SCDJWS 1.4 (CX-310-220) - CONTENTS

Revision #	*Date*	*Changes*
0.1	4th Aug, 2006	Initial Version.

Disclaimer: This document is my notes from Richard Monson-Haefel's book J2EE Web services and Sun's Blueprint book on J2EE Webservices namely, Designing Webservices with J2EE 1.4 Platform by Inderjeet Singh et al. For WS-Security, I have referred http://www.xyzws.com, SCDJWS study guide. The purpose of this document is to capture atleast matter on all topics prescribed in the syllabus for the SCDJWS 1.4 exam.

Peer Reynders at JavaRanch.com forum for SCDJWS, said the following:

Unlike the SCWCD 1.4 and SCBCD 1.3 exams this exam is about breadth not depth (well at least not too deep).
You should only need to use the specifications if there is material that you do not understand or if you come across an explanation that is ambiguous.

- RMH – the information here is covered in more depth that the objectives (that RMH addresses) require. But you will probably need that extra depth to understand why things are the way they are. Make sure you have an absolute solid understanding of XML, XML Schema, SOAP and WSDL as described by the book. Get a good understanding of SAAJ (SwA), JAX-RPC and know the basic steps necessary for some of the more basic procedures that can be implemented with those API (usually detailed memorization of the entire API in not required – understand *what* the API was created for and how the API is applied to the fundamental problem(s) that the API is supposed to help you solve - that's how you identify which API methods are important). For UDDI understand the entities and the role they play in the problem that UDDI tries to address. For JAXR know how UDDI entities map to JAXR structures and understand the basic procedures of manipulation of these structures - don't try to memorize the entire API. (Also don't skip the relevant appendices) (1)
- Blueprints – Read the whole thing. Some recommend to read this one first – I feel that your learning at this point is more effective if you already have RMH under your belt – so that you understand the details. (1)
- MZ's Guide – if you can, go through the entire thing (for review purposes). But whatever you do, understand the security section – this isn't covered anywhere else. (1)
- JAXB Tutorial – do as many of the relevant J2EE and web service tutorials that relate the material you have read about. Minimally do the JAXB tutorial because it isn't covered anywhere else (in MZ's Guide maybe – but it's better to experience this one).(1) Chapter 1: Binding XML Schema to Java Classes with JAXB
- Basic Profile – read through this one. By now you should understand the issues involved, this reinforces your understanding of what interoperability is all about.
- TrAX – Do a tutorial or read about the basics of the JAXP Transformation API for XML – no

need to know it in much depth. (Elliotte Rusty Harold; Processing XML with Java: Chapter 17.2 XSLT - TrAX)

- Mock exam material:
  - http://java.sun.com/developer/Quizzes/misc/jwsa.html
  - Mikalai Zaikin's SCWSD Quiz (90+ questions)
  - Whizlab's SCDJWS Sample exam.
  - http://www.xyzws.com SCDJWS Free mock exams (200 questions).
- Links:
  - http://www.valoxo.ch/jr/SCDJWS_Links.html

JavaRanch SCDJWS Links

Objectives and Specifications	
Sun Education for Certifications	SCJDWS Exam Objectives
WS-I Basic Profile 1.0a	J2EE 1.4
XML 1.0	Namespaces in XML
XML Schema Specification Part 0, Part 1, Part 2	XPointer Scheme
SOAP 1.1	SOAP with Attachments
SOAP Digital Signatures	WSDL 1.1
UDDI 2.04	UDDI Core tModels
Providing a Taxonomy for Use in UDDI	
Servlet 2.4	EJB 2.1
Implementing Enterprise Web Services 1.1	JAX-RPC 1.1
JAXP 1.2	JAXR 1.0
SAAJ 1.2	JAXB 1.0
Tutorials	
Sun's Java Web Services tutorial	Sun's Code Camps
Mock exams	
Whizlabs' SCDJWS Exam Simulator	XYZWS.com's SCDJWS Mock Exam
Mikalai Zaikin's SCDJWS Quiz	Sun Microsystems' Web Services Quiz
Notes and Information	
Sathya Srinivasan's notes	Mikalai Zaikin's SCDJWS Study Guide
Val's Specs-Objectives Mapping	Whizlabs's SCDJWS Certification Primer
SCDJWS Certification Groups on Yahoo!	SCDJWS Certification Center
Artima's Web Services Corner	IBM's Web Services Zone
XYZWS.com's SCDJWS Study Guide	

XML Web service Standards (Objective 1)

[RMH – 2,3]
1.1 Given **XML documents, schemas, and fragments** determine whether their syntax and form are correct (according to W3C schema) and whether they **conform to the WS-I Basic Profile 1.0a**.
1.2 Describe the **use of XML schema** in J2EE Web services.
1.3 Describe the **use of namespaces** in an XML document.

J2EE Web Services is about interoperability. Web service technologies are platform-agnostic; in other words, the medium used to communicate is not specific to any combination of programming language, operating system, and hardware.

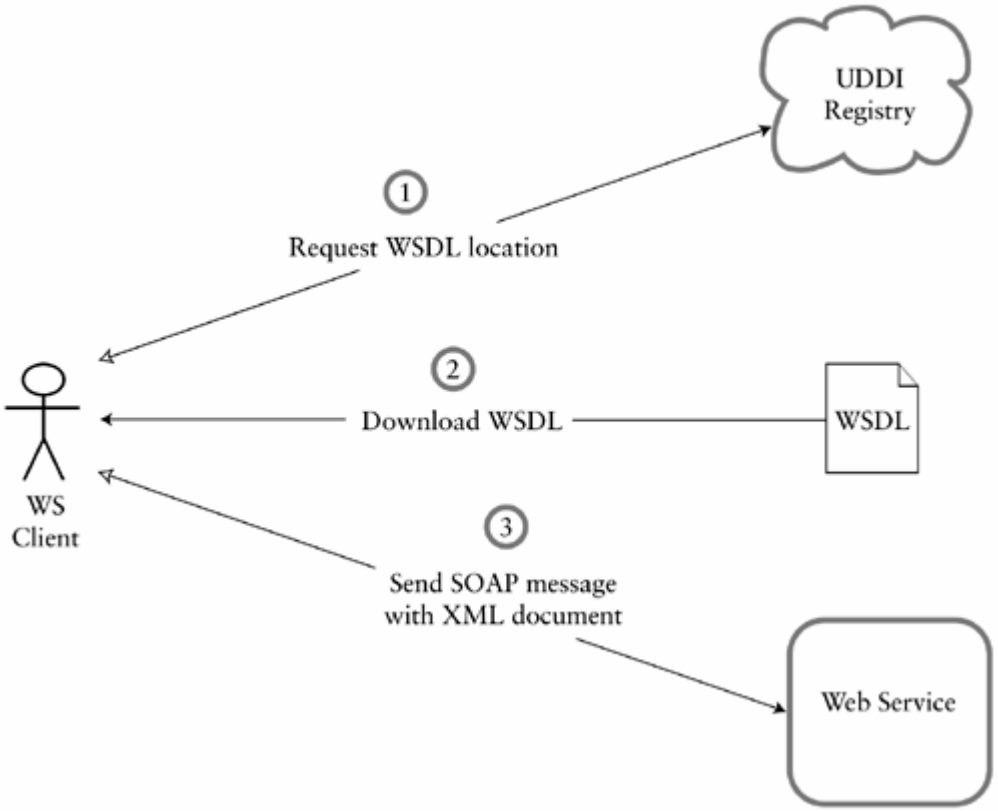
At the heart of J2EE Web Services interoperability, and of Web services interoperability in general, is the **Basic Profile 1.0 (BP),** published by the **Web Services Interoperability Organization (WS-I)**. **The BP provides a set of rules that govern how applications make use of common Web service technologies so that everyone is speaking the same language.**

It's not mandatory that you use it. Only J2EE 1.4 vendors are required to support BP. In fact, J2EE 1.4 Web Services APIs are also required to support technologies that don't conform to the BP, like SOAP Messages with Attachments and RPC/Encoded messaging. The Web services components and APIs are actually described by the Web Services for J2EE 1.1 specification. **WS-J2EE 1.1** requires compliance with the WS-I Basic Profile 1.0.

A Web service is a software application that conforms to the Web Service interoperability Organization's Basic Profile 1.0.

The main purpose of Web service technologies is to allow applications on different platforms to exchange business data. Web service technologies are used for Application-to-Application (A2A) integration or Business-to-Business (B2B) communication. A2A refers to disparate applications within a single organization communicating and exchanging data—A2A is also known as Enterprise Application Integration (EAI). B2B refers to multiple organizations, typically business partners, exchanging data. Web service technologies today are used mostly in A2A/EAI settings, but they are also seeing growth in the B2B arena.

Figure 1-2. Web Services Interaction Diagram



XML (eXtensible Markup Language), SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery, and Integration) are used in concert to provide Web service applications with a type system (XML), a messaging protocol (SOAP), an interface definition language (WSDL), and a registry for publishing Web services (UDDI). XML documents contain the information being exchanged between two parties, while SOAP provides a packaging and routing standard for exchanging XML documents over a network. WSDL allows an organization to describe the types of XML documents and SOAP messages that must be used to interact with their Web services. Finally, UDDI allows organizations to register their Web services in a uniform manner within a common directory, so clients can locate their Web services and learn how to access them.

WS-I Basic Profile 1.0

The BP defines a set of conformance rules that clear up ambiguities in the specifications of XML, WSDL, SOAP, and UDDI, and defines in concrete terms how to use these technologies in concert to register, describe, and communicate with Web services. WSDL is a very generalized technology, which allows you to describe any kind of Web service (REST based or SOAP based). Unfortunately, WSDL is so general that it's difficult in some circumstances to determine exactly how, for example, a SOAP message exchanged with the Web service should be formatted. In addition, WSDL defines features that have, in practice, made interoperability more difficult, such as the SMTP and MIME bindings. The **BP fixes these interoperability problems by telling us exactly how WSDL should describe SOAP-based Web services, and by restricting the use of WSDL.**

Introducing XML, SOAP, WSDL and UDDI

Currently there is only one version, **XML 1.0**, which is managed by the World Wide Web Consortium (W3C). **SOAP 1.1** (from Simple Object Access Protocol) defines a standard packaging format for

transmitting XML data between applications on a network. It is not specific to any programming language, product, or hardware platform, so it can be used by just about any kind of application (C++, Java, .NET, Perl, legacy systems, and others). A SOAP message is just an XML document. SOAP is specially designed, however, to contain and transmit other XML documents as well as information related to routing, processing, security, transactions, and other qualities of service.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:addr="http://www.Monson-Haefel.com/jwsbook/ADDR" >
  <soap:Body>
    <addr:address>
      <addr:name>Amazon.com</addr:name>
      <addr:street>1516 2nd Ave</addr:street>
      <addr:city>Seattle</addr:city>
      <addr:state>WA</addr:state>
      <addr:zip>90952</addr:zip>
    </addr:address>
  </soap:Body>
</soap:Envelope>
```

SOAP takes advantage of advanced XML features like **XML namespaces** (similar to Java package names) and **XML schemas** (used to type data). The important thing to understand at this point is that **SOAP messages serve as a network envelope for exchanging XML documents and data.** Having a single industry standard for packaging and exchanging XML data makes Web service applications more interoperable, more understandable, and easier to implement. The BP requires the use of SOAP 1.1 and provides a number of clarifications and restrictions that largely eliminate interoperability problems associated with this sometimes poorly specified standard. There is an ancillary specification called **SOAP Messages with Attachments** (**SwA**). SwA defines a message format for attaching binary data (images, sound files, documents, and so on) to SOAP messages. Although SwA is not sanctioned by the BP, support for it is required by the J2EE 1.4 Web Services specifications—SwA will also be supported in the next version of the BP, version 1.1, which is still in development.

WSDL (Web Services Description Language) is a standard for describing the structure of the XML data exchanged between two systems using SOAP. The BP requires the use of **WSDL 1.1**, but also provides strict rules on how it's used, to improve interoperability.

UDDI (Universal Description, Discovery, and Integration) defines a standard set of Web service operations (methods) that are used to store and look up information about other Web service applications. In other words, UDDI defines a standard SOAP-based interface for a Web services registry. You can use a UDDI registry to find a particular type of Web service, or to find out about the Web services hosted by a specific organization. A UDDI registry is often referred to as a "Yellow Pages" for Web services. When you look up information about a Web service in a UDDI registry, you can narrow your search using various categories (technologies used, business types, industry, and so on). Each entry in a UDDI registry provides information on where the Web service is located and how to communicate with it. The UDDI specification is now maintained by the Organization for the Advancement of Structured Information Standards (OASIS), but was originally created by Microsoft and IBM, which led a multi-vendor organization called UDDI.org. The UDDI.org has set up a free UDDI registry open to everyone—something like a Yahoo! for Web services. Most UDDI registries, however, are private systems deployed by individual companies or trade organizations.

Of the four principal Web services technologies, **UDDI is the only one the BP says is optional.** You don't have to use UDDI, but if you need a Web service registry, UDDI is the preferred technology. The BP specifies the use of **UDDI version 2.0.**

Introducing J2EE Webservices APIs

You can think of **JAX-RPC 1.1** (Java API for XML-based RPC) as Java RMI over SOAP. JAX-RPC is divided into two parts: a set of client-side APIs, and a set of server-side components, called endpoints. The client-side APIs can be used from standalone Java applications or from J2EE components like servlets,

JSPs, or EJBs. There are three client-side APIs: **generated stub**, **dynamic proxy**, and **DII** (Dynamic Invocation Interface). The generated stub is the one you will use the most, and its semantics closely resemble those of Java RMI. The dynamic proxy API also follows many of the Java RMI semantics, but is used less often. The DII is a very low-level API used primarily by vendor tools, but can also be employed by Web services developers if necessary.

The server-side components include the JAX-RPC service endpoint (**JSE**) and the **EJB endpoint**. The JSE component is actually a type of servlet that has been adapted for use as a Web services component. It's very easy to implement, yet it has access to the full array of services and interfaces common to servlets. The EJB endpoint is simply a type of stateless session EJB that has been adapted for use as a Web service endpoint. The EJB endpoint provides all the transactional and security features of a normal stateless session bean, but it's specifically designed to process SOAP requests.

SAAJ (SOAP with Attachments API for Java) is **a low-level SOAP API that complies with SOAP 1.1 and the SOAP Messages with Attachments** (**SwA**) specification. SAAJ allows you to build SOAP messages from scratch as well as read and manipulate SOAP messages. You can use it alone to create, transmit, and process SOAP messages, but you're more likely to use it in conjunction with JAX-RPC. In JAX-RPC, SAAJ is used primarily to process SOAP header blocks (the SOAP message meta-data).

JAXR 1.0 (Java API for XML Registries) provides an API for accessing UDDI registries. It simplifies the process of publishing and searching for Web service endpoints. JAXR was originally intended for ebXML registries, a standard that competes with UDDI, but was adapted for UDDI. JAXR has a set of business-domain types like Organization, PostalAddress, and Contact as well as technical-domain types like ServiceBinding, ExternalLink, and Classification. These domain models map nicely to UDDI data types. JAXR also defines APIs for publishing and searching for information in a UDDI registry.

JAXP 1.2 (Java API for XML Processing) provides a framework for using DOM 2 and SAX2, standard Java APIs that read, write, and modify XML documents.

DOM 2 (Document Object Model, Level 2) is a Java API that models XML documents as trees of objects. It contains objects that represent elements, attributes, values, and so on. DOM 2 is used a lot in situations where speed and memory are not factors, but complex manipulation of XML documents is required. DOM 2 is also the basis of SAAJ 1.1.

SAX2 (Simple API for XML, version 2) is very different in functionality from DOM 2. When a SAX parser reads an XML document, it fires events as it encounters start and end tags, attributes, values, etc. You can register listeners for these events, and they will be notified as the SAX2 parser detects changes in the XML document it is reading.

XML 1.0 Primer (Objective 1.1)

```
<?xml version="1.0" encoding="UTF-8" ?>
<addresses>
   <address category="friend">
       <name>Bill Frankenfiller</name>
       <street>3243 West 1st Ave.</street>
       <city>Madison</city>
       <state>WI</state>
       <zip>53591</zip>
   </address>
   <address category="business">
       <name>Amazon.com</name>
       <street>1516 2nd Ave</street>
       <city>Seattle</city>
       <state>WA</state>
       <zip>90952</zip>
   </address>
</addresses>
```


XML documents are composed of Unicode text (usually UTF-8), so people as well as software can understand them. A specific XML markup language describes which element names are used and how they are organized. Because anyone can make up a new markup language at any time, the number of them is potentially infinite. The ability to create an infinite number of new markup languages is why XML is called eXtensible. While XML itself is not a programming language, there are XML markup languages that can be compiled and interpreted. For example, XSLT (eXtensible Stylesheet Language Transformation) is a programming language based on XML.
XML is used for two different purposes: **document-oriented** and **data-oriented** applications. Document-oriented markup languages like XHTML and DocBook are focused on the format and presentation of literature. Data-oriented markup languages focus on how data is organized and typed; they define a schema for storing and exchanging data between software applications. Some XML markup languages are industry standards, like SOAP and XHTML, while most are designed to serve a single application, organization, or individual.
An XML parser is a utility that can read and analyze an XML document. In most cases an XML parser is combined with a parser API (such as SAX2 or DOM 2) that allows a developer to interact with the XML document while it's being parsed, or after.
XML document instance, means it represents one possible set of data for a particular markup language. An XML document is made up of declarations, elements, attributes, text data, comments, and other components.
An XML document may start with an **XML declaration**, **but it's not required**. An XML declaration declares the version of XML used to define the document (there is only one version at this time, version 1.0). It may also indicate the character encoding used to store or transfer the document, and whether the document is standalone or not.

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

XML markup languages organize data hierarchically, in a tree structure, where each branch of the tree is called an **element** and is delimited by a pair of tags. All elements are named and have a start tag and an end tag.

```
<address>
   <name>Amazon.com</name>
   <street>1516 2nd Ave</street>
   <city>Seattle</city>
   <state>WA</state>
   <zip>90952</zip>
</address>
```

According to the WS-I Basic Profile 1.0, XML documents used in Web services must use either UTF-8 or UTF-16 encoding. This limitation simplifies things for Web service vendors and makes interoperability easier, because there is only one character encoding standard to worry about, Unicode. UTF-8 and UTF-16 encoding allows you to use characters from English, Chinese, French, German, Japanese, and many other languages.
An element **name must always begin with a letter or underscore**, but can contain pretty much any Unicode character you like, including underscores, letters, digits, hyphens, and periods. Also, **an element name must never start with the string xml,** as this is reserved by the XML 1.0 specification.
An element may have one or more **attributes**. You use an attribute to supplement the data contained by an element, to provide information about it not captured by its contents.

```
<address category="business" >
```

Each attribute is a name-value pair. The value must be in single or double quotes. Attribute names have the same restrictions as element names.
In many cases, empty-element tags are used when the attributes contain all the data.

```
<phone countrycode="01" areacode="715" number="55529482" ext="341" />
```

Using attributes instead of nested elements is considered a matter of style, rather than convention. There are no "standard" design conventions for using attributes or elements.
Comments are placed between a <!-- designator and a --> designator, as in HTML: <!-- comment goes here -->.
When an element contains text, you have to be careful about which characters you use because certain characters have special meaning in XML. Using quotes (single or double), less-than and greater-than signs (< and >), the ampersand (&), and other special characters in the contents of an element will confuse parsers, which consider these characters to be special parsing symbols. To avoid parsing problems you can use escape characters like &gt for greater-than or &amp for ampersand, but this technique can become

cumbersome. A **CDATA section** allows you to **mark a section of text as literal so that it will not be parsed for tags and symbols, but will instead be considered just a string of characters.** For example, if you want to put HTML in an XML document, but you don't want it parsed, you can embed it in a CDATA section.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This document contains address information -->
<address category="business" >
   <name>Amazon.com</name>
   <street>1516 2nd Ave</street>
   <city>Seattle</city>
   <state>WA</state>
   <zip>90952</zip>
   <note>
   <![CDATA[
     <html>
       <body>
            <p>
            Last time I contacted <b>Amazon.com</b> I spoke to ...
       </body>
     </html>
   ]]>
  </note>
</address>
```

Parsers usually provide a programming API that allows developers to access elements, attributes, text, and other constructs in XML documents. There are basically two standard kinds of XML parser APIs: **SAX** and **DOM**.

SAX2 is based on an event model. As the SAX2 parser reads an XML document, starting at the beginning, it fires off events every time it encounters a new element, attribute, piece of text, or other component. SAX2 parsers are generally very fast because they read an XML document sequentially and report on the markup as it's encountered.

DOM 2 presents the programmer with a generic, object-oriented model of an XML document. Elements, attributes, and text values are represented as objects organized into a hierarchical tree structure that reflects the hierarchy of the XML document being processed. DOM 2 allows an application to navigate the tree structure, modify elements and attributes, and generate new XML documents in memory. It's a very powerful and flexible programming model, but it's also slow compared to SAX2, and consumes a lot more memory.

In addition to providing a programming model for reading and manipulating XML documents, the parser's primary responsibility is checking that documents are **well formed**; that is, that their elements, attributes, and other constructs conform to the syntax prescribed by the XML 1.0 specification. For example, an element without an end tag, or with an attribute name that contains invalid characters, will result in a syntax error. A parser may also, optionally, enforce **validity** of an XML document. An XML document may be well formed, but invalid because it is not organized according to its schema.

Two popular Java parser libraries, **Crimson** and **Xerces-J**, include both SAX2 and DOM 2, so you can pick the API that better meets your needs. **Crimson is a part of the Java 2 platform (JDK 1.4)**, which means it's available to you automatically. Xerces, which some people feel is better, is maintained by the Apache Software Foundation. You must download it as a JAR file and place it in your classpath (or ext directory) before you can use it. Either parser library is fine for most cases, but **Xerces supports W3C XML Schema validation while Crimson doesn't**.

JAXP (Java API for XML Processing), which is part of the J2EE platform, **is not a parser**. **It's a set of factory classes and wrappers for DOM 2 and SAX2 parsers.** Java-based DOM 2 and SAX2 parsers, while conforming to standard DOM 2 or SAX2 programming models, are instantiated and configured differently, which inhibits their portability. JAXP eliminates this portability problem by providing a consistent programming model for instantiating and configuring DOM 2 and SAX2 parsers. JAXP can be used with Crimson or Xerces-J. JAXP is a standard Java extension library, so using it will help keep your J2EE applications portable.

Other non-standard XML APIs are also available to Java developers, including **JDOM, dom4j, and XOM**. If ease of use is important, you may want to use one of these non-standard parser libraries, but if J2EE portability is more important, stick with JAXP, DOM 2, and SAX2.

XML Namespaces (Objective 1.3)

An XML namespace provides **a qualified name for an XML element or attribute**, the same way that a Java package provides a qualified name for a Java class.

Creating XML documents based on multiple markup languages is often desirable. For example, suppose we are building a billing and inventory control system for a company called Monson-Haefel Books. We can define a standard markup language for address information, the Address Markup Language, to be used whenever an XML document needs to contain address information. An instance of Address Markup is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<address category="business" >
   <name>Amazon.com</name>
   <street>1516 2nd Ave</street>
   <city>Seattle</city>
   <state>WA</state>
   <zip>90952</zip>
</address>
```

The Address Markup will also be reused in other XML markup languages (types of XML documents): Invoice, Purchase Order, Shipping, Marketing, and others. Address Markup has its own schema, defined using either DTD (Document Type Definition) or the W3C XML Schema Language, which dictates how its elements are organized. Every time we use address information in an XML document, it should be validated against **Address Markup's schema**.

