requester. As the name suggests, a web service is a kind of *webified application*, that is, an application typically delivered over *HTTP* (Hyper Text Transport Protocol). A web service is thus a distributed application whose components can be deployed and executed on distinct devices. For instance, a stock-picking web service might consist of several code components, each hosted on a separate business-grade server, and the web service might be consumed on PCs, handhelds, and other devices.

Web services can be divided roughly into two groups, *SOAP*-based and *REST*-style. The distinction is not sharp because, as a code example later illustrates, a SOAP-based service delivered over HTTP is a special case of a REST-style service. *SOAP* originally stood for Simple Object Access Protocol but, by serendipity, now may stand for Service Oriented Architecture (SOA) Protocol. Deconstructing SOA is nontrivial but one point is indisputable: whatever SOA may be, web services play a central role in the SOA approach to software design and development. (This is written with tongue only partly in cheek. SOAP is officially no longer an acronym, and SOAP and SOA can live apart from one another.) For now, SOAP is just an *XML* (EXtensible Markup Language) dialect in which documents are messages. In SOAP-based web services, the SOAP is mostly unseen infrastructure. For example, in a typical scenario, called the request/response *message exchange pattern* (MEP), the client's underlying SOAP library sends a SOAP message as a service request, and the web service's underlying SOAP library sends another SOAP message as the corresponding service response. The client and the web service source code may provide few hints, if any, about the underlying SOAP (see Figure 1-1).

Figure 1-1. Architecture of a typical SOAP-based web service

REST stands for REpresentational State Transfer. Roy Fielding, one of the main authors of the HTTP specification, coined the acronym in his Ph.D. dissertation to describe an architectural style in the design of web services. SOAP has standards (under the World Wide Web Consortium [W3C]), toolkits, and bountiful software libraries. REST has no standards, few

toolkits, and meager software libraries. The REST style is often seen as an antidote to the creeping complexity of SOAP-based web services. This book covers SOAP-based and REST-style web services, starting with the SOAP-based ones.

Except in test mode, the client of either a SOAP-based or REST-style service is rarely a web browser but rather an application without a graphical user interface. The client may be written in any language with the appropriate support libraries. Indeed, a major appeal of web services is language transparency: the service and its clients need not *be written in the same* language. Language transparency is the key to web service *interoperability*; that is, the ability of web services and requesters to interact seamlessly *despite* differences in programming languages, support libraries, and platforms. To *underscore* this appeal, clients against our Java web services will be written in various languages such as C#, Perl, and Ruby, and Java clients will consume services written in other languages, including languages unknown.

There is no magic in language transparency, of course. If a SOAP-based web service written in Java can have a Perl or a Ruby consumer, there must be an intermediary that handles the differences in data types between the service and the requester languages. XML technologies, which support structured document interchange and processing, act as the intermediary. For example, in a typical SOAP-based web service, a client transparently sends a SOAP document as a request to a web service, which transparently returns another SOAP document as a response. In a REST-style service, a client might send a standard HTTP request to a web service and receive an appropriate XML document as a response.

Several features distinguish web services from other distributed software systems. Here are three:

Open infrastructure

> Web services are deployed using industry-standard, vendor-independent protocols such as HTTP and XML, which are ubiquitous and well understood. Web services can piggyback on networking, data formatting, security, and other infrastructures already in place, which lowers entry costs and promotes interoperability among *services*.

Language transparency

Web services and their clients can interoperate even if written in different programming languages. Languages such as C/C++, C#, Java, Perl, Python, Ruby, and others provide libraries, utilities, and even frameworks in support of web *services*.

Modular design

Web services are meant to be modular in design so that new services can be generated through the integration and layering of existing services. Imagine, for example, an inventory-tracking service integrated with an online ordering service to yield a service that automatically orders the appropriate products in response to inventory levels.

1.1.1. What Good Are Web Services?

This obvious question has no simple, single answer. Nonetheless, the chief benefits and promises of web services are clear. Modern software systems are written in a variety of languages—a variety that seems likely to increase. These software systems will continue to be hosted on a variety of platforms. Institutions large and small have significant investment in legacy software systems whose functionality is useful and perhaps mission critical; and few of these institutions have the will and the resources, human or financial, to rewrite their legacy systems.

It is rare that a software system gets to run in splendid isolation. The typical software system must interoperate with others, which may reside on different hosts and be written in different languages. Interoperability is not just a long-term challenge but also a current requirement of production software.

Web services address these issues directly because such services are, first and foremost, language- and platform-neutral. If a legacy COBOL system is exposed through a web service, the system is thereby interoperable with service clients written in other programming languages.

Web services are inherently *distributed* systems that communicate mostly over HTTP but can communicate over other popular transports as well. The communication payloads of web services are structured text (that is, XML documents), which can be inspected, transformed, persisted, and otherwise

processed with widely and even freely available tools. When efficiency demands it, however, web services also can deliver binary payloads. Finally, web services are a work in progress with real-world distributed systems as their test bed. For all of these reasons, web services are an essential tool in any modern programmer's toolbox.

The examples that follow, in this chapter and the others, are meant to be simple enough to isolate critical features of web services but also realistic enough to illustrate the power and flexibility that such services bring to software development. Let the examples begin.

1.2. A First Example

The first example is a SOAP-based web service in Java and clients in Perl, Ruby, and Java. The Java-based web service consists of an interface and an implementation.

1.2.1. The Service Endpoint Interface and Service Implementation Bean

The first web service in Java, like almost all of the others in this book, can be compiled and deployed using core Java SE 6 (Java Standard Edition 6) or greater without any additional software. All of the libraries required to compile, execute, and consume web services are available in core Java 6, which supports *JAX-WS* (Java API for XML-Web Services). JAX-WS supports SOAP-based and REST-style services. JAX-WS is commonly shortened to *JWS* for Java Web Services. The current version of JAX-WS is 2.x, which is a bit confusing because version 1.x has a different label: JAX-RPC. JAX-WS preserves but also significantly extends the capabilities of JAX-RPC.

A SOAP-based web service could be implemented as a single Java class but, following best practices, there should be an interface that declares the methods, which are the web service operations, and an implementation, which defines the methods declared in the interface. The interface is called the *SEI*: Service Endpoint Interface. The implementation is called the *SIB*: Service Implementation Bean. The SIB can be either a POJO or a Stateless Session *EJB* (Enterprise Java Bean). Chapter 6, which deals with the GlassFish Application Server, shows how to implement a web service as an EJB. Until then, the SOAP-based web services will be implemented as POJOs, that is, as instances of regular Java classes. These web services will be published using library classes that come with core Java 6 and, a bit later, with standalone Tomcat and *GlassFish*.

Core Java 6, JAX-WS, and Metro

Java SE 6 ships with JAX-WS. However, JAX-WS has a life outside of core Java 6 and a separate development team. The bleeding edge of JAX-WS is the *Metro Web Services Stack* (https://wsit.dev.java.net), which includes Project Tango to promote interoperability between the Java platform and *WCF* (Windows Communication Foundation), also known as Indigo. The interoperability initiative goes by the acronym *WSIT* (Web Services Interoperability Technologies). In any case, the current Metro version of JAX-WS, hereafter the *Metro release*, is typically ahead of the JAX-WS that ships with the core Java 6 SDK. With Update 4, the JAX-WS in core Java 6 went from JAX-WS 2.0 to JAX-WS 2.1, a significant improvement.

The frequent Metro releases fix bugs, add features, lighten the load on the programmer, and in general strengthen JAX-WS. At the start my goal is to introduce JAX-WS with as little fuss as possible; for now, then, the JAX-WS that comes with core Java 6 is just fine. From time to time an example may involve more work than is needed under the current Metro release; in such cases, the idea is to explain what is really going on before introducing a Metro shortcut.

The Metro home page provides an easy download. Once installed, the Metro release resides in a directory named *jaxws-ri*. Subsequent examples that use the Metro release assume an environment variable `METRO_HOME`, whose value is the install directory for *jaxws-ri*. The *ri*, by the way, is short for *reference implementation*.

Finally, the downloaded Metro release is a way to do JAX-WS under core Java 5. *JAX-WS* requires at least core Java 5 because support for annotations begins with core *Java 5*.

Example 1-1 is the SEI for a web service that returns the current time as either a string or as the elapsed milliseconds from the Unix epoch, midnight January 1, 1970 GMT.

Example 1-1. Service Endpoint Interface for the TimeServer

```
package ch01.ts;  // time server

import javax.jws.WebService;
import javax.jws.WebMethod;
```

```
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

/**
 *  The annotation @WebService signals that this is the
 *  SEI (Service Endpoint Interface). @WebMethod signals
 *  that each method is a service operation.
 *
 *  The @SOAPBinding annotation impacts the under-the-hood
 *  construction of the service contract, the WSDL
 *  (Web Services Definition Language) document. Style.RPC
 *  simplifies the contract and makes deployment easier.
*/
@WebService
@SOAPBinding(style = Style.RPC) // more on this later
public interface TimeServer {
    @WebMethod String getTimeAsString();
    @WebMethod long getTimeAsElapsed();
}
```

Example 1-2 is the SIB, which implements the SEI.

Example 1-2. Service Implementation Bean for the TimeServer

```
package ch01.ts;

import java.util.Date;
import javax.jws.WebService;

/**
 *  The @WebService property endpointInterface links the
 *  SIB (this class) to the SEI (ch01.ts.TimeServer).
 *  Note that the method implementations are not annotated
 *  as @WebMethods.
*/
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl implements TimeServer {
    public String getTimeAsString() { return new Date().toString(); }
    public long getTimeAsElapsed() { return new Date().getTime(); }
}
```

The two files are compiled in the usual way from the current working directory, which in this case is immediately above the subdirectory *ch01*. The symbol `%` represents the command prompt:

```
% javac ch01/ts/*.java
```

1.2.2. A Java Application to Publish the Web Service

Once the SEI and SIB have been compiled, the web service is ready to be published. In full production mode, a Java Application Server such as BEA WebLogic, GlassFish, JBoss, or WebSphere might be used; but in development and even light production mode, a simple Java application can be used. Example 1-3 is the publisher application for the TimeServer service.

Example 1-3. Endpoint publisher for the TimeServer

```
package ch01.ts;

import javax.xml.ws.Endpoint;

/**
 * This application publishes the web service whose
 * SIB is ch01.ts.TimeServerImpl. For now, the
 * service is published at network address 127.0.0.1.,
 * which is localhost, and at port number 9876, as this
 * port is likely available on any desktop machine. The
 * publication path is /ts, an arbitrary name.
 *
 * The Endpoint class has an overloaded publish method.
 * In this two-argument version, the first argument is the
 * publication URL as a string and the second argument is
 * an instance of the service SIB, in this case
 * ch01.ts.TimeServerImpl.
 *
 * The application runs indefinitely, awaiting service requests.
 * It needs to be terminated at the command prompt with control-C
 * or the equivalent.
 *
 * Once the applicatation is started, open a browser to the URL
 *
 *     http://127.0.0.1:9876/ts?wsdl
 *
 * to view the service contract, the WSDL document. This is an
 * easy test to determine whether the service has deployed
 * successfully. If the test succeeds, a client then can be
 * executed against the service.
*/
public class TimeServerPublisher {
    public static void main(String[ ] args) {
      // 1st argument is the publication URL
      // 2nd argument is an SIB instance
      Endpoint.publish("http://127.0.0.1:9876/ts", new TimeServerImpl());
    }
}
```

Once compiled, the publisher can be executed in the usual way:

```
% java ch01.ts.TimeServerPublisher
```

How the Endpoint Publisher Handles Requests

Out of the box, the `Endpoint` publisher handles one client request at a time. This is fine for getting web services up and running in development mode. However, if the processing of a given request should hang, then all other client requests are effectively blocked. An example at the end of this chapter shows how `Endpoint` can handle requests concurrently so that one hung request does not block the others.

1.2.3. Testing the Web Service with a Browser

We can test the deployed service by opening a browser and viewing the *WSDL* (Web Service Definition Language) document, which is an automatically generated service contract. (WSDL is pronounced "whiz dull.") The browser is opened to a URL that has two parts. The first part is the URL published in the Java TimeServerPublisher application: http://127.0.0.1:9876/ts. Appended to this URL is the query string *?wsdl* in *upper*-, lower-, or mixed case. The result is http://127.0.0.1:9876/ts?wsdl. Example 1-4 is the WSDL document that the browser displays.

Example 1-4. WSDL document for the TimeServer service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://ts.ch01/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    targetNamespace="http://ts.ch01/"
    name="TimeServerImplService">
  <types></types>

  <message name="getTimeAsString"></message>
  <message name="getTimeAsStringResponse">
    <part name="return" type="xsd:string"></part>
  </message>
  <message name="getTimeAsElapsed"></message>
```

```
  <message name="getTimeAsElapsedResponse">
    <part name="return" type="xsd:long"></part>
  </message>

  <portType name="TimeServer">
    <operation name="getTimeAsString" parameterOrder="">
      <input message="tns:getTimeAsString"></input>
      <output message="tns:getTimeAsStringResponse"></output>
    </operation>
    <operation name="getTimeAsElapsed" parameterOrder="">
      <input message="tns:getTimeAsElapsed"></input>
      <output message="tns:getTimeAsElapsedResponse"></output>
    </operation>
  </portType>

  <binding name="TimeServerImplPortBinding" type="tns:TimeServer">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http">
    </soap:binding>
    <operation name="getTimeAsString">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </input>
      <output>
        <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </output>
    </operation>
    <operation name="getTimeAsElapsed">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </input>
      <output>
        <soap:body use="literal" namespace="http://ts.ch01/"></soap:body>
      </output>
    </operation>
  </binding>

  <service name="TimeServerImplService">
      <port name="TimeServerImplPort" binding="tns:TimeServerImplPortBinding">
      <soap:address location="http://localhost:9876/ts"></soap:address>
    </port>
  </service>
</definitions>
```

Chapter 2 examines the WSDL in detail and introduces Java utilities associated with the service contract. For now, two sections of the WSDL (both shown in bold) deserve a quick look. The `portType` section, near the top, groups the operations that the web service delivers, in this case the operations `getTimeAsString` and `getTimeAsElapsed`, which are the two Java

methods declared in the SEI and implemented in the SIB. The WSDL `portType` is like a Java interface in that the `portType` presents the service operations abstractly but provides no implementation detail. Each operation in the web service consists of an `input` and an `output` message, where *input* means *input for the web service*. At runtime, each message is a SOAP document. The other WSDL section of interest is the last, the `service` section, and in particular the service `location`, in this case the URL http://localhost:9876/ts. The URL is called the *service endpoint* and it informs clients about where the service can be accessed.

The WSDL document is useful for both creating and executing clients against a web service. Various languages have utilities for generating client-support code from a WSDL. The core Java utility is now called *wsimport* but the earlier names *wsdl2java* and *java2wsdl* were more descriptive. At runtime, a client can consume the WSDL document associated with a web service in order to get critical information about the data types associated with the operations bundled in the service. For example, a client could determine from our first WSDL that the operation `getTimeAsElapsed` returns an integer and expects no arguments.

The WSDL also can be accessed with various utilities such as *curl*. For example, the command:

```
% curl http://localhost:9876/ts?wsdl
```

also displays the WSDL.

Avoiding a Subtle Problem in the Web Service Implementation

This example departs from the all-too-common practice of having the service's SIB (the class `TimeServerImpl`) connected to the SEI (the interface `TimeServer`) only through the `endpointInterface` attribute in the `@WebService` annotation. It is not unusual to see this:

```
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl { // implements TimeServer removed
```

The style is popular but unsafe. It is far better to have the `implements` clause so that the compiler checks whether the SIB implements the methods declared in the SEI. Remove the `implements` clause and comment out the definitions of the two web service *operations*:

```
@WebService(endpointInterface = "ch01.ts.TimeServer")
public class TimeServerImpl {
    // public String getTimeAsString() { return new Date().toString(); }
    // public long gettimeAsElapsed() { return new Date().getTime(); }
}
```

The code still compiles. With the `implements` clause in place, the compiler issues a fatal error because the SIB fails to define the methods declared in the SEI.

1.3. A Perl and a Ruby Requester of the Web Service

To illustrate the language transparency of web services, the first client against the Java-based web service is not in Java but rather in Perl. The second client is in Ruby. Example 1-5 is the Perl client.

Example 1-5. Perl client for the TimeServer client

```
#!/usr/bin/perl -w

use SOAP::Lite;
my $url = 'http://127.0.0.1:9876/ts?wsdl';
my $service = SOAP::Lite->service($url);

print "\nCurrent time is: ", $service->getTimeAsString();
print "\nElapsed milliseconds from the epoch: ", $service->getTimeAsElapsed();
```

On a sample run, the output was:

```
Current time is: Thu Oct 16 21:37:35 CDT 2008
Elapsed milliseconds from the epoch: 1224211055700
```

The Perl module `SOAP::Lite` provides the under-the-hood functionality that allows the client to issue the appropriate SOAP request and to process the resulting SOAP response. The request URL, the same URL used to test the web service in the browser, ends with a query string that asks for the WSDL document. The Perl client gets the WSDL document from which the `SOAP::Lite` library then generates the appropriate service object (in Perl syntax, the scalar variable `$service`). By consuming the WSDL document, the `SOAP::Lite` library gets the information needed, in particular, the names of the web service operations and the data types involved in these operations. Figure 1-2 depicts the architecture.

Figure 1-2. Architecture of the Perl client and Java service

After the setup, the Perl client invokes the web service operations without any fuss. The SOAP messages remain unseen.

Example 1-6 is a Ruby client that is functionally equivalent to the Perl client.

Example 1-6. Ruby client for the TimeServer client

```
#!/usr/bin/ruby

# one Ruby package for SOAP-based services
require 'soap/wsdlDriver'

wsdl_url = 'http://127.0.0.1:9876/ts?wsdl'


service = SOAP::WSDLDriverFactory.new(wsdl_url).create_rpc_driver

# Save request/response messages in files named '...soapmsgs...'
service.wiredump_file_base = 'soapmsgs'

# Invoke service operations.
result1 = service.getTimeAsString
result2 = service.getTimeAsElapsed

# Output results.
puts "Current time is: #{result1}"
puts "Elapsed milliseconds from the epoch: #{result2}"
```

1.4. The Hidden SOAP

In SOAP-based web services, a client typically makes a remote procedure call against the service by invoking one of the web service operations. As mentioned earlier, this back and forth between the client and service is the request/response message exchange pattern, and the SOAP messages exchanged in this pattern allow the web service and a consumer to be programmed in different languages. We now look more closely at what happens under the hood in our first example. The Perl client generates an HTTP request, which is itself a formatted message whose *body* is a SOAP message. Example 1-7 is the HTTP request from a sample run.

Example 1-7. HTTP request for the TimeServer service

```
POST http://127.0.0.1:9876/ts HTTP/ 1.1
Accept: text/xml
Accept: multipart/*
Accept: application/soap
User-Agent: SOAP::Lite/Perl/0.69
Content-Length: 434
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
     soap:encodingStyle="http:// schemas.xmlsoap.org/soap/encoding/"
     xmlns:soap="http://schemas.xmlsoap.org/soap/ envelope/"
     xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:tns="http://ts.ch01/"
     xmlns:xsd ="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <tns:getTimeAsString xsi:nil="true" />
  </soap:Body>
</soap:Envelope>
```

The HTTP request is a message with a structure of its own. In particular:

- The HTTP *start line* comes first and specifies the request method, in this case the POST method, which is typical of requests for dynamic resources such as web services or other web application code (for example, a Java servlet) as opposed to requests for a static HTML page. In this case, a POST rather than a GET request is needed because only a POST request has a body, which encapsulates the SOAP message. Next comes the request URL followed by the HTTP version, in this case 1.1, that the requester understands. HTTP 1.1 is the current version.

- Next come the HTTP *headers*, which are key/value pairs in which a colon (`:`) separates the key from the value. The order of the key/value pairs is arbitrary. The key `Accept` occurs three times, with a *MIME* (Multiple Internet Mail Extension) type/subtype as the value: `text/xml`, `multipart/*`, and `application/soap`. These three pairs signal that the requester is ready to accept an arbitrary XML response, a response with arbitrarily many attachments of any type (a SOAP message can have arbitrarily many attachments), and a SOAP document, respectively. The HTTP key `SOAPAction` is often present in the HTTP header of a web service request and the key's value may be the empty string, as in this case; but the value also might be the name of the requested web service operation.

- Two *CRLF* (Carriage Return Line Feed) characters, which correspond to two Java `\n` characters, separate the HTTP headers from the HTTP body, which is required for the POST verb but may be empty. In this case, the HTTP body contains the SOAP document, commonly called the SOAP envelope because the outermost or *document* element is named `Envelope`. In this SOAP envelope, the SOAP body contains a single element whose *local name* is `getTimeAsString`, which is the name of the web service operation that the client wants to invoke. The SOAP request envelope is simple in this example because the requested operation takes no *arguments*.

On the web service side, the underlying Java libraries process the HTTP request, extract the SOAP envelope, determine the identity of the requested service operation, invoke the corresponding Java method `getTimeAsString`, and then generate the appropriate SOAP message to carry the method's return value back to the client. Example 1-8 is the HTTP response from the Java `TimeServerImpl` service request shown in Example 1-7.

Example 1-8. HTTP response from the TimeServer service

```
HTTP/1.1 200 OK
Content-Length: 323
Content-Type: text/xml; charset=utf-8
Client-Date: Mon, 28 Apr 2008 02:12:54 GMT
Client-Peer: 127.0.0.1:9876
Client-Response-Num: 1

<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
  <soapenv:Body>
    <ans:getTimeAsStringResponse xmlns:ans="http://ts.ch01/">
      <return>Mon Apr 28 14:12:54 CST 2008</return>
    </ans:getTimeAsStringResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Once again the SOAP envelope is the body of an HTTP message, in this case the HTTP response to the client. The HTTP *start line* now contains the status code as the integer 200 and the corresponding text `OK`, which signal that the client request was handled successfully. The SOAP envelope in the HTTP response's body contains the current time as a string between the XML start and end tags named `return`. The Perl SOAP library extracts the SOAP envelope from the HTTP response and, because of information in the WSDL document, expects the desired return value from the web service operation to occur in the XML `return` element.

1.5. A Java Requester of the Web Service

Example 1-9 is a Java client functionally equivalent to the Perl and Ruby clients shown in Examples Example 1-5 and Example 1-6, respectively.

Example 1-9. Java client for the Java web service

```
package ch01.ts;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import java.net.URL;
class TimeClient {
    public static void main(String args[ ]) throws Exception {
        URL url = new URL("http://localhost:9876/ts?wsdl");

        // Qualified name of the service:
        //   1st arg is the service URI
        //   2nd is the service name published in the WSDL
        QName qname = new QName("http://ts.ch01/", "TimeServerImplService");

        // Create, in effect, a factory for the service.
        Service service = Service.create(url, qname);

        // Extract the endpoint interface, the service "port".
        TimeServer eif = service.getPort(TimeServer.class);

        System.out.println(eif.getTimeAsString());
        System.out.println(eif.getTimeAsElapsed());
   }
}
```

The Java client uses the same URL with a query string as do the Perl and Ruby clients, but the Java client explicitly creates an XML *qualified name*, which has the syntax *namespace URI:local name*. A *URI* is a Uniform Resource Identifier and differs from the more common URL in that a URL specifies a *location*, whereas a URI need not specify a location. In short, a URI need not be a URL. For now, it is enough to underscore that the Java class `java.xml.namespace.QName` represents an XML-qualified name. In this example, the namespace URI is provided in the WSDL, and the local name is the SIB class name `TimeServerImpl` with the word `Service` appended. The local name occurs in the `service` section, the last section of the WSDL document.

Once the `URL` and `QName` objects have been constructed and the `Service.create` method has been invoked, the statement of interest:

```
TimeServer port = service.getPort(TimeServer.class);
```

executes. Recall that, in the WSDL document, the `portType` section describes, in the style of an interface, the operations included in the web service. The `getPort` method returns a reference to a Java object that can invoke the `portType` operations. The `port` object reference is of type `ch01.ts.TimeServer`, which is the SEI type. The Java client, like the Perl client, invokes the two web service methods; and the Java libraries, like the Perl and Ruby libraries, generate and process the SOAP messages exchanged transparently to enable the successful method invocations.

1.6. Wire-Level Tracking of HTTP and SOAP Messages

Example 1-7 and Example 1-8 show an HTTP request message and an HTTP response message, respectively. Each HTTP message encapsulates a SOAP envelope. These message traces were done with the Perl client by changing the Perl `use` directive in Example 1-5:

```
use SOAP::Lite;
```

to:

```
use SOAP::Lite +trace;
```

The Ruby client in Example 1-6 contains a line:

```
service.wiredump_file_base = 'soapmsgs'
```

that causes the SOAP envelopes to be saved in files on the local disk. It is possible to capture the wire-level traffic directly in Java as well, as later examples illustrate. Various options are available for tracking SOAP and HTTP messages at the wire level. Here is a short introduction to some of them.

The *tcpmon* utility (available at https://tcpmon.dev.java.net) is free and downloads as an executable JAR file. Its graphical user interface (GUI) is easy to use. The utility requires only three settings: the server's name, which defaults to `localhost`; the server's port, which would be set to `9876` for the `TimeServer` example because this is the port at which `Endpoint` publishes the service; and the local port, which defaults to `8080` and is the port at which *tcpmon* listens. With *tcpmon* in use, the `TimeClient` would send its requests to port `8080` instead of port `9876`. The *tcpmon* utility intercepts HTTP traffic between the client and web service, displaying the full messages in its GUI.

The Metro release has utility classes for tracking HTTP and SOAP traffic. This approach does not require any change to the client or to the service code; however, an additional package must be put on the classpath and a system property must be set either at the command line or in code. The required package is in the file *jaxws_ri/jaxws-rt.jar*. Assuming that the environment variable `METRO_HOME` points to the *jaxws-ri* directory, here is the command that tracks HTTP and SOAP traffic between the `TimeClient`, which connects to the

service on port 9876, and the `TimeServer` service. (Under Windows, `$METRO_HOME` becomes `%METRO_HOME%`.) The command is on three lines for readability:

```
% java -cp ".":$METRO_HOME/lib/jaxws-rt.jar \
  -Dcom.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true \
  ch01.ts.TimeClient
```

The resulting dump shows all of the SOAP traffic but not all of the HTTP headers. Message tracking also can be done on the service side.

There are various other open source and commercial products available for tracking the SOAP traffic. Among the products that are worth a look at are SOAPscope, NetSniffer, and Wireshark. The *tcpdump* utility comes with most Unix-type systems, including Linux and OS X, and is available on Windows as WinDump. Besides being free, *tcpdump* is nonintrusive in that it requires no change to either a web service or a client. The *tcpdump* utility dumps message traffic to the standard output. The companion utility *tcptrace* (*http://www.tcptrace.org*) can be used to analyze the dump. The remainder of this section briefly covers *tcpdump* as a flexible and powerful trace utility.

Under Unix-type systems, the *tcpdump* utility typically must be executed as *superuser*. There are various flagged arguments that determine how the utility works. Here is a sample invocation:

```
% tcpdump -i lo -A -s 1024 -l 'dst host localhost and port 9876' | tee dump.log
```

The utility can capture packets on any network interface. A list of such interfaces is available with the *tcpdump -D* (under Windows, *WinDump -D*), which is equivalent to the *ifconfig -a* command on Unix-like systems. In this example, the flag/value pair `-i lo` means *capture packets from the interface lo*, where *lo* is short for the *localhost* network interface on many Unix-like systems. The flag `-A` means that the captured packets should be presented in ASCII, which is useful for web packets as these typically contain text. The `-s 1024` flag sets the *snap length*, the number of bytes that should be captured from each packet. The flag `-l` forces the standard output to be line buffered and easier to read; and, on the same theme, the construct `| tee dump.log` at the end pipes the same output that shows up on the screen (the standard

output) into a local file named *dump.log*. Finally, the expression:

```
'dst host localhost and port 9876'
```

acts as a filter, capturing only packets whose destination is *localhost* on port 9876, the port on which `TimeServerPublisher` of Example 1-3 publishes the `TimeServer` service.

The *tcpdump* utility and the `TimeServerPublisher` application can be started in any order. Once both are running, the `TimeClient` or one of the other clients can be executed. With the sample use of *tcpdump* shown above, the underlying network packets are saved in the file *dump.log*. The file does require some editing to make it easily readable. In any case, the *dump.log* file captures the same SOAP envelopes shown in *Examples* Example 1-7 and Example 1-8.

1.7. What's Clear So Far?

The first example is a web service with two operations, each of which delivers the current time but in different representations: in one case as a human-readable string, and in the other case as the elapsed milliseconds from the Unix epoch. The two operations are implemented as independent, self-contained methods. From the service requester's perspective, either method may be invoked independently of the other and one invocation of a service method has no impact on any subsequent invocation of the same service method. The two Java methods depend neither on one another nor on any instance field to which both have access; indeed, the SIB class `TimeServerImpl` has no fields at all. In short, the two method invocations are stateless.

In the first example, neither method expects arguments. In general, web service operations may be parameterized so that appropriate information can be passed to the operation as part of the service request. Regardless of whether the web service operations are parameterized, they still should appear to the requester as independent and self-contained. This design principle will guide all of the samples that we consider, even ones that are richer than the first.

1.7.1. Key Features of the First Code Example

The `TimeServerImpl` class implements a web service with a distinctive message *exchange pattern (MEP)*—request/response. The service allows a client to make a *language-neutral* remote procedure call, invoking the methods `getTimeAsString` and `getTimeAsElapsed`. Other message patterns are possible. Imagine, for example, a web service that tracks new snow amounts for ski areas. Some participating clients, perhaps snow-measuring electrical devices strategically placed around the ski slopes, might use the one-way pattern by sending a snow amount from a particular location but *without* expecting a response from the service. The service might exhibit the notification pattern by multicasting to subscribing clients (for instance, travel bureaus) information about current snow conditions. Finally, the service might periodically use the solicit/response pattern to ask a subscribing client whether the client wishes to continue receiving notifications. In summary, SOAP-based web services support various patterns. The *request/response* pattern of RPC remains the dominant one. The infrastructure needed to support this pattern in particular is worth summarizing:

Message transport

SOAP is designed to be transport-neutral, a design goal that complicates matters because SOAP messages cannot rely on protocol-specific information included in the transport infrastructure. For instance, SOAP delivered over HTTP should not differ from SOAP delivered over some other transport protocol such as *SMTP* (Simple Mail Transfer Protocol), *FTP* (File Transfer Protocol), or even *JMS* (Java Message Service). In practice, however, HTTP is the usual transport for SOAP-based services, a point underscored in the usual name: SOAP-based *web* services.

Service contract

The service client requires information about the service's operations in order to invoke them. In particular, the client needs information about the invocation syntax: the operation's name, the order and types of the arguments passed to the operation, and the type of the returned value. The client also requires the service endpoint, typically the service URL. The WSDL document provides these pieces of information and others. Although a client could invoke a service without first accessing the WSDL, this would make things harder than they need to be.

Type system

The key to language neutrality and, therefore, service/consumer interoperability is a shared type system so that the data types used in the client's invocation coordinate with the types used in the service operation. Consider a simple example. Suppose that a Java web service has the operation:

```
boolean bytes_ok(byte[ ] some_bytes)
```

The `bytes_ok` operation performs some validation test on the bytes passed to the operation as an array argument, returning either `true` or `false`. Now assume that a client written in C needs to invoke `bytes_ok`. C has no types named `boolean` and `byte`. C represents boolean values with integers, with nonzero as `true` and zero as `false`; and the C type `signed char` corresponds to the Java type `byte`. A web service would be

cumbersome to consume if clients had to map client-language types to service-language types. In SOAP-based web services, the XML Schema type system is the default type system that mediates between the client's types and the service's types. In the example above, the XML Schema type `xsd:byte` is the type that mediates between the C `signed char` and the Java `byte`; and the XML Schema type `xsd:boolean` is the mediating type for the C integers nonzero and zero and the Java `boolean` values `true` and `false`. In the notation `xsd:byte`, the prefix `xsd` (XML Schema Definition) underscores that this is an XML Schema type because `xsd` is the usual extension for a file that contains an XML Schema definition; for instance, *purchaseOrder.xsd*.

1.8. Java's SOAP API

A major appeal of SOAP-based web services is that the SOAP usually remains hidden. Nonetheless, it may be useful to glance at Java's underlying support for generating and processing SOAP messages. Chapter 3, which introduces SOAP handlers, puts the SOAP API to practical use. This section provides a first look at the SOAP API through a simulation example. The application consists of one class, `DemoSoap`, but simulates sending a SOAP message as a request and receiving another as a response. Example 1-10 shows the full application.