```
<?xml version="1.0" encoding="UTF-8" ?>
<purchaseOrder orderDate="2003-09-22" >
   <accountName>Amazon.com</accountName>
   <accountNumber>923</accountNumber>
   <address>
      <name>AMAZON.COM</name>
      <street>1850 Mercer Drive</street>
      <city>Lexington</city>
      <state>KY</state>
      <zip>40511</zip>
   </address>
   <book>
      <title>J2EE Web Services</title>
      <quantity>300</quantity>
      <wholesale-price>29.99</wholesale-price>
   </book>
   <total>8997.00</total>
</purchaseOrder>
```

If the purchase-order document has its own schema (defined by the Purchase Order Markup Language) and the address information has its own schema (defined by the Address Markup Language), how do we indicate that the address element should conform to the Address Markup Language, while the rest of the elements conform to the Purchase Order Markup Language? We use **namespaces**. We can state that the address elements conform to Address Markup by declaring the namespace of Address Markup in the address element. We can do the same thing for the purchase order elements by declaring, in the purchaseOrder element, that they conform to the Purchase Order Markup.

```
<?xml version="1.0" encoding="UTF-8" ?>
<purchaseOrder orderDate="2003-09-22"
 xmlns="http://www.Monson-Haefel.com/jwsbook/PO">
   <accountName>Amazon.com</accountName>
   <accountNumber>923</accountNumber>

   <address xmlns="http://www.Monson-Haefel.com/jwsbook/ADDR">
      <name>AMAZON.COM</name>
      <street>1850 Mercer Drive</street>
      <city>Lexington</city>
      <state>KY</state>
      <zip>40511</zip>
   </address>

   <book>
      <title>J2EE Web Services</title>
```

```
      <quantity>300</quantity>
      <wholesale-price>29.99</wholesale-price>
   </book>
   <total>8997.00</total>
</purchaseOrder>
```

The **xmlns** attribute declares a specific XML namespace in the form **xmlns="someURI".** The value of an xmlns attribute is a **URI reference**, which must conform to the URI specification (RFC2396) defined by the IETF (Internet Engineering Task Force). URIs **(Uniform Resource Identifiers) can take many different forms; the most common is the URL** (Universal Resource Locator) as is used in the above example. It's important to remember that the **URI used for the XML namespace should be unique to that markup language, but it doesn't have to point to an actual resource or document.**

The scope of a **default namespace** applies only to the element and its descendants, so the xmlns used in the address element applies only to the address, name, street, city, state, and zip elements. The default xmlns declared in the purchaseOrder element applies to all the elements except the address elements, because the address element overrides the default namespace of the purchaseOrder element to define its own default namespace.

To simplify things, XML Namespaces defines **a shorthand notation** for associating elements and attributes with namespaces. You can assign an XML namespace to a **prefix**, then **use that prefix to fully qualify each element name**.

```
<?xml version="1.0" encoding="UTF-8" ?>
<po:purchaseOrder orderDate="2003-09-22"
   xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
   xmlns:addr="http://www.Monson-Haefel.com/jwsbook/ADDR">

   <po:accountName>Amazon.com</po:accountName>
   <po:accountNumber>923</po:accountNumber>

   <addr:address>
      <addr:name>AMAZON.COM</addr:name>
      <addr:street>1850 Mercer Drive</addr:street>
      <addr:city>Lexington</addr:city>
      <addr:state>KY</addr:state>
      <addr:zip>40511</addr:zip>
   </addr:address>

   <po:book>
      <po:title>J2EE Web Services</po:title>
      <po:quantity>300</po:quantity>
      <po:wholesale-price>29.99</po:wholesale-price>
   </po:book>
   <po:total>8997.00</po:total>
</po:purchaseOrder>
```

It's not necessary to qualify every element with a namespace prefix. You can rely on default namespaces to determine the namespaces of all elements not explicitly prefixed,

```
<?xml version="1.0" encoding="UTF-8" ?>
<purchaseOrder orderDate="2003-09-22"
   xmlns="http://www.Monson-Haefel.com/jwsbook/PO"
   xmlns:addr="http://www.Monson-Haefel.com/jwsbook/ADDR">

   <accountName>Amazon.com</accountName>
   <accountNumber>923</accountNumber>

   <addr:address>
      <addr:name>AMAZON.COM</addr:name>
      <addr:street>1850 Mercer Drive</addr:street>
      <addr:city>Lexington</addr:city>
      <addr:state>KY</addr:state>
```

```
      <addr:zip>40511</addr:zip>
   </addr:address>

   <book>
      <title>J2EE Web Services</title>
      <quantity>300</quantity>
      <wholesale-price>29.99</wholesale-price>
   </book>
   <total>8997.00</total>
</purchaseOrder>
```

Any element that doesn't have a prefix is, by default, a member of http://www.Monson-Haefel.com/jwsbook/PO.

In XML-speak, a prefix combined with an element name is called a **QName**, which stands for "**qualified name**." A QName has two parts, the XML namespace and the local name. In addr:street, http://www.Monson-Haefel.com/jwsbook/ADDR is the namespace and street is the local name.

XML namespaces based on URLs tend to be universally unique, which makes it easy for parsers and software applications to distinguish between instances of different markup languages within the same document. Namespaces help avoid name collisions, where two elements from different markups share a common local name. For example, a WSDL document can use Monson-Haefel's postal address element as well as the SOAP-binding address element in the same document. Although both elements are named address, they belong to different namespaces with different QNames, so there is no name conflict.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="Address-Update"
   targetNamespace="http://www.monson-haefel.org/jwsbook/Address-Update"
   xmlns:tns="http://www.monson-haefel.org/jwsbook/Address-Update"
   xmlns:addr="http://www.Monson-Haefel.com/jwsbook/ADDR"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
...

<!-- message elements describe the paramters and return values -->
<message name="AddressMessage">
   <part name="address" element="addr:address" />
</message>

...

<!-- service tells us the Internet address of a Web service -->
<service name="AddressUpdateService">
  <documentation>Update a customers mailing address</documentation>
  <port name="AddressUpdate_Port" binding="tns:AddressUpdate_Binding">
     <soap:address
      location="http://www.monson-haefel.org/jwsbook/BookPrice" />
  </port>
</service>

</definitions>
```

XML parsers and other tools can use XML namespaces to process, sort, and search XML elements in a document according to their QNames. This allows reusable code modules to be invoked for specific namespaces. For example, you can create a custom Java tool to map an instance of Address Markup to a relational database. It will be invoked only for address elements that belong to the Address Markup namespace, http://www.Monson-Haefel.org/addr, and not for address elements of any other namespace.

XML namespaces also allow for a great **versioning system**. If the Address Markup changes, we can assign the new version its own namespace, such as http://www.Monson-Haefel.org/ADDR-2, so it can be

distinguished from its predecessor. We can support both the old and new versions of the Address Markup Language simultaneously, because the parser can uniquely identify each version by its namespace.

XML Schema (Objective 1.2)

Basics

The XML specification includes the Document Type Definition (**DTD**), which can be used to describe XML markup languages and to validate instances of them (XML documents). To address limitations of DTDs, the W3C (World Wide Web Consortium), which manages the fundamental XML standards, created a new way to describe markup languages called XML schema.

DTD failed to address data typing.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT address (street+, city, state, zip)>
<!ELEMENT street (#PCDATA) >
<!ELEMENT city (#PCDATA) >
<!ELEMENT state (#PCDATA) >
<!ELEMENT zip (#PCDATA) >
<!ATTLIST address category CDATA #REQUIRED >
```

DTD declares that an address element may contain one or more street elements and must contain exactly one of each of the city, state, and zip elements. It also declares that the address element must have a category attribute.

To be valid, an XML instance must conform to its DTD, which means it must use the elements specified by the DTD in the correct order and multiplicity (zero, one, or many times). DTDs have a very weak typing system that restricts elements to four broad types of data: EMPTY, ANY, element content, or mixed element-and-text content. In other words, **DTDs can only restrict elements to containing nothing, other elements, or text**—not a very granular typing system. **DTDs don't support types like integer, decimal, boolean, and enumeration.** For example, the Address Markup DTD cannot restrict the contents of the zip element to an integer value or the state element to a set of valid state codes.

XML schema, by contrast, provides a much stronger type system. Many believe that XML schema is superior to DTD because it defines a richer type system, which includes simple primitives (integer, double, boolean, among others) as well as facilities for more complex types. XML schema facilitates type inheritance, which allows simple or complex types to be extended or restricted to create new types. In addition, XML schema supports the use of XML namespaces to create compound documents composed of multiple markup languages.

A schema describes an XML markup language. **Specifically it defines which elements and attributes are used in a markup language, how they are ordered and nested, and what their data types are.**

A schema describes the structure of an XML document in terms of **complex types** and **simple types**. **Complex types describe how elements are organized and nested. Simple types are the primitive data types contained by elements and attributes.**

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook">

    <element name="address" type="mh:USAddress" />

    <complexType name="USAddress">
      <sequence>
        <element name="name"    type="string" />
        <element name="street"  type="string" />
        <element name="city"    type="string" />
        <element name="state"   type="string" />
        <element name="zip"     type="string" />
      </sequence>
    </complexType>
    ...
</schema>
```

That schemas are XML documents is a critical point: It makes the development of validating parsers and other software tools easier, because the operations that manipulate schemas can be based on XML parsers, which are already widely available. DTDs, the predecessor to schemas, were not based on XML, so processing them required special parsing.

1. The root element of a schema document is always the schema element. Nested within the schema element are element and type declarations.
2. The schema element assigns the XML schema namespace ("http://www.w3.org/2001/XMLSchema") as the **default namespace**. This namespace is the standard namespace defined by the XML schema specification—all the XML schema elements must belong to this namespace. The schema element also defines the **targetNamespace** attribute, which declares the XML namespace of all new types explicitly created within the schema. For example, the USAddress type is automatically assigned to targetNamespace, "http://www.Monson-Haefel.com/jwsbook".
3. The schema element also uses an XML namespace declaration to assign the prefix mh to the targetNamespace. Subsequently, newly created types in the schema can be referred to as "mh:Typename".
4. An instance document based on this schema would use the address element directly or refer to the USAddress type. When a parser that supports XML schema reads the document, it can validate the contents of the XML document against the USAddress type definition.

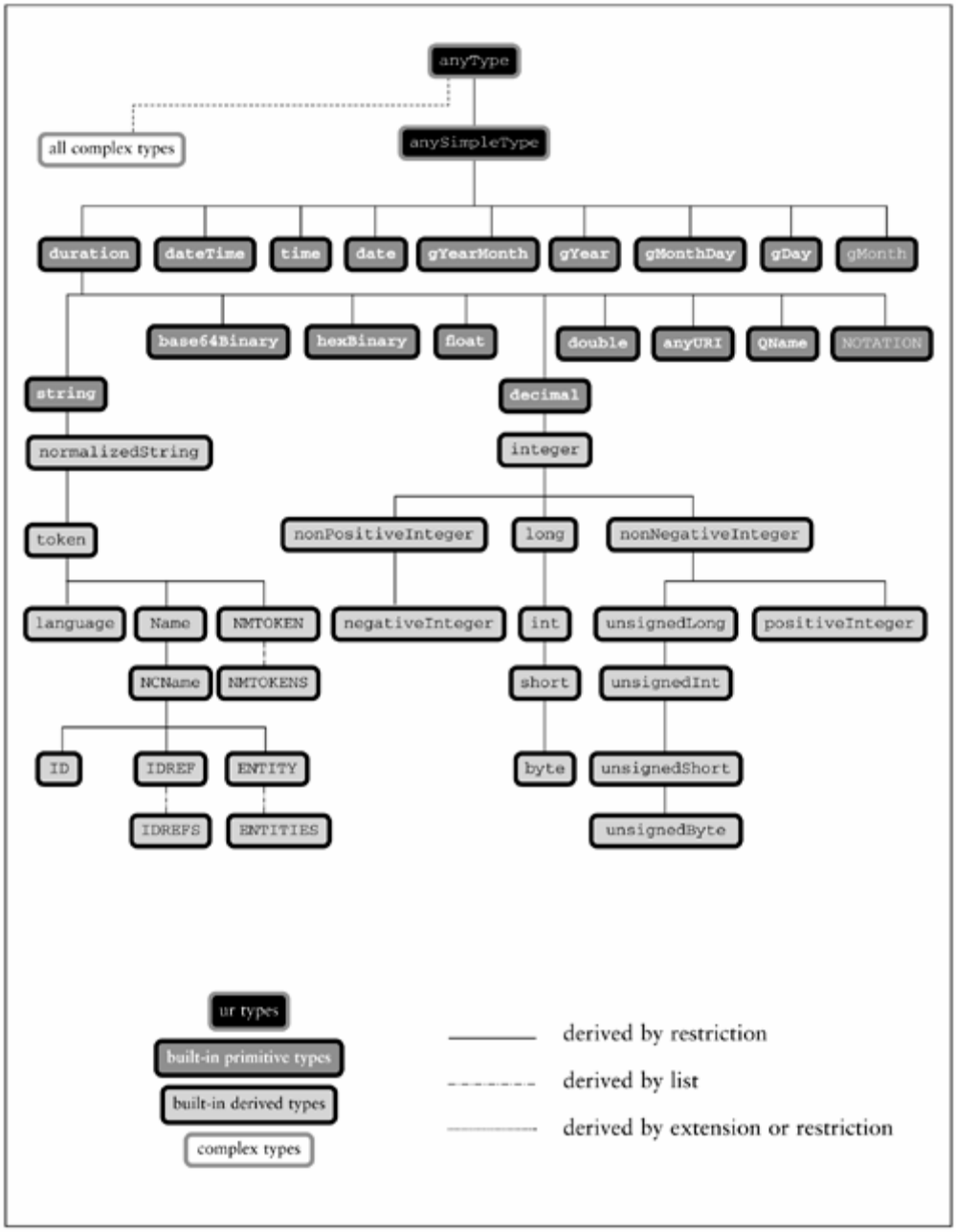
```
<?xml version="1.0" encoding="UTF-8"?>
<addr:address xmlns:addr="http://www.Monson-Haefel.com/jwsbook">
   <name>Amazon.com</name>
   <street>1516 2nd Ave</street>
   <city>Seattle</city>
   <state>WA</state>
   <zip>90952</zip>
</addr:address>
```

The XML schema specification defines **44 standard simple types**, called **built-in types**. The built-in types are the standard building blocks of an XML schema document. They are members of the XML schema namespace, "http://www.w3.org/2001/XMLSchema".

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
    targetNamespace="http://www.Monson-Haefel.com/jwsbook">
  ...
  <complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName"   type="string" />
      <element name="accountNumber" type="integer" />
      <element name="total"         type="float" />
      <!-- More stuff follows -->
    </sequence>
  </complexType>
  ...
</schema>
```

All built-in simple and complex types are ultimately derived from anyType, which is the ultimate base type, like the Object class in Java. The XML Schema Part 2: Datatypes specification offers a diagram of the data type hierarchy:

Figure 3-1. XML Schema Type Hierarchy



A **complex type** is analogous to a Java class definition with fields but no methods. An instance of a complex type is an element in an XML document. Most **complexType** declarations in schemas will contain a **sequence** element that lists one or more **element** definitions. The element definitions tell you which elements are nested in the type, the order in which they appear, and the kind of data each element contains.
A **complex type may contain a sequence of elements that are simple types or other complex types**. For example, we can define an element for a purchase-order document by adding a PurchaseOrder type to the Monson-Haefel Markup Language the new PurchaseOrder type has two nested elements, billAddress and shipAddress, both of type USAddress.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook" >
```



```
<element name="purchaseOrder" type="mh:PurchaseOrder" />
<element name="address" type="mh:USAddress" />

<complexType name="PurchaseOrder">
   <sequence>
      <element name="accountName"   type="string" />
      <element name="accountNumber" type="unsignedShort" />
      <element name="shipAddress"   type="mh:USAddress" />
      <element name="billAddress"   type="mh:USAddress" />
      <element name="book"          type="mh:Book" />
      <element name="total"         type="float" />
   </sequence>
   <attribute name="orderDate" type="date"/>
</complexType>

<complexType name="USAddress">
   <sequence>
      <element name="name"    type="string" />
      <element name="street"  type="string" />
      <element name="city"    type="string" />
      <element name="state"   type="string" />
      <element name="zip"     type="string" />
   </sequence>
</complexType>
<complexType name="Book">
   <sequence>
     <element name="title"           type="string" />
     <element name="quantity"        type="unsignedShort" />
     <element name="wholesale-price" type="float" />
   </sequence>
</complexType>

</schema>
```

The names of **XML schema types are case-sensitive**. When an element declares that it is of a particular type, it must specify both the namespace and the name of that type exactly as the type declares them.
In addition to sequences of elements, a complex type may also define its own **attributes**. An **instance** of the above schema is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<po:purchaseOrder orderDate="2003-09-22"
 xmlns:po="http://www.Monson-Haefel.com/jwsbook">
                <accountName>Amazon.com</accountName>
                <accountNumber>923</accountNumber>
                <shipAddress>
                       <name>AMAZON.COM</name>
                       <street>1850 Mercer Drive</street>
                       <city>Lexington</city>
                       <state>KY</state>
                       <zip>40511</zip>
                 </shipAddress>
                 <billAddress>
                       <name>Amazon.com</name>
                       <street>1516 2nd Ave</street>
                       <city>Seattle</city>
                       <state>WA</state>
                       <zip>90952</zip>
                  </billAddress>
                  <book>
                       <title>J2EE Web Services</title>
                       <quantity>300</quantity>
                       <wholesale-price>24.99</wholesale-price>
                  </book>
 <total>8997.00</total>
</po:purchaseOrder>
```

The **multiplicity** of an element, the number of times it occurs in an instance document, is controlled by occurrence constraints, which are declared by the **maxOccurs** and **minOccurs** attributes. For eg:

```
<complexType name="USAddress">
   <sequence>
```



```
      <element name="name"    type="string" />
      <element name="street"  type="string"
                              minOccurs="1" maxOccurs="2" />
      <element name="city"    type="string" />
      <element name="state"   type="string" />
      <element name="zip"     type="string" />
   </sequence>
   <attribute name="orderDate" type="date" use="required" />
</complexType>
```

The **default value for both maxOccurs and minOccurs is "1",** so if these attributes are not specified the element must be present exactly once. **0 <= minOccurs <= maxOccurs** or maxOccurs can be **unbounded** to specify that the element may occur an unlimited number of times.

Attributes also have occurrence constraints, but they are different from those of elements. Attribute types declare the **use** occurrence constraint, which may be **"required", "optional", or "prohibited"**, indicating that the attribute must, may, or may not be used, respectively. The default is "optional". An attribute might be "prohibited" if you want to stop the use of a particular attribute, perhaps one that is inappropriate or no longer in use.

An attribute may also have a default value, to be assigned if no value is explicitly declared in the instance document. For example, the USAddress type may include an attribute called category that can have the value "business" (default), "private", or "government".

```
<complexType name="USAddress">
   <sequence>
      <element name="name"    type="string" />
      <element name="street"  type="string" />
      <element name="city"    type="string" />
      <element name="state"   type="string" />
      <element name="zip"     type="string" />
   </sequence>
   <attribute name="category" type="string" default="business" />
</complexType>
```

The default attribute can be used only when the use attribute is "optional" (recall that "optional" is the default value for the use attribute). It wouldn't make sense to declare a default when the use is "required" or "prohibited". If the use attribute is "required", there is no need for a default because the attribute must appear in the instance document. If the use is "prohibited", the attribute's not allowed so there is no sense having a default value.

An attribute may also be declared **fixed**: A fixed value is assigned to the attribute no matter what value appears in the XML instance document. This feature is useful in rare situations where you want to force a particular attribute always to have the same value. For example, if a particular schema is assigned a version number, then that version number should be fixed for that schema (**UDDI does this**).

Most of the time you'll base complex types on **sequence** elements, but occasionally you may want to use the **all** element. Unlike sequence, which defines the exact order of child elements, **the XML schema all element allows the elements in it to appear in any order. Each element in an all group may occur once or not at all**; no other multiplicity is allowed. In other words, minOccurs is always "0" and maxOccurs is always "1". Finally, only single elements may be used in an all group; it can't include other groupings like sequence or all.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
            targetNamespace="http://www.Monson-Haefel.com/jwsbook" >
   ...
   <complexType name="USAddress">
      <all>
         <element name="name"    type="string" />
         <element name="street"  type="string" />
         <element name="city"    type="string" minOccurs="0"/>
         <element name="state"   type="string" minOccurs="0"/>
         <element name="zip"     type="string" />
      </all>
   </complexType>
   ...
</schema>
```

name, street, and zip elements must be present in the instance document, but the city and state elements may be absent. The elements can be in any order, but none of the elements may occur more than once.

In addition to declaring simple and complex types, a schema may also declare **global elements**, which XML instance documents can refer to directly. Global elements are declared as direct children of the schema element, rather than children of a complex type.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
    targetNamespace="http://www.Monson-Haefel.com/jwsbook">

  <element name="purchaseOrder" type="mh:PurchaseOrder"/>
  <complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="unsignedShort"/>
      <element name="shipAddress" type="mh:USAddress"/>
      <element name="billAddress" type="mh:USAddress"/>
      <element name="book" type="mh:Book"/>
      <element name="total" type="float"/>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>
  ...
</schema>
```

An XML document can use the purchaseOrder as:

```
<?xml version="1.0" encoding="UTF-8"?>
<po:purchaseOrder orderDate="2003-09-22"
 xmlns:po="http://www.Monson-Haefel.com/jwsbook">

   <accountName>Amazon.com</accountName>
   <accountNumber>923</accountNumber>
   <shipAddress>
   ...

</po:purchaseOrder>
```

The root element of a valid XML document must have a corresponding global element declaration in the schema. A schema may define more than one global element. For example, we can modify the schema for Monson-Haefel Books so that it declares two global elements: purchaseOrder and address.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
    targetNamespace="http://www.Monson-Haefel.com/jwsbook">

  <element name="address" type="mh:USAddress"/>
  <element name="purchaseOrder" type="mh:PurchaseOrder"/>
  <complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="unsignedShort"/>
      <element name="shipAddress" type="mh:USAddress"/>
      <element name="billAddress" type="mh:USAddress"/>
      <element name="book" type="mh:Book"/>
      <element name="total" type="float"/>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>
  <complexType name="USAddress">
    <sequence>
      <element name="name" type="string"/>
      <element name="street" type="string"/>
      <element name="city" type="string"/>
      <element name="state" type="string"/>
      <element name="zip" type="string"/>
    </sequence>
```

```
  </complexType>
  ...
</schema>
```

This schema **allows you to create XML documents in which the purchaseOrder element is the root, but it also allows you to create XML documents in which the address element is the root**. A valid xml document that conforms to the above schema is:

```
<?xml version="1.0" encoding="UTF-8"?>
<addr:address
    xmlns:addr="http://www.Monson-Haefel.com/jwsbook">

  <name>AMAZON.COM</name>
  <street>1850 Mercer Drive</street>
  <city>Lexington</city>
  <state>KY</state>
  <zip>40511</zip>
</addr:address>
```

By declaring two different global elements in the Monson-Haefel Books schema, you effectively create two schema-verifiable markup languages, a Purchase Order Markup Language and a U.S. Address Markup Language. The implication here is that **a single schema can be used to validate two—indeed many—different kinds of documents.**

XML schema also supports **global attributes** that can be referred to anywhere in the schema, and that provide **a consistent attribute name and type across elements.** An example of a standard global attribute is **xml:lang**, which any element can use to indicate the language used in an element's value ("es" for Spanish, "en" for English, and so on).