Example 1-10. A demonstration of Java's SOAP API

```
package ch01.soap;

import java.util.Date;
import java.util.Iterator;
import java.io.InputStream;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.Node;
import javax.xml.soap.Name;

public class DemoSoap {
    private static final String LocalName = "TimeRequest";
    private static final String Namespace = "http://ch01/mysoap/";
    private static final String NamespacePrefix = "ms";

    private ByteArrayOutputStream out;
    private ByteArrayInputStream in;

    public static void main(String[ ] args) {
       new DemoSoap().request();
    }

    private void request() {
       try {
         // Build a SOAP message to send to an output stream.
         SOAPMessage msg = create_soap_message();
```

```
      // Inject the appropriate information into the message.
      // In this case, only the (optional) message header is used
      // and the body is empty.
      SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
      SOAPHeader hdr = env.getHeader();

      // Add an element to the SOAP header.
      Name lookup_name = create_qname(msg);
      hdr.addHeaderElement(lookup_name).addTextNode("time_request");

      // Simulate sending the SOAP message to a remote system by
      // writing it to a ByteArrayOutputStream.
      out = new ByteArrayOutputStream();
      msg.writeTo(out);

      trace("The sent SOAP message:", msg);

      SOAPMessage response = process_request();
      extract_contents_and_print(response);
    }
    catch(SOAPException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }
  }

  private SOAPMessage process_request() {
    process_incoming_soap();
    coordinate_streams();
    return create_soap_message(in);
  }

  private void process_incoming_soap() {
    try {
      // Copy output stream to input stream to simulate
      // coordinated streams over a network connection.
      coordinate_streams();

      // Create the "received" SOAP message from the
      // input stream.
      SOAPMessage msg = create_soap_message(in);

      // Inspect the SOAP header for the keyword 'time_request'
      // and process the request if the keyword occurs.
      Name lookup_name = create_qname(msg);

      SOAPHeader header = msg.getSOAPHeader();
      Iterator it = header.getChildElements(lookup_name);
      Node next = (Node) it.next();
      String value = (next == null) ? "Error!" : next.getValue();

      // If SOAP message contains request for the time, create a
      // new SOAP message with the current time in the body.
      if (value.toLowerCase().contains("time_request")) {

        // Extract the body and add the current time as an element.
        String now = new Date().toString();
        SOAPBody body = msg.getSOAPBody();
        body.addBodyElement(lookup_name).addTextNode(now);
        msg.saveChanges();
```

```
        // Write to the output stream.
        msg.writeTo(out);
        trace("The received/processed SOAP message:", msg);
      }
    }
    catch(SOAPException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }
  }

  private void extract_contents_and_print(SOAPMessage msg) {
    try {
      SOAPBody body = msg.getSOAPBody();

      Name lookup_name = create_qname(msg);
      Iterator it = body.getChildElements(lookup_name);
      Node next = (Node) it.next();

      String value = (next == null) ? "Error!" : next.getValue();
      System.out.println("\n\nReturned from server: " + value);
    }
    catch(SOAPException e) { System.err.println(e); }
  }

  private SOAPMessage create_soap_message() {
    SOAPMessage msg = null;
    try {
      MessageFactory mf = MessageFactory.newInstance();
      msg = mf.createMessage();
    }
    catch(SOAPException e) { System.err.println(e); }
    return msg;
  }

  private SOAPMessage create_soap_message(InputStream in) {
    SOAPMessage msg = null;
    try {
      MessageFactory mf = MessageFactory.newInstance();
      msg = mf.createMessage(null, // ignore MIME headers
                             in);  // stream source
    }
    catch(SOAPException e) { System.err.println(e); }
    catch(IOException e) { System.err.println(e); }
    return msg;
  }

  private Name create_qname(SOAPMessage msg) {
    Name name = null;
    try {
      SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
      name = env.createName(LocalName, NamespacePrefix, Namespace);
    }
    catch(SOAPException e) { System.err.println(e); }
    return name;
  }

  private void trace(String s, SOAPMessage m) {
    System.out.println("\n");
```

```
        System.out.println(s);
        try {
          m.writeTo(System.out);
        }
        catch(SOAPException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }
    }

    private void coordinate_streams() {
        in = new ByteArrayInputStream(out.toByteArray());
        out.reset();
    }
}
```

Here is a summary of how the application runs, with emphasis on the code involving SOAP messages. The `DemoSoap` application's `request` method generates a SOAP message and adds the string `time_request` to the SOAP envelope's header. The code segment, with comments removed, is:

```
SOAPMessage msg = create_soap_message();
SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
SOAPHeader hdr = env.getHeader();
Name lookup_name = create_qname(msg);
hdr.addHeaderElement(lookup_name).addTextNode("time_request");
```

There are two basic ways to create a SOAP message. The simple way is illustrated in this code segment:

```
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();
```

In the more complicated way, the `MessageFactory` code is the same, but the creation call becomes:

```
SOAPMessage msg = mf.createMessage(mime_headers, input_stream);
```

The first argument to `createMessage` is a collection of the transport-layer headers (for instance, the key/value pairs that make up an HTTP header), and the second argument is an input stream that provides the bytes to create the message (for instance, the input stream encapsulated in a Java `Socket` instance).

Once the SOAP message is created, the header is extracted from the SOAP envelope and an XML text node is inserted with the value `time_request`. The resulting SOAP message is:

```
<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header>
      <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
        time_request
      </ms:TimeRequest>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body/>
</SOAP-ENV:Envelope>
```

There is no need right now to examine every detail of this SOAP message. Here is a summary of some key points. The SOAP body is always required but, as in this case, the body may be empty. The SOAP header is optional but, in this case, the header contains the text `time_request`. Message contents such as `time_request` normally would be placed in the SOAP body and special processing information (for instance, user authentication data) would be placed in the header. The point here is to illustrate how the SOAP header and the SOAP body can be manipulated.

The `request` method writes the SOAP message to a `ByteArrayOutputStream`, which simulates sending the message over a network connection to a receiver on a different host. The `request` method invokes the `process_request` method, which in turn delegates the remaining tasks to other methods. The processing goes as follows. The received SOAP message is created from a `ByteArrayInputStream`, which simulates an input stream on the receiver's side; this stream contains the sent SOAP message. The SOAP message now is created from the input stream:

```
SOAPMessage msg = null;
try {
   MessageFactory mf = MessageFactory.newInstance();
   msg = mf.createMessage(null, // ignore MIME headers
                          in);  // stream source (ByteArrayInputStream)
}
```

and then the SOAP message is processed to extract the `time_request` string. The extraction goes as follows. First, the SOAP header is extracted from the SOAP message and an iterator over the elements with the tag name:

```
<ms:TimeRequest xmlns:ms="http://ch01/mysoap/>
```

is created. In this example, there is one element with this tag name and the element should contain the string `time_request`. The lookup code is:

```
SOAPHeader header = msg.getSOAPHeader();
Iterator it = header.getChildElements(lookup_name);
Node next = (Node) it.next();
String value = (next == null) ? "Error!" : next.getValue();
```

If the SOAP header contains the proper request string, the SOAP body is extracted from the incoming SOAP message and an element containing the current time as a string is added to the SOAP body. The revised SOAP message then is sent as a response. Here is the code segment with the comments removed:

```
if (value.toLowerCase().contains("time_request")) {
  String now = new Date().toString();
  SOAPBody body = msg.getSOAPBody();
  body.addBodyElement(lookup_name).addTextNode(now);
  msg.saveChanges();

  msg.writeTo(out);
  trace("The received/processed SOAP message:", msg);
}
```

The outgoing SOAP message on a sample run was:

```
<SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header>
       <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
          time_request
       </ms:TimeRequest>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
       <ms:TimeRequest xmlns:ms="http://ch01/mysoap/">
          Mon Oct 27 14:45:53 CDT 2008
       </ms:TimeRequest>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This example provides a first look at Java's API. Later examples illustrate production-level use of the SOAP API.

1.9. An Example with Richer Data Types

The operations in the `TimeServer` service take no arguments and return simple types, a string and an integer. This section offers a richer example whose deta are clarified in the next chapter.

The `Teams` web service in Example 1-11 differs from the `TimeServer` service in several important ways.

Example 1-11. The Teams document-style web service

```
package ch01.team;

import java.util.List;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Teams {
    private TeamsUtility utils;

    public Teams() {
       utils = new TeamsUtility();
       utils.make_test_teams();
    }

    @WebMethod
    public Team getTeam(String name) { return utils.getTeam(name); }

    @WebMethod
    public List<Team> getTeams() { return utils.getTeams(); }
}
```

For one, the `Teams` service is implemented as a single Java class rather than as separate SEI and SIB. This is done simply to illustrate the possibility. A more important difference is in the return types of the two `Teams` operations. The operation `getTeam` is parameterized and returns an object of the programmer-defined type `Team`, which is a list of `Player` instances, another programmer-defi type. The operation `getTeams` returns a `List<Team>`, that is, a Java `Collection`.

The utility class `TeamsUtility` generates the data. In a production environment, this utility might retrieve a team or list of teams from a database. To keep thi example simple, the utility instead creates the teams and their players on the Here is part of the utility:

```
package ch01.team;

import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

public class TeamsUtility {
    private Map<String, Team> team_map;

    public TeamsUtility() {
       team_map = new HashMap<String, Team>();
    }

    public Team getTeam(String name) { return team_map.get(name); }
    public List<Team> getTeams() {
       List<Team> list = new ArrayList<Team>();
       Set<String> keys = team_map.keySet();
       for (String key : keys)
          list.add(team_map.get(key));
       return list;
    }

    public void make_test_teams() {
       List<Team> teams = new ArrayList<Team>();
       ...
       Player chico = new Player("Leonard Marx", "Chico");
       Player groucho = new Player("Julius Marx", "Groucho");
       Player harpo = new Player("Adolph Marx", "Harpo");
       List<Player> mb = new ArrayList<Player>();
       mb.add(chico); mb.add(groucho); mb.add(harpo);
       Team marx_brothers = new Team("Marx Brothers", mb);
       teams.add(marx_brothers);

       store_teams(teams);
    }

    private void store_teams(List<Team> teams) {
       for (Team team : teams)
          team_map.put(team.getName(), team);
    }
}
```

1.9.1. Publishing the Service and Writing a Client

Recall that the SEI for the `TimeServer` service contains the annotation:

```
@SOAPBinding(style = Style.RPC)
```

This annotation requires that the service use only very simple types such as string and integer. By contrast, the `Teams` service uses richer data types, which means that `Style.DOCUMENT`, the default, should replace `Style.RPC`. The document style does require more setup, which is given below but not explained until the next chapter. Here, then, are the steps required to get the web service deployed and a sample client written quickly:

1. The source files are compiled in the usual way. From the working directory which has *ch01* as a subdirectory, the command is:

```
% javac ch01/team/*.java
```

   In addition to the `@WebService`-annotated `Teams` class, the *ch01/team* directory contains the `Team`, `Player`, `TeamsUtility`, and `TeamsPublisher` classes shown below all together:

```
package ch01.team;
public class Player {
    private String name;
    private String nickname;

    public Player() { }
    public Player(String name, String nickname) {
       setName(name);
       setNickname(nickname);
    }

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
    public void setNickname(String nickname) { this.nickname = nickname; }
    public String getNickname() { return nickname; }
}
// end of Player.java

package ch01.team;

import java.util.List;
public class Team {
    private List<Player> players;
    private String name;

    public Team() { }
    public Team(String name, List<Player> players) {
       setName(name);
       setPlayers(players);
    }

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
    public void setPlayers(List<Player> players) { this.players = players; }
    public List<Player> getPlayers() { return players; }
```

```
    public void setRosterCount(int n) { } // no-op but needed for property
 public int getRosterCount() { return (players == null) ? 0 : players.size(

    }
}
// end of Team.java

package ch01.team;
import javax.xml.ws.Endpoint;
class TeamsPublisher {
    public static void main(String[ ] args) {
       int port = 8888;
       String url = "http://localhost:" + port + "/teams";
       System.out.println("Publishing Teams on port " + port);
       Endpoint.publish(url, new Teams());
    }
}
```

2. In the working directory, invoke the *wsgen* utility, which comes with core Java 6:

   ```
   % wsgen -cp . ch01.team.Teams
   ```

   This utility generates various *artifacts*; that is, Java types needed by the method `Endpoint.publish` to generate the service's WSDL. Chapter 2 looks closely at these artifacts and how they contribute to the WSDL.

3. Execute the `TeamsPublisher` application.

4. In the working directory, invoke the *wsimport* utility, which likewise come with core Java 6:

   ```
   % wsimport -p teamsC -keep http://localhost:8888/teams?wsdl
   ```

   This utility generates various classes in the subdirectory *teamsC* (the `-p` fl stands for `package`). These classes make it easier to write a client against t service.

Step 4 expedites the coding of a client, which is shown here:

```
import teamsC.TeamsService;
```

```
import teamsC.Teams;
import teamsC.Team;
import teamsC.Player;
import java.util.List;
class TeamClient {
    public static void main(String[ ] args) {
        TeamsService service = new TeamsService();
        Teams port = service.getTeamsPort();
        List<Team> teams = port.getTeams();
        for (Team team : teams) {
            System.out.println("Team name: " + team.getName() +
                               " (roster count: " + team.getRosterCount() + ")"
            for (Player player : team.getPlayers())
                System.out.println("  Player: " + player.getNickname());
        }
    }
}
```

When the client executes, the output is:

```
Team name: Abbott and Costello (roster count: 2)
  Player: Bud
  Player: Lou
Team name: Marx Brothers (roster count: 3)
  Player: Chico
  Player: Groucho
  Player: Harpo
Team name: Burns and Allen (roster count: 2)
  Player: George
  Player: Gracie
```

This example hints at what is possible in a commercial-grade, SOAP-based we service. Programmer-defined types such as `Player` and `Team`, along with arbitra collections of these, can be arguments passed to or values returned from a we service so long as certain guidelines are followed. One guideline comes into pl in this example. For the `Team` and the `Player` classes, the JavaBean properties a of types `String` or `int`; and a `List`, like any Java `Collection`, has a `toArray` meth In the end, a `List<Team>` reduces to arrays of simple types; in this case `String` instances or `int` values. The next chapter covers the details, in particular how *wsgen* and the *wsimport* utilities facilitate the development of JWS services ar clients.

1.10. Multithreading the Endpoint Publisher

In the examples so far, the `Endpoint` publisher has been single-threaded and, therefore, capable of handling only one client request at a time: the published service completes the processing of one request before beginning the processing of another request. If the processing of the current request hangs, then no subsequent request can be processed unless and until the hung request is processed to completion.

In production mode, the `Endpoint` publisher would need to handle concurrent requests so that several pending requests could be processed at the same time. If the underlying computer system is, for example, a symmetric multiprocessor (SMP), then separate CPUs could process different requests concurrently. On a single-CPU machine, the concurrency would occur through time sharing; that is, each request would get a share of the available CPU cycles so that several requests would be in some stage of processing at any given time. In Java, concurrency is achieved through multithreading. At issue, then, is how to make the `Endpoint` publisher multithreaded. The JWS framework supports `Endpoint` multithreading without forcing the programmer to work with difficult, error-prone constructs such as the `synchronized` block or the `wait` and `notify` method invocations.

An `Endpoint` object has an `Executor` property defined with the standard `get`/`set` methods. An `Executor` is an object that executes `Runnable` tasks; for example, standard Java `Thread` instances. (The `Runnable` interface declares only one method whose declaration is `public void run()`.) An `Executor` object is a nice alternative to `Thread` instances, as the `Executor` provides high-level constructs for submitting and managing tasks that are to be executed concurrently. The first step to making the `Endpoint` publisher multithreaded is thus to create an `Executor` class such as the following very basic one:

```
package ch01.ts;

import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class MyThreadPool extends ThreadPoolExecutor {
    private static final int pool_size = 10;
```

```
    private boolean is_paused;
    private ReentrantLock pause_lock = new ReentrantLock();
    private Condition unpaused = pause_lock.newCondition();

    public MyThreadPool(){
       super(pool_size,        // core pool size
             pool_size,        // maximum pool size
             0L,               // keep-alive time for idle thread
             TimeUnit.SECONDS, // time unit for keep-alive setting
             new LinkedBlockingQueue<Runnable>(pool_size)); // work queue
    }

    // some overrides
    protected void beforeExecute(Thread t, Runnable r) {
       super.beforeExecute(t, r);
       pause_lock.lock();
       try {
          while (is_paused) unpaused.await();
       }
       catch (InterruptedException e) { t.interrupt(); }
       finally { pause_lock.unlock(); }
    }

    public void pause() {
       pause_lock.lock();
       try {
          is_paused = true;
       }
       finally { pause_lock.unlock(); }
    }

    public void resume() {
       pause_lock.lock();
       try {
          is_paused = false;
          unpaused.signalAll();
       }
       finally { pause_lock.unlock(); }
    }
}
```

The class `MyThreadPool` creates a pool of 10 threads, using a fixed-size queue to store the threads that are created under the hood. If the pooled threads are all in use, then the next task in line must wait until one of the busy threads becomes available. All of these management details are handled automatically. The `MyThreadPool` class overrides a few of the available methods to give the flavor.

A `MyThreadPool` object can be used to make a multithreaded `Endpoint` publisher. Here is the revised publisher, which now consists of several methods to divide the work:

```
package ch01.ts;

import javax.xml.ws.Endpoint;

class TimePublisherMT { // MT for multithreaded
    private Endpoint endpoint;

    public static void main(String[ ] args) {
        TimePublisherMT self = new TimePublisherMT();
        self.create_endpoint();
        self.configure_endpoint();
        self.publish();
    }
    private void create_endpoint() {
        endpoint = Endpoint.create(new TimeServerImpl());
    }
    private void configure_endpoint() {
        endpoint.setExecutor(new MyThreadPool());
    }
    private void publish() {
        int port = 8888;
        String url = "http://localhost:" + port + "/ts";
        endpoint.publish(url);
        System.out.println("Publishing TimeServer on port " + port);
    }
}
```

Once the `ThreadPoolWorker` has been coded, all that remains is to set the `Endpoint` publisher's executor property to an instance of the worker class. The details of thread management do not intrude at all into the publisher.

The multithreaded `Endpoint` publisher is suited for lightweight production, but this publisher is not a service *container* in the true sense; that is, a software application that readily can deploy many web services at the same port. A web container such as Tomcat, which is the reference implementation, is better suited to publish multiple web services. Tomcat is introduced in later examples.

1.11. What's Next?

A SOAP-based web service should provide, as a WSDL document, a service contract for its potential clients. So far we have seen how a Perl, a Ruby, and a Java client can request the WSDL at runtime for use in the underlying SOAP libraries. Chapter 2 studies the WSDL more closely and illustrates how it may be used to generate client-side artifacts such as Java classes, which in turn ease the coding of web service clients. The Java clients in Chapter 2 will not be written from scratch, as is our first Java client. Instead such clients will be written with the considerable aid of the *wsimport* utility, as was the `TeamClient` shown earlier. Chapter 2 also introduces *JAX-B* (Java API for XML-Binding), a collection of Java packages that coordinate Java data types and XML data types. The *wsgen* utility generates JAX-B artifacts that play a key role in this coordination; hence, *wsgen* also will get a closer look.

Chapter 2. All About WSDLs

2.1. What Good Is a WSDL?

The usefulness of WSDLs, the service contracts for SOAP-based web services, is shown best through examples. The original Java client against the `TimeServer` service invokes the *Service.create* method with two arguments: a URL, which provides the endpoint at which the service can be accessed, and an XML-qualified name (a Java `QName`), which in turn consists of the service's local name (in this case, `TimeServerImplService`) and a namespace identifier (in this case, the URI `http://ts.ch01/`). Here is the relevant code without the comments:

```
URL url = new URL("http://localhost:9876/ts?wsdl");
QName qname = new QName("http://ts.ch01/", "TimeServerImplService");
Service service = Service.create(url, qname);
```

Note that the automatically generated namespace URI *inverts* the package name of the service implementation bean (SIB), `ch01.ts.TimeServerImpl`. The package `ch01.ts` becomes `ts.ch01` in the URI. This detail is critical. If the first argument to the `QName` constructor is changed to the URI `http://ch01.ts/`, the Java `TimeClient` throws a exception, complaining that the service's automatically generated WSDL does not describe a service with this namespace URI. The programmer must figure out the namespace URI—presumably by inspecting the WSDL! By the way, even the trailing slash in the URI is critical. The URI `http://ts.ch01`, with a slash missing at the end, causes the same exception as does `http://ch01.ts/`, with *ch01* and *ts* in the wrong order.

The same point can be illustrated with a revised Perl client, which accesses the web service but without requesting its WSDL, as shown in Example 2-1.

Example 2-1. A revised Perl client for the TimeServer service

```
#!/usr/bin/perl -w

use SOAP::Lite;

my $endpoint = 'http://127.0.0.1:9876/ts'; # endpoint
my $uri      = 'http://ts.ch01/';          # namespace

my $client = SOAP::Lite->uri($uri)->proxy($endpoint);

my $response = $client->getTimeAsString()->result();
print $response, "\n";
$response = $client->getTimeAsElapsed()->result();
print $response, "\n";
```

The revised Perl client is functionally equivalent to the original. However, the revised Perl client must specify the service's namespace URI: `http://ts.ch01/`. No other URI would work because the Java-generated WSDL produces exactly this one. In effect, the web service is accessible through a pair: a service endpoint (URL) and a service namespace (URI). It is harder to code the Perl client in Example 2-1 than the original Perl client. The revised client requires the programmer to know the service namespace URI in addition to the service endpoint URL. The original Perl client sidesteps the problem by requesting the WSDL document, which includes the service URI. The original Perl client, which gets the URI from the consumed WSDL document, is the easier way to go.

2.1.1. Generating Client-Support Code from a WSDL

Java has a *wsimport* utility that eases the task of writing a Java client against a SOAP-based web service. The utility generates client-support code or *artifacts* from the service contract, the WSDL document. At a command prompt, the command:

```
% wsimport
```

displays a short report on how the utility can be used. The first example with *wsimport* produces client artifacts against the `TimeServer` service.

After the `ch01.ts.TimeServerPublisher` application has been started, the command:

```
% wsimport -keep -p client http://localhost:9876/ts?wsdl
```

generates two source and two compiled files in the subdirectory *client*. The URL at the end—the same one used in the original Perl, Ruby, and Java clients to request the service's WSDL document—gives the location of the service contract. The option `-p` specifies the Java package in which the generated files are to be placed, in this case a package named `client`; the package name is arbitrary and the utility uses or creates a subdirectory with the package name. The option `-keep` indicates that the source files should be kept, in this case for inspection. The `-p` option is important because *wsimport* generates the file *TimeServer.class*, which has the same

name as the compiled versions of the original service endpoint interface (SEI). If a package is not specified with the *wsimport* utility, then the default package is the package of the service implementation, in this case `ch01.ts`. In short, using the `-p` option prevents the compiled SEI file from being overwritten by the file generated with the *wsimport* utility. If a local copy of the WSDL document is available (for instance, the file named *ts.wsdl*), then the command would be:

```
% wsimport -keep -p client ts.wsdl
```

Examples Example 2-2 and Example 2-3 are the two source files, with comments removed, that the *wsimport* command generates.

Example 2-2. The wsimport-generated TimeServer

```
package client;

import javax.jws.WebMethod;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name = "TimeServer", targetNamespace = "http://ts.ch01/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface TimeServer {
    @WebMethod
    @WebResult(partName = "return")
    public String getTimeAsString();

    @WebMethod
    @WebResult(partName = "return")
    public long getTimeAsElapsed();
}
```

Example 2-3. The wsimport-generated TimeServerImplService

```
package client;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;

@WebServiceClient(name = "TimeServerImplService",
                  targetNamespace = "http://ts.ch01/",
                  wsdlLocation = "http://localhost:9876/ts?wsdl")
public class TimeServerImplService extends Service {
```

```
    private final static URL TIMESERVERIMPLSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new URL("http://localhost:9876/ts?wsdl");
        }
        catch (MalformedURLException e) {
            e.printStackTrace();
        }
        TIMESERVERIMPLSERVICE_WSDL_LOCATION = url;
    }

    public TimeServerImplService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public TimeServerImplService() {
        super(TIMESERVERIMPLSERVICE_WSDL_LOCATION,
              new QName("http://ts.ch01/", "TimeServerImplService"));
    }

    @WebEndpoint(name = "TimeServerImplPort")
    public TimeServer getTimeServerImplPort() {
        return (TimeServer)super.getPort(new QName("http://ts.ch01/",
                                                   "TimeServerImplPort"),
                                         TimeServer.class);
    }
}
```

Three points about these generated source files deserve mention. First, the interface `client.TimeServer` declares the very same methods as the original SEI, `TimeServer`. The methods are the web service operation `getTimeAsString` and the operation `getTimeAsElapsed`. Second, the class `client.TimeServerImplService` has a no-argument constructor that constructs the very same `Service` object as the original Java client `TimeClient`. Third, `TimeServerImplService` encapsulates the `getTimeServerImplPort` method, which returns an instance of type `client.TimeServer`, which in turn supports invocations of the two web service operations. Together the two generated types, the interface `client.TimeServer` and the class `client.TimeServerImplService`, ease the task of writing a Java client against the web service. Example 2-4 shows a client that uses the client-support code from the *wsimport* utility.

Example 2-4. A Java client that uses wsimport artifacts

```
package client;
```

```
class TimeClientWSDL {
    public static void main(String[ ] args) {
       // The TimeServerImplService class is the Java type bound to
       // the service section of the WSDL document.
       TimeServerImplService service = new TimeServerImplService();

       // The TimeServer interface is the Java type bound to
       // the portType section of the WSDL document.
       TimeServer eif = service.getTimeServerImplPort();

       // Invoke the methods.
       System.out.println(eif.getTimeAsString());
       System.out.println(eif.getTimeAsElapsed());
    }
}
```

The client in Example 2-4 is functionally equivalent to the original `TimeClient`, but this client is far easier to write. In particular, troublesome yet critical details such as the appropriate `QName` and service endpoint now are hidden in the *wsimport*-generated class, `client.TempServerImplService`. The idiom that is illustrated here works in *general* for *writing clients* with help from WSDL-based artifacts such as `TimeServer` and `TimeServerImplService`:

- First, construct a `Service` object using one of two constructors in the *wsimport*-generated class, in this example `client.TimeServerImplService`. The no-argument constructor is preferable because of its simplicity. However, a two-argument constructor is also available in case the web service's namespace (URI) or the service endpoint (URL) have changed. Even in this case, however, it would be advisable to regenerate the WSDL-based Java files with another use of the *wsimport* utility.

- Invoke the `get...Port` method on the constructed `Service` object, in this example, the method `getTimeServerImplPort`. The method returns an object that *encapsulates* the web service operations, in this case `getTimeAsString` and `getTimeAsElapsed`, declared in the original SEI.

2.1.2. The @WebResult Annotation

The WSDL-based `client.TimeServer` interface introduces the `@WebResult` annotation. To show how this annotation works, Example 2-5 is a revised version of the `ch01.ts.TimeServer` SEI (with comments removed).

Example 2-5. A more annotated version of the TimeServer service

```
package ch01.ts;  // time server

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
@SOAPBinding(style = Style.RPC) // more on this later
public interface TimeServer {
    @WebMethod
    @WebResult(partName = "time_response")
    String getTimeAsString();

    @WebMethod
    @WebResult(partName = "time_response")
    long getTimeAsElapsed();
}
```

The `@WebResult` annotates the two web service operations. In the resulting WSDL, the `message` section reflects this change:

```
<message name="getTimeAsString"></message>
<message name="getTimeAsStringResponse">
    <part name="time_response" type="xsd:string"></part>
</message>
<message name="getTimeAsElapsed"></message>
<message name="getTimeAsElapsedResponse">
    <part name="time_response" type="xsd:long"></part>
</message>
```

Note that `time_response` now replaces `return` from the original WDSL. The SOAP response document from the web service likewise reflects the change:

```
<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
     <ans:getTimeAsStringResponse xmlns:ans="http://ts.ch01/">
       <time_response>
          Thu Mar 27 21:20:09 CDT 2008
       </time_response>
     </ans:getTimeAsStringResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Once again, the `time_response` tag replaces the `return` tag from the original SOAP response. If the `@WebResult` annotation were applied, say, only to the `getTimeAsString` operation, then the SOAP response for this operation would use the `time_response` tag but the response for the `getTimeAsElapsed` operation still would use the default `return` tag.

The point of interest is that various annotations are available to determine what the generated WSDL document will look like. Such annotations will be introduced in small doses. It is easiest to minimize the use of annotations, using only the ones that serve some critical purpose. The `TimeServer` works exactly the same whether the `time_response` or the default `return` tag is used in the WSDL and in the SOAP response; hence, there is usually no need for programmer-generated code to use the `@WebResult` annotation.

Various commercial-grade WSDLs are available for generating Java support code. An example that will be introduced shortly uses a WSDL from Amazon's Associates Web Service, more popularly known as Amazon's E-Commerce service. However, the Amazon example first requires a closer look at WSDL structure. My plan is to keep the tedious details to a minimum. The goal of the next section is to move from basic WSDL structure to the unofficial but popular distinction between *wrapped* and *unwrapped* SOAP message bodies.

2.2. WSDL Structure

At a high level, a WSDL document is a contract between a service and its consumers. The contract provides such critical information as the service endpoint, the service operations, and the data types required for these operations. The service contract also indicates, in describing the messages exchanged in the service, the underlying service pattern, for instance, request/response or solicit/response. The outermost element (called the *document* or *root* element) in a WSDL is named `definitions` because the WSDL provides definitions grouped into the following sections:

- The `types` section, which is optional, provides data type definitions under some data type system such as XML Schema. A particular document that defines data types is an *XSD* (XML Schema Definition). The `types` section holds, points to, or imports an XSD. If the `types` section is empty, as in the case of the `TimeServer` service, then the service uses only simple data types such as `xsd:string` and `xsd:long`.

  Although the WSDL 2.0 specification allows for alternatives to XML Schema (see http://www.w3.org/TR/wsdl20-altschemalangs), XML Schema is the default and the dominant type system used in WSDLs. Accordingly, the following examples *assume* XML Schema unless otherwise noted.

- The `message` section defines the messages that implement the service. Messages are constructed from data types either defined in the immediately preceding section or, if the `types` section is empty, available as defaults. Further, the order of the messages indicates the service pattern. Recall that, for messages, the directional properties `in` and `out` are from the service's perspective: an `in` message is to the service, whereas an `out` message is from the service. Accordingly, the message order `in/out` indicates the request/response pattern, whereas the message order `out/in` indicates the solicit/response pattern. For the `TimeServer` service, there are four messages: a request and a response for the two operations, `getTimeAsString` and `getTimeAsElapsed`. The `in/out` order in each pair indicates a request/response pattern for the web service operations.

- The `portType` section presents the service as named operations, with each operation as one or more messages. Note that the operations are named after methods annotated as `@WebMethod`s, a point to be discussed

in detail shortly. A web service's `portType` is akin to a Java interface in presenting the service abstractly, that is, with no implementation details.

- The `binding` section is where the WSDL definitions go from the abstract to the concrete. A WSDL binding is akin to a Java implementation of an interface (that is, a WSDL `portType`). Like a Java implementation class, a WSDL `binding` provides important concrete details about the service. The `binding` section is the most complicated one because it must specify these implementation details of a service *defined* abstractly in the `portType` section:

  - The transport protocol to be used in sending and receiving the underlying SOAP messages. Either HTTP or *SMTP* (Simple Mail Transport Protocol) may be used for what is called the *application-layer* protocol; that is, the protocol for transporting the SOAP messages that implement the service. HTTP is by far the more popular choice. The WSDL for the `TimeServer` service contains this segment:

    ```
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http">
    ```

    The value of the `transport` attribute signals that the service's SOAP messages will be sent and received over HTTP, which is captured in the slogan *SOAP over HTTP*.

  - The `style` of the service, shown earlier as the value of the `style` attribute, takes either `rpc` or `document` as a value. The `document` style is the default, which explains why the SEI for the `TimeServer` service contains the annotation:

    ```
    @SOAPBinding(style = Style.RPC)
    ```

    This annotation forces the `style` attribute to have the value `rpc` in the Java-generated WSDL. The difference between the `rpc` and the `document` style will be clarified shortly.

  - The data format to be used in the SOAP messages. There are two choices, `literal` and `encoded`. These choices also will be clarified shortly.