Local elements are those declared within the scope of a complex type. So street, city etc are local elements and orderDate is a **local attribute**. Address and purchaseOrder are global elements and xml:lang is a global attribute.

In a nutshell, **global elements and attributes** are declared as direct children of the schema element, while **local elements and attributes** are not; they are the children of complex types.

Elements can be **qualified** by a namespace, or **unqualified**; that is, that elements in an XML document may or may not require QName prefixes. **Global elements and attributes must always be qualified**, which means that in an XML instance you must prefix them to form a QName. The exception is when a global element is a member of the default namespace, in which case it does not have to be qualified with a prefix—all unqualified elements are assumed to be part of the default namespace. The default namespace does not apply to global attributes**; global attributes must always be prefixed**. While global elements and attributes must always be qualified, local elements may not need to be qualified. XML schema defines two attributes, **elementFormDefault** and **attributeFormDefault**, that determine whether local elements in an XML instance need to be qualified with a prefix or not.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook"
 elementFormDefault="qualified" >

    <element name="address" type="mh:USAddress" />

    <complexType name="USAddress">
      <sequence>
        <element name="name"    type="string" />
        <element name="street"  type="string" />
        <element name="city"    type="string" />
        <element name="state"   type="string" />
        <element name="zip"     type="string" />
      </sequence>
    </complexType>
    ...
</schema>
```

When the elementFormDefault attribute is set to "qualified", in any XML instance all the local elements in the targetNamespace must be qualified with a prefix.

The default value of the fromElementDefault and the attributeElementDefault attributes is "unqualified", so if they're not used then the local attributes and elements of targetNamespace do not need to be qualified.

To validate an XML document against one or more schemas, you need to specify which schemas to use. You do so by identifying the schemas' locations, using the **schemaLocation** attribute, which is an **XML schema-instance** attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="2003-09-22"
   xmlns="http://www.Monson-Haefel.com/jwsbook"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.Monson-Haefel.com/jwsbook
       http://www.Monson-Haefel.com/jwsbook/po.xsd">
  <accountName>Amazon.com</accountName>
  <accountNumber>923</accountNumber>
  <shipAddress>
    <name>AMAZON.COM</name>
    <street>1850 Mercer Drive</street>
    <city>Lexington</city>
    <state>KY</state>
    <zip>40511</zip>
  </shipAddress>
  <billAddress>
    <name>Amazon.com</name>
    <street>1516 2nd Ave</street>
    <city>Seattle</city>
    <state>WA</state>
    <zip>90952</zip>
  </billAddress>
  <book>
    <title>J2EE Web Services</title>
    <quantity>300</quantity>
    <wholesale-price>24.99</wholesale-price>
  </book>
  <total>8997.00</total>
</purchaseOrder>
```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" is the XML schema-instance namespace, which is defined by the XML schema specification. The XML schema specification explicitly defines a few attributes belonging to this namespace, which can be used in XML documents, including the xsi:schemaLocation attribute.

The **xsi:schemaLocation** attribute helps an XML processor locate the actual physical schema document used by the XML instance. Each schema is listed in an xsi:schemaLocation attribute as a **namespace-location pair**, which **associates a namespace with a physical URL**. the Monson-Haefel namespace, "http://www.Monson-Haefel.com/jwsbook", is associated with a schema file located at Monson-Haefel Books' Web site. You can use xsi:schemaLocation to point at several schemas if you need to.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="2003-09-22"
  xmlns="http://www.Monson-Haefel.com/jwsbook"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.Monson-Haefel.com/jwsbook
                      http://www.Monson-Haefel.com/jwsbook/po.xsd

                      http://www.w3.org/2001/XMLSchema-instance
                      http://www.w3.org/2001/XMLSchema.xsd">
```

You don't actually need to specify the XML schema-instance schema location, because it must be supported natively by any XML schema validating parser, but **you should list any other schemas used in an XML document.**

When using a local schema, specify the location relative to the directory in which the XML document is located.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="2003-09-22"
  xmlns="http://www.Monson-Haefel.com/jwsbook"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.Monson-Haefel.com/jwsbook
                      file://./monsonhaefel/jwsbook/po.xsd">
```

It's important to note that the **xsi:schemaLocation attribute is considered a "hint"** by the XML schema specification, which means that **XML parsers are not required to use the schema identified by xsi:schemaLocation, but a good parser will, and some, like Xerces-J, allow you to override the location identified by the xsi:schemaLocation attribute programmatically**—useful if you want to avoid downloading the schema every time an XML document based on it is parsed; you can use a cached copy instead of the original. (More on this in the XML Processing Section.)

The xsi:schemaLocation attribute is **usually declared in the root element of an XML document**, but it doesn't have to be. **You can declare it later in the document, as long as it's in the scope of the elements it applies to.**

Given:

```
<!-- http://www.w3.org is bound to n1 and n2 -->

<x xmlns:n1="http://www.w3.org"

   xmlns:n2="http://www.w3.org" >

  <bad a="1"     a="2" />

  <bad n1:a="1"  n2:a="2" />

</x>
```

What is wrong above:

Ans: No tag may contain two attributes which:

1. have identical names
2. have qualified names with same local part and with prefixes which have been bound to namespace names that are identical (n1 and n2 above).

However, the following is correct.

```
<!-- http://www.w3.org is bound to n1 and is the default -->

<x xmlns:n1="http://www.w3.org"

   xmlns="http://www.w3.org" >

  <good a="1"     b="2" />

  <good a="1"     n1:a="2" />

</x>
```

The second element is correct above as default namespace does not apply to attribute names. (http://www.w3.org/TR/REC-xml-names/#uniqAttrs) .

Note: To practice XML Schema related examples one free tool you can use is http://architag.com/xray/ (XRay Editor) which provides real time xml schema validation.

Advanced

Many Web service platforms map XML schema types to native primitive types, structures, and objects so that developers can manipulate XML data using constructs native to their programming environment. For example, JAX-RPC maps some of the XML schema built-in types to Java primitives, and basic complex types to Java beans (Refer JAX-RPC Mapping Section). JAX-RPC can map most derived complex types to Java beans, but not all. Similar limitations are found in other platforms like .NET and SOAP::Lite for Perl. **Most object-oriented languages do not support the full scope of inheritance defined by the XML schema specification**. For this reason, you should use **type inheritance in schemas** with care.

Complex types can use two types of inheritance: **extension** and **restriction**. Both allow you to derive new complex types from existing complex types.

1. **Extension** broadens a derived type by adding elements or attributes not present in the base type,
2. **Restriction** narrows a derived type by omitting or constraining elements and attributes defined by the base type.

An **extension** type inherits the elements and attributes of its base type, and adds new ones.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<schema
  targetNamespace="http://www.Monson-Haefel.com/jwsbook"
  xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

<element name="address" type="mh:Address"/>

<complexType name="Address">
    <sequence>
      <element name="name" type="string"/>
      <element name="street" type="string" maxOccurs="unbounded"/>
      <element name="city" type="string"/>
      <element name="country" type="string"/>
    </sequence>
    <attribute name="category" type="string" default="business"/>
  </complexType>

  <complexType name="USAddress">
    <complexContent>
      <extension base="mh:Address">
        <sequence>
          <element name="state" type="string"/>
          <element name="zip" type="string"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
  ...
</schema>
```

USAddress extends Address. USAddress type has a total of six elements (name, street, city, state, zip, and country). Base type Address can be used to create other derived types as well.

With **Restriction**, you simply redefine or omit those elements and attributes that change, and list all the other elements and attributes exactly as they were in the base type.

```
<complexType name="BriefUSAddress">
  <complexContent>
    <restriction base="mh:USAddress">
      <sequence>
        <element name="name" type="string"/>
        <element name="street" type="string"/>
        <element name="zip" type="string"/>
      </sequence>
      <attribute name="category" type="string" default="business"/>
    </restriction>
  </complexContent>
</complexType>
```

The derived type, BriefUSAddress, contains the name, street, and zip elements, but not the city, state, and country elements, because the schema simply omits them (we will first have to make these omitted elements omittable in the base type by making them optional there ie minOccurs= 0). We have redefined the occurrence constraints on the street element so that it may occur only once (restricted to occur once from unbounded in the base type).

Rules for restriction:

1. You cannot omit an element from a restriction unless the parent type declared it to be optional(minOccurs="0").
2. In addition, the derived type's occurrence constraints cannot be less strict than those of its base type.

While restriction is useful, it's used less than extension because it doesn't map as well to programming languages. For this reason, **it's risky to use restriction when defining complex types** in your XML documents.

The real power of extension, and of restriction for that matter, is that **derived types can be used polymorphically with elements of the base type**. In other words, you can use a derived type in an instance document in place of the base type specified in the schema.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
    targetNamespace="http://www.Monson-Haefel.com/jwsbook"
    elementFormDefault="qualified">

  <element name="address" type="mh:Address"/>
  <element name="purchaseOrder" type="mh:PurchaseOrder"/>
  <complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="unsignedShort"/>
      <element name="shipAddress" type="mh:Address"/>
      <element name="billAddress" type="mh:Address"/>
      <element name="book" type="mh:Book"/>
      <element name="total" type="float"/>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>
  ...
</schema>
```

Because XML schema supports polymorphism, an instance document can now use any type derived from Address for the shipAddress and billAddress elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="2003-09-22"
  xmlns="http://www.Monson-Haefel.com/jwsbook"
  xmlns:mh="http://www.Monson-Haefel.com/jwsbook"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.Monson-Haefel.com/jwsbook
                      http://www.Monson-Haefel.com/jwsbook/po2.xsd">

   <accountName>Amazon.com</accountName>
   <accountNumber>923</accountNumber>
   <shipAddress xsi:type="mh:UKAddress">
       <name>Amazon.co.uk</name>
```

```
        <street>Ridgmont Road</street>
        <city>Bedford</city>
        <country>United Kingdom</country>
        <postcode>MK43 0ZA</postcode>
    </shipAddress>
    <billAddress xsi:type="mh:BriefUSAddress">
       <name>Amazon.com</name>
       <street>1516 2nd Ave</street>
       <zip>90952</zip>
    </billAddress>
    <book>
       <title>Java Web Services</title>
       <quantity>300</quantity>
       <wholesale-price>24.99</wholesale-price>
    </book>
    <total>8997.00</total>
</purchaseOrder>
```

The **xsi:type** attribute explicitly declares the type of the element in the instance document. **Explicitly declaring an element's type with xsi:type tells the parser to validate the element against the derived type instead of the type declared in the schema.** You can think of this as "**casting**" an element, similar to casting a value in Java.
You can declare complex types to be **abstract** much as you do Java classes.

```
<complexType name="Address" abstract="true">
  <sequence>
    <element name="name" type="string"/>
    <element name="street" type="string" maxOccurs="unbounded"/>
    <element name="city" type="string"/>
    <element name="country" type="string"/>
  </sequence>
  <attribute name="category" type="string" default="business"/>
</complexType>
```

If we add **abstract="true"** to the earlier definition of Address, as in the above snippet, **it cannot be used directly in an instance document**.

You can also declare complex types to be **final**, just as Java classes can be final, **to prevent a complex type from being used as a base type for restriction or extension.** The possible values for the final attribute are "**restriction**", "**extension**", and "**#all**".

```
<complexType name="USAddress" final="extension">
  <complexContent>
    <extension base="mh:Address">
      <sequence>
        <element name="state" type="string"/>
        <element name="zip" type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

If a type is declared final="restriction", it can be extended but not restricted. If the final attribute equals "#all", the type cannot used as a base type at all.

XML schema allows us to **create new simple types that are derived from existing simple types** in order to constrain further the range of possible values that a simple type may represent.
To constrain data in the total element to this range, we restrict the built-in float type to create a new type called Total.

```
<simpleType name="Total">
  <restriction base="float">
    <minInclusive value="0"/>
    <maxExclusive value="100000"/>
  </restriction>
</simpleType>
```

The restriction element for simple types contains one or more facet elements. A facet is an element that represents an aspect or characteristic of the built-in type that can be modified. For example, the Total simple type declares that its **minInclusive** facet is "0" and its **maxExclusive** facet is "100000", thereby specifying that values held by elements of this type must be at least zero and less then 100,000.
Simple type facets are:

maxInclusive	
maxExclusive	
minInclusive	
minExclusive	
pattern	Regexp for value
enumeration	Set of allowed values
whiteSpace	Constrain.

```
<complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="unsignedShort"/>
      <element name="shipAddress" type="mh:Address"/>
      <element name="billAddress" type="mh:Address"/>
      <element name="book" type="mh:Book"/>
      <element name="total" type="mh:Total"/>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>
```

In XML schema, a regular expression is used to verify that the contents of an element or attribute adhere to a predefined character pattern.

```
<simpleType name="Total">
   <restriction base="float">
      <pattern value="[0-9]+\.[0-9]{2}" />
      <minInclusive value="0"/>
      <maxExclusive value="100000" />
   </restriction>
</simpleType>
```

The regular expression "[0-9]+\.[0-9]{2}" specifies that there must be at least one digit before the decimal point and exactly two digits following the decimal point.

The pattern facet is commonly applied to string types. For example, we can define a USZipCode type that restricts a string value either to five digits, or to nine digits with the last four set off by a hyphen.

```
<simpleType name="USZipCode">
   <restriction base="string">
      <pattern value="[0-9]{5}(-[0-9]{4})?" />
   </restriction>
</simpleType>
<complexType name="USAddress" final="extension">
  <complexContent>
    <extension base="mh:Address">
      <sequence>
        <element name="state" type="string"/>
        <element name="zip" type="mh:USZipCode"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

The **enumeration** facet restricts the value of any simple type (except boolean) to a set of distinct values.

```
<simpleType name="USState">
   <restriction base="string">
      <enumeration value="AK"/> <!-- Alaska   -->
      <enumeration value="AL"/> <!-- Alabama  -->
      <enumeration value="AR"/> <!-- Arkansas -->
      <!-- and so on -->
   </restriction>
</simpleType>
```

Union and List types:

Union and **list types** should be used with care, especially when interoperability across programming environments is important.

A **list** is a sequence of simple-type values separated by white space.

```
<simpleType name="USStateList">
    <list itemType="mh:USState"/>
</simpleType>
```

In an instance document, an element of the USStateList type could contain zero or more state abbreviations separated by spaces.

```
<list-of-states>CA  NY  FL  AR  NH</list-of-states>
```

A **union** is a set of valid simple types. The union type USStateOrZipUnion allows the value to be either a USStateList type or a USZipCode type.

```
<simpleType name="USStateOrZipUnion">
   <union memberTypes="mh:USStateList mh:USZipCode"/>
</simpleType>
```

An element or attribute based on this type can hold either a USStateList or a USZipCode. It cannot, however, contain a mix of values.

```
<!-- valid use of union type -->
<location>CA NJ AK</location>
<location>94108</location>

<!-- invalid use of union type -->
<location>94108 CA 554011 MN</location>
```

Anonymous types:

You can **combine an element declaration with a complex or simple type declaration to create an anonymous type.** An anonymous type is not named and cannot be referred to outside the element that declares it.

```
<element name="purchaseOrder"
type="mh:PurchaseOrder"/>

<complexType name="PurchaseOrder">
  <sequence>
    <element name="accountName"
type="string"/>
    <element name="accountNumber"
type="unsignedShort"/>
    <element name="shipAddress"
type="mh:Address"/>
    <element name="billAddress"
type="mh:Address"/>
    <element name="book" type="mh:Book"/>
    <element name="total" type="mh:Total"/>
  </sequence>
  <attribute name="orderDate" type="date"/>
</complexType>
```

```
<element name="purchaseOrder">
  <complexType>
    <sequence>
      <element name="accountName"
type="string"/>
      <element name="accountNumber"
type="unsignedShort"/>
      <element name="shipAddress"
type="mh:Address"/>
      <element name="billAddress"
type="mh:Address"/>
      <element name="book" type="mh:Book"/>
      <element name="total"
type="mh:Total"/>
    </sequence>
    <attribute name="orderDate"
type="date"/>
  </complexType>
</element>
```

The PurchaseOrder type is not very useful outside the purchaseOrder element, so we can combine the two declarations into one (as shown in the rhs above).

Notice that the element declaration doesn't need a type attribute because it defines its own type, and that the complexType declaration doesn't declare a name attribute; it's anonymous.

Anonymous types can be nested:

```
<element name="purchaseOrder">
  <complexType>
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="unsignedShort"/>
      <element name="shipAddress" type="mh:Address"/>
      <element name="billAddress" type="mh:Address"/>
      <element name="book">
        <complexType>
          <sequence>
            <element name="title" type="string"/>
            <element name="quantity" type="unsignedShort"/>
            <element name="wholesale-price" type="float"/>
          </sequence>
        </complexType>
      </element>
      <element name="total">
        <simpleType>
          <restriction base='float'>
```

```
            <minInclusive value="0"/>
            <maxExclusive value="100000"/>
            <pattern value="[0-9]+\.[0-9]{2}"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>
</element>
```

Anonymous types are not reusable, **and you should employ them only when you know that the type won't be useful in other schemas**. For example, the book and total elements are based on anonymous types that might well be useful in other circumstances; you might benefit from defining them separately as named types.

Importing and Including Schemas:

An **import** allows you to combine schemas from different namespaces, while an **include** lets you combine schemas from the same namespace.

Addr.xsd:

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
   targetNamespace="http://www.Monson-Haefel.com/addr"
   xmlns:addr="http://www.Monson-Haefel.com/addr"
   xmlns="http://www.w3.org/2001/XMLSchema">

  <element name="address" type="addr:Address"/>

  <simpleType name="USZipCode">
    <restriction base="string">
      <pattern value="[0-9]{5}(-[0-9]{4})?"/>
    </restriction>
  </simpleType>

  <simpleType name="USState">
    <restriction base="string">
      <enumeration value="AK"/> <!-- Alaska   -->
      <enumeration value="AL"/> <!-- Alabama  -->
      <enumeration value="AR"/> <!-- Arkansas -->
      <!-- and so on -->
    </restriction>
  </simpleType>

  <complexType name="Address" abstract="true">
    <sequence>
      <element name="name" type="string"/>
      <element name="street" type="string" maxOccurs="unbounded"/>
      <element name="city" type="string"/>
      <element name="country" type="string"/>
    </sequence>
    <attribute name="category" type="string" default="business"/>
  </complexType>

  <complexType name="USAddress" final="extension">
    <complexContent>
      <extension base="addr:Address">
        <sequence>
          <element name="state" type="addr:USState"/>
          <element name="zip" type="addr:USZipCode"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="UKAddress">
```

```
    <complexContent>
      <extension base="addr:Address">
        <sequence>
          <element name="postcode" type="string"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="BriefUSAddress">
    <complexContent>
      <restriction base="addr:USAddress">
        <sequence>
          <element name="name" type="string"/>
          <element name="street" type="string"/>
          <element name="zip" type="addr:USZipCode"/>
        </sequence>
        <attribute name="category" type="string" default="business"/>
      </restriction>
    </complexContent>
  </complexType>

</schema>
```

The targetNamespace of the Address Markup schema is "http://www.Monson-Haefel.com/jwsbook/ADDR", which is a separate namespace from that of the purchase-order elements. Because the PurchaseOrder type depends on the Address type, we'll need to import the Address Markup schema into the Purchase Order schema (**po.xsd**):

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
  targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns:addr="http://www.Monson-Haefel.com/jwsbook/ADDR"
  xmlns="http://www.w3.org/2001/XMLSchema">

  <import namespace="http://www.Monson-Haefel.com/jwsbook/ADDR"
   schemaLocation="http://www.Monson-Haefel.com/jwsbook/addr.xsd" />
  <element name="purchaseOrder" type="po:PurchaseOrder"/>
  <simpleType name="Total">
    <restriction base="float">
      <minInclusive value="0.00"/>
      <maxExclusive value="100000.00"/>
      <pattern value="[0-9]+\.[0-9]{2}"/>
    </restriction>
  </simpleType>
  <complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="unsignedShort"/>
      <element name="shipAddress" type="addr:Address"/>
      <element name="billAddress" type="addr:Address"/>
      <element name="book" type="po:Book"/>
      <element name="total" type="po:Total"/>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>
  <complexType name="Book">
    <sequence>
      <element name="title" type="string"/>
      <element name="quantity" type="unsignedShort"/>
      <element name="wholesale-price" type="float"/>
    </sequence>
  </complexType>
</schema>
```

Including is useful when a schema becomes large and difficult to maintain. The Purchase Order schema has not become that unwieldy, but just as an example, we could place the definitions of the Total and Book types into a separate schema, then use an include element to combine them with the Purchase Order schema. **PO.XSD:**

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
```



```
  targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns="http://www.w3.org/2001/XMLSchema">

  <simpleType name="Total">
    <restriction base="float">
      <minInclusive value="0.00"/>
      <maxExclusive value="100000.00"/>
      <pattern value="[0-9]+\.[0-9]{2}"/>
    </restriction>
  </simpleType>

  <complexType name="Book">
    <sequence>
      <element name="title" type="string"/>
      <element name="quantity" type="unsignedShort"/>
      <element name="wholesale-price" type="float"/>
    </sequence>
  </complexType>
</schema>
```

We can combine these two schemas using an include statement. **PO2.XSD:**

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
  targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns:addr="http://www.Monson-Haefel.com/jwsbook/ADDR"
  xmlns="http://www.w3.org/2001/XMLSchema">

 <include
  schemaLocation="http://www.Monson-Haefel.com/jwsbook/po.xsd" />

 <import namespace="http://www.Monson-Haefel.com/jwsbook/ADDR"
  schemaLocation="http://www.Monson-Haefel.com/jwsbook/addr.xsd" />

  <element name="purchaseOrder" type="po:PurchaseOrder"/>
  <complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="unsignedShort"/>
      <element name="shipAddress" type="addr:Address"/>
      <element name="billAddress" type="addr:Address"/>
      <element name="book" type="po:Book"/>
      <element name="total" type="po:Total"/>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>
</schema>
```

Conclusion:

Although XML schema is the basis of Web services in J2EE, it's not the only XML schema language available today. In fact there are a couple of other schema languages, including DTDs, Schematron, RELAX-NG, and a few others. Of these, Schematron appears to be the best complement to XML schema, or at least to offer validation checks that XML schema cannot duplicate. Schematron is based on Xpath and XSLT and is used for defining context-dependent rules for validating XML documents. For example, in the purchase-order document you could use Schematron to ensure that the value of the total element equals the value of the quantity element multiplied by the value of the wholesale-price element.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="2003-09-22"
     xmlns:mh="http://www.Monson-Haefel.com/jwsbook">
                  ...
                  <book>
                       <title>J2EE Web Services</title>
                       <quantity>300</quantity>
                       <wholesale-price>24.99</wholesale-price>
                  </book>
 <total>7485.00</total>
```



```
</purchaseOrder>
```

XML schema does not provide this type of business-rule support, **so you may well want to use Schematron in combination with XML schema to provide more robust validation.**

For Revision: Here are some xsd's which finally are imported and included to make the complete PO.xsd.

1. **addr.xsd**

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
   targetNamespace="http://www.Monson-Haefel.com/jwsbook/ADDR"
   xmlns:addr="http://www.Monson-Haefel.com/jwsbook/ADDR"
   xmlns="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified">

  <element name="address" type="addr:Address"/>

  <simpleType name="USZipCode">
    <restriction base="string">
      <pattern value="[0-9]{5}(-[0-9]{4})?"/>
    </restriction>
  </simpleType>

  <simpleType name="USState">
    <restriction base="string">
      <enumeration value="AK"/> <!-- Alaska   -->
      <enumeration value="AL"/> <!-- Alabama  -->
      <enumeration value="AR"/> <!-- Arkansas -->
      <enumeration value="KY"/> <!-- Kenttucky -->
      <!-- and so on -->
    </restriction>
  </simpleType>

  <complexType name="Address" abstract="true">
    <sequence>
      <element name="name" type="string"/>
      <element name="street" type="string" maxOccurs="unbounded"/>
      <element name="city" type="string" minOccurs="0"/>
      <element name="country" type="string" minOccurs="0"/>
    </sequence>
    <attribute name="category" type="string" default="business"/>
  </complexType>

  <complexType name="USAddress" final="extension">
    <complexContent>
      <extension base="addr:Address">
        <sequence>
          <element name="state" type="addr:USState" minOccurs="0"/>
          <element name="zip" type="addr:USZipCode"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="UKAddress">
    <complexContent>
      <extension base="addr:Address">
        <sequence>
          <element name="postcode" type="string"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="BriefUSAddress">
    <complexContent>
      <restriction base="addr:USAddress">
        <sequence>
          <element name="name" type="string"/>
          <element name="street" type="string"/>
          <element name="zip" type="addr:USZipCode"/>
        </sequence>
```

```
        <attribute name="category" type="string" default="business"/>
      </restriction>
    </complexContent>
  </complexType>
</schema>
```

2. **book.xsd:**

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
  targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <simpleType name="Total">
    <restriction base="float">
      <minInclusive value="0.00"/>
      <maxExclusive value="100000.00"/>
      <pattern value="[0-9]+\.[0-9]{2}"/>
    </restriction>
  </simpleType>

  <complexType name="Book">
    <sequence>
      <element name="title" type="string"/>
      <element name="quantity" type="unsignedShort"/>
      <element name="wholesale-price" type="float"/>
    </sequence>
  </complexType>
</schema>
```

3. po.xsd: You will need to change the schemaLocation path to refer to the actual path in your setup for the book.xsd and addr.xsd files.

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
  targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
  xmlns:addr="http://www.Monson-Haefel.com/jwsbook/ADDR"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

 <include
  schemaLocation="file:/E:/Work/staffing/staffing-war/web/WEB-INF/book.xsd" />

 <import namespace="http://www.Monson-Haefel.com/jwsbook/ADDR"
  schemaLocation="file:/E:/Work/staffing/staffing-war/web/WEB-INF/addr.xsd" />

  <element name="purchaseOrder" type="po:PurchaseOrder"/>
  <complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="unsignedShort"/>
      <element name="shipAddress" type="addr:Address"/>
      <element name="billAddress" type="addr:Address"/>
      <element name="book" type="po:Book"/>
      <element name="total" type="po:Total"/>
    </sequence>
    <attribute name="orderDate" type="date" use="required"/>
  </complexType>
</schema>
```

4. And here's the **xml document instance** conforming to the above xsd:

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

<po:purchaseOrder xmlns='http://www.Monson-Haefel.com/jwsbook/PO'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://www.Monson-Haefel.com/jwsbook/PO
                      file:/E:/Work/staffing/staffing-war/web/WEB-INF/po.xsd'
  xmlns:po='http://www.Monson-Haefel.com/jwsbook/PO'
  xmlns:addr='http://www.Monson-Haefel.com/jwsbook/ADDR'
  orderDate="2003-09-22">
<!-- You will need to modify the schemaLocation path above to
       refer to the actual path for the referenced po.xsd.
```

```
-->
    <po:accountName>Amazon.com</po:accountName>
    <po:accountNumber>923</po:accountNumber>
    <po:shipAddress xsi:type="addr:USAddress">
        <addr:name>AMAZON.COM</addr:name>
        <addr:street>1850 Mercer Drive</addr:street>
        <addr:city>Lexington</addr:city>
        <addr:country>USA</addr:country>
        <addr:state>KY</addr:state>
        <addr:zip>40511</addr:zip>
    </po:shipAddress>
    <po:billAddress xsi:type="addr:BriefUSAddress" category="business">
        <addr:name>AMAZON.COM</addr:name>
        <addr:street>1516 2nd Ave</addr:street>
        <addr:zip>90952</addr:zip>
    </po:billAddress>
    <po:book>
        <po:title>J2EE Webservices</po:title>
        <po:quantity>300</po:quantity>
        <po:wholesale-price>24.99</po:wholesale-price>
    </po:book>
    <po:total>8997.00</po:total>
</po:purchaseOrder>
```

SOAP 1.1 Web Service Standards (Objective 2)

[RMH – 4, Appendix E and D]
2.1 List and describe the encoding types used in a SOAP message.
2.2 Describe how SOAP message header blocks are used and processed.
2.3 Describe the function of each element contained in a SOAP message, the SOAP binding to HTTP, and how to represent faults that occur when processing a SOAP message.
2.4 Create a SOAP message that contains an attachment.
2.5 Describe the restrictions placed on the use of SOAP by the WS-I Basic Profile 1.0a.
2.6 Describe the function of SOAP in a Web service interaction and the advantages and disadvantages of using SOAP messages.

SOAP(Objective 2.1/2/3/5/6)

SOAP was originally an acronym for Simple Object Access Protocol. (Now it's just a name.) SOAP is just another XML markup language accompanied by rules that dictate its use. SOAP has a clear purpose: exchanging data over networks. Specifically, it concerns itself with encapsulating and encoding XML data and defining the rules for transmitting and receiving that data. In a nutshell, **SOAP is a network application protocol**.

A SOAP XML document instance, which is called a SOAP message (also called SOAP envelope) is usually carried as the payload of some other network protocol. For example, the most common way to exchange SOAP messages is via HTTP (HyperText Transfer Protocol), used by Web browsers to access HTML Web pages. The big difference is that you don't view SOAP messages with a browser as you do HTML. SOAP messages are exchanged between applications on a network and are not meant for human consumption. HTTP is just a convenient way of sending and receiving SOAP messages.
SOAP messages can also be carried by e-mail using SMTP (Simple Mail Transfer Protocol) and by other network protocols, such as FTP (File Transfer Protocol) and raw TCP/IP (Transmission Control Protocol/Internet Protocol). At this time, however, the WS-I Basic Profile 1.0 sanctions the use of SOAP only over HTTP.

Web services can use One-Way messaging or Request/Response messaging. Advantages of SOAP:

1. The SOAP message format is defined by an XML schema, which exploits XML namespaces to make SOAP very extensible.
2. Another advantage of SOAP is its explicit definition of an HTTP binding, a standard method for **HTTP tunneling.** HTTP tunneling is the process of hiding another protocol inside HTTP

messages in order to pass through a firewall unimpeded. Firewalls will usually allow HTTP traffic through port 80, but will restrict or prohibit the use of other protocols and ports.

Basic Structure of SOAP

SOAP has its own XML schema, namespaces, and processing rules.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO">
      <po:accountName>Amazon.com</po:accountName>
      <po:accountNumber>923</po:accountNumber>
      <po:address>
        <po:name>AMAZON.COM</po:name>
        <po:street>1850 Mercer Drive</po:street>
        <po:city>Lexington</po:city>
        <po:state>KY</po:state>
        <po:zip>40511</po:zip>
      </po:address>
      <po:book>
        <po:title>J2EE Web Services</po:title>
        <po:quantity>300</po:quantity>
        <po:wholesale-price>24.99</po:wholesale-price>
      </po:book>
     </po:purchaseOrder>
   </soap:Body>
</soap:Envelope>
```

A SOAP message that contains an instance of Purchase Order markup. A SOAP message may have an XML declaration, which states the version of XML used and the encoding format. If an xml declaration is used, the version of XML must be 1.0 and the encoding must be either UTF-8 or UTF-16. **An XML declaration isn't mandatory. Web services are required to accept messages with or without them.**

Envelope may contain an optional **Header** element, and must contain a **Body** element. If you use a Header element, it must be the immediate child of the Envelope element, and precede the Body element. The Body element contains, in XML format, the actual application data being exchanged between applications.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <!-- Header blocks go here -->
  </soap:Header>
  <soap:Body>
    <!-- Application data goes here -->
  </soap:Body>
</soap:Envelope>
```

A SOAP message adheres to the **SOAP 1.1 XML schema**, which requires that elements and attributes be fully qualified (use prefixes or default namespaces). A SOAP message may have a single Body element preceded, optionally, by one Header element. The Envelope element cannot contain any other children.
The application data could be an arbitrary XML element like a purchaseOrder, or an element that maps to the arguments of a procedure call.
The **Header** element contains **information about the message** (describe security credentials, transaction IDs, routing instructions, debugging information, payment tokens, or any other information about the message that is important in processing the data in the Body element.), in the form of one or more distinct XML elements, each of which describes some aspect or quality of service associated with the message.
You can place any number of header blocks in the Header element. For example, a SOAP message with an XML Digital signature and a message id header blocks:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id">
```

```
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <sec:Signature >
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm=
           "http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
          <ds:SignatureMethod Algorithm=
           "http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
          <ds:Reference URI="#Body">
            <ds:Transforms>
              <ds:Transform Algorithm=
               "http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm=
             "http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>u29dj93nnfksu937w93u8sjd9=
            </ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>CFFOMFCtVLrklR…</ds:SignatureValue>
      </ds:Signature>
    </sec:Signature>
  </soap:Header>
  <soap:Body sec:id="Body">
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

Each header block element must be identified by its own namespace. Each header block in the Header element should have its own namespace. This is particularly important because namespaces help SOAP applications identify header blocks and process them separately. A variety of "standard" header blocks that address topics such as security, transactions, and other qualities of service are in development by several organizations, including W3C, OASIS, IETF, Microsoft, BEA, and IBM. All of the proposed standards define their own namespaces and XML schemas, as well as processing requirements

SOAP Namespaces

A SOAP message may include several different XML elements in the Header and Body elements, and to avoid name collisions each of these elements should be identified by a unique namespace.

Namespace **xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"** `defines the namespace of the standard SOAP elements`–Envelope, Header, `and` Body. SOAP messages must declare the namespace of the Envelope element to be the standard SOAP 1.1 envelope namespace, "http://schemas.xmlsoap.org/soap/envelope/". If a SOAP application receives a message based on some other namespace, it must generate a fault. **[TBD – Copy notes from the other doc at home.]**

SOAP Headers

The SOAP specification defines rules by which header blocks must be processed in the **message path**. The message path is simply the route that a SOAP message takes from the initial sender to the ultimate receiver. It includes processing by any intermediaries. The SOAP rules specify which nodes must process particular header blocks and what should be done with header blocks after they've been processed.

SOAP is a protocol used to exchange messages between **SOAP applications** on a network, usually an intranet or the Internet. A SOAP application is simply any piece of software that generates or processes SOAP messages. The application sending a SOAP message is called the **sender**, and the application receiving it is called the **receiver**. All SOAP messages start with the **initial sender**, which creates the SOAP message, and end with the **ultimate receiver**. As a SOAP message travels along the message path, its header blocks may be intercepted and processed by any number of **SOAP intermediaries** along the way. A SOAP intermediary is both a receiver and a sender. It receives a SOAP message, processes one or more of the header blocks, and sends it on to another SOAP application. The applications along the message path (the initial sender, intermediaries, and ultimate receiver) are also called **SOAP nodes**. Intermediaries in a SOAP message path must not modify the application-specific contents of the SOAP Body element, but they may, and often do, manipulate the SOAP header blocks. When processing a header

block, each node reads, acts on, and removes the header block from the SOAP message before sending it along to the next receiver. Any node in a message path may also add a new header block to a SOAP message. SOAP 1.1 applications use the **actor** attribute to identify the nodes that should process a specific header block. SOAP also employs the **mustUnderstand** attribute to indicate whether a node processing the block needs to recognize the header block and know how to process it.

The actor attribute

You use an actor attribute to identify a function to be performed by a particular node. a node can play one or more roles in a SOAP message path. **You must identify the roles a node will play by declaring an actor attribute.** In SOAP 1.2 this attribute has been renamed role.
The actor attribute uses a URI (Uniform Resource Identifier) to **identify the role that a node must perform in order to process that header block. When a node receives a SOAP message, it examines each of the header blocks to determine which ones are targeted to roles supported by that node.** For example, every SOAP message processed by a Monson-Haefel Books Web service might pass through a logging intermediary, a code module that records information about incoming messages in a log, to be used for debugging. The logging module represents a particular role played by a node. A **node may have many modules that operate on a message, and therefore many roles, so every node in a message path may identify itself with several different roles**.
The receiving node will first determine whether it plays the role designated by the actor attribute, and then choose the correct code module to process the header block, based on the XML namespace of the header block. Therefore, the receiving node must recognize the role designated by the actor attribute assigned to a header block, as well as the XML namespace associated with the header block. For example, the actor attribute identifies the logger role with the URL "http://www.Monson-Haefel.com/logger". A node that's intended to perform the logger role will look for header blocks where that URL is the value of the actor attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <mi:message-id soap:actor="http://www.Monson-Haefel.com/logger" >
      11d1def534ea:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
    <proc:processed-by soap:actor=" http://schemas.xmlsoap.org/soap/actor/next">
      <node>
        <time-in-millis>1013694680000</time-in-millis>
        <identity>http://www.customer.com</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
      <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

Only those nodes in the message path that identify themselves with the actor value "http://www.Monson-Haefel.com/logger" will process the message-id header block; all other nodes will ignore it.

In addition to custom URIs like "http://www.Monson-Haefel.com/logger", SOAP identifies two standard roles for the actor attribute: **next** and **ultimate receiver**. The next role indicates that the next node in the message path must process the header. The next role has a designated URI, which must be used as the value of the actor attribute: **"http://schemas.xmlsoap.org/soap/actor/next".** The ultimate receiver role indicates that only the ultimate receiver of the message should process the header block. The protocol doesn't specify an explicit URI for this purpose; **it's the absence of an actor attribute in the header block that signals that the role is ultimate receiver**.
In the above example, the next receiver in the message path, no matter what other purpose it may serve, should process the processed-by header block. If an intermediary node in the message path supports the logger role, then it should process the processed-by header block in addition to the message-id header block.

When a node processes a header block, it must remove it from the SOAP message. The node may also add new header blocks to the SOAP message. **SOAP nodes frequently feign removal of a header block by simply modifying it**, which is logically the same as removing it, modifying it, and then adding it back to the SOAP message—a little trick that allows a node to adhere to the SOAP specifications while propagating header blocks without losing any data. For example, the logger node may **remove the message-id header block**, but we don't want it to remove the processed-by header block, because we want all the nodes in the message path to add information to it. Therefore, the logger node will **simply add its own data to the processed-by header block**, then pass the SOAP message to the next node in the message path, as shown in the below example, after the logger node has processed the SOAP message.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <proc:processed-by
     soap:actor="http://schemas.xmlsoap.org/soap/actor/next">
      <node>
        <time-in-millis>1013694680000</time-in-millis>
        <identity>http://www.customer.com</identity>
      </node>
      <node>
        <time-in-millis>1013694680010</time-in-millis>
        <identity>http://www.Monson-Haefel.com/sales</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
      <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

The mustUnderstand attribute

We don't always know whether nodes can process header blocks correctly.
Header blocks may indicate **whether processing is mandatory or not** by using the mustUnderstand attribute. The mustUnderstand attribute can have the value of either "1" or "0", to represent true and false, respectively. **According to the BP, SOAP applications must set the mustUnderstand attribute to "1" or "0"—"true" and "false" are not allowed.** If the mustUnderstand attribute is omitted, then its default value is "0" (false). When a header block has a mustUnderstand attribute equal to "1", it's called a mandatory header block. SOAP nodes must be able to process any header block that is marked as mandatory if they play the role specified by the actor attribute of the header block.
If a node doesn't understand a mandatory header block, it must generate a SOAP fault (similar to a remote exception in Java) and discard the message; it must not forward the message to the next node in the message path – the BP 1.0 requires this behavior. A SOAP receiver is required to generate a fault with the fault code MustUnderstand if it fails to understand a mandatory header block. Whether or not a fault is sent back to the sender depends on whether the messaging exchange pattern (MEP) is One-Way or Request/Response**. If a SOAP application uses Request/Response messaging, it's required to send a SOAP fault back to the sender; if it uses One-Way messaging, it's not**.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <proc:processed-by
     soap:actor="http://schemas.xmlsoap.org/soap/actor/next"
     soap:mustUnderstand="1" >
      <node>
        <time-in-millis>1013694684723</time-in-millis>
        <identity>http://local/SOAPClient2</identity>
      </node>
      <node>
        <time-in-millis>1013694685023</time-in-millis>
        <identity>http://www.Monson-Haefel.com/logger</identity>
```

```
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

If a node performs the role declared by a non-mandatory header block, and an application fails to understand the header (it doesn't recognize the XML structure or the namespace), it must remove the header block.

Receivers should not reject a message simply because a header block targeted at some other node has not been processed (and removed). In other words, receivers should not attempt to determine whether a message was successfully processed by previous nodes in the path based on which header blocks are present. Because nodes are required to "mind their own business," message paths can evolve and are very dynamic. Adding new intermediaries (or removing them) doesn't require adjustments to every other node in a message path.

WS-I Conformance Header Block

Although the BP doesn't endorse any particular type of header block, it does specify an **optional** conformance header block that indicates that the SOAP message complies with the BP.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  <soap:Header>
    <wsi:Claim conformsTo="http://ws-i.org/profiles/basic/1.0"
     xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/" />
  </soap:Header>
  <soap:Body sec:id="Body">
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

A SOAP message can declare a separate Claim header for each profile it adheres to. At the time of this writing the WS-I has defined only the Basic Profile 1.0, but it's expected to release other profiles. In the future, it's possible that a SOAP message will conform to both the Basic Profile 1.0 and other, as yet undefined, profiles. The Claim header block is always considered optional, so its **mustUnderstand attribute must not be "1".** You cannot require receivers to process a Claim header block.

Conclusion

SOAP headers are a very powerful way of extending the SOAP protocol. Standards bodies frequently drive the definition of general-purpose SOAP header blocks. These organizations are primarily concerned with header blocks that address qualities of service, such as security, transactions, message persistence, and routing. OASIS, for example, is defining the WS-Security SOAP headers used with XML digital signatures—an XML security mechanism. The BP does not address any of these potential standards, but WS-I will eventually create more advanced profiles that incorporate many of the proposals evolving at OASIS, W3C, Microsoft, IBM, and other organizations - WS-I has started defining the WS-I Security Profile based on the OASIS WS-Security standard.

SOAP Body

All SOAP messages must contain exactly one Body element. The Body element contains either the application-specific data or a fault message. Application-specific data is the information that we want to exchange with a Web service. It can be arbitrary XML data or parameters to a procedure call. A fault message is used only when an error occurs. The receiving node that discovers a problem, such as a processing error or a message that's improperly structured, sends it back to the sender just before it in the message path. **A SOAP message may carry either application-specific data or a fault, but not both.** Intermediary nodes in the message path may view the Body element, but they should not alter its contents in any way. Only the ultimate receiver should alter the contents of the Body element.

Neither SOAP 1.1 nor the BP explicitly prohibits intermediaries from modifying the contents of the Body element. As a result, the ultimate receiver has no way of knowing if the application-specific data has changed somewhere along the message path. SOAP 1.2 reduces this uncertainty by explicitly prohibiting certain intermediaries, called forwarding intermediaries, from changing the contents of the Body element and recommending that all other intermediaries, called active intermediaries, use a header block to document any changes to the Body element.

SOAP Messaging Modes

Except in the case of fault messages, SOAP does not specify the contents of the Body element (although it does specify the general structure of RPC-type messages). As long as the Body contains well-formed XML, the application-specific data can be anything. The Body element may contain any XML element or it can be empty.

Although SOAP supports four modes of messaging (RPC/Literal, Document/ Literal, RPC/Encoded, and Document/Encoded) the BP permits the use of RPC/ Literal or Document/Literal only. The RPC/Encoded and Document/Encoded modes are explicitly prohibited.

A messaging mode is defined by its **messaging style** (**RPC** or **Document**) and its encoding style. There are **two common types of encoding** used in SOAP messaging: **SOAP encoding** as described in Section 5 of the SOAP 1.1 specification, and **Literal encoding**. **SOAP encoding is not supported by WS-I-conformant Web services because it causes significant interoperability problems.**[BP] **The term "Literal" means that the XML document fragment can be validated against its XML schema.** The RPC/Encoded relies on built-in XML Schemas and is designed to represent a graph of objects. Document/Encoded messaging is not supported by J2EE Web services and is rarely used in practice. Both these SOAP encoded messaging modes cause interoperability issues.

Document/Literal

In the Document/Literal mode of messaging, a SOAP Body element contains an XML document fragment, a well-formed XML element that contains arbitrary application data (text and other elements) that belongs to an XML schema and namespace separate from the SOAP message's. For example, a set of XML elements that describes a purchase order, embedded within a SOAP message, is considered an XML document fragment.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <!-- Header blocks go here -->
  </soap:Header>
  <soap:Body>
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO">
      <po:accountName>Amazon.com</po:accountName>
      <po:accountNumber>923</po:accountNumber>
      ...
      <po:book>
        <po:title>J2EE Web Services</po:title>
        <po:quantity>300</po:quantity>
        <po:wholesale-price>24.99</po:wholesale-price>
      </po:book>
    </po:purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

RPC/Literal

The RPC/Literal mode of messaging enables SOAP messages to model calls to procedures or method calls with parameters and return values. In RPC/Literal messaging, the contents of the Body are always formatted as a struct. An RPC request message contains the method name and the input parameters of the call. An RPC response message contains the return value and any output parameters (or a fault). In many

cases, RPC/Literal messaging is used to expose traditional components (servlets, stateless session beans, CORBA object, Java RMI object, DCOM etc) as Web services.

```
package com.jwsbook.soap;
import java.rmi.RemoteException;

public interface BookQuote extends java.rmi.Remote {
    // Get the wholesale price of a book
    public float getBookPrice(String ISBN)
      throws RemoteException, InvalidISBNException;
}
```

This JAX-RPC service endpoint can use the RPC/Literal mode of messaging. The Web service uses two SOAP messages: a request message and a reply message. The request message is sent from an initial sender to the Web service and contains the method name, getBookPrice, and the ISBN string parameter. The reply message is sent back to the initial sender and contains the price of the book as a float value.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

And the response is:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
   <soap:Body>
      <mh:getBookPriceResponse>
          <result>24.99</result>
      </mh:getBookPriceResponse>
   </soap:Body>
</soap:Envelope>
```

Unlike Document/Literal messaging, which makes no assumptions about the type and structure of elements contained in the Body of the message—except that the document fragment adheres to some XML schema—RPC/Literal messages carry a simple set of arguments.

It's important to understand that RPC/Literal and Document/Literal may be indistinguishable from the perspective of a developer using tools like JAX-RPC, because JAX-RPC can present procedure-call semantics for both RPC/Literal and Document/Literal. So its debatable, Why use RPC/Literal when you can use Document/Literal, which is arguably simpler to implement in some respects, and can exploit XML schema validation?

SOAP Faults

SOAP faults are returned to the receiver's immediate sender. When that sender receives the fault message, it may take some action, such as undoing operations, and may send another fault further upstream to the next sender if there is one.

A SOAP message that contains a Fault element in the Body is called a **fault message**. A fault message is analogous to a Java exception; it's generated when an error occurs. Fault messages are used in Request/Response messaging. Faults are caused by improper message formatting, version mismatches, trouble processing a header, and application-specific errors.

When a fault message is generated, the Body of the SOAP message must contain only a single Fault element and nothing else. The Fault element itself **must contain** a **faultcode** element and a **faultstring** element, and **optionally** **faultactor** and **detail** elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
  <soap:Body>
```



```
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring>
        The ISBN value contains invalid characters
      </faultstring>
      <faultactor>http://www.xyzcorp.com</faultactor>
      <detail>
        <mh:InvalidIsbnFaultDetail>
          <offending-value>19318224-D</offending-value>
          <conformance-rules>
            The first nine characters must be digits. The last
            character may be a digit or the letter 'X'. Case is
            not important.
          </conformance-rules>
        </mh:InvalidIsbnFaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Children of the Fault element may be **unqualified**. It's forbidden for the Fault element to contain any immediate child elements other than faultcode, faultstring, faultactor, and detail.

SOAP Standard Fault Codes:

1. Client
2. Server
3. VersionMismatch
4. MustUnderstand

BP dictates that you must use the above four fault codes only.

The **Client** fault code signifies that the node that sent the SOAP message caused the error. Basically, if the receiver cannot process the SOAP message because there is something wrong with the message or its data, it's considered the fault of the client, the sender. The receiving node generates a Client fault if the message is not well formed, or contains invalid data, or lacks information that was expected, like a specific header. When a node receives a fault message with a Client code, it should not attempt to resend the same message. It should take some action to correct the problem or abort completely.

The **Server** fault code indicates that the node that received the SOAP message malfunctioned or was otherwise unable to process the SOAP message. This fault is a reflection of an error by the receiving node (either an intermediary or the ultimate receiver) and doesn't point to any problems with the SOAP message itself. In this case the sender can assume the SOAP message to be correct, and can redeliver it after pausing some period of time to give the receiver time to recover.

A receiving node generates a **VersionMismatch** fault when it doesn't recognize the namespace of a SOAP message's Envelope element. For example, a SOAP 1.1 node will generate a fault with a VersionMismatch code if it receives a SOAP 1.2 message, because it finds an unexpected namespace in the Envelope. The VersionMismatch fault applies only to the namespace assigned to the Envelope, Header, Body, and Fault elements. It does not apply to other parts of the SOAP message, like the header blocks, XML document version, or application-specific elements in the Body.

When a node receives a SOAP message, it must examine the Header element to determine which header blocks, if any, are targeted at that node. If a header block is targeted at the current node (via the actor attribute) and sets the mustUnderstand attribute equal to "1", then the node is required to know how to process the header block. If the node doesn't recognize the header block, it must generate a fault with the **MustUnderstand** code.

It is also possible to use **non-standard SOAP fault codes** that are prescribed by other organizations and belong to a separate namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/06/secext">
  <soap:Body>
    <soap:Fault>
```



```
      <faultcode>wsse:InvalidSecurityToken</faultcode>
      <faultstring>An invalid security token was provided</faultstring>
      <detail/>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Optionally, the **faultstring** element (it's a must tag to use in case of fault message) can indicate the language of the text message using a special attribute, xml:lang.

```
      <faultstring xml:lang="es" >
        El ISBN tiene letras invalidas
      </faultstring>
```

Although it's not specified, it's assumed that, in the absence of the xml:lang attribute, the default is English (xml:lang="en"). The xml:lang attribute is part of the XML 1.0 namespace, which does not need to be declared in an XML document.

The **faultactor** element indicates which node encountered the error and generated the fault (the **faulting node**). This element is **required** if the faulting node is an **intermediary**, but optional if it's the ultimate receiver. The faultactor element may contain any URI, but is usually the Internet address of the faulting node, or the URI used by the actor attribute if a header block was the source of the error. SOAP 1.1 doesn't recognize the concept of a role as distinct from a node. In fact, it lumps these two concepts together into the single concept actor. Thus you can see the faultactor as identifying both the node that generated the fault and the role that it was manifesting when it generated the fault.

The **detail** element of a fault message **must be included if the fault was caused by the contents of the Body element** (though its legal to have the detail element empty <detail/>), but it **must not be included if the error occurred while processing a header block**. The detail element may contain any number of application-specific elements, which may be qualified or unqualified, according to their XML schema. In addition, the detail element itself may contain any number of qualified attributes, as long as they do not belong to the SOAP 1.1 namespace, "http://schemas.xmlsoap.org/soap/envelope".

Recap:

Fault	Fault Code
The message received by the receiver is improperly structured or contains invalid data.	**Client**
The incoming message is properly structured, but it uses elements and namespaces in the Body element that the receiver doesn't recognize.	
The incoming message contains a mandatory header block that the receiver doesn't recognize.	**MustUnderstand**
The incoming message specifies an XML namespace for the SOAP Envelope and its children (Body, Fault, Header) that is not the SOAP 1.1 namespace	**VersionMismatch**
The SOAP receiver has encountered an abnormal condition that prevents it from processing an otherwise valid SOAP message	**Server**

SOAP Over HTTP

SOAP messages sent over HTTP are placed in the payload of an HTTP request or response. **Web services that use SOAP 1.1 with HTTP always use HTTP POST** and not HTTP GET messages as HTTP POST requests have a payload area for the SOAP message to be tunneled.

```
POST /jwsbook/BookQuote HTTP/1.1
Host: www.Monson-Haefel.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 295
SOAPAction=""

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
```

```
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

The HTTP POST message must contain a **SOAPAction** header field, but the value of this header field is not specified. The SOAPAction header field can improve throughput by providing routing information outside the SOAP payload. A node can then do some of the routing work using the SOAPAction, rather than having to parse the SOAP XML payload. The BP requires that the SOAPAction header field be present and that its value be a quoted string that matches the value of the **soapAction** attribute declared by the corresponding WSDL document. If that document declares no soapAction attribute, the SOAPAction header field can be an empty string.

The WS-I Basic Profile 1.0 prefers that the **text/xml Content-Type** be used with SOAP over HTTP. It's possible to use others (for example, SOAP with Attachments would specify **multipart/related**) but it's not recommended.

The reply to the SOAP message is placed in an HTTP reply message that is similar in structure to the request message, but contains no SOAPAction header.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset='utf-8'
Content-Length: 311

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
   <soap:Body>
      <mh:getBookPriceResponse>
          <result>24.99</result>
      </mh:getBookPriceResponse>
   </soap:Body>
</soap:Envelope>
```

Although HTTP 1.1 is the preferred protocol, you may also use HTTP 1.0. HTTP defines a number of success and failure codes that can be included in an HTTP reply message, but the BP takes special care to specify exactly which codes can be used by conformant SOAP applications. The types of response codes used depend on the success or failure of the SOAP request and the type of messaging exchange pattern used, Request/Response or One-Way.

Success Codes (200 level codes)	
200	**OK.** When a SOAP operation generates a response SOAP message, the HTTP response code for successful processing is 200 OK. This response code indicates that the reply message is not a fault, that it does contain a normal SOAP response message.
202	**Accepted**. This response code means that the request was processed successfully but that there is no SOAP response data. This type of SOAP operation is similar to a Java method that has a return type of **void**. One-Way SOAP messages do not return SOAP faults or results of any kind, so the HTTP 202 Accepted response code indicates only that the message made it to the receiver—it doesn't indicate whether the message was successfully processed.
Error Codes (400 level codes)	
400	**Bad Request**. This error code is used to indicate that either the HTTP request or the XML in the SOAP message was not well formed.
405	**Method Not Allowed.** f a Web service receives a SOAP message via any HTTP method other than HTTP POST, the service should return a 405 Method Not Allowed error to the sender.
415	**Unsupported Media Type.** HTTP POST messages must include a Content-Type header with a value of text/xml. If it's any other value, the server must return a 415 Unsupported Media Type error.
500	**Internal Server Error.** This code must be used when the response message in a Request/Response MEP is a SOAP fault.

SwA (SOAP with Attachments – Objective 2.4)

[RMH – Appendix E, F and G]

SOAP employs MIME for the same reason as e-mail: to transfer binary data. A SOAP message doesn't contain a MIME package, however. On the contrary, the SOAP message is actually part of the MIME package. HTTP already supports the use of MIME packages, so SOAP Messages with Attachments wisely exploits the existing infrastructure for MIME and simply embeds the SOAP message as a MIME part, along with its related binary data.

```
POST /submitBook HTTP/1.1
Host: www.Monson-Haefel.com
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
        start="<submitBook_1>"
Content-Length: XXXX

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <submitBook_1>

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
     <mh:submitBook>
       <isbn>0596002262</isbn>
       <coverImage href=:"cid:submitBook_2"/>
       <manuscript href=:"cid:submitBook_3"/>
     </mh:submitBook>
   <soap:Body>
</soap:Envelope>

--MIME_boundary
Content-Type: image/gif
Content-Transfer-Encoding: binary
Content-ID: <submitBook_2>

   R0lGODlhHAJSAaIAAKOgpc/N0E9PTyAgIHJDknp5e0oNc////yH5BAAAAAAALAAAAAAcAlIBAAP /aLrc
   /jDKSau9OOvNu/9gKI5kaZ5oqq5s675wLM90bRtHru987//AoHBILBqPyKRyyWw6n9CoEq tWq8/BnbL7Xq
   /4LB4TC6bqdqzes1uu9/wuLyantvv+Lx
...
+z7/X+4CBgoOEhX1/homK nN7pnt6YCQAAOw==
--MIME_boundary
Content-Type: application/pdf;
Content-Transfer-Encoding: binary
Content-ID: <submitBook_3>

JVBERi0xLjIKJeLjz9MNCjEgMCBvYmoKPDwKL1Byb2R1Y2VyIChBY3JvYmF0IERpc3RpbGxlciB
    Db21tYW5kIDMuMDEgZm9yIFNvbGFyaXMgMi4zIGFuZCBsYXRlciBcKFNQQVJDXCkpCi9UaXRsZS
AoanRzKQovQ3JlYXRvciAoRnJhbWVNYWtlciA1LjUuNi4...
OTkxMTI0MTUyMzI5KQo+PgplbmRv RU9GCg==
--MIME_boundary--
```

A CID (Content-ID) is a unique ID that is assigned to a MIME part. A CID must start with the scheme identifier `cid:` followed by a set of characters. In the example, we use identifiers like `submitBook_n`, but in practice it's more likely that some random, unique value will be used (for example, `cid:20040709.125120`). When the SubmitBook Web service receives the HTTP request in above listing, it opens the MIME package and extracts the SOAP message. If the SOAP message is well formed and properly structured, the Web service then uses the content-id references to locate the MIME parts containing binary data, then decodes them and processes them.

SAAJ Attachments

Simply put, JAF (Java Activation Framework) provides a framework for dynamically discovering visual widgets to handle (view, edit, print, and so on) any kind of data described by MIME headers. While JAF is focused on the GUI side of things, as a framework for dynamically discovering objects that can manipulate specific MIME types, it's useful for non-visual systems like SAAJ as well. In particular JAF can map Java

types, like `java.awt.Image`, to special handlers that seamlessly convert them to streams of data. This mechanism is important in SOAP messaging because it allows SAAJ to convert Java objects (such as AWT images, DOM `Document` objects, and files) into raw data contained by SwA MIME parts, automatically. For example, using SAAJ, you can add an `Image` object to a SOAP message without having to convert the image to a stream of bytes first—JAF will take care of that for you, behind the scenes.

```
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage message = mf.createMessage();
java.awt.Image image = ...; // get an image form somewhere
AttachmentPart jpegAttach =
               message.createAttachmentPart(image ,"image/jpeg");
```

A SAAJ client is a standalone client. That is, it sends point-to-point messages directly to a Web service that is implemented for request-response messaging. A request-response message is sent over a SOAPConnection object via the method SOAPConnection.call, which sends the message and blocks until it receives a response. A standalone client can operate only in a client role, that is, it can only send requests and receive their responses. A SOAPMessage object always has a required SOAP part, and it may also have one or more attachment parts. The SOAP part must always have a SOAPEnvelope object, which must in turn always contain a SOAPBody object. The SOAPEnvelope object may also contain a SOAPHeader object, to which one or more headers can be added. A SOAPMessage object always has a required SOAP part, and it may also have one or more attachment parts. The SOAP part must always have a SOAPEnvelope object, which must in turn always contain a SOAPBody object. The SOAPEnvelope object may also contain a SOAPHeader object, to which one or more headers can be added.

The SOAPBody object can hold XML fragments as the content of the message being sent. If you want to send content that is not in XML format or that is an entire XML document, your message will need to contain an attachment part in addition to the SOAP part. The connection is created by a SOAPConnectionFactory object. A client obtains the default implementation for SOAPConnectionFactory by calling the following line of code:

```
SOAPConnectionFactory factory = SOAPConnectionFactory.newInstance();
SOAPConnection connection = factory.createConnection();

MessageFactory messageFactory = MessageFactory.newInstance();
SOAPMessage message = messageFactory.createMessage();
```

All of the SOAPMessage objects that messageFactory creates, including message in the previous line of code, will be SOAP messages. This means that they will have no pre-defined headers. The new SOAPMessage object message automatically contains the required elements SOAPPart, SOAPEnvelope, and SOAPBody, plus the optional element SOAPHeader (which is included for convenience). The SOAPHeader and SOAPBody objects are initially empty. Content can be added to the SOAPPart object, to one or more AttachmentPart objects, or to both parts of a message.

One way to add content to the SOAP part of a message is to create a SOAPHeaderElement object or a SOAPBodyElement object and add an XML fragment that you build with the method SOAPElement.addTextNode.

```
SOAPPart soapPart = message.getSOAPPart();
SOAPEnvelope envelope = soapPart.getEnvelope();
SOAPBody body = envelope.getBody();
SOAPBodyElement bodyElement = body.addBodyElement(
   envelope.createName("text", "hotitems",
                  "http://hotitems.com/products/gizmo");
bodyElement.addTextNode("some-xml-text");
```

Another way is to add content to the SOAPPart object by passing it a javax.xml.transform.Source object, which may be a SAXSource, DOMSource, or StreamSource object. The Source object contains content for the SOAP part of the message and also the information needed for it to act as source input. A StreamSource object will contain the content as an XML document; the SAXSource or DOMSource object will contain content and instructions for transforming it into an XML document.


```
SOAPPart soapPart = message.getSOAPPart();
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbFactory.newDocumentBuilder();
Document document = builder.parse("file:///foo.bar/soap.xml");
DOMSource domSource = new DOMSource(document);
soapPart.setContent(domSource);
```

You use the setContent method when you want to send an existing SOAP message. If you have an XML document that you want to send as the content of a SOAP message, you use the addDocument method on the body of the message:

```
SOAPBodyElement docElement = body.addDocument(document);
```

A SOAPMessage object may have no attachment parts, but if it is to contain anything that is not in XML format, that content must be contained in an attachment part. In the following code fragment, the content is an image in a JPEG file, whose URL is used to initialize the javax.activation.DataHandler object handler.

```
URL url = new URL("http://foo.bar/img.jpg");
DataHandler handler = new DataHandler(url);
AttachmentPart attachPart = message.createAttachmentPart(handler);
message.addAttachmentPart(attachPart);
SOAPMessage response = soapConnection.call(message, endpoint);
```

Once you have populated a SOAPMessage object, you are ready to send it. A client uses the SOAPConnection method call to send a message. This method sends the message and then blocks until it gets back a response.

The following listing illustrates the creation of the SOAP request. The request asks a server to resize an image. The procedure is as follows:

1. Create SOAP connection and SOAP message objects through factories.
2. Retrieve the message body from the message object (intermediary steps: retrieve the SOAP part and envelope).
3. Create a new XML element to represent the request.
4. Create the attachment and initialize it with a DataHandler object.
5. Create more elements to represent the two parameters (source and percent).
6. Associate the attachment to the first parameter by adding an href attribute. The attachment is referred to through a cid (Content-ID) URI.
7. Set the value of the second parameter directly as text and call the service.

The service replies with the resized image, again as an attachment. To retrieve it, you can test for a SOAP fault (which indicates an error). If there are no faults, retrieve the attachment as a file and process it.

```
public File resize(String endPoint,File file) {
   SOAPConnection connection = SOAPConnectionFactory.newInstance().createConnection();
   SOAPMessage message = MessageFactory.newInstance().createMessage();
   SOAPPart part = message.getSOAPPart();
   SOAPEnvelope envelope = part.getEnvelope();
   SOAPBody body = envelope.getBody();
   SOAPBodyElement operation = body.addBodyElement(
       envelope.createName("resize", "ps", "http://example.com"));
   DataHandler dh = new DataHandler(new FileDataSource(file));
   AttachmentPart attachment = message.createAttachmentPart(dh);
   SOAPElement source  = operation.addChildElement("source","");
   SOAPElement percent = operation.addChildElement("percent","");
   message.addAttachmentPart(attachment);
   source.addAttribute(envelope.createName("href"), "cid:" + attachment.getContentId());
   percent.addTextNode("20");
```



```
   SOAPMessage result = connection.call(message,endPoint);
   part = result.getSOAPPart();
   envelope = part.getEnvelope();
   body = envelope.getBody();
   if(!body.hasFault())  {
      Iterator iterator = result.getAttachments();
      if(iterator.hasNext()) {
         dh = ((AttachmentPart)iterator.next()).getDataHandler();
         String fname = dh.getName();
         if (null != fname) return new File(fname);
      }
   }
   return null;
}
```

The code above will produce following SOAP 1.1 message with attachment:

```
POST /ws/resize HTTP/1.0
Content-Type: multipart/related; type="text/xml";
     start="<EB6FC7EDE9EF4E510F641C481A9FF1F3>";
     boundary="----=_Part_0_7145370.1075485514903"
Accept: application/soap+xml, multipart/related, text/*
Host: example.com:8080
SOAPAction: ""
Content-Length: 1506005

------=_Part_0_7145370.1075485514903
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-Id: <EB6FC7EDE9EF4E510F641C481A9FF1F3>

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ps:resize  xmlns:ps="http://example.com">
      <source href="cid:E1A97E9D40359F85CA19D1B8A7C52AA3"/>
      <percent>20</percent>
    </ps:resize>
  </soapenv:Body>
</soapenv:Envelope>

------=_Part_0_7145370.1075485514903
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-Id: <E1A97E9D40359F85CA19D1B8A7C52AA3>

d3d3Lm1hcmNoYWwuY29taesgfSEVFES45345sdvgfszd==
------=_Part_0_7145370.1075485514903--
```

Describing and Publishing (WSDL 1.1 + UDDI 2.0)

[RMH – 5 for WSDL, 6,7,8 for UDDI]
3.1 Explain the use of WSDL in Web services, including a description of WSDL's basic elements, binding mechanisms and the basic WSDL operation types as limited by the WS-I Basic Profile 1.0a.
3.2 Describe how W3C XML Schema is used as a typing mechanism in WSDL 1.1.
3.3 Describe the use of UDDI data structures. Consider the requirements imposed on UDDI by the WS-I Basic Profile 1.0a.
3.4 Describe the basic functions provided by the UDDI Publish and Inquiry APIs to interact with a UDDI business registry.

WSDL 1.1

Web Services Description Language is a document format for precisely defining web services. WSDL is used to specify the exact message format, Internet protocol, and address that a client must use to communicate with a particular Web service.

Note that WSDL 1.1 is not specific to SOAP; it can be used to describe non-SOAP-based Web services as well. **WSDL is especially useful for code generators** like the JAX-RPC API which take a WSDL and produce Java RMI interfaces (Service end point interfaces) and network stubs that implement the SEIs such that the client can use the stub to invoke remote methods on the web service. J2EE 1.4 application servers also provide their own JAX-RPC API implementations for eg. BEA Weblogic JAX-RPC implementation. There are other code generators, like .NET and Axis which use WSDL for code generation of programmatic interfaces to access a web service. Many Web service clients, for example, use SOAP APIs instead of generated call interfaces and stubs. These APIs usually model the structure of the SOAP message using objects like Envelope, Header, Body, and Fault. Examples of SOAP APIs include SAAJ (SOAP with Attachments API for Java), Perl::Lite, Apache SOAP, and others. When you use a SOAP API, you can use a Web service's WSDL as a guide for exchanging SOAP messages with that Web service.

Basic Structure of a WSDL

A WSDL document is an XML document that adheres to the WSDL XML schema. A WSDL document contains seven important elements: **types, import, message, portType, operations, binding,** and **service,** which are nested in the **definitions** element, the root element of a WSDL document.

Types	uses the XML schema language to declare complex data types and elements that are used elsewhere in the WSDL document.
Import	is similar to an import element in an XML schema document; it's used to import WSDL definitions from other WSDL documents.
Message	describes the message's payload using XML schema built-in types, complex types, or elements that are defined in the WSDL document's types element, or defined in an external WSDL document the import element refers to.
portType and Operation	describe a Web service's interface and define its methods. A portType and its operation elements are analogous to a Java interface and its method declarations. An operation element uses one or more message types to define its input and output payloads.
Binding	assigns a portType and its operation elements to a particular protocol (for instance, SOAP 1.1) and encoding style.
Service	is responsible for assigning an Internet address to a specific binding.
Documentation	explains some aspect of the WSDL document to human readers. Any of the other WSDL elements may contain documentation elements.

```
package com.jwsbook.soap;
import java.rmi.RemoteException;

public interface BookQuote extends java.rmi.Remote {
    // Get the wholesale price of a book
    public float getBookPrice(String ISBN)
      throws RemoteException, InvalidISBNException;
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>

    <xsd:schema
     targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote">
      <!-- The ISBN simple type -->
      <xsd:simpleType name="ISBN">
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[0-9]{9}[0-9Xx]" />
        </xsd:restriction>
      </xsd:simpleType>
```



```
    </xsd:schema>

  </types>

  <!-- message elements describe the input and output parameters -->
  <message name="GetBookPriceRequest">
    <part name="isbn" type="mh:ISBN" />
  </message>
  <message name="GetBookPriceResponse">
    <part name="price" type="xsd:float" />
  </message>

  <!-- portType element describes the abstract interface of a Web service -->
  <portType name="BookQuote">
    <operation name="getBookPrice">
      <input name="isbn" message="mh:GetBookPriceRequest"/>
      <output name="price" message="mh:GetBookPriceResponse"/>
    </operation>
  </portType>

  <!-- binding tells us which protocols and encoding styles are used -->
  <binding name="BookPrice_Binding" type="mh:BookQuote">
    <soapbind:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getBookPrice">
      <soapbind:operation style="rpc"
       soapAction=
       "http://www.Monson-Haefel.com/jwsbook/BookQuote/GetBookPrice"/>
        <input>
          <soapbind:body use="literal"
           namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
        </input>
        <output>
          <soapbind:body use="literal"
           namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
        </output>
    </operation>
  </binding>

  <!-- service tells us the Internet address of a Web service -->
  <service name="BookPriceService">
    <port name="BookPrice_Port" binding="mh:BookPrice_Binding">
      <soapbind:address location=
       "http://www.Monson-Haefel.com/jwsbook/BookQuote" />
    </port>
  </service>

</definitions>
```

In the sections below is the piece-meal analysis of the above WSDL listing for the BookQuote web service.

WSDL Declarations : definitions, types and import

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

A WSDL document must use either UTF-8 or UTF-16 encoding; other encoding systems are not allowed.

definitions element

The root element of all WSDL documents is the definitions element, which encapsulates the entire document and also provides a WSDL document with its name.

```
<definitions name="BookQuoteWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Declaring the WSDL namespace (http://schemas.xmlsoap.org/wsdl/) as the **default namespace** avoids having to qualify every WSDL element explicitly, with a prefix.
Name attribute is used to give a WSDL name (which is not used in practice) and is optional.
The definitions element also declares a **targetNamespace** attribute, which identifies the namespace of elements defined in the WSDL document—much as it does in XML schema documents.
The **message**, **portType**, and **binding** elements are assigned labels using their name attributes; these labels automatically take on the namespace specified by the **targetNamespace** attribute. (In this book the URL of the targetNamespace is also assigned a prefix of **mh**.) Labeled message, portType, and binding elements are commonly called **definitions**. These definitions assume the namespace specified by targetNamespace.
Other elements in the document refer to the definitions using their label and namespace prefix. A prefixed label is considered a fully qualified name (**QName**) for a definition.

```
<!-- message elements describe the input and output parameters -->
<message name="GetBookPriceRequest">
  <part name="isbn" type="xsd:string" />
</message>
<message name="GetBookPriceResponse">
  <part name="price" type="xsd:float" />
</message>

<!-- portType element describes the abstract interface of a Web service -->
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input name="isbn" message="mh:GetBookPriceRequest"/>
    <output name="price" message="mh:GetBookPriceResponse"/>
  </operation>
</portType>
```

Input and output elements above refer to the message definitions using their QNames.

types element

WSDL adopts, as its basic type system, the W3C XML schema built-in types. The types element serves as a container for defining any data types that are not described by the XML schema built-in types: complex types and custom simple types. The data types and elements defined in the types element are used by message definitions when declaring the parts (payloads) of messages.

The targetNamespace attribute of the XML schema must be a valid non-null value, otherwise the types and element will not belong to a valid namespace. In addition, the XML schema defined in the types element must belong to a namespace specified by the WSDL document (usually in the definitions element) or to a namespace of an imported WSDL document.

In a nutshell, you are not allowed to use the Array type, or the arrayType attribute defined for SOAP 1.1 Encoding (SOAP 1.1 Note, Section 5), or the WSDL arrayType attribute defined by WSDL. In addition you should not label array types as "ArrayOfXXX" as suggested by the WSDL 1.1 Note. The wayaround is, to **define a complex type with a maxOccurs value greater than 0** (for example, "10", "32000", or "unbounded") and **you have yourself a basic array**.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>

    <xsd:schema
     targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote">
      <!--A simple array-like type -->
      <xsd:complexType name="IntArray">
        <xsd:sequence>
          <xsd:element name="arg" type="xsd:int" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
```

```
    </xsd:schema>
  </types>
```

import Element

The import element makes available in the present WSDL document the definitions from a specified namespace in another WSDL document. This feature can be useful if you want to modularize WSDL documents—for example, to separate the abstract definitions (the types, message, and portType elements) from the concrete definitions (the binding, service, and port elements). Another reason to use import is to consolidate into one WSDL document several definitions you want to maintain separately.

```
<definitions name="AllMhWebServices"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

    <import namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
     location="http://www.Monson-Haefel.com/jwsbook/BookPrice.wsdl"/>
    <import namespace="http://www.Monson-Haefel.com/jwsbook/po"
     location="http://www.Monson-Haefel.com/jwsbook/wsdl/PurchaseOrder.wsdl"/>
    <import namespace="http://www.Monson-Haefel.com/jwsbook/Shipping"
     location="http://www.Monson-Haefel.com/jwsbook/wsdl/Shipping.wsdl"/>

</definitions >
```

The WSDL import element must declare two attributes: namespace and location. The value of the namespace attribute must match the targetNamespace declared by the WSDL document being imported. The location attribute must point to an actual WSDL document; it cannot be empty or null. That said, the location is considered a hint. If the application reading the WSDL document has cached or stored copies of the imported WSDL document locally, it may use those instead.

You can use import and types together, **but you should list the import elements before the types element** in a WSDL document.

Basic Profile specifies that a WSDL import element may refer only to WSDL documents. If you need to import an XML schema element, you should do so in the XML schema definition contained in the WSDL types element, using the standard XML schema import statement**. You cannot use the XML schema import statement to import an XML schema directly from the types element of some other WSDL document.**

WSDL Abstract Interface : message, portType and operation

The portType combines the operation and message definitions into an abstract interface that is analogous to a Java interface definition.

message Element

The **message** element **describes the payload of a message used by a Web service**. A message element can describe the payloads of outgoing or incoming messages—that is, messages that are directly sent to or received from a Web service. In addition, the message element can describe the contents of SOAP header blocks and fault detail elements. The way to define a message element depends on whether you use RPC-style or document-style messaging.

When **RPC-style messaging** is used, message elements describe the payloads of the SOAP request and reply messages. They may describe call parameters, call return values, header blocks, or faults.

```
<!-- message elements describe the input and output parameters -->
<message name="GetBookPriceRequest">
  <part name="isbn" type="xsd:string" />
</message>
<message name="GetBookPriceResponse">
  <part name="price" type="xsd:float" />
</message>
```

Message names are arbitrary and only serve to qualify a message definition.

Both input and output messages in Web services can have multiple parts, which is a departure from Java method-call semantics, which recognize multiple inputs (parameters) but only one output (the return value). Both SAAJ and JAX-RPC have facilities to support output messages with multiple parts.

When you use **document-style messaging**, the **message definition refers to a top-level element in the types definition**. For eg; WSDL document that describes a SubmitPurchaseOrder Web Service. The Purchase Order schema is imported into the types element of the WSDL document. In this case, the Web service uses One-Way messaging, so there is no reply message, which is common in document-style messaging. A message **part** may declare either a **type** attribute or an **element** attribute, but not both. Which to use depends on the kind of messaging you're doing. **If you're using RPC-style messaging, the part elements must use the type attribute; if you're using document-style messaging, the part elements must use the element attribute**.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PurchaseOrderWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/PO"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO">
      <!-- Import the PurchaseOrder XML schema document -->
      <xsd:import namespace="http://www.Monson-Haefel.com/jwsbook/PO"
       schemaLocation="http://www.Monson-Haefel.com/jwsbook/po.xsd" />
    </xsd:schema>
  </types>
  <!-- message elements describe the input and output parameters -->
  <message name="SubmitPurchaseOrderMessage">
    <part name="order" element="mh:purchaseOrder" />
  </message>
  ...
</definitions>
```

RPC-style messaging uses types to define procedure calls, where each element represents a type of parameter. Document-style messaging, on the other hand, exchanges XML document fragments and refers to their top-level (global) elements.

You can use message definitions to **declare faults** in the same way you use them to declare input and output messages.

```
<definitions name="BookQuote" ...>

  <types>
    <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO">
      <!-- Import the PurchaseOrder XML schema document -->
      <xsd:element name="InvalidIsbnFaultDetail" >
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="offending-value" type="xsd:string"/>
            <xsd:element name="conformance-rules" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>

  <!-- message elements describe the input and output parameters -->
  <message name="GetBookPriceRequest">
    <part name="isbn" type="xsd:string" />
  </message>
  <message name="GetBookPriceResponse">
    <part name="price" type="xsd:float" />
  </message>
  <message name="InvalidArgumentFault">
    <part name="error_message" element="mh:InvalidIsbnFaultDetail" />
  </message>
</definitions>
```

The **types** element defines the **InvalidIsbnFaultDetail** top-level element as an anonymous type.
Fault messages used by SOAP-based Web services can have only one part. In the case of a SOAP fault, the part definition refers to the contents of the **detail** section of the fault message. The message definitions used by faults use Document/Literal encoding style and therefore must be based on a top-level element defined in the types element or imported in a WSDL or XML schema document.
Message definitions can also be used to describe SOAP header blocks and fault header blocks.

portType Element

A `portType` defines the abstract interface of a Web service. In WSDL the `portType` is implemented by the `binding` and `service` elements, which dictate the Internet protocols, encoding schemes, and an Internet address used by a Web service implementation. The "methods" of the `portType` are its `operation` elements. A `portType` may have one or more `operation` elements, each of which defines an RPC- or document-style Web service method. Each `operation` is composed of at most one `input` or `output` element and any number of `fault` elements.

WSDL portType	Java Interface

<pre><portType name="BookQuote">
  <operation name="GetBookPrice">
  <input
  name="isbn"
  message="mh:GetBookPriceRequest"/>
  <output
  name="price"
  message="mh:GetBookPriceResponse"/>
  </operation>
  <operation name="GetBulkBookPrice">
  <input
  name="request"
  message="mh:GetBulkBookPriceRequest"/>
  <output name="prices"
 	
message="mh:GetBulkBookPriceResponse"/>
  </operation>
  <operation name="GetBookIsbn">
  <input
  name="title"
  message="mh:GetBookIsbnRequest"/>
  <output
  name="isbn"
  message="mh:GetBookIsbnResponse"/>
  </operation>
</portType></pre>
	<pre>public interface BookQuote {
	  public float getBookPrice
	  (String isbn);
	
	  public float getBulkBookPrice
	  (String isbn, int quantity);
	
	  public String getBookIsbn
	  (String bookTitle);
	}</pre>

Code generators can generate Java interfaces from WSDL `portType` elements—and can also generate WSDL `portType`, `operation`, and `message` elements from simple Java interfaces. A WSDL document can have one or more `portType` elements, each of which describes the abstract interface to a different Web service. For example, a WSDL Web service might define one `portType` named `"BookQuote"` and another named `"SubmitPurchaseOrder"`.

The operation Element

Each `operation` element declared by a `portType` uses one or more message definitions to define its input, output, and faults.

```
<portType name="BookQuote">
  <operation name="getBookPrice">
     <input name="isbn" message="mh:GetBookPriceRequest"/>
     <output name="price" message="mh:GetBookPriceResponse"/>
     <fault name="InvalidArgumentFault" message="mh:InvalidArgumentFault"/>
  </operation>
```

```
</portType>
```

In many cases the parameters of the `input` and `output` messages must be transferred in a specific order, the same order as the parameters of the procedure call. To enforce proper ordering of `input` or `output` message parameters, the `operation` element may declare a `parameterOrder` attribute.

```
<message name="GetBulkBookPriceRequest">
  <part name="isbn" type="xsd:string"/>
  <part name="quantity" type="xsd:int"/>
</message>
<message name="GetBulkBookPriceResponse">
  <part name="prices" type="mh:prices" />
</message>
<portType name="GetBulkBookPrice" >
  <operation name="getBulkBookPrice" parameterOrder="isbn quantity">
     <input name="request" message="mh:GetBulkBookPriceRequest"/>
     <output name="prices" message="mh:GetBulkBookPriceResponse"/>
  </operation>
</portType>
```

When a `parameterOrder` attribute is used, it must include all the input parts and only the output parts that are not the return type.

The order of parameters transmitted from sender to receiver must follow the order of `part` declarations made in `input` and `output` message definitions. The purpose of the `parameterOrder` attribute is to indicate which part, if any, is the return type. Any part that is omitted from the list provided by the `parameterOrder` attribute is assumed to be the return type of the operation. A procedure call can have only one return type, so only a single output part may be omitted from the `parameterOrder` attribute.

Operation Overloading

In WSDL, two operations may have the same name, provided their input or output messages differ. Unfortunately, this feature has caused enough interoperability problems that the Basic Profile prohibits operation overloading. So across portType definitions (each portType corresponds to single Java interface) the method names (operation names) can be same but operation names within the same portType definition cannot be same as mandated by BP.

WSDL Message Exchange Pattern

Most WSDL-based Web services today use either Request/Response or One-Way messaging and are the ones supported by J2EE Web services.

A WSDL document can dictate the MEP for a specific operation by the way it declares its input and output elements.

If an `operation` is declared with a single `input` element followed by a single `output` element, it defines a **Request/Response** operation. In addition to its one `input` and one `output`, a Request/Response operation may also include `fault` elements (0 or more), which are returned to the client in the event of an error.

```
<portType name="BookQuote">
  <operation name="getBookPrice">
     <input name="isbn" message="mh:GetBookPriceRequest"/>
     <output name="price" message="mh:GetBookPriceResponse"/>
     <fault name="InvalidArgumentFault" message="mh:InvalidArgumentFault"/>
     <fault name="SecurityFault" message="mh:SecurityFault"/>
  </operation>
</portType>
```

If an operation is declared with a single `input` but no `output`, it defines a **One-Way** operation. Unlike Request/Response operations, One-Way operations may not specify fault elements and do not generate fault messages.

```
<portType name="SubmitPurchaseOrder_PortType">
  <operation name="SubmitPurchaseOrder">
     <input name="order" message="mh:SubmitPurchaseOrderMessage"/>
  </operation>
```

```
</portType>
```

Neither the Notification nor the Solicit/Response MEP can be used in J2EE Web Services.

WSDL Implementation : binding, service and port

The `binding` element maps an abstract `portType` to a set of concrete protocols such as SOAP and HTTP, messaging styles (RPC or document), and encoding styles (Literal or SOAP Encoding). The `binding` element, and its subelements, are used in combination with protocol-specific elements. The `binding` elements identify which `portType` and `operation` elements are being bound, while the protocol-specific elements declare the protocol and encoding style to be associated with the `portType`. Each type of protocol (SOAP, MIME, and HTTP) has its own set of protocol-specific elements and its own namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
  ...
  <!-- binding tells us which protocols and encoding styles are used -->
  <binding name="BookPrice_Binding" type="mh:BookQuote">
    <soapbind:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getBookPrice">
      <soapbind:operation style="rpc"
       soapAction=
       "http://www.Monson-Haefel.com/jwsbook/BookQuote/GetBookPrice"/>
        <input>
          <soapbind:body use="literal"
           namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
        </input>
        <output>
          <soapbind:body use="literal"
           namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
        </output>
    </operation>
  </binding>
  ...
</definitions>
```

The elements without a namespace prefix (shown in bold) are members of the WSDL 1.1 namespace `"http://schemas.xmlsoap.org/wsdl/"`, which is the default namespace of the WSDL document. The WSDL 1.1-generic binding elements are `binding`, `operation`, `input`, and `output`. The `soapbind:binding`, `soapbind:operation`, and `soapbind:body` elements, on the other hand, are protocol-specific. They are members of the namespace for the SOAP-WSDL binding, `"http://schemas.xmlsoap.org/wsdl/soap/"`.

The `soapbind:binding` and `soapbind:body` elements are responsible for expressing the SOAP-specific details of the Web service. For example, `soapbind:binding` tells us that the messaging style is RPC and that the network application protocol is HTTP. The `soapbind:body` element tells us that both the input and output messages use literal encoding. The children of the `binding` element (`operation`, `input`, and `output`) map directly to the corresponding children of the `portType` element.

SOAP Binding

Several SOAP 1.1-specific binding elements are used in combination with the WSDL binding elements. These include `soapbind:binding`, `soapbind:operation`, `soapbind:body`, `soapbind:fault`, `soapbind:header`, and `soapbind:headerfault`. The `soapbind:binding` and `soapbind:body` elements are required, but the other elements are optional.

```
<!-- binding tells us which protocols and encoding styles are used  -->
<binding name="BookPrice_Binding" type="mh:BookQuote">
  <soapbind:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getBookPrice">
    <soapbind:operation style="rpc"
     soapAction=
     "http://www.Monson-Haefel.com/jwsbook/BookQuote/GetBookPrice"/>
      <input>
        <soapbind:body use="literal"
         namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
      </input>
      <output>
        <soapbind:body use="literal"
         namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
      </output>
    <fault name="InvalidArgumentFault">
     <soapbind:fault name="InvalidArgumentFault" use="literal" />
    </fault>
  </operation>
</binding>
```

The **`soapbind:binding`** element identifies the Internet protocol used to transport SOAP messages and the default messaging style (RPC or document) of its operations. The `style` attribute must be declared as either `"rpc"` or `"document"`; no other values are acceptable. The `transport` attribute must be declared to be HTTP, which means its value must be `"http://schemas.xmlsoap.org/soap/http/"`.

The **`soapbind:operation`** element is required. It specifies the messaging style (RPC or document) for a specific operation and the value of the `SOAPAction` header field.

1. The Basic Profile requires that `style` attributes declared by `soapbind:operation` elements have the same value as the `style` attribute of their `soapbind:binding` element.

2. You aren't required to declare the WSDL `soapAction` attribute; it can be omitted. You can also declare the `soapAction` attribute's value to be empty (indicated by two quotes), which is the same as omitting it. If this attribute is omitted or empty, then the `SOAPAction` HTTP header field must be present and must contain an empty string.
3. When you use RPC-style messaging, the `Body` of the SOAP message will contain an element that represents the operation to be performed. This element gets its name from the `operation` defined in the `portType`. The `operation` element will contain zero or more parameter elements, which are derived from the `input` message's parts—each parameter element maps directly to a message part.
4. When you use document-style messaging, the XML document fragment will be the direct child of the `Body` element of the SOAP message. The operation is not identified.

Attributes of the **`soapbind:body`** element change depending on whether you use RPC- or document-style messaging. The `soapbind:body` element has four kinds of attributes: `use`, `namespace`, `part`, and `encodingStyle`.

1. The `use` attribute is required to be `"literal"`—and that value is assumed if the `soapbind:body` element fails to declare the `use` attribute.
2. The `encodingStyle` attribute is never used at all, because WS-I-conformant Web services are based on the W3C XML schema, which is implied by the `use="literal"` declaration. Other encoding styles, like SOAP 1.1 Encoding, are not used.
3. The `part` attribute specifies which `part` elements in the message definition are being used. The `part` attribute is necessary only if you are using a subset of the `part` elements declared by a message.
4. In `"rpc"`-style messages, the `namespace` attribute must be specified with a valid URI.[BP] The URI can be the same as the `targetNamespace` of the WSDL document.
5. In contrast, document-style messages must not specify the `namespace` attribute in the `soapbind:body` element. The namespace of the XML document fragment is derived from its XML schema. This is shown below:

```
<!-- binding tells us which protocols and encoding styles are used -->
<binding name="SubmitPurchaseOrder_Binding" type="mh:SubmitPurchaseOrder">
  <soapbind:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="submit">
    <soapbind:operation style="document"/>
      <input>
        <soapbind:body use="literal" />
      </input>
      <output>
        <soapbind:body use="literal" />
      </output>
  </operation>
</binding>
```

In addition to the `soapbind:body` element, a binding `operation` may also declare **`fault`** elements.

1. The WSDL `fault` and `soapbind:fault` elements include a mandatory `name` attribute, which refers to a specific fault message declared in the associated `portType`.
2. An operation may have zero or more `fault` elements, each with its own `soapbind:fault` element. Each `soapbind:fault` element may declare a `use` attribute. If it does, the value must be `"literal"`. If it doesn't, the value is `"literal"` by default.

WSDL explicitly identifies a SOAP header block by using the **`soapbind:header`** element in the binding's `input` element, its `output` element, or both.

```
<types>
    <xsd:schema targetNamespace=
```



```
    "http://www.Monson-Haefel.com/jwsbook/BookQuote"
    xmlns="http://www.w3.org/2001/XMLSchema">
     <xsd:element name="message-id" type="string" />
   </xsd:schema>
 </types>

 <!-- message elements describe the input and output parameters -->
 <message name="Headers">
   <part name="message-id" element="mh:message-id" />
 </message>
 <!-- message elements describe the input and output parameters -->
 <message name="HeaderFault">
   <part name="faultDetail" element="mh:detailMessage" />
 </message>
 <message name="GetBookPriceRequest">
   <part name="isbn" type="xsd:string" />
 </message>
 <message name="GetBookPriceResponse">
   <part name="price" type="xsd:float" />
 </message>

 <!-- portType element describes the abstract interface of a Web service -->
 <portType name="BookQuote">
   <operation name="getBookPrice">
     <input name="isbn" message="mh:GetBookPriceRequest"/>
     <output name="price" message="mh:GetBookPriceResponse"/>
   </operation>
 </portType>

 <!-- binding tells us which protocols and encoding styles are used -->
 <binding name="BookPrice_Binding" type="mh:BookQuote">
   <soapbind:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="getBookPrice">
     <soapbind:operation style="rpc"
      soapAction=
      "http://www.Monson-Haefel.com/jwsbook/BookQuote/GetBookPrice"/>
       <input>
         <soapbind:header message="mh:Headers" part="message-id"
          use="literal">
           <soapbind:headerfault message="mh:HeaderFault" use="literal" />
         </soapbind:header>
         <soapbind:body use="literal"
          namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
       </input>
       <output>
         <soapbind:body use="literal"
          namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
       </output>
   </operation>
 </binding>
```

1. The `part` referred to must use an `element` attribute. In other words, you can't base a header block on a type definition, it must be based on a top-level element defined in the `types` element, or imported in some other XML schema document or WSDL document.
2. The `use` attribute is always equal to `"literal"`, whether it's explicitly declared or not.

The **`soapbind:headerfault`** element describes a header block-specific fault message. if there is a response message, any header block-specific faults must be returned in its `Header` element. WSDL maintains this scoping by requiring that `soapbind:headerfault` elements be nested in their associated headers.

1. It must declare a `message` attribute that points to the appropriate `message` definition, and
2. its `use` attribute must be equal to `"literal"`, whether explicitly declared or by default.

WSDL service and port Elements

The service element contains one or more port elements, each of which represents a different Web service. The port element assigns the URL to a specific binding. It's even possible for two or more port elements to assign different URLs to the same binding, which might be useful for load balancing or failover.

```
<service name="BookPriceService">
  <port name="BookPrice_Port" binding="mh:BookPrice_Binding">
    <soapbind:address location=
     "http://www.Monson-Haefel.com/jwsbook/BookQuote" />
  </port>
  <port name="BookPrice_Failover_Port" binding="mh:BookPrice_Binding">
    <soapbind:address location=
     "http://www.monson-haefel.org/jwsbook/BookPrice" />
  </port>
  <port name="SubmitPurchaseOrder_Port"
   binding="mh:SubmitPurchaseOrder_Binding">
    <soapbind:address location=
     "https://www.monson-haefel.org/jwsbook/po" />
  </port>
</service>
```

1. The soapbind:address element is pretty straightforward; it simply assigns an Internet address to a SOAP binding via its location attribute (its only attribute). Although WSDL allows any type of address (HTTP, FTP, SMTP, and so on), the Basic Profile allows only those URLs that use the HTTP or HTTPS schema.
2. Two or more port elements within the same WSDL document must not specify exactly the same URL value for the location attribute of the soapbind:address.

WS-I Conformance Claims

A WS-I conformance claim can be assigned to any WSDL definition, asserting adherence to the WS-I Basic Profile 1.0 specification. Child elements inherit their parents' conformance claims; for example, a portType's claim that it conforms to the BP also applies to all the operation and message definitions associated with that portType. The best place to put a conformance claim is inside the port definition, because it applies to all the other definitions associated with that port (binding, portType, operation, and message).

UDDI 2.0(Universal Description, Discovery and Integration)

The basic goal of UDDI is to provide a standard data model for storing information about organizations and their Web services. UDDI defines a SOAP-based API that allows remote clients to access, update, and search the information in a UDDI registry. UDDI is the specification and an UDDI registry is an implementation. The usual analogy is that a UDDI registry is like an electronic "Yellow Pages" that businesses can search to find other organizations, and specific types of Web services. You can access the data in a UDDI directory using SOAP, which makes a **UDDI registry a Web service**. Currently the specs are maintained by **OASIS** (Organisation for Advancement of Structured Information Standards). Although most UDDI products today run on a relational database management system, they can be implemented using other technologies, including LDAP servers and XML databases.

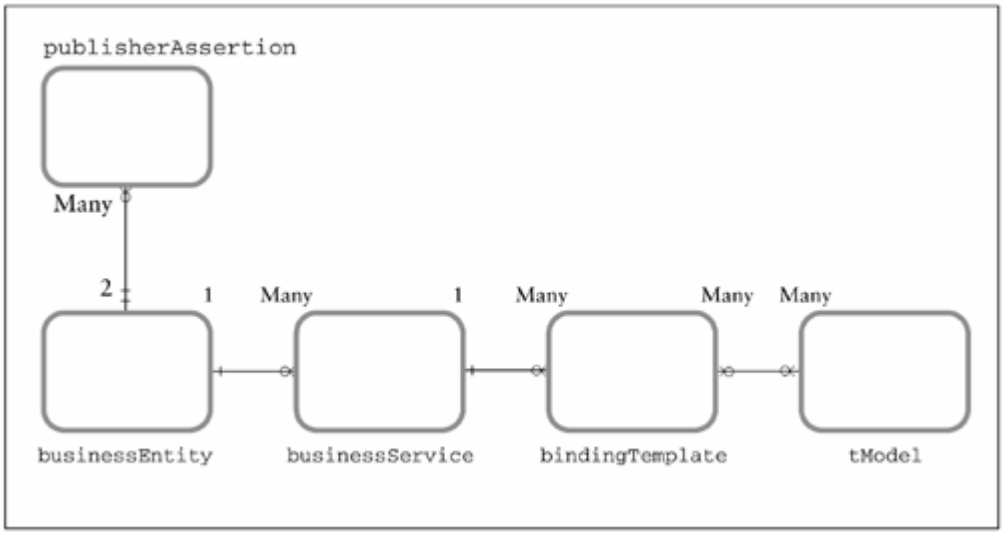
UDDI Data Structures

A UDDI registry must provide access to its data by way of a set of SOAP-based Web services. The UDDI specification describes about thirty different SOAP operations that allow you to add, update, delete, and find information contained in a UDDI registry. Most UDDI products also provide a Web interface for access to information in the registry. Many people think of UDDI as a good way of cataloging a corporation's software applications, in order to provide a centralized source for documentation and discovery.

UDDI defines five primary data structures, which are used to represent an organization, its services, implementation technologies, and relationships to other businesses:

1. A **businessEntity** represents the business or organization that provides the Web service.

2. A **businessService** represents a Web service or some other electronic service.
3. A **bindingTemplate** represents the technical binding of a Web service to its access point (its URL), and to tModels.
4. A **tModel** represents a specific kind of technology, such as SOAP or WSDL, or a type of categorization system, such as U.S. Federal Tax ID numbers or D-U-N-S numbers. The tModels that a bindingTemplate refers to reveal the type of technologies used by a Web service. Most of the data structures also use tModels to indicate the categorization system of an identifier assigned to the structure.
5. A **publisherAssertion** represents a relationship between two business entities.



The five primary data structures are not the only data types defined by UDDI, but they are the most important ones, and the only ones assigned unique keys that allow them to be stored separately, yet still refer to each other. There are also data structures that represent address information, categories, contact information, URLs, and other common business data. All the standard data structures in UDDI are defined as XML schema complex types in a single XML schema document.

businessEntity Structure

A businessEntity is a data structure that represents any individual or enterprise that is publishing one or more Web services to a UDDI registry. The businessEntity provides very general information: names used by the organization, descriptions, contact information, Web services offered, and identity and categorization information.

```
<schema targetNamespace="urn:uddi-org:api_v2"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2"
            version="2.03" id="uddi">
...
<element name="businessEntity" type="uddi:businessEntity"/>
<complexType name="businessEntity">
   <sequence>
      <element ref="uddi:discoveryURLs" minOccurs="0"/>
      <element ref="uddi:name" maxOccurs="unbounded"/>
      <element ref="uddi:description" minOccurs="0"
                                      maxOccurs="unbounded"/>
      <element ref="uddi:contacts" minOccurs="0"/>
      <element ref="uddi:businessServices" minOccurs="0"/>
      <element ref="uddi:identifierBag" minOccurs="0"/>
      <element ref="uddi:categoryBag" minOccurs="0"/>
   </sequence>
   <attribute name="businessKey" type="uddi:businessKey" use="required"/>
   <attribute name="operator" type="string" use="optional"/>
   <attribute name="authorizedName" type="string" use="optional"/>
```



```
</complexType>
 ...
</schema>
```

And here's an instance of businessEntity:

```
<?xml version='1.0' encoding='UTF-8'?>
<businessEntity businessKey="01B1FA80-2A15-11D6-9B59-000629DC0A53"
                xmlns="urn:uddi-org:api_v2">
  <discoveryURLs>
    <discoveryURL useType="buinessEntity">
      http://uddi.ibm.com/registry/uddiget?businessKey=01B1FA80...000629DC0A53
    </discoveryURL>
  </discoveryURLs>
  <name>Monson-Haefel Books, Inc.</name>
  <description xml:lang="en">Technical Book Wholesaler</description>
  <contacts>
    <contact>
      <description xml:lang="en">Web Service Tech. Support</description>
      <personName>Stanley Kubrick</personName>
      <phone useType="Voice">01-555-222-4000</phone>
    </contact>
  </contacts>
  <businessServices>
    <!-- businessService data goes here -->
  </businessServices>
  <identifierBag>
    <!-- D-U-N-S Number Identifier System -->
    <keyedReference
        keyName="D-U-N-S"
        keyValue="03-892-4499"
        tModelKey="uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823"/>
  </identifierBag>
  <categoryBag>
    <!-- North American Industry Classification System (NAICS) 1997 -->
    <keyedReference
        keyName="Book, Periodical, and Newspaper Wholesalers"
        keyValue="42292"
        tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"/>
    <!-- Universal Standard Products and Services Classification (UNSPSC)
         Version 6.3 -->
    <keyedReference
        keyName="Educational or vocational textbooks"
        keyValue="55.10.15.09.00"
        tModelKey="uuid:CD153256-086A-4236-B336-6BDCBDCC6634"/>
    <!-- ISO 3166 Geographic Taxonomy -->
    <keyedReference
        keyName="Minnesota, USA"
        keyValue="US-MN"
        tModelKey="uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88"/>
  </categoryBag>
</businessEntity>
```

The value of **businessKey** attribute is a UUID (Universally Unique Identifier) key that is automatically generated from the data structure by the UDDI registry when it's first created. The `businessService`, `bindingTemplate`, and `tModel` data structures also have UUID keys.

The **`discoveryURL`** element contains the Web address where the raw `businessEntity` can be accessed using an HTTP GET message. A `discoveryURL` can refer to two types of documents: a required `businessEntity` type and zero or more `businessEntityExt` types. The `businessEntity`'s `discoveryURL` is generated by the UDDI registry automatically when the `businessEntity` data is added or modified. The `businessEntity`'s `discoveryURL` is useful when perusing a UDDI registry with an HTML browser. When the `discoveryURL` declares a `useType` attribute value to be `"businessEntity"`, then it was generated by the UDDI registry and contains the raw `businessEntity` data. If, however, the `discoveryURL`'s `useType` attribute value is `"businessEntityExt"`, then the URL refers to non-UDDI data provided by the publisher. This might include technical specifications, or financial documents, etc.

The **`name`** element holds the common name of the organization that the `businessEntity` represents.

description element presents a description of the organization.
The **contacts** element contains contact information for people at the organization who maintain the UDDI information or fulfill some other function. The contacts element is **optional**. Following is the definition for the uddi:contact:

```
<schema targetNamespace="urn:uddi-org:api_v2"
           xmlns="http://www.w3.org/2001/XMLSchema"
           xmlns:uddi="urn:uddi-org:api_v2"
           version="2.03" id="uddi">
  ...
  <xsd:element name="contact" type="uddi:contact"/>
  <xsd:complexType name="contact">
    <xsd:sequence>
      <xsd:element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="uddi:personName"/>
      <xsd:element ref="uddi:phone" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="uddi:email" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="uddi:address" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="useType" type="string" use="optional"/>
  </xsd:complexType>
 ...
</schema>
```

The **identifierBag** element is **optional** and may contain one or more keyed Reference elements. When used with a businessEntity, a keyedReference element contains a name-value pair that declares some type of identifier for the organization: a D-U-N-S number (from an identification system created by Dun & Bradstreet), a tax identification number, or some other form of identification. A tModel is a data structure that identifies the specifications for a specific taxonomy or classification system. Every tModel has a unique tModelKey, which can be used to look up the tModel in a UDDI registry. In the case of both D-U-N-S and Thomas Register, the tModel referred to by the tModelKey points to the organization's Web site.
A **categoryBag** element has the same structure as an identifierBag element, but its keyedReference elements refer to categorization codes rather than unique identifiers. For example, a businessEntity might be categorized as a manufacturer, or as providing a specific kind of product, or by its geographic location. The UDDI specification requires that all UDDI registries support three industry-standard categorization code sets: NAICS, UNSPSC, and ISO 3166. Operator Sites automatically provide validated categorization support for three taxonomies that cover industry codes (via NAICS), product and service classifications (via UNSPC) and geography (via ISO 3166).

businessService Structure

The **businessServices** element contains one or more **businessService** data structures, each of which represents a Web service implementation. Each businessService contains one or more bindingTemplate entries. The relationship between a businessService data structure and the bindingTemplate data structure is similar to the relationship between a WSDL service and the WSDL port elements: A businessService represents a grouping of related bindingTemplate data entries. A bindingTemplate describes a Web service endpoint and represents the "technical fingerprint" of a Web service. In other words, it lists all the tModel types that describe a Web service, which uniquely identify the Web service's technical specifications.

```
<schema targetNamespace="urn:uddi-org:api_v2"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2"
            version="2.03" id="uddi">
  ...
  <element name="businessService" type="uddi:businessService"/>
  <complexType name="businessService">
    <sequence>
      <element ref="uddi:name" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="uddi:description" minOccurs="0"
                                      maxOccurs="unbounded"/>
      <element ref="uddi:bindingTemplates" minOccurs="0"/>
```

```
      <element ref="uddi:categoryBag" minOccurs="0"/>
    </sequence>
    <attribute name="serviceKey" type="uddi:serviceKey" use="required"/>
    <attribute name="businessKey" type="uddi:businessKey" use="optional"/>
  </complexType>
  ...
</schema>
```

And here's a sample businessService instance:

```
<?xml version='1.0' encoding='UTF-8' ?>
<businessEntity businessKey="01B1FA80-2A15-11D6-9B59-000629DC0A53"
                operator="uddi:ibm"
                xmlns="urn:uddi-org:api_v2">
  ...
  <businessServices>
    <businessService serviceKey="3902AEE0-3301-11D6-9F18-000629DC0A53">
      <name>BookQuoteService</name>
      <description xml:lang="en">
        Given an ISBN number, this service returns the wholesale price
      </description>
      <bindingTemplates>
        <bindingTemplate bindingKey="391E2620-3301-11D6-9F18-000629DC0A53">
          <description xml:lang="en">
            This service uses a SOAP RPC/Literal Endpoint
          </description>
          <accessPoint URLType="http">
            http://www.Monson-Haefel.com/jwsed1/BookQuote
          </accessPoint>
          <tModelInstanceDetails>
            <tModelInstanceInfo
              tModelKey="uddi:4C9D3FE0-2A16-11D6-9B59-000629DC0A53"/>
          </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
    </businessService>
  </businessServices>
```

In private UDDI registries, the **categoryBag** of the businessService is used to indicate things like department number, version number, and such.

bindingTemplate Structure

Like the WSDL port element, a **bindingTemplate** declares the Internet address of a Web service, and a reference to the implementation details. The definition of bindingTemplate structure is:

```
<schema targetNamespace="urn:uddi-org:api_v2"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2"
            version="2.03" id="uddi">
  ...
  <element name="bindingTemplate" type="uddi:bindingTemplate"/>
  <complexType name="bindingTemplate">
    <sequence>
      <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
      <choice>
        <element ref="uddi:accessPoint"/>
        <element ref="uddi:hostingRedirector"/>
      </choice>
      <element ref="uddi:tModelInstanceDetails"/>
    </sequence>
    <attribute name="serviceKey" type="uddi:serviceKey" use="optional"/>
    <attribute name="bindingKey" type="uddi:bindingKey" use="required"/>
  </complexType>
```

The **accessPoint** element provides the exact electronic address of the service. This address may be one of several standard types, including "mailto" (e-mail), "http", "https", "ftp", "phone", or "other". The type of address is indicated by the URLType attribute of the accessPoint element. A bindingTemplate may have only one accessPoint element. If a Web service is accessible via more than one URL, you must define a different bindingTemplate for each URL.

The **tModelInstanceDetails** element contains one or more **tModelInstanceInfo** elements, each of which refers to a tModel that describes some technical aspect of the Web service. For example,

when using a WSDL-based Web service with UDDI, the convention is to refer to a single `tModel` that refers to a specific `port` of a particular WSDL document. In this case the `tModelInstanceDetails` element would be very simple, containing one empty `tModelInstanceInfo` element, with a `tModelKey` attribute that refers to the WSDL `tModel` in the UDDI directory.

tModel Structure

tModel stands for type or technical model. A `tModel` is essentially a namespace with meta-data. It represents a single specification, taxonomy, model, category, identity system, or any other technical concept. There are many standard `tModels`, and you can specify your own custom `tModels` that identify just about anything that can be named. For example, the following is an example of the kind of information required for a WSDL `tModel`:

Name: Monson-Haefel:BookQuote

Description: Given an ISBN, this Web service returns the wholesale price of the book.

UUID: uuid:4C9D3FE0-2A16-11D6-9B59-000629DC0A53

Overview URL: http://www.Monson-Haefel.com/jwsed1/BookQuote.wsdl#xmlns (wsdl=http://schemas.xmlsoap.org/wsdl/) xpointer(/wsdl:definitions/wsdl:portType [@name="BookQuoteBinding"])

Categorization: wsdlSpec

Why use a `tModel` to represent a WSDL document? For one thing, UDDI is very generalized and is not specifically aligned with WSDL, so there is no way to store a WSDL document directly in a UDDI registry. A WSDL document can be stored somewhere else and referred to by a UDDI entry, however. The WSDL `tModel` provides that reference. More importantly, the WSDL `tModel` allows us to attach meta-data to a WSDL reference in the form of categorizations. The BookQuote `tModel` is identified with a categorization of **`wsdlSpec`**, which indicates that this `tModel` refers to a WSDL document. The D-U-N-S `tModel`, in contrast, declares the categorization of `identifier`, which means it refers to some type of identification system. `tModels` make it easy to search for different types of organizations and Web services by categorization or identifiers.

A `tModel` may be assigned any number of categorizations, but every `tModel` must be assigned one of **16 standard categorizations**, called the **UDDI types,** or a subtype thereof. These are a predefined set of types that allow us to categorize all `tModels` using a single system of categorization.

Few UDDI types:

tModel	The root type of all UDDI types. Every `tModel` is based on this type, or a subtype of it. The `tModel` UDDI type is analogous to the `java.lang.Object` type in Java.
identifier	A type of identification system, like the D-U-N-S or Thomas Registry `tModel`.
categorization	A type of taxonomy, like NAICS, UNSPSC, or WAND.
specification	A type of `tModel` that represents some kind of specification for a Web service, like a WSDL document, a RosettaNet PIP document, or a CORBA IDL file.
wsdlSpec	A type of `tModel` that represents a WSDL document.

The `wsdlSpec` UDDI type is applied to a `tModel` when the `tModel` refers to a WSDL document. The BookQuote `tModel` is a good example. `wsdlSpec`, like all UDDI types, is itself categorized by a

tModel called specification, and the specification tModel is in turn categorized as a tModel UDDI type.
When publishing a Web service in UDDI, the bindingTemplate must refer to a WSDL tModel.[BP] In addition, the tModel must be categorized as a wsdlSpec UDDI type (it may have other categorizations as well) and it must provide an XPointer to a specific WSDL binding element in an accessible WSDL document.

```
<tModel tModelKey="UUID:4C9D3FE0-2A16-11D6-9B59-000629DC0A53">
  <name>Monson-Haefel:BookQuote</name>
  <description xml:lang="en">
     Provides a wholesale price given for a ISBN number.
  </description>
  <overviewDoc>
    <description xml:lang="en">
       This URL points to the BookQuote WSDL document.
    </description>
    <overviewURL>http://www.Monson-Haefel.com/jwsed1/BookQuote.wsdl
#xmlns(wsdl=http://schemas.xmlsoap.org/wsdl/) xpointer(/wsdl:definitions/wsdl:portType[
   @name="BookQuoteBinding"])
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference
      tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
      keyName="types"
      keyValue="wsdlSpec"/>
  </categoryBag>
</tModel>
```

Of particular importance is the overviewURL element, which points to the actual WSDL binding represented by the tModel. A WSDL tModel follows the recommendations of version 1.08 of the UDDI Best Practices for Using WSDL in a UDDI Registry.[BP] This document defines **the use of the XPointer reference system to identify a specific binding in a specific WSDL document.** You just have to insert into a small template the proper URL and the name of the WSDL binding element you are referring to. The template to use the xpointer reference is:

```
url-goes-here#xmlns(wsdl=http://schemas.xmlsoap.org/wsdl/) xpointer(/wsdl: definitions
  /wsdl:portType[ @name="binding-name-goes-here"])
```

For example, if your WSDL document is located at http://www.xyxcorp.com/my_wsdl.wsdl and the WSDL binding element is named FooBinding, you create an XPointer that looks like this:

```
http://www.xyxcorp.com/my_wsdl.wsdl#xmlns(wsdl=http://schemas.xmlsoap.org/wsdl/)
  xpointer(/wsdl:definitions/wsdl:portType[ @name="FooBinding"])
```

When accessing a WSDL binding from a UDDI registry, the port elements in the WSDL document are ignored, because the bindingTemplate should provide the correct URL for the Web service. This arrangement allows you to change the access point dynamically in UDDI, without having to modify the WSDL document. If you access a WSDL document via UDDI, use the accessPoint specified by the UDDI bindingTemplate, but if you access a WSDL document without UDDI, use the URL specified by the WSDL port element.

When a tModel is used as a **taxonomy identifier,** it is basically providing context for keyedReference elements in identifierBag and categoryBag elements. Adding a reference to a specific tModel, using a tModelKey attribute, gives the keyedReference context. For example, if the keyedReference refers to a NAICS 1997 number, then we know it refers to the UDDI standard tModel for NAICS 1997:

```
<keyedReference keyName="Book, Periodical, and Newspaper Wholesalers"
                keyValue="42292"
                tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"/>
```

By looking up the tModel associated with the keyedReference we can determine the type of system to which the name-value pair belongs. The NAICS 1997 tModel is:

```
<tModel tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2">
  <name>ntis-gov:naics:1997</name>
  <description xml:lang="en">
```



```
    Business Taxonomy: NAICS (1997 Release)
  </description>
  <overviewDoc>
    <description xml:lang="en">
      This tModel defines the NAICS industry taxonomy.
    </description>
    <overviewURL>
      http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#NAICS
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference
      tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
      keyName="types"
      keyValue="categorization"/>
    <keyedReference
      tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
      keyName="types"
      keyValue="checked"/>
  </categoryBag>
</tModel>
```

The **real power of `tModels`, when used as taxonomies, is that they allow for more concise searching of UDDI registries.** For example, you can conduct a search for a specific business by its D-U-N-S number, or you can search for businesses that fall under a specific NAICS category or are located in a specific ISO 3166 region.

UDDI Standard tModels:

uddi-org:types	UDDI categorization codes for `tModels`
`ntis-gov:naics:1997`	North American Industry Classification System (NAICS), 1997 release
`unspsc-org:unspsc`	Universal Standard Products and Services Classification (UNSPSC), version 6.3
`uddi-org:iso-ch:3166-1999`	ISO 3166 Geographic Taxonomy
`dnb-com:D-U-N-S`	D-U-N-S Number Identifier System
`thomasregister-com:supplierID`	Thomas Register Supplier Identifier Code System
`uddi-org:general-keywords`	UDDI "other" taxonomy

The UDDI type `tModel` provides a taxonomy for categorizing all other `tModels`. **uddi-org:types** kind of a super `tModel`, that all other `tModels` can refer to from their `categoryBag` elements. The `uddi-org:types tModel` includes 16 categories to which a `tModel` can belong.

Key Value	Description
`tModel`	The UDDI type taxonomy is structured to allow for categorization of registry entries other than `tModels`. This key is the root of the branch of the taxonomy that is intended for use in categorization of `tModels` within the UDDI registry. Categorization is not allowed with this key.
`identifier`	An `identifier tModel` represents a specific set of values used to uniquely identify information. `identifier tModels` are intended to be used in `keyedReferences` inside of `identifierBags`. For example, a Dun & Bradstreet D-U-N-S number uniquely identifies companies globally. The D-U-N-S number taxonomy is an `identifier` taxonomy.
`namespace`	A `namespace tModel` represents a scoping constraint or domain for a set of information. In contrast to an `identifier`, a `namespace` does not have a predefined set of values within the domain, but acts to avoid collisions. It is similar to the namespace functionality used for XML. For example, the `uddi-org:relationship tModel`, which is used to assert relationships between business entities, is a `namespace tModel`.
`categorization`	A `categorization tModel` is used for information taxonomies within the UDDI registry. NAICS and UNSPSC are examples of

	categorization tModels.
relationship	A relationship tModel is used for relationship categorizations within the UDDI registry; relationship tModels are typically used in connection with publisher-relationship assertions.
postalAddress	A postalAddress tModel is used to identify different forms of postal address within the UDDI registry; postalAddress tModels may be used with the address element to distinguish different forms of postal address.
specification	A specification tModel is used for tModels that define interactions with a Web service. These interactions typically include the definition of the set of requests and responses, or other types of interaction, that are prescribed by the service. tModels describing XML, COM, CORBA, or any other service are specification tModels.
xmlSpec	An xmlSpec tModel is a refinement of the specification tModel type. It is used to indicate that the interaction with the service is via XML. The UDDI API tModels are xmlSpec tModels.
soapSpec	Further refining the xmlSpec tModel type, a soapSpec is used to indicate that the interaction with the service is via SOAP. The UDDI API tModels are soapSpec tModels, in addition to xmlSpec tModels.
wsdlSpec	A tModel for a Web Service described using WSDL is categorized as a wsdlSpec.
protocol	A tModel describing a protocol of any sort.
transport	A transport tModel is a specific type of protocol. HTTP, FTP, and SMTP are types of transport tModels.
signatureComponent	A signatureComponent is used in cases where a single tModel cannot represent a complete specification for a Web service. This is the case for specifications like RosettaNet, where implementation requires the composition of three tModels to be complete—a general tModel indicating RNIF, one for the specific PIP, and one for the error-handling services. Each of these tModels would be of type signatureComponent, in addition to any others as appropriate.
unvalidatable	Used to mark a categorization or identifier tModel as unavailable for use. tModels representing checked value sets are marked unvalidatable as they are brought on-line, and to retire them.
checked	Marking a tModel with this classification asserts that it represents a categorization, identifier, or namespace tModel that has a properly registered validation service per the UDDI Version 2.0 Operators Specification Appendix A.
unchecked	Marking a tModel with this classification asserts that it represents a categorization, identifier, or namespace tModel that does not have a validation service.

The UDDI types are also handy for searches. If, for example, you wanted a list of all the organizations that had WSDL-based Web services, you could get it by searching for organizations whose technology tModel was categorized with a keyValue of "wsdlSpec".

A **checked** tModel is a categorization or identification system that requires validation to ensure that values added in a keyedReference element are correct. If a tModel is checked, it will contain a uddi-org:type categorization of checked. A tModel not categorized as checked is assumed to be unchecked, but you can also explicitly categorize it as such. Except for General Keyword, all of the UDDI-standard tModels listed in above table are checked tModels.

Structure for tModels:

Although the use of tModels varies, their structures are all the same. They include name, description, overviewDoc, identifierBag, and categoryBag elements.

```
<schema targetNamespace="urn:uddi-org:api_v2"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2"
            version="2.03" id="uddi">
  ...
  <element name="tModel" type="uddi:tModel"/>
  <complexType name="tModel">
    <sequence>
      <element ref="uddi:name"/>
      <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="uddi:overviewDoc" minOccurs="0"/>
      <element ref="uddi:identifierBag" minOccurs="0"/>
      <element ref="uddi:categoryBag" minOccurs="0"/>
    </sequence>
    <attribute name="tModelKey" type="uddi:tModelKey" use="required"/>
    <attribute name="operator" type="string" use="optional"/>
    <attribute name="authorizedName" type="string" use="optional"/>
  </complexType>
```

The **name** element is, by convention, based on a URI and is used to identify the tModel succinctly. The name doesn't have to be unique within a particular UDDI registry, however; the tModel's UUID is its unique identifier. The **description** element may appear multiple times for different languages—qualified by the xml:lang attribute, as in the businessEntity structure.

```
<tModel tModelKey="UUID:4C9D3FE0-2A16-11D6-9B59-000629DC0A53">
  <name>Monson-Haefel:BookQuote</name>
  <description xml:lang="en">The WSDL document for BookQuote</description>
  ...
</tModel>
```

The **overviewDoc** may contain an optional **description** element (xml:lang-qualified) and an **overviewURL** element. The overviewURL element can be any valid URL, but the convention is to use a URL that points to a file you can obtain with a standard HTTP GET operation, or download using a common Web browser. The tModel used for a WSDL-based Web service will usually point to a WSDL document, which can be accessed with an HTTP GET operation.

```
    <overviewURL> http://www.Monson-Haefel.com/jwsed1/BookQuote.wsdl
  #xmlns(wsdl=http://schemas.xmlsoap.org/wsdl/) xpointer(/wsdl:definitions/wsdl:portType[
   @name="BookQuoteBinding"])
    </overviewURL>
```

The **identifierBag** element is not used much with tModels except for a special UDDI identifier called **isReplacedBy**. The isReplacedBy identifier indicates that one tModel has been replaced by another—this might occur if a new version of the technology represented by the tModel is used.

```
<tModel tModelKey="UUID:4C9D3FE0-2A16-11D6-9B59-000629DC0A53">
  <name>Monson-Haefel:BookQuote</name>
  <description xml:lang="en">
     Provides a wholesale price given for an ISBN number.
  </description>
  <overviewDoc>
    <description xml:lang="en">
       This URL points to the BookQuote WSDL document.
    </description>
    <overviewURL> http://www.Monson-Haefel.com/jwsed1/BookQuote.wsdl
#xmlns{wsdl=http://www.Monson-Haefel.com/BookQuote}
xpointer{//wsdl:binding[@name='BookQuoteBinding']}
    </overviewURL>
  </overviewDoc>
  <identifierBag>
    <keyedReference keyName="uddi-org:isReplacedBy"
                    keyValue= "uuid:A84ED203-33D5-04A2-C262-49C293E82DE2"
                    tModelKey="uuid:E59AE320-77A5-11D5-B898-0004AC49CC1E" />
  </identifierBag>
```



```
  <categoryBag>
    <keyedReference
         tModelKey="uuid:65719168-72c6-3f29-8c20-62defb0961c0"
         keyName="ws-I_conformance:BasicProfile1.0"
         keyValue="http://ws-i.org/profiles/basic/1.0" />
    <keyedReference
         tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
         keyName="uddi-org:types"
         keyValue="wsdlSpec"/>

  </categoryBag>
</tModel>
```

The **categoryBag**, on the other hand, is used with tModels quite a bit. the tModel for Monson-Haefel Books' BookQuote Web service, which is categorized as both a wsdlSpec tModel and a ws-I_conformance:BasicProfile1.0 tModel.

publisherAssertion Structure

A publisherAssertion structure defines relationships between two business entities. The data structure identifies both participants, as well as a keyedReference to a tModel that defines the relationship.

Structure of publisherAssertion:

```
<schema targetNamespace="urn:uddi-org:api_v2"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2"
            version="2.03" id="uddi">
  ...
  <element name="publisherAssertion" type="uddi:publisherAssertion"/>
  <complexType name="publisherAssertion">
    <sequence>
      <element ref="uddi:fromKey"/>
      <element ref="uddi:toKey"/>
      <element ref="uddi:keyedReference"/>
    </sequence>
  </complexType>
```

In an instance of this structure, the fromKey and toKey elements will contain the unique UDDI businessEntity identifiers of the entities involved in the relationship. For example, the fromKey might contain the UUID of the IBM Corporation, while the toKey contains the UUID of IBM's Professional Services division. The keyedReference element will point to a tModel that represents the relationship between these organizations.

```
<publisherAssertion>
    <fromKey>0207DE98-9C61-4138-A121-4B9E636B7649</fromKey>
    <toKey>1EE48BF0-9356-11D5-8838-002035229C64</toKey>
    <keyedReference keyName="subsidiary"
                    keyValue="parent-child"
                    tModelKey="uuid:807A2C62-EE22-470D-ADC6-E0424A337C03"/>
</publisherAssertion>
```

In order for the publisherAssertion to be valid and therefore visible in the UDDI registry, both entities must submit complementary publisherAssertion entries. If only one of them does, that assertion will not be visible to anyone but the publisher, and will not be considered valid. This requirement prevents misrepresentation. An organization cannot feign a relationship that the other organization does not recognize—for example, that it's a division of another company, a member of a consortium, or a preferred vendor.

UUID Keys

Most of the primary UDDI data structures (businessEntity, business Service, bindingTemplate, and tModel) are automatically assigned a Universally Unique Identifier (UUID) key when they are added to the UDDI registry**. The UUID is a hexadecimal-encoded number that is about 30 characters long and is generated using the DCE UUID-generation algorithm**. The UUID is globally unique—there will not be duplicates even in other registries. For eg; 01B1FA80-2A15-11D6-9B59-000629DC0A53.

The UUIDs for the primary data structures all take the same form, except for the tModel, which prefixes its UUID value with "uuid:". The tModel is the only data structure that does this, because the tModelKey is supposed to be a valid URI (Uniform Resource Identifier)—hence the uuid prefix.

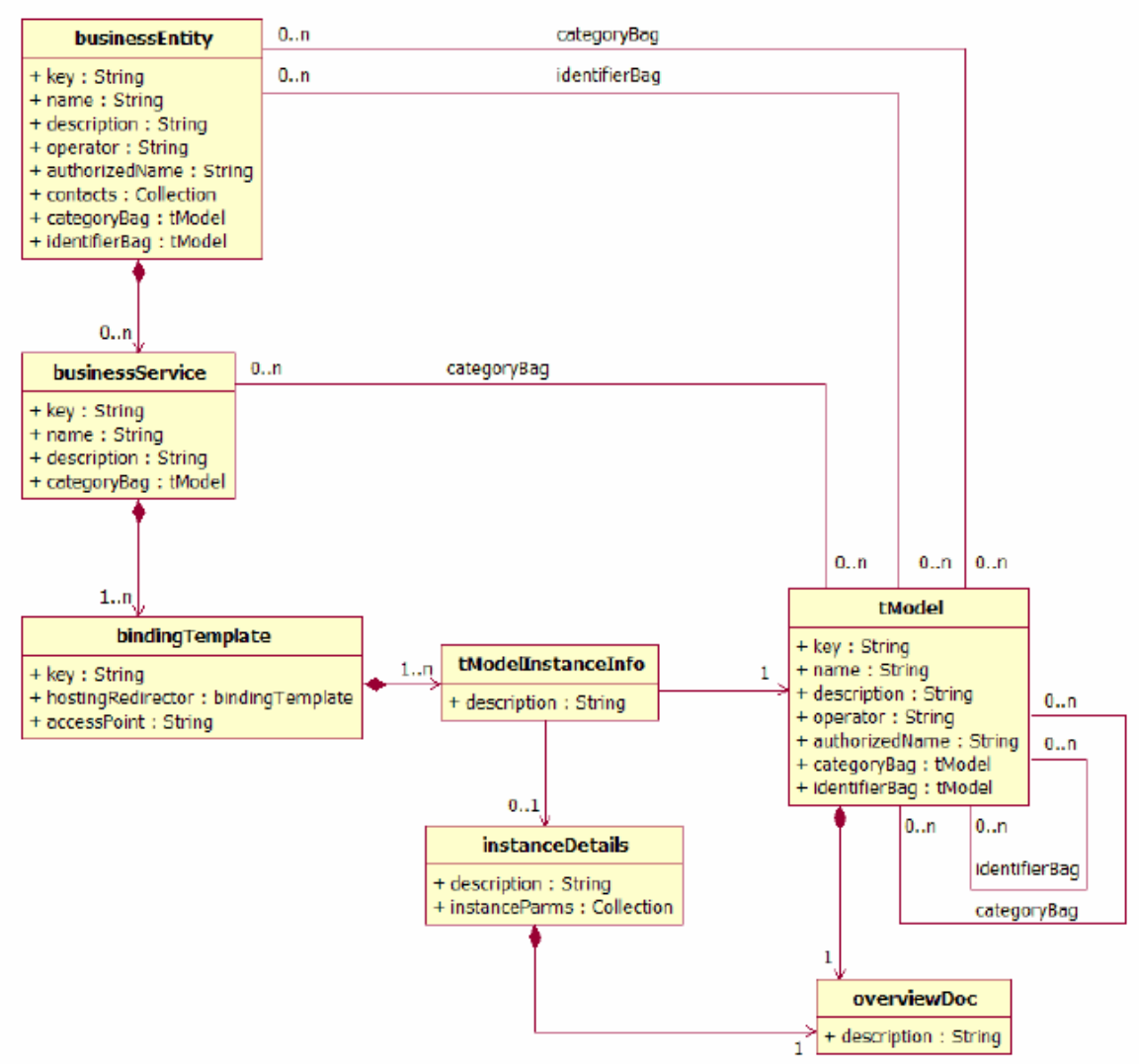
WS-I Conformance Claims

A `tModel` that represents a Web service may claim that it adheres to the WS-I Basic Profile 1.0 by including a WS-I conformance claim categorization.

```
<tModel tModelKey="UUID:4C9D3FE0-2A16-11D6-9B59-000629DC0A53">
  <name>Monson-Haefel:BookQuote</name>
  <description xml:lang="en">
     Provides a wholesale price given for a ISBN number.
  </description>
  <overviewDoc>
    ...
  </overviewDoc>
  <categoryBag>
    <keyedReference
         tModelKey="uuid:65719168-72c6-3f29-8c20-62defb0961c0"
         keyName="ws-I_conformance:BasicProfile1.0"
         keyValue="http://ws-i.org/profiles/basic/1.0" />
  </categoryBag>
</tModel>
```

If a conformance claim is used in UDDI, there must be a corresponding WS-I conformance claim in the WSDL `binding` element.

Summary:



UDDI Inquiry API

The UDDI specification requires that **UDDI registries support a specific set of SOAP-based Web service** operations called the **UDDI Programming API**. These SOAP messages use the **Document/Literal** mode of messaging and are described in detail by **WSDL documents located at the UDDI.org** Web site. This makes **UDDI a Web service**, just like any other WSDL/SOAP-based Web service. UDDI's standard Web services are divided into two WSDL/SOAP-based APIs: The Inquiry API and the Publishing API. The Inquiry API is used to search and read data in a UDDI registry, while the Publishing API is used to add, modify, and delete data in a UDDI registry.

All UDDI Inquiry and Publishing operations use **Document/Literal** SOAP messages and all of them are **Request/Response**; meaning that the UDDI registry always replies with either a SOAP message or a SOAP fault. UDDI requires the use of UTF-8 only (and not UTF-16). This means that UDDI is not exactly conformant with the WS-I Basic Profile 1.0, which requires Web services to support both UTF-8 and UTF-16. The basic structure of a UDDI SOAP message and its HTTP header is shown below:

```
POST /someVerbHere HTTP/1.1
Host: www.someoperator.org
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <some-uddi-element generic="2.0" xmlns="urn:uddi-org:api_v2">
      ...
    </some-uddi-element>
  </Body>
</Envelope>
```

Because **UDDI doesn't support the use of the `Header` element** and all messages are Document/Literal, all you need to know is the structure of the XML document fragment in the `Body` element. Just replace `some-uddi-element` with the proper UDDI data structure and you have a UDDI SOAP message.

All UDDI XML types are part of the `"urn:uddi-org:api_v2"` namespace for UDDI 2.0. In addition to specifying the proper namespace, you also have to include the `generic` attribute with a value of `"2.0"`, to signify that the SOAP message conforms to the UDDI 2.0 Programming API. All SOAP messages in UDDI are carried in HTTP POST and reply messages. An HTTP POST request message must declare a `SOAPAction` header, but the value can be an empty string or any value.

You can access a UDDI Inquiry or Publishing API using any SOAP 1.1 toolkit. Generating interfaces from UDDI's WSDL documents can result in a convenient Java API for UDDI (using JAX-RPC code generator). But using JAXR is more convenient.

UDDI Operations

You use the Inquiry API for querying the UDDI registry and fetching specific UDDI data structures. When you want to search the registry, you use a SOAP **find** operation. When you want to get the data associated with a specific entry, such as a `businessEntity`, `businessService`, or `tModel`, you use a SOAP **get** operation.

Find Operations

The find operations allow you to search the UDDI registry for data structures that match some criteria. These are:

Operation	Description
`find_business`	finds matching `businessEntity` entries.
`find_relatedBusiness`	finds matching `publisherAssertion` entries.
`find_service`	finds matching `businessService` entries.
`find_binding`	finds matching `bindingTemplate` entries.
`find_tModel`	finds matching `tModel` entries.

```
POST /someVerbHere HTTP/1.1
Host: www.someoperator.org
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
       <categoryBag>
         <!-- ISO 3166 -->
         <keyedReference
                      keyName="Minnesota, USA"
                      keyValue="US-MN"
                      tModelKey="uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88" />
         <!-- NAICS -->
         <keyedReference
                      keyName="Book, Periodical, and Newspaper Wholesalers"
                      keyValue="42292"
                      tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2" />
       </categoryBag>
    </find_business>
  </Body>
</Envelope>
```

Only those businessEntity entries that declare both of these keyedReference values in their **categoryBag** will be a match. **For categoryBag, the default behavior is an AND condition**: Both keyedReference values must match. In this case the client is searching for all book wholesalers that are also located in the state of Minnesota (United States). This query would find a positive match with Monson-Haefel's business because its businessEntity structure contains matching keyedReference values in its categoryBag element.

```
<?xml version='1.0' encoding='UTF-8'?>
<businessEntity businessKey="01B1FA80-2A15-11D7-9B59-000629DC0A53"
                xmlns="urn:uddi-org:api_v2">
  ...
  <identifierBag>
    <!-- D-U-N-S® Number Identifier System -->
    <keyedReference keyName="Monson-Haefel Books, Inc."
        keyValue="038924499"
        tModelKey="uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823"/>
  </identifierBag>
  <categoryBag>
    <!-- North American Industry Classification System (NAICS) 1997 -->
    <keyedReference keyName="Book, Periodical, and Newspaper Wholesalers"
        keyValue="42292"
        tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"/>
    <!-- Universal Standard Products and Services Classification (UNSPSC)
         Version 7.3 -->
    <keyedReference keyName="Educational or vocational textbooks"
        keyValue="55.10.15.09.00"
        tModelKey="uuid:CD153257-086A-4237-B336-6BDCBDCC6634"/>
    <!-- ISO 3166 Geographic Taxonomy -->
    <keyedReference keyName="Minnesota, USA"
        keyValue="US-MN"
        tModelKey="uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88"/>
  </categoryBag>
</businessEntity>
```

Search Elements

A variety of search elements can be used in the find operations, including identifierBag, categoryBag, name, tModelBag, and findQualifiers.

You can use the identifierBag in the same way. You can search for entries that have matching keyedReference values in their **identifierBag** elements, using the find_business and find_tModel operations. **The default behavior for identifierBag searches is an OR match:** Any entity that contains at least one matching value is considered a match.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
       <identifierBag>
          <!-- D-U-N-S -->
          <keyedReference keyName="Monson-Haefel Books, Inc."
                    keyValue="038924499"
                    tModelKey="uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823" />
       </identifierBag>
    </find_business>
  </Body>
</Envelope>
```

You can search for matching values in both categoryBag and identifierBag in a single search operation (find_business and find_tModel only), but you'll get more matches if you modify the default behavior to use OR search criteria—otherwise the categoryBag is likely to force an exact match. To override the default search behavior you use the **findQualifiers** element.

The **findQualifiers** element (plural) may contain one or more findQualifier elements (singular), each of which may specify a different qualifier for modifying the default matching behavior of the find operation.

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

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
       <findQualifiers>
          <findQualifier>orAllKeys</findQualifier>
       </findQualifiers>
       <categoryBag>
         <keyedReference keyName="Minnesota, USA"
                         keyValue="US-MN"
                         tModelKey="uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88" />
         <keyedReference keyName="Book, Periodical, and Newspaper Wholesalers"
                         keyValue="42292"
                         tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2" />
       </categoryBag>
    </find_business>
  </Body>
</Envelope>
```

This find operation will return a list of all businesses that have at least one matching `keyedReference` element: all the companies located in the state of Minnesota, regardless of their industry, as well as all the wholesale book companies, regardless of their location.

There are 11 qualifier values you can use with find operations, some of which apply only to certain find operations. They are, exactNameMatch, caseSensitiveMatch, sortByName{AsclDesc}, sortByDate{AsclDesc}, orLikeKeys, orAllKeys, combineCategoryBags, serviceSubset, andAllKeys. **[For description read from J2EE Webservices book.]**

name search element:

You can also **search for UDDI entries by name,** with the `find_business`, `find_service`, and `find_tModel` operations.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
       <name>Monson-Haefel</name>
       <name xml:lang="en">Addison</name>
    </find_business>
  </Body>
</Envelope>
```

You can specify multiple `name` elements to perform, by default, an **ORed search.** In the above example the reply message will contain all the business whose name starts with either "Monson-Haefel" or "Addison." You can **use a wild-card character (`%`) in `name` searches.** By default, a wild card is assumed to be at the end of each name, so a search on `"Monson-Haefel"`, for example, might return entries named Monson-Haefel Books and Monson-Haefel Industries, Inc. You can explicitly specify one or more wild-card characters, overriding the default behavior. For example, a `name` search on the value **`"Am%com"` might return entries for Amazon.com and American Qualicom.**

tModelBag search element:

The `find_business`, `find_service`, and `find_binding` operations can all use the `tModelBag` search element in their request messages. The `tModelBag` search element contains one or more `tModelKey` elements, each of which specifics a unique identifier for a technical `tModel`. The search is actually applied to the `tModels` that describe the Web service.

```
<?xml version="1.0" encoding="UTF-8"?>
<businessEntity businessKey="01B1FA80-2A15-11D7-9B59-000629DC0A53"
 xmlns="urn:uddi-org:api_v2">
  ...
  <name>Monson-Haefel Books, Inc.</name>
  <description xml:lang="en">Technical Book Wholesaler</description>
  <businessServices>
    <businessService serviceKey="3902AEE0-3301-11D7-9F17-000629DC0A53">
      <name>BookQuote</name>
      <bindingTemplates>
        <bindingTemplate bindingKey="391E2620-3301-11D7-9F17-000629DC0A53">
          <accessPoint URLType="http">
            http://www.Monson-Haefel.com/jwsed1/BookQuote
          </accessPoint>
```

```
          <tModelInstanceDetails>
            <tModelInstanceInfo tModelKey="uddi:4C9D3FE0…9B59-000629DC0A53"/>
            <tModelInstanceInfo tModelKey="uddi:2B4C3DE0…7B22-000438FE0C22"/>
          </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
    </businessService>
  </businessServices>
</businessEntity>
```

A `tModelBag` search element will list only those data structures that contain all the `tModelKeys` specified—in other words **it's an AND search.**

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_service generic="2.0" xmlns="urn:uddi-org:api_v2">
       <tModelBag>
           <tModelKey>uddi:4C9D3FE0-2A16-11D7-9B59-000629DC0A53</tModelKey>
           <tModelKey>uddi:2B4C3DE0-23B4-84FE-7B22-000438FE0C22</tModelKey>
       </tModelBag>
    </find_service>
  </Body>
</Envelope>
```

Operation Definition and Payload

Every `find` operation takes exactly the same form, because they are all **Document/ Literal SOAP messages.** The data structure, which contains the criteria of the search, is nested directly in the SOAP `Body` element:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <!-- the find data structure goes here -->
  </Body>
</Envelope>
```

A find request message may specify a **`maxRows`** attribute that allows the UDDI registry to limit the number of results returned. Response messages will include the **`operator`** attribute, which provides the URI of the UDDI operator you are querying, and possibly a **`truncation`** attribute, which indicates that the data returned was too large and was truncated.
For detailed UDDI v2 schema refer to http://uddi.org/schema/uddi_v2.xsd.

Given a set of criteria (categories, identifiers, `tModels`, or `discoveryURLs`), the **`find_business`** operation will return a lightweight list of `businessEntity` listings, including their keys, names, descriptions, and `businessService` names and keys. You are most likely to use this operation to browse for businesses.

```
<schema targetNamespace="urn:uddi-org:api_v2"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2"
            version="2.03" id="uddi">
  ...
  <!-- Request Structure -->
  <element name="find_business" type="uddi:find_business"/>
  <complexType name="find_business">
    <sequence>
      <element ref="uddi:findQualifiers" minOccurs="0"/>
      <element ref="uddi:name" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="uddi:identifierBag" minOccurs="0"/>
      <element ref="uddi:categoryBag" minOccurs="0"/>
      <element ref="uddi:tModelBag" minOccurs="0"/>
      <element ref="uddi:discoveryURLs" minOccurs="0"/>
    </sequence>
    <attribute name="generic" type="string" use="required"/>
    <attribute name="maxRows" type="int" use="optional"/>
  </complexType>
  <!-- Response Structure -->
  <xsd:element name="businessList" type="uddi:businessList"/>
  <complexType name="businessList">
```

```
    <sequence>
      <element ref="uddi:businessInfos"/>
    </sequence>
    <attribute name="generic" type="string" use="required"/>
    <attribute name="operator" type="string" use="required"/>
    <attribute name="truncated" type="uddi:truncated" use="optional"/>
  </complexType>
...
</schema>
```

The WSDL message and portType definitions for find_business method is shown below:

```
<definitions ...
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:uddi="urn:uddi-org:api_v2">
  ...
  <message name="find_business">
    <part name="body" element="uddi:find_business"/>
  </message>
  <message name="businessList">
    <part name="body" element="uddi:businessList"/>
  </message>
  ...
  <portType name="Inquire">
    ...
    <operation name="find_business">
      <input message="tns:find_business"/>
      <output message="tns:businessList"/>
      <fault name="error" message="tns:dispositionReport"/>
    </operation>
    ...
  </portType>
  ...
</definitions>
```

The **find_relatedBusinesses** operation returns a lightweight list of all the businesses that have visible publisherAssertion relationships with a specified organization. The search can be modified to list a subset of all related business, according to keyedReference elements. [For schema definition for this method and its WSDL message and portType