- The `service` section specifies one or more endpoints at which the service's functionality, the sum of its operations, is available. In technical terms, the `service` section lists one or more `port` elements, where a `port` consists of a `portType` (interface) together with a corresponding `binding` (implementation). The term `port` *derives* from distributed systems. An application hosted at a particular network *address* (for instance, 127.0.0.1) is available to clients, local or remote, through a specified port. For example, the `TimeServer` application is available to clients at port 9876.

The tricky part of the binding section involves the possible combinations of the style and the use attributes. The next subsection looks more closely at the relationships between these attributes.

2.2.1. A Closer Look at WSDL Bindings

In the WSDL `binding` section, the `style` attribute has `rpc` and `document` as possible values, with `document` as the default. The `use` attribute has `literal` and `encoded` as possible values, with `literal` as the default. In theory, there are four possibilities, as shown in Table 2-1.

Table 2-1. Possible combinations of style and use

style	use
document	literal
document	encoded
rpc	literal
rpc	encoded

Of the four possible combinations listed in Table 2-1, only two occur regularly in contemporary SOAP-based web services: `document/literal` and `rpc/literal`. For one thing, the `encoded` use, though valid in a WSDL document, does not comply with the *WS-I* (Web Services-Interoperability) guidelines (see http://www.ws-i.org). As the name indicates, the WS-I initiative is meant to help software architects and developers produce web services that can interoperate seamlessly despite differences in platforms and programming languages.

Before going any further into the details, it will be helpful to have a sample WSDL for a `document`-style service, which then can be contrasted with the WSDL for the `rpc`-style `TimeServer` service. The `use` attribute will be clarified after the `style` attribute.

The `TimeServer` service can be changed to a `document`-style service in two quick steps. The first is to comment out the line:

```
@SOAPBinding(style = Style.RPC)
```

in the SEI source, `ch02.ts.TimeServer.java`, before recompiling. (The package has been changed from `ch01.ts` to `ch02.ts` to reflect that this is Chapter 2.) Commenting out this annotation means that the default style, `Style.DOCUMENT`, is in effect. The second step is to execute, in the working directory, the command:

```
% wsgen -keep -cp . ch02.ts.TimeServerImpl
```

The revised SEI and the corresponding SIB must be recompiled before the *wsgen* utility is used so that the `document`-style version is current. The *wsgen* utility then generates four source and four compiled files in the subdirectory *ch02/ts/jaxws*. These files provide the data types needed, in the `document`-style service, to produce the WSDL automatically. For example, among the files are source and compiled versions of the classes `GetTimeAsElapsed`, which is a request type, and `GetTimeAsElapsedResponse`, which is a response type. As expected, these two types support requests to and responses from the service operation `getTimeAsElapsed`. The *wsgen* utility generates comparable types for the `getTimeAsString` operation as well. The *wsgen* utility will be examined again later in this chapter.

The revised program can be published as before with the `TimeServerPublisher`, and the WSDL then is available at the published URL http://localhost:9876/ts?wsdl. The contrasting WSDLs now can be used to explain in detail how the `document` and `rpc` styles differ. There is one other important change to the `TimeServer`, which is reflected in the WSDL: the package name changes from `ch01.ts` to `ch02.ts`, a change that is reflected in a namespace throughout the WSDL.

2.2.2. Key Features of Document-Style Services

The `document` style indicates that a SOAP-based web service's underlying messages contain full XML documents; for instance, a company's product list as an XML document or a customer's order for some products from this list as another XML document. By contrast, the `rpc` style indicates that the underlying SOAP messages contain parameters in the request messages

and return values in the response messages. Following is a segment of the `rpc`-style WSDL for the original `TimeServer` service:

```
<types></types>
<message name="getTimeAsString"></message>
<message name="getTimeAsStringResponse">
  <part name="time_response" type="xsd:string"></part>
</message>
<message name="getTimeAsElapsed"></message>
<message name="getTimeAsElapsedResponse">
  <part name="time_response" type="xsd:long"></part>
</message>
```

The types section is empty because the service uses, as return values, only the simple types `xsd:string` and `xsd:long`. The simple types do not require a definition in the WSDL's `types` section. Further, the message names show their relationships to *the corresponding Java* `@WebMethod`s; in this case, the method `getTimeAsString` and the method `getTimeAsElapsed`. The response messages have a `part` subelement that gives the data type of the returned value, but because the request messages expect no arguments, the request messages do not need `part` subelements to describe parameters.

By contrast, here is the same WSDL segment for the `document`-style version of the `TimeServer` service:

```
<types>
  <xsd:schema>
    <xsd:import schemaLocation="http://localhost:9876/ts?xsd=1"
         namespace="http://ts.ch02/">
    </xsd:import>
  </xsd:schema>
</types>
<message name="getTimeAsString">
  <part element="tns:getTimeAsString" name="parameters"></part>
</message>
<message name="getTimeAsStringResponse">
  <part element="tns:getTimeAsStringResponse" name="parameters"></part>
</message>
<message name="getTimeAsElapsed">
  <part element="tns:getTimeAsElapsed" name="parameters"></part>
</message>
<message name="getTimeAsElapsedResponse">
  <part element="tns:getTimeAsElapsedResponse" name="parameters"></part>
</message>
```

The `types` now contains an `import` directive for the associated XSD document. The URL for the XSD is http://localhost:9876/ts?xsd=1. Example

2-6 is the XSD associated with the WSDL.

Example 2-6. The XSD for the TimeServer WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://ts.ch02/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://ts.ch02/" version="1.0">
  <xs:element name="getTimeAsElapsed"
              type="tns:getTimeAsElapsed">
  </xs:element>
  <xs:element name="getTimeAsElapsedResponse"
              type="tns:getTimeAsElapsedResponse">
  </xs:element>
  <xs:element name="getTimeAsString"
              type="tns:getTimeAsString">
  </xs:element>
  <xs:element name="getTimeAsStringResponse"
              type="tns:getTimeAsStringResponse">
  </xs:element>
  <xs:complexType name="getTimeAsString"></xs:complexType>
  <xs:complexType name="getTimeAsStringResponse">
    <xs:sequence>
      <xs:element name="return"
                  type="xs:string" minOccurs="0">
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="getTimeAsElapsed"></xs:complexType>
  <xs:complexType name="getTimeAsElapsedResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:long"></xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The XSD document defines four complex types whose names (for instance, the class `getTimeAsElapsedResponse`) are the names of corresponding messages in the `message` section. Under the `rpc`-style, the messages carry the names of the `@WebMethods`; under the `document`-style, there is an added level of complexity in that the messages carry the names of XSD types defined in the WSDL's `types` section.

The request/response pattern in a web service is possible under either the `document` or the `rpc` style, although the style named *rpc* obviously

underscores this pattern. Under the `rpc` style, the messages are named but not explicitly typed; under the `document` style, the messages are explicitly typed in an XSD document.

The `document` style deserves to be the default. This style can support services with rich, explicitly defined data types because the service's WSDL can define the required types in an XSD document. Further, the `document` style can support any service pattern, including request/response. Indeed, the SOAP messages exchanged in the `TimeServer` application look the same regardless of whether the `style` is `rpc` or `document`. From an architectural perspective, the `document` style is the simpler of the two in that the body of a SOAP message is a self-contained, precisely defined document. The `rpc` style requires messages with the names of the associated operations (in Java, the `@WebMethod`s) with parameters as subelements. From a developer perspective, however, the `rpc` style is the simpler of the two because the *wsgen* utility is not needed to generate Java types that correspond to XML Schema types. (The current Metro release generates the *wsgen* artifacts automatically. The details follow shortly.)

Finally, the `use` attribute determines how the service's data types are to be encoded and decoded. The WSDL service contract has to specify how data types used in the *implementation* language (for instance, Java) are serialized to WSDL-compliant types (by default, XML Schema types). On the client side, the WSDL-compliant types then must be deserialized into client-language types (for instance, C or Ruby types). The setting:

```
use = 'literal'
```

means that the service's type definitions *literally* follow the WSDL document's XML Schema. By contrast, the setting:

```
use = 'encoded'
```

means that the service's type definitions come from encoding rules, typically the encoding rules in the SOAP 1.1 specification. As noted earlier, the `document/encoded` and `rpc/encoded` combinations are not WS-I compliant. The real choices are, therefore, `document/literal` and `rpc/literal`.

2.2.3. Validating a SOAP Message Against a WSDL's XML Schema

One more point about `rpc`-style versus `document`-style bindings needs to be made. On the receiving end, a `document`-style SOAP message can be validated straightforwardly against the associated XML Schema. In `rpc`-style, by contrast, the validation is trickier precisely because there is no associated XML Schema. Example 2-7 is a short Java program that validates an arbitrary XML document against an arbitrary XSD document.

Example 2-7. A short program to validate an XML document

```
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Schema;
import javax.xml.XMLConstants;
import javax.xml.validation.Validator;

class ValidateXML {
    public static void main(String[ ] args) {
       if (args.length != 2) {
          String msg = "\nUsage: java ValidateXML XMLfile XSDfile";
          System.err.println(msg);
          return;
       }
       try {
          // Read and validate the XML Schema (XSD document).
          final String schema_uri = XMLConstants.W3C_XML_SCHEMA_NS_URI;
          SchemaFactory factory = SchemaFactory.newInstance(schema_uri);
          Schema schema = factory.newSchema(new StreamSource(args[1]));
          // Validate the XML document against the XML Schema.
          Validator val = schema.newValidator();
          val.validate(new StreamSource(args[0]));
       }
       // Return on any validation error.
       catch(Exception e) {
          System.err.println(e);
          return;
       }
       System.out.println(args[0] + " validated against " + args[1]);
    }
}
```

Here is the extracted body of a SOAP response to the `document`-style `TimeServer` service:

```
<ns1:getTimeAsElapsedResponse xmlns:ns1="http://ts.ch02/">
   <return>1208229395922</return>
</ns1:getTimeAsElapsedResponse>
```

The body has been edited slightly by the addition of the namespace declaration `xmlns:ns1="http://ts.ch02/`, which sets `ns1` as an alias or proxy for the namespace URI `http://ts.ch02/`. This URI does occur in the SOAP message but in the element `SOAP::Envelope` rather than in the `ns1:getTimeAsElapsedResponse` subelement. For a review of the full XSD, see Example 2-6. To keep the processing simple, the response body and the XSD are in the local files *body.xml* and *ts.xsd*, respectively. The command:

```
% java ValidateXML body.xml ts.xsd
```

validates the response's body against the XSD. If the return element in the body were changed to, say, `foo bar`, the attempted validation would produce the error message:

```
'foo bar' is not a valid value for 'integer'
```

thereby indicating that the SOAP message did not conform to the associated WSDL's XSD document.

2.2.4. The Wrapped and Unwrapped Document Styles

The `document` style is the default under the WS-I Basic profile for web services interoperability (see http://www.ws-i.org/Profiles/BasicProfile-1.1.html). This default style provides, through the XSD in the WSDL's `types` section, an explicit and precise definition of the data types in the service's underlying SOAP messages. The `document` style thereby promotes web service interoperability because a service client can determine precisely which data types are involved in the service and how the document contained in the body of an underlying SOAP message should be structured. However, the `rpc` style still has appeal in that the web service's operations have names linked directly to the underlying implementations; for example, to Java `@WebMethod`s. For instance, the `TimeServer` service in the `rpc` style has a `@WebMethod` with the name `getTimeAsString` whose WSDL counterparts are the request message named `getTimeAsString` and the response message named `getTimeAsStringResponse`. The `rpc` style is programmer-friendly.

The gist of the wrapped convention, which is unofficial but widely followed, is to give a `document`-style service the look and feel of an `rpc`-style service.

The wrapped convention seeks to combine the benefits of `document` and `rpc` styles.

To begin, Example 2-8 is an example of an unwrapped SOAP envelope, and Example 2-9 is an example of a wrapped SOAP envelope.

Example 2-8. Unwrapped document style

```
<?xml version="1.0" ?>
<!-- Unwrapped document style -->
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
     <num1 xmlns:ans="http://ts.ch01/">27</num1>
     <num2 xmlns:ans="http://ts.ch01/">94</num1>
  </soapenv:Body>
</soapenv:Envelope>
```

Example 2-9. Wrapped document style

```
<?xml version="1.0" ?>
<!-- Wrapped document style -->
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
     <addNums xmlns:ans="http://ts.ch01/">
       <num1>27</num1>
       <num2>94</num1>
     </addNums>
  </soapenv:Body>
</soapenv:Envelope>
```

The body of the unwrapped SOAP request envelope has two elements, named `num1` and `num2`. These are numbers to be added. The SOAP body does *not* contain the name of the service operation that is to perform the addition and send the sum as a response. By contrast, the body of a wrapped SOAP request envelope has a single element named `addNums`, the name of the requested service operation, and two subelements, each holding a number to be added. The wrapped version makes the service operation explicit. The arguments for the operation are nested in an intuitive manner; that is, as *subelements* within the operation element `addNums`.

Guidelines for the wrapped `document` convention are straightforward. Here is

a summary of the guidelines:

- The SOAP envelope's body should have only one part, that is, it should contain a single XML element with however many XML subelements are required. For example, even if a service operation expects arguments and returns a value, the parameters and return value do not occur as standalone XML elements in the SOAP body but, rather, as XML subelements within the main element. Example 2-9 *illustrates* with the `addNums` element as the single XML element in the SOAP body and the pair `num1` and `num2` as XML subelements.

- The relationship between the WSDL's XSD and the single XML element in the SOAP body is well defined. The `document`-style version of the `TimeServer` can be used to illustrate. In the XSD, there are four XSD `complexType` elements, each of which defines a data type. For example, there is a `complexType` with the name `getTimeAsString` and another with the name `getTimeAsStringResponse`. These definitions occur in roughly the bottom half of the XSD. In the top half are XML `element` definitions, each of which has a name and a type attribute. The `complexType`s also have names, which are coordinated with the `element` names. For example, the `complexType` named `getTimeAsString` is matched with an element of the same name. Here is a segment of the XSD that shows the name coordination:

```
<xs:element name="getTimeAsString"
            type="tns:getTimeAsString">
</xs:element>
<xs:element name="getTimeAsStringResponse"
            type="tns:getTimeAsStringResponse">
</xs:element>
...
<xs:complexType name="getTimeAsString"></xs:complexType>
<xs:complexType name="getTimeAsStringResponse">
    <xs:sequence>
      <xs:element name="return"
                  type="xs:string" minOccurs="0">
      </xs:element>
    </xs:sequence>
</xs:complexType>
```

  Further, each `complexType` is either empty (for instance, `getTimeAsString`) or contains an `xs:sequence` (for instance, `getTimeAsStringResponse`, which has an `xs:sequence` of one XML element). The `xs:sequence` contains typed arguments and typed returned values. The `TimeServer` example is quite simple in that the requests contain no arguments and the responses

contain just one return value. Nonetheless, this segment of XSD shows the structure for the general case. For instance, if the `getTimeAsStringResponse` had several return values, then each would occur as an XML subelement within the `xs:sequence`. Finally, note that every XML element in this XSD segment (and, indeed, in the full XSD) is named and typed.

- The XML `element`s in the XSD serve as the *wrappers* for the SOAP message body. For the `ch01.ts.TimeServer`, a sample wrapped `document` is:

```
<?xml version="1.0" ?>
<soapenv:Envelope
     xmlns:soapenv="http://schemas.xmlsoap.or g/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
    <ans:getTimeAsElapsedResponse xmlns:ans="http://ts.ch01/">
      <return>1205030105192</return>
    </ans:getTimeAsElapsedResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

  This is the same kind of SOAP body generated for an `rpc`-style service, which is precisely the point. The difference, again, is that the wrapped `document`-style service, unlike its `rpc`-style counterpart, includes explicit type and format information in an XSD from the WSDL's `types` section.

- The request wrapper has the same name as the service operation (for instance, `addNums` in Example 2-9), and the response wrapper should be the request wrapper's name with `Response` appended (for instance, `addNumsResponse`).

- The WSDL `portType` section now has named operations (e.g., `getTimeAsString`) whose messages are *typed*. For instance, the input (request) message `getTimeAsString` has the type `tns:getTimeAsString`, which is defined as one of the four `complexType`s in the WSDL's XSD. For the document-style version of the service, here is the `portType` segment:

```
<portType name="TimeServer">
  <operation name="getTimeAsString">
    <input message="tns:getTimeAsString"></input>
    <output message="tns:getTimeAsStringResponse"></output>
  </operation>
  <operation name="getTimeAsElapsed">
    <input message="tns:getTimeAsElapsed"></input>
    <output message="tns:getTimeAsElapsedResponse"></output>
  </operation>
```

```
    </portType>
```

The wrapped `document` convention, despite its unofficial status, has become prevalent. JWS supports the convention. By default, a Java SOAP-based web service is wrapped *doc/lit*, that is, wrapped `document` style with `literal` encoding.

This excursion into the details of WSDL `binding` section will be helpful in the next example, which uses the *wsimport* utility to generate client-support code against Amazon's E-Commerce web service. The contrast between wrapped and unwrapped is helpful in understanding the client-support code.

2.3. Amazon's E-Commerce Web Service

The section title has the popular and formerly official name for one of the web services that Amazon hosts. The official name is now *Amazon Associates Web Service*. The service in question is accessible as SOAP-based or REST-style. Th service is free of charge, but it does require registration at http://affiliate-program.amazon.com/gp/associates/join. For the examples in this section, an Amazon *access key* (as opposed to the *secret access key* used to generate an authentication token) is required.

Amazon's E-Commerce service replicates the interactive experience at the Amazon website. For example, the service supports searching for items, biddi on items and putting items up for bid, creating a shopping cart and filling it wi items, and so on. The two sample clients illustrate item search.

This section examines two Java clients against the Amazon E-Commerce servi Each client is generated with Java support code from the *wsimport* utility introduced earlier. The difference between the two clients refines the distincti between the wrapped and unwrapped conventions.

2.3.1. An E-Commerce Client in Wrapped Style

The Java support code for the client can be generated with the command:

```
% wsimport -keep -p awsClient \
http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl
```

Recall that the `-p awsClient` part of the command generates a package (and, therefore, a subdirectory) named *awsClient*.

The source code for the first Amazon client, `AmazonClientW`, resides in the worki directory; that is, the parent directory of *awsClient*. Example 2-10 is the application code, which searches for books about quantum gravity.

Example 2-10. An E-Commerce Java client in wrapped style

```
import awsClient.AWSECommerceService;
import awsClient.AWSECommerceServicePortType;
import awsClient.ItemSearchRequest;
import awsClient.ItemSearch;
import awsClient.Items;
import awsClient.Item;
```

```
import awsClient.OperationRequest;
import awsClient.SearchResultsMap;
import javax.xml.ws.Holder;
import java.util.List;
import java.util.ArrayList;

class AmazonClientW { // W is for Wrapped style
    public static void main(String[ ] args) {
      if (args.length < 1) {
        System.err.println("Usage: java AmazonClientW <access key>");
        return;
      }
      final String access_key = args[0];

      // Construct a service object to get the port object.
      AWSECommerceService service = new AWSECommerceService();
      AWSECommerceServicePortType port = service.getAWSECommerceServicePort();

      // Construct an empty request object and then add details.
      ItemSearchRequest request = new ItemSearchRequest();
      request.setSearchIndex("Books");
      request.setKeywords("quantum gravity");

      ItemSearch search = new ItemSearch();
      search.getRequest().add(request);
      search.setAWSAccessKeyId(access_key);

      Holder<OperationRequest> operation_request = null;
      Holder<List<Items>> items = new Holder<List<Items>>();

      port.itemSearch(search.getMarketplaceDomain(),
                      search.getAWSAccessKeyId(),
                      search.getSubscriptionId(),
                      search.getAssociateTag(),
                      search.getXMLEscaping(),
                      search.getValidate(),
                      search.getShared(),
                      search.getRequest(),
                      operation_request,
                      items);

      // Unpack the response to print the book titles.
      Items retval = items.value.get(0);       // first and only Items element
      List<Item> item_list = retval.getItem(); // list of Item subelements
      for (Item item : item_list)              // each Item in the list
        System.out.println(item.getItemAttributes().getTitle());
    }
}
```

The code is compiled and executed in the usual way but requires your access k
(a string such as 1A67QRNF7AGRQ1XXMJ07) as a command-line argument. C
sample run, the output for an item search among books on the string `quantum`

`gravity` was:

```
The Trouble With Physics
The Final Theory: Rethinking Our Scientific Legacy
Three Roads to Quantum Gravity
Keeping It Real (Quantum Gravity, Book 1)
Selling Out (Quantum Gravity, Book 2)
Mr Tompkins in Paperback
Head First Physics
Introduction to Quantum Effects in Gravity
The Large, the Small and the Human Mind
Feynman Lectures on Gravitation
```

The `AmazonClientW` client is not intuitive. Indeed, the code uses relatively obscu types such as `Holder`. The `itemSearch` method, which does the actual search, tak 10 arguments, the last of which, named `items` in this example, holds the servi response. This response needs to be unpacked in order to get a list of the boo titles returned from the search. The next client is far simpler. Before looking a the simpler client, however, it will be instructive to consider what makes the f client so tricky.

In the `binding` section of the WSDL for the E-Commerce service, the `style` is se the default, `document`, and the encoding is likewise set to the default, `literal`. Further, the wrapped convention is in play, as this segment from the XSD illustrates:

```
<xs:element name="ItemSearch">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="MarketplaceDomain"
                  type="xs:string" minOccurs="0"/>
      <xs:element name="AWSAccessKeyId"
                  type="xs:string" minOccurs="0"/>
      <xs:element name="SubscriptionId"
                  type="xs:string" minOccurs="0"/>
      <xs:element name="AssociateTag"
                  type="xs:string" minOccurs="0"/>
      <xs:element name="XMLEscaping"
                  type="xs:string" minOccurs="0"/>
      <xs:element name="Validate"
                  type="xs:string" minOccurs="0"/>
      <xs:element name="Shared"
                  type="tns:ItemSearchRequest" minOccurs="0"/>
      <xs:element name="Request"
                  type="tns:ItemSearchRequest" minOccurs="0" maxOccurs="unbound
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

This XSD segment defines the `ItemSearch` element, which is the wrapper type i
body of a SOAP request. Here is a segment from the XSD's `message` section, wh
shows that the request message is the type defined above:

```
<message name="ItemSearchRequestMsg">
   <part name="body" element="tns:ItemSearch"/>
</message>
```

For the service response to an `ItemSearch`, the wrapper type defined in the XSI

```
<xs:element name="ItemSearchResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="tns:OperationRequest" minOccurs="0"/>
      <xs:element ref="tns:Items" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The impact of these WSDL definitions is evident in the SOAP request message
from a client and the SOAP response message from the server. Example 2-11
shows a SOAP request from a sample run.

Example 2-11. A SOAP request against Amazon's E-Commerce service

```
<?xml version="1.0" ?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns1="http://webservices.amazon.com/AWSECommerceService/2008-03-03">
   <soapenv:Body>
     <ns1:ItemSearch>
       <ns1:AWSAccessKeyId>...</ns1:AWSAccessKeyId>
       <ns1:Request>
          <ns1:Keywords>quantum gravity</ns1:Keywords>
          <ns1:SearchIndex>Books</ns1:SearchIndex>
       </ns1:Request>
     </ns1:ItemSearch>
   </soapenv:Body>
</soapenv:Envelope>
```

The `ItemSearch` wrapper is the single XML element in the SOAP body and this element has two subelements, one with the name `ns1:AWSAccessKeyId` and the o with the name `ns1:Request`, whose subelements specify the search string (in th case, quantum gravity) and the search category (in this case, `Books`).

Here is part of the SOAP response for the request above:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ItemSearchResponse
       xmlns="http://webservices.amazon.com/AWSECommerceService/2008-03-03">
      <OperationRequest>
        <HTTPHeaders>
          <Header Name="UserAgent" Value="Java/1.6.0"></Header>
        </HTTPHeaders>
        <RequestId>0040N1YEKV0CRCT2B5PR</RequestId>
        <Arguments>
          <Argument Name="Service" Value="AWSECommerceService"></Argument>
        </Arguments>
        <RequestProcessingTime>0.0566580295562744</RequestProcessingTime>
      </OperationRequest>
      <Items>
        <Request>
          <IsValid>True</IsValid>
          <ItemSearchRequest>
            <Keywords>quantum gravity</Keywords>
            <SearchIndex>Books</SearchIndex>
          </ItemSearchRequest>
        </Request>
        <TotalResults>207</TotalResults>
        <TotalPages>21</TotalPages>
        <Item>
          <ASIN>061891868X</ASIN>
          <DetailPageURL>http://www.amazon.com/gp/redirect.html...
          </DetailPageURL>
          <ItemAttributes>
            <Author>Lee Smolin</Author>
            <Manufacturer>Mariner Books</Manufacturer>
            <ProductGroup>Book</ProductGroup>
            <Title>The Trouble With Physics</Title>
          </ItemAttributes>
        </Item>
        ...
      </Items>
    </ItemSearchResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP body now consists of a single element named `ItemSearchResponse`, wh is defined as a type in the service's WSDL. This element contains various subelements, the most interesting of which is named `Items`. The `Items` subelem contains multiple `Item` subelements, one apiece for a book on quantum gravity Only one `Item` is shown, but this is enough to see the structure of the respons document. The code near the end of the `AmazonClientW` reflects the structure of SOAP response: first the `Items` element is extracted, then a list of `Item` subelements, each of which contains a book's title as an XML attribute.

Now we can look at a *wsimport*-generated artifact for the E-Commerce service particular at the annotations on the `itemSearch` method with its 10 arguments. method is declared in the `AWSECommerceServicePortType` interface, which declares single `@WebMethod` for each of the E-Commerce service's operations. Example 2- shows the declaration of interest.

Example 2-12. Java code generated from Amazon's E-Commerce WSDI

```
@WebMethod(operationName = "ItemSearch",
           action = "http://soap.amazon.com")
@RequestWrapper(localName = "ItemSearch",
                targetNamespace =
                 "http://webservices.amazon.com/AWSECommerceService/2008-03-03"
                className = "awsClient.ItemSearch")
@ResponseWrapper(localName = "ItemSearchResponse",
                 targetNamespace =
                  "http://webservices.amazon.com/AWSECommerceService/2008-03-03
                 className = "awsClient.ItemSearchResponse")
public void itemSearch(
  @WebParam(name = "MarketplaceDomain",
            targetNamespace =
             "http://webservices.amazon.com/AWSECommerceService/2008-03-03")
  String marketplaceDomain,
  @WebParam(name = "AWSAccessKeyId",
            targetNamespace =
             "http://webservices.amazon.com/AWSECommerceService/2008-03-03")
  String awsAccessKeyId,
  ...
  ItemSearchRequest shared,
  @WebParam(name = "Request",
            targetNamespace =
             "http://webservices.amazon.com/AWSECommerceService/2008-03-03")
  List<ItemSearchRequest> request,
  @WebParam(name = "OperationRequest",
            targetNamespace =
             "http://webservices.amazon.com/AWSECommerceService/2008-03-03",
```

```
            mode = WebParam.Mode.OUT)
  Holder<OperationRequest> operationRequest,
  @WebParam(name = "Items",
            targetNamespace =
             "http://webservices.amazon.com/AWSECommerceService/2008-03-03",
            mode = WebParam.Mode.OUT)
  Holder<List<Items>> items);
```

The last two parameters, named `operationRequest` and `items`, are described as `WebParam.Mode.OUT` to signal that they represent return values from the E-Comm service to the requester. An `OUT` parameter is returned in a Java `Holder` object. method `itemSearch` thus reflects the XSD request and response types from the WSDL. The XSD type `ItemSearch`, which is the request wrapper, has eight subelements such as `MarketplaceDomain`, `AWSAccessKeyId`, and `ItemSearchRequest`. E of these *subelements* occurs as a parameter in the `itemSearch` method. The XS type `ItemSearchResponse` has two subelements, named `OperationRequest` and `Item` which are the last two parameters (the `OUT` parameters) in the `itemSearch` meth The tricky part for the programmer, of course, is that `itemSearch` becomes hard invoke precisely because of the 10 arguments, especially because the last 2 arguments hold the return values.

Why does the wrapped style result in such a complicated client? Recall that th wrapped style is meant to give a `document`-style service the look and feel of an style service without giving up the advantages of `document` style. The wrapped `document` style requires a wrapper XML `element`, typically with the name of a we service operation such as `ItemSearch`, that has a typed XML `subelement` per operation parameter. In the case of the `itemSearch` operation in the E-Commer service, there are eight `in` or request parameters and two `out` or response parameters, including the critical `Items` response parameter. The XML subelem that represent the parameters occur in an `xs:sequence`, which means that each parameter is positional. For example, the response parameter `Items` must com last in the list of 10 parameters because the part `@ResponseWrapper` comes after `@RequestWrapper` and the `Items` parameter comes last in the `@ResponseWrapper`. Th upshot is that the wrapped style makes for a very tricky invocation of the `itemSearch` method. The wrapped style, without a workaround, may well produ irritated programmer. The next section presents a workaround.

2.3.2. An E-Commerce Client in Unwrapped Style

The client built from artifacts in the unwrapped style is simpler than the client built from artifacts in the wrapped style. However, artifacts for the simplified E

Commerce client are generated from the very same WSDL as the artifacts for more complicated, wrapped-style client. The underlying SOAP messages have same structure with either the complicated or the simplified client. Yet the clie differ significantly—the simplified client is far easier to code and to understanc Here is the call to `invokeSearch` in the simplified client:

```
ItemSearchResponse response = port.itemSearch(item_search);
```

The `itemSearch` method now takes one argument and returns a value, which is assigned to the object reference `response`. Unlike the complicated client, the simplified client has no `Holder` of a return value, which is an exotic and difficul construct. The invocation of `itemSearch` is familiar, identical in style to method in standalone applications. The full simplified client is in Example 2-13.

Example 2-13. An E-Commerce Java client in unwrapped style

```
import awsClient2.AWSECommerceService;
import awsClient2.AWSECommerceServicePortType;
import awsClient2.ItemSearchRequest;
import awsClient2.ItemSearchResponse;
import awsClient2.ItemSearch;
import awsClient2.Items;
import awsClient2.Item;
import java.util.List;

class AmazonClientU { // U is for Unwrapped style
    public static void main(String[ ] args) {
        // Usage
        if (args.length != 1) {
            System.err.println("Usage: java AmazonClientW <access key<");
            return;
        }
        final String access_key = args[0];

        // Create service and get portType reference.
        AWSECommerceService service = new AWSECommerceService();
        AWSECommerceServicePortType port =
          service.getAWSECommerceServicePort();

        // Create request.
        ItemSearchRequest request = new ItemSearchRequest();

        // Add details to request.
        request.setSearchIndex("Books");
        request.setKeywords("quantum gravity");
        ItemSearch item_search= new ItemSearch();
        item_search.setAWSAccessKeyId(access_key);
        item_search.getRequest().add(request);
```

```
        // Invoke service operation and get response.
        ItemSearchResponse response = port.itemSearch(item_search);

        List<Items> item_list = response.getItems();
        for (Items next : item_list)
           for (Item item : next.getItem())
              System.out.println(item.getItemAttributes().getTitle());
    }
}
```

Generating the simplified client with the *wsimport* is ironically more complicat than generating the complicated client. Here is the command, on three lines t enhance readability:

```
% wsimport -keep -p awsClient2 \
  http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl \
  -b custom.xml .
```

The `-b` flag at the end specifies a customized `jaxws:bindings` document, in this in the file *custom.xml*, that overrides *wsimport* defaults, in this case a `WrapperS` setting of `true`. Example 2-14 shows the document with the customized bindin information.

Example 2-14. A customized bindings document for wsimport

```
<jaxws:bindings
    wsdlLocation =
     "http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
  <jaxws:enableWrapperStyle>false</jaxws:enableWrapperStyle>
</jaxws:bindings>
```

The impact of the customized binding document is evident in the generated artifacts. For example, the segment of the `AWSECommerceServicePortType` artifact becomes:

```
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public interface AWSECommerceServicePortType {
...
  @WebMethod(operationName = "ItemSearch",
             action = "http://soap.amazon.com")
  @WebResult(name = "ItemSearchResponse",
             targetNamespace =
              "http://webservices.amazon.com/AWSECommerceService/2008-04-07",
             partName = "body")
  public ItemSearchResponse itemSearch(
      @WebParam(name = "ItemSearch",
                targetNamespace =
                 "http://webservices.amazon.com/AWSECommerceService/2008-04-07"
                partName = "body")
      ItemSearch body);
```

The *wsimport*-generated interface `AWSECommerceServicePortType` now has the annotation `@SOAPBinding.ParameterStyle.BARE`, where `BARE` is the alternative to `WRAPPED`. JWS names the attribute `parameterStyle` because the contrast between wrapped and unwrapped in `document`-style web services comes down to how parameters are represented in the SOAP body. In the unwrapped style, the parameters occur *bare*; that is, as a sequence of unwrapped XML subelements the SOAP body. In the wrapped style, the parameters occur as *wrapped* XML subelements of an XML element with the name of the service operation; and t wrapper XML element is the only direct subelement of the SOAP body.

What may be surprising is that the structure of the underlying SOAP message both the request and the response, remain unchanged. For instance, the requ message from the simplified client `AmazonClientU` is identical in structure to the request message from the complicated client `AmazonClientW`. Here is the body o SOAP envelope from a request that the simplified client generates:

```
<soapenv:Body>
  <ns1:ItemSearch>
    <ns1:AWSAccessKeyId>...</ns1:AWSAccessKeyId>
    <ns1:Request>
      <ns1:Keywords>quantum gravity</ns1:Keywords>
      <ns1:SearchIndex>Books</ns1:SearchIndex>
    </ns1:Request>
</ns1:ItemSearch>
</soapenv:Body>
```

The SOAP body contains a single wrapper element, `ns1:ItemSearch`, with subelements for the access key identifier and the request details. The complic

wrapped-style client generates requests with the identical structure.

The key difference is in the Java client code, of course. The simplified client, w the unwrapped `parameterStyle`, calls `invokeSearch` with one argument of type `ItemSearch` and expects a single response of type `ItemSearchResponse`. So the `parameterStyle` with a value of `BARE` eliminates the complicated call to `invokeSea` with 10 arguments, 8 of which are arguments bound to the subelements in th `@RequestWrapper`, and 2 of which are arguments bound to subelements in the `@ResponseWrapper`.

The E-Commerce example shows that the wrapped `document`-style, despite its advantages, can complicate the programming of a client. In that example, the simplified client with the `BARE` or unwrapped `parameterStyle` is a workaround.

The underlying WSDL for the E-Commerce example could be simplified, which would make clients against the service easier to code. Among SOAP-based we services available commercially, it is not unusual to find complicated WSDLs t in turn complicate the clients written against the service.

2.3.3. Tradeoffs Between the RPC and Document Styles

JWS still supports both `rpc` and `document` styles, with `document` as the default; fo both styles, only `literal` encoding is supported in compliance with the WS-I Ba Profile. The issue of `rpc` versus `document` often is touted as a *freedom of choice* issue. Nonetheless, it is clear that `document` style, especially in the wrapped fla is rapidly gaining mind share. This subsection briefly reviews some tradeoffs between the two styles.

As in any tradeoff scenario, the pros and cons need to be read with a critical e especially because a particular point may be cited as both a pro and a con. On familiar complaint against the `rpc` style is that it imposes the request/response pattern on the service. However, this pattern remains the dominant one in rea world, SOAP-based web services, and there are obviously situations (for instan validating a credit card to be used in a purchase) in which the request/respons pattern is needed.

Here are some upsides of the `rpc` style:

- The automatically generated WSDL is relatively short and simple because there is no `types` section.
- Messages in the WSDL carry the names of the underlying web service operations, which are `@WebMethod`s in a Java-based service. The WSDL thus

a *what you see is what you get* style with respect to the service's operatio

- Message throughput may improve because the messages do not carry any type-encoding information.

Here are some downsides to the `rpc` style:

- The WSDL, with its empty `types` section, does not provide an XSD against which the body of a SOAP message can be validated.
- The service cannot use arbitrarily rich data types because there is no XSD define such types. The service is thus restricted to relatively simple types as integers, strings, dates, and arrays of such.
- This style, with its obvious link to the request/response pattern, encourag *tight coupling* between the service and the client. For example, the Java cl `ch01.ts.TimeClient` *blocks* on the call:

  ```
  port.getTimeAsString()
  ```

  until either the service responds or an exception is thrown. This same poi sometimes made by noting that the `rpc` style has an inherently *synchrono* opposed to *asynchronous* invocation idiom. The next section offers a workaround, which shows how JWS supports nonblocking clients under the request/response pattern.
- Java services written in this style may not be consumable in other frameworks, thus undermining interoperability. Further, long-term suppor within the web services community and from the WS-I group is doubtful.

Here are some upsides of the `document` style:

- The body of a SOAP message can be validated against the XSD in the `type` section of the WSDL.
- A service in this style can use arbitrarily rich data types, as the XML Sche language supports not only simple types such as integers, strings, and dat but also arbitrarily rich complex types.
- There is great flexibility in how the body of a SOAP message is structured long as the structure is clearly defined in an XSD.
- The *wrapped* convention provides a way to enjoy a key upside of the `rpc` s

—naming the body of a SOAP message after the corresponding service operation—without enduring the downsides of the `rpc` style.

Here are some downsides of the `document` style:

- In the unwrapped variant, the SOAP message does not carry the name of service operation, which can complicate the dispatching of messages to th appropriate program code.
- The wrapped variant adds a level of complexity, in particular at the API le Writing a client against a wrapped-`document` service can be challenging, as `AmazonClientW` example shows.
- The wrapped variant does not support overloaded service operations beca the XML wrapper `element` in the body of a SOAP message must have the na of *the service operation.* In effect, then, there can be only one operation f given `element` name.

2.3.4. An Asynchronous E-Commerce Client

As noted earlier, the original `TimeClient` blocks on calls against the `TimeServer` service operations. For example, the call:

```
port.getTimeAsString()
```

blocks until either a response from the web service occurs or an exception is thrown. The call to `getTimeAsString` is, therefore, known as a *blocking* or *synchronous* call. JWS also supports *nonblocking* or *asynchronous* clients agai web services.

Example 2-15 shows a client that makes an asynchronous call against the E-Commerce service.

Example 2-15. An asynchronous client against the E-Commerce servic

```
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

import awsClient3.AWSECommerceService;
import awsClient3.AWSECommerceServicePortType;
import awsClient3.ItemSearchRequest;
import awsClient3.ItemSearchResponse;
```

```
import awsClient3.ItemSearch;
import awsClient3.Items;
import awsClient3.Item;

import java.util.List;
import java.util.concurrent.ExecutionException;

class AmazonAsyncClient {
    public static void main(String[ ] args) {
        // Usage
        if (args.length != 1) {
            System.err.println("Usage: java AmazonAsyncClient <access key>");
            return;
        }
        final String access_key = args[0];

        // Create service and get portType reference.
        AWSECommerceService service = new AWSECommerceService();
        AWSECommerceServicePortType port = service.getAWSECommerceServicePort()

        // Create request.
        ItemSearchRequest request = new ItemSearchRequest();

        // Add details to request.
        request.setSearchIndex("Books");
        request.setKeywords("quantum gravity");
        ItemSearch item_search= new ItemSearch();
        item_search.setAWSAccessKeyId(access_key);
        item_search.getRequest().add(request);

        port.itemSearchAsync(item_search, new MyHandler());

        // In this case, just sleep to give the search process time.
        // In a production application, other useful tasks could be
        // performed and the application could run indefinitely.
        try {
           Thread.sleep(400);
        }
        catch(InterruptedException e) { System.err.println(e); }
    }

    // The handler class implements handleResponse, which executes
    // if and when there's a response.
    static class MyHandler implements AsyncHandler<ItemSearchResponse> {
        public void handleResponse(Response<ItemSearchResponse> future) {
          try {
             ItemSearchResponse response = future.get();
             List<Items> item_list = response.getItems();
             for (Items next : item_list)
               for (Item item : next.getItem())
                  System.out.println(item.getItemAttributes().getTitle());
          }
          catch(InterruptedException e) { System.err.println(e); }
          catch(ExecutionException e) { System.err.println(e); }
        }
    }
}
```

The nonblocking E-Commerce client uses artifacts generated by *wsimport*, aga
with a customized bindings file. Here is the file:

```
<jaxws:bindings
    wsdlLocation=
      "http://ecs.amazonaws.com/AWSECommerceService/AWSECommerceService.wsdl"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
  <jaxws:enableWrapperStyle>false</jaxws:enableWrapperStyle>
  <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
</jaxws:bindings>
```

with the `enableAsyncMapping` attribute set to `true`.

The nonblocking call can follow different styles. In the style shown here, the c
to `itemSearchAsync` takes two arguments: the first is an `ItemSearchRequest`, and t
second is a class that implements the `AsyncHandler` interface, which declares a
single method named `handleResponse`. The call is:

```
port.itemSearchAsync(item_search, new MyHandler());
```

If and when an `ItemSearchResponse` comes from the E-Commerce service, the
method `handleResponse` in the `MyHandler` class executes as a separate thread and
prints out the books' titles.

There is a version of `itemSearchAsync` that takes one argument, an
`ItemSearchRequest`. In this version the call also returns if and when the E-Comm
service sends a response, which then can be processed as in the other E-
Commerce clients. In this style, the application might start a separate thread
execute this code segment:

```
Response<ItemSearchResponse> res = port.itemSearchAsync(item_search);
try {
   ItemSearchResponse response = res.get();
   List<Items> item_list = response.getItems();
   for (Items next : item_list)
      for (Item item : next.getItem())
         System.out.println(item.getItemAttributes().getTitle());
```

```
}
catch(InterruptedException e) { System.err.println(e); }
catch(ExecutionException e) { System.err.println(e); }
```

JWS is flexible in supporting nonblocking as well as the default blocking client
the end, it is application logic that determines which type of client is suitable.

2.4. The wsgen Utility and JAX-B Artifacts

Any `document`-style service, wrapped or unwrapped, requires the kind of artifac that the *wsgen* utility produces. It is time to look again at this utility. A simple experiment underscores the role that the utility plays. To begin, the line:

```
@SOAPBinding(style = Style.RPC)
```

should be commented out or deleted from the SEI `ch01.ts.TimeServer`. With thi *annotation* gone, the web service becomes `document` style rather than `rpc` style After *recompiling* the altered SEI, try to publish the service with the command

```
% java ch01.ts.TimeServerPublisher
```

The resulting error message is:

```
Exception in thread "main" com.sun.xml.internal.ws.model.RuntimeModelerExceptio
runtime modeler error: Wrapper class ch01.ts.jaxws.GetTimeAsString is not found
```

The message is obscure in citing the immediate problem rather than the unde cause. The immediate problem is that the publisher cannot find the class `ch01.ts.jaxws.GetTimeAsString`. Indeed, at this point the package `ch01.ts.jaxws` t houses the class does not even exist. The publisher cannot generate the WSD because the publisher needs Java classes such as `GetTimeAsString` to do so. The *wsgen* utility produces the classes required to build the WSDL, classes known *wsgen* artifacts. The command:

```
% wsgen -keep -cp . ch01.ts.TimeServerImpl
```

produces the artifacts and, if necessary, the package `ch01.ts.jaxws` that houses these artifacts. In the `TimeServer` example, there are four messages altogether request and response messages for the `getTimeAsString` operation, and the req and response messages for the `getTimeAsElapsed` operation. The *wsgen* utility generates a Java *class*—hence, a Java data type—for each of these messages. these Java types that the publisher uses to generate the WSDL for a `document`-

service. So each of the Java data types is bound to an XML Schema type, whic serves as a type for one of the four messages involved in the service. Conside from the other direction, a `document`-style web service has *typed* messages. Th *wsgen* artifacts are the Java types from which the XML Schema types for the messages are derived.

A SOAP-based web service, in either `document` or `rpc` style, should be interoper a client application written in one language should be able to interact seamles with a service written in another despite any differences in data types betwee two languages. A shared type system such as XML Schema is, therefore, the k interoperability. The `document` style extends typing to the service messages; ar typed messages requires the explicit binding of Java and XML Schema types.

Any `document`-style service, wrapped or unwrapped, has a WSDL with an XSD i `types` section. (The term *in* is being used loosely here. The `types` section could import the XSD or link to the XSD.) The types in the WSDL's associated XSD b to types in a service-implementation or client language such as Java. The *wsg* utility, introduced earlier to change the `TimeServer` service from `rpc` to `document` style, generates Java types that bind to XML Schema types. Under the hood, t utility uses the packages *associated* with *JAX-B* (Java API for XML-Binding). In nutshell, JAX-B supports conversions between Java and XML types.

2.4.1. A JAX-B Example

Before looking again at the artifacts that *wsgen* generates, let's consider a JA example to get a better sense of what is going on in *wsgen*. Example 2-16 is t code for a programmer-defined Java type, the `Person` class, and Example 2-17 another programmer-defined Java type, the `Skier` class. Each class declaration begins with a single annotation for Java-to-XML binding. The `Person` class is annotated as an `@XmlType`, whereas the `Skier` class is annotated as an `@XmlRootElement`. At the design level, a `Skier` is also a `Person`; hence, the `Skier` c has a `Person` field.

Example 2-16. The Person class for a JAX-B example

```
import javax.xml.bind.annotation.XmlType;

@XmlType
public class Person  {
    // fields
    private String name;
    private int    age;
    private String gender;
```

```
    // constructors
    public Person() { }

    public Person(String name, int age, String gender){
        setName(name);
        setAge(age);
        setGender(gender);
    }

    // properties: name, age, gender
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return age;  }
    public void setAge(int age) { this.age = age; }

    public String getGender() { return gender; }
    public void setGender(String gender) { this.gender = gender; }
}
```

Example 2-17. The Skier class for a JAX-B example

```
import javax.xml.bind.annotation.XmlRootElement;
import java.util.Collection;

@XmlRootElement
public class Skier  {
    // fields
    private Person person;
    private String national_team;
    private Collection major_achievements;
    // constructors
    public Skier() { }
    public Skier (Person person,
                  String national_team,
                  Collection<String> major_achievements) {
        setPerson(person);
        setNationalTeam(national_team);
        setMajorAchievements(major_achievements);
    }
    // properties
    public Person getPerson() { return person; }
    public void setPerson (Person person) { this.person = person; }

    public String getNationalTeam() { return national_team; }
    public void setNationalTeam(String national_team) {
        this.national_team = national_team;
    }
```

```
    public Collection getMajorAchievements() { return major_achievements; }
    public void setMajorAchievements(Collection major_achievements) {
        this.major_achievements = major_achievements;
    }
}
```

The `@XmlType` and `@XmlRootElement` annotations direct the marshaling of `Skier` ob
where *marshaling* is the process of encoding an in-memory object (for exampl
`Skier`) as an XML document so that, for instance, the encoding can be sent acr
network to be *unmarshaled* or decoded at the other end. In common usage, th
distinction between *marshal* and *unmarshal* is very close and perhaps identica
the *serialize/deserialize* distinction. I use the distinctions interchangeably. JAX
supports the marshaling of in-memory Java objects to XML documents and the
unmarshaling of XML documents to in-memory Java objects.

The annotation `@XmlType`, applied in this example to the `Person` class, indicates
JAX-B should generate an XML Schema type from the Java type. The annotati
`@XmlRootElement`, applied in this example to the `Skier` class, indicates that JAX-B
should generate an XML *document* (outermost or root) element from the Java
class. Accordingly, the resulting XML in this example is a document whose
outermost element represents a skier; and the document has a nested elemer
that represents a person.

The `Marshal` application in Example 2-18 illustrates marshaling and unmarshal

Example 2-18. The Marshal application for a JAX-B example

```
import java.io.File;
import java.io.OutputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.IOException;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.JAXBException;
import java.util.List;
import java.util.ArrayList;

class Marshal {
    private static final String file_name = "bd.mar";
    public static void main(String[ ] args) {
        new Marshal().run_example();
```

```
    }

    private void run_example() {
        try {
            JAXBContext ctx = JAXBContext.newInstance(Skier.class);
            Marshaller m = ctx.createMarshaller();
            m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

            // Marshal a Skier object: 1st to stdout, 2nd to file
            Skier skier = createSkier();
            m.marshal(skier, System.out);

            FileOutputStream out = new FileOutputStream(file_name);
            m.marshal(skier, out);
            out.close();

            // Unmarshal as proof of concept
            Unmarshaller u = ctx.createUnmarshaller();
            Skier bd_clone = (Skier) u.unmarshal(new File(file_name));
            System.out.println();
            m.marshal(bd_clone, System.out);
        }
        catch(JAXBException e) { System.err.println(e); }
        catch(IOException e) { System.err.println(e); }
    }

    private Skier createSkier() {
        Person bd = new Person("Bjoern Daehlie", 41, "Male");
        List<String> list = new ArrayList<String>();
        list.add("12 Olympic Medals");
        list.add("9 World Championships");
        list.add("Winningest Winter Olympian");
        list.add("Greatest Nordic Skier");
        return new Skier(bd, "Norway", list);
    }
}
```

The application constructs a `Skier` object, including a `Person` object encapsulate
a `Skier` field, and sets the appropriate `Skier` and `Person` properties. A critical fea
of marshaling is that the process preserves an object's state in the encoding, v
an object's *state* comprises the values of its instance (that is, non-`static`) field
the case of a `Skier` object, the marshaling must preserve state information suc
the skier's major accomplishments together with `Person`-specific state informat
such as the skier's name and age. The `@XmlRootElement` annotation in the declar
of the `Skier` class directs the marshaling as follows: the `Skier` object is encoded
an XML document whose outermost (that is, *document*) element is named `skie`
The `@XmlType` annotation in the `Person` class directs the marshaling as follows: t
`skier` XML document has a `person` subelement, which in turn has subelements

the `name`, `age`, and `gender` properties of the marshaled skier.

By default, JAX-B marshaling follows standard Java and JavaBean naming conventions. For example, the Java class `Skier` becomes the XML tag named `sk` and the Java class `Person` becomes the XML tag named `person`. For both the `Ski` and the encapsulated `Person` objects, the JavaBean-style `get` methods are invo (e.g., `getNationalTeam` in `Skier` and `getAge` in `Person`) to populate the XML docum with state information about the skier.

It is possible to override, with annotations, JavaBean naming conventions in marshaling and unmarshaling classes. Following, for example, is a *wsgen*-generated artifact with the `get` and `set` methods that do not follow JavaBean naming conventions. The annotation of interest is in bold:

```
...
@XmlRootElement(name = "getTimeAsElapsedResponse", namespace = "http://ts.ch02/
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getTimeAsElapsedResponse", namespace = "http://ts.ch02/")
public class GetTimeAsElapsedResponse {
    @XmlElement(name = "return", namespace = "")
    private long _return;
    public long get_return() { return this._return; }

    public void set_return(long _return) { this._return = _return; }
}
```

The annotation in bold indicates that the field named `_return` will be marshale unmarshaled rather than a property defined with a `get`/`set` method pair that fo the usual JavaBean naming conventions.

Other annotation attributes can be set to override the default naming convent For example, if the annotation in the `Skier` class declaration is changed to:

```
@XmlRootElement(name = "NordicSkier")
```

then the resulting XML document begins:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<NordicSkier>
```

Once the `Skier` object has been constructed, the `Marshal` application marshals t
object to the standard output and, to set up unmarshaling, to a local file. Here
the XML document that results from marshaling the legendary Nordic skier Bj
Daehlie:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<skier>
    <majorAchievements
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xsi:type="xs:string">
      12 Olympic Medals
    </majorAchievements>
    <majorAchievements
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xsi:type="xs:string">
      9 World Championships
    </majorAchievements>
    <majorAchievements
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xsi:type="xs:string">
      Winningest Winter Olympian
    </majorAchievements>
    <majorAchievements
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xsi:type="xs:string">
      Greatest Nordic Skier
    </majorAchievements>
    <nationalTeam>Norway</nationalTeam>
    <person>
        <age>41</age>
        <gender>Male</gender>
        <name>Bjoern Daehlie</name>
    </person>
</skier>
```

The unmarshaling process constructs a `Skier`, with its encapsulated `Person` field
from the XML document. JAX-B unmarshaling requires that each class have a
no-argument constructor, which is used in the construction. After the object is
constructed, the appropriate `set` methods are invoked (for instance, `setNationa`
in `Skier` and `setAge` in `Person`) to restore the marshaled skier's state. The marsh
and unmarshaling processes are remarkably clear and straightforward at the c
level.

2.4.2. Marshaling and wsgen Artifacts

Now we can tie the *wsgen* utility and marshaling together. Recall that there w
two steps to changing the `TimeServer` application from `rpc` to `document` style. Firs
annotation:

```
@SOAPBinding(style = Style.RPC)
```

is commented out in the SEI, `ch01.ts.TimeServer`. Second, the *wsgen* utility is
executed in the working directory against the `ch01.ts.TimeServerImpl` class:

```
% wsgen -keep -cp . ch01.ts.TimeServerImpl
```

The *wsgen* invocation generates two source and two compiled files in the
automatically created subdirectory *ch01/ts/jaxws*. Example 2-19 shows the *ws*
artifact, the class named `GetTimeAsElapsedResponse`, with the comments remove

Example 2-19. A Java class generated with wsgen

```
package ch01.ts.jaxws;

import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "getTimeAsElapsedResponse", namespace = "http://ts.ch01/
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "getTimeAsElapsedResponse", namespace = "http://ts.ch01/")
public class GetTimeAsElapsedResponse {
    @XmlElement(name = "return", namespace = "")
    private long _return;
    public long get_return() { return this._return; }
    public void set_return(long _return) { this._return = _return; }
}
```

Of particular interest is the `@XmlType` annotation applied to the class. The anno
sets the `name` attribute to `getTimeAsElapsedResponse`, the name of a SOAP messag
returned from the web service to the client on a call to the `getTimeAsElapsed`

operation. The `@XmlType` annotation means that such SOAP response messages *typed*; that is, that they satisfy an XML Schema. In a `document`-style service, a message is typed.

Example 2-20 shows a modified version of the `Marshal` application, renamed `MarshalGTER` to signal that the revised application is to be run against the *wsgen-generated* class named `GetTimeAsElapsedResponse`.

Example 2-20. Code to illustrate the link between wsgen and JAX-B

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.JAXBException;

import ch01.ts.jaxws.GetTimeAsElapsedResponse;

class MarshalGTER {
    private static final String file_name = "gter.mar";

    public static void main(String[ ] args) {
        new MarshalGTER().run_example();
    }

    private void run_example() {
        try {
            JAXBContext ctx =
                JAXBContext.newInstance(GetTimeAsElapsedResponse.class);
            Marshaller m = ctx.createMarshaller();
            m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

            GetTimeAsElapsedResponse tr = new GetTimeAsElapsedResponse();
            tr.set_return(new java.util.Date().getTime());

            m.marshal(tr, System.out);
        }
        catch(JAXBException e) { System.err.println(e); }
    }
}
```

The marshaled XML document on a sample run was:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:getTimeAsElapsedResponse xmlns:ns2="http://ts.ch01/">
    <return>1209174518855</return>
</ns2:getTimeAsElapsedResponse>
```

For reference, here is the response message from the `document`-style version o
`TimeServer` service:

```
<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns1="http://ts.ch01/">
  <soapenv:Body>
    <ns1:getTimeAsElapsedResponse>
      <return>1209181713849</return>
    </ns1:getTimeAsElapsedResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

There are some incidental differences between the body of the SOAP message
the output of the `MarshalGTER` application. For instance, the namespace prefix i
in the SOAP message but `ns2` in the marshaled XML document. Note, however
the all-important namespace URI is the same in both: `http://ts.ch01`. In the S
message, the namespace prefix is defined in the `Envelope` element rather than
the element `getTimeAsElapsedResponse`. Of interest here is that the *wsgen*-gener
artifacts such as the `GetTimeAsElapsedResponse` class provide the annotated type
the underlying SOAP libraries can use to marshal Java objects of some type in
XML documents and to unmarshal such documents into Java objects of the
appropriate type.

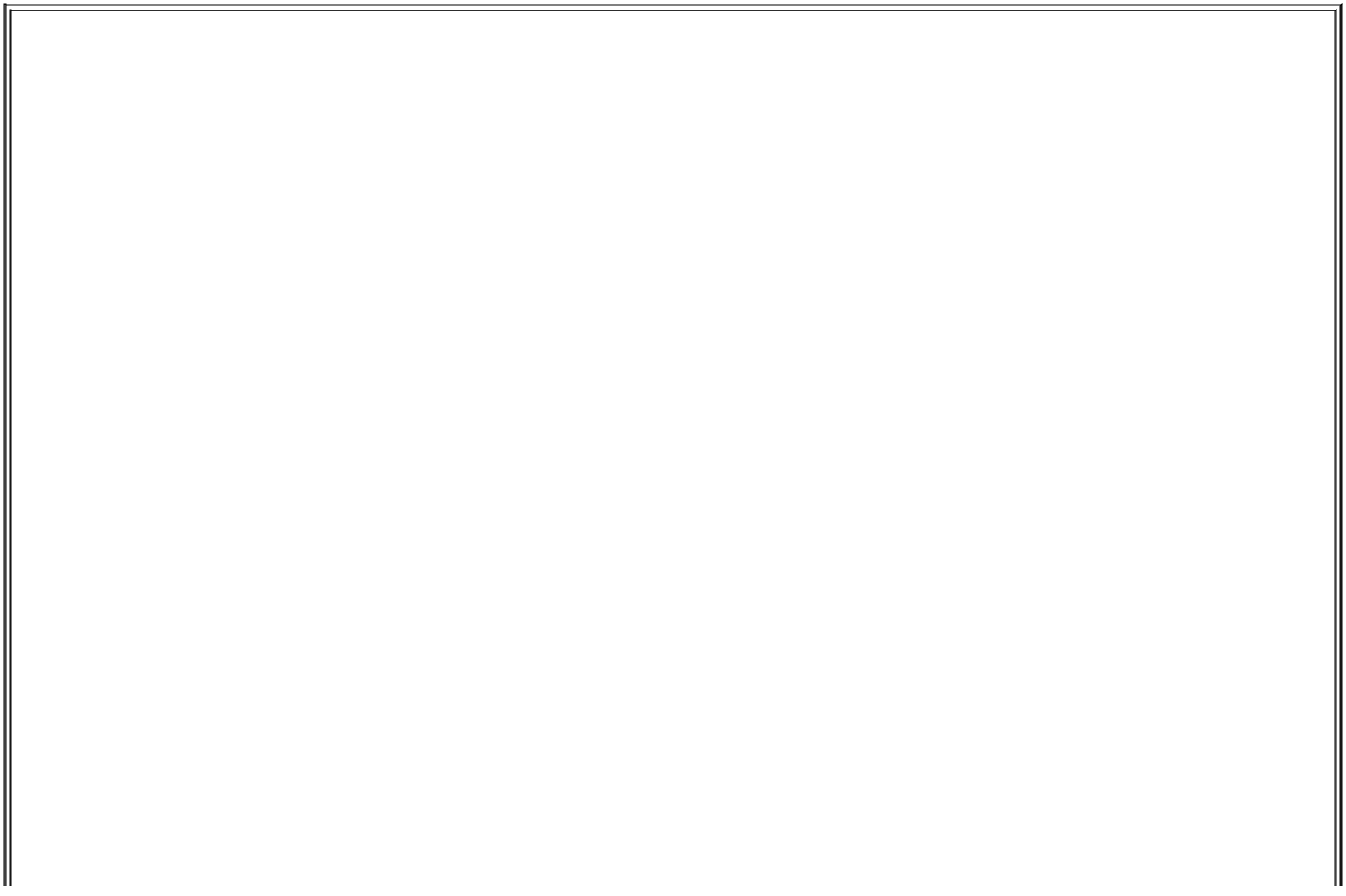
2.4.3. An Overview of Java Types and XML Schema Types

Java's primitive types such as `int` and `byte` bind to similarly named XML Schem
types, in this case `xsd:int` and `xsd:byte`, respectively. The `java.lang.String` type
binds to `xsd:string`, and the `java.util.Calendar` typebinds to each of `xsd:date`,
`xsd:time`, and `xsd:dateTime`. The XML Schema type `xsd:decimal` binds to the Java
`BigDecimal`. Not all bindings are obvious and the same Java *type*—for example,
may match up with several XSD types, for instance, `xsd:int` and `xsd:unsignedSh`
Here is the reason for the apparent mismatch between a Java `int` and an XSD
`xsd:unsignedShort`. Java technically does not have unsigned integer types. (You
could argue, of course, that a Java `char` is really an unsigned 16-bit integer.) T
maximam value of the Java 16-bit short integer is 32,767, but the maximum
of the `xsd:unsignedShort` integer is 65,535, a value within the range of a 32-bit
`int`.

What JAX-B brings to the table is a framework for binding arbitrarily rich Java

types, with the `Skier` class as but a hint of the possibilities, to XML types. Insta of these Java types can be marshaled to XML document instances, which in tu can be unmarshaled to instances of either Java types or types in some other language.

How the *wsgen* utility and the JAX-B packages interact in JWS now can be summarized. A Java web service in `document` rather than `rpc` style has a nonem `types` section in its WSDL. This section defines, in the XML Schema language, t types required for the web service. The *wsgen* utility generates, from the SIB, classes that are counterparts of XSD types. These *wsgen* artifacts are availabl the underlying JWS libraries, in particular for the JAX-B family of packages, to convert (marshal) instances of Java types (that is, Java in-memory objects) in XML instances of XML types (that is, into XML document instances that satisfy XML Schema document). The inverse operation is used to convert (unmarshal XML document instance to an in-memory object, an object of a Java type or a comparable type in some other language. The *wsgen* utility thus produces the artifacts that support interoperability for a Java-based web service. The JAX-B libraries provide the under-the-hood support to *convert between* Java and XSI types. For the most part, the *wsgen* utility can be used without our bothering inspect the artifacts that it produces. For the most part, JAX-B remains unseen infrastructure.



Beyond the Client-Side wsgen

A web service's SEI contains all of the information required to generate the *wsgen artifacts*. After all, the SEI declares the service operations as `@WebMethod`s and these *declarations* specify argument and return types.

In the current Metro release, the `Endpoint` publisher automatically generates the *wsgen* artifacts if the programmer does not. For example, once the sample service has been compiled, the command:

```
% java -cp .:$METRO_HOME/lib/jaxws-rt.jar test.WS1Pub
```

publishes the web service even though no *wsgen* artifacts were generated beforehand. (Under Windows, `$METRO_HOME` becomes `%METRO_HOME%`.) Here is the output:

```
com.sun.xml.ws.model.RuntimeModeler getRequestWrapperClass
INFO: Dynamically creating request wrapper Class test.jaxws.Op
com.sun.xml.ws.model.WrapperBeanGenerator createBeanImage
INFO:
@XmlRootElement(name=op, namespace=http://test/)
@XmlType(name=op, namespace=http://test/)
public class test.jaxws.Op {
    @XmlRootElement(name=arg0, namespace=)
    public I arg0
}
com.sun.xml.ws.model.RuntimeModeler getResponseWrapperClass
INFO: Dynamically creating response wrapper bean Class
      test.jaxws.OpResponse
com.sun.xml.ws.model.WrapperBeanGenerator createBeanImage
INFO:
@XmlRootElement(name=opResponse, namespace=http://test/)
@XmlType(name=opResponse, namespace=http://test/)
public class test.jaxws.OpResponse {
    @XmlRootElement(name=return, namespace=)
    public I _return
}
```

In time this convenient feature of the Metro release will make its way into core Java so that the *wsgen* step in `document`-style services can be avoided. It is still helpful to understand exactly how the JAX-B artifacts from the *wsgen* utility figure in Java-based web services.

2.4.4. Generating a WSDL with the wsgen Utility

The *wsgen* utility also can be used to generate a WSDL document for a web se
For example, the command:

```
% wsgen -cp "." -wsdl ch01.ts.TimeServerImpl
```

generates a WSDL for the original `TimeServer` service. The `TimeServer` service is
rather than `document` style, which is reflected in the WSDL. There is one critica
difference between the WSDL generated with the *wsgen* utility and the one
retrieved at runtime after the web service has been published: the *wsgen*-
generated WSDL does not include the service endpoint, as this URL depends o
actual publication of the service. Here is the relevant segment from the *wsger*
generated WSDL:

```
<service name="TimeServerImplService">
  <port name="TimeServerImplPort" binding="tns:TimeServerImplPortBinding">
    <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
  </port>
</service>
```

Except for this difference, the two WSDLs are the same service contract.

Generating a WSDL directly from *wsgen* will be useful in later security exampl
a web service is secured, then so is its WSDL; hence, the first step in writing a
wsimport-supported client to test the service is to access the WSDL, and *wsge*
an easy way to take this step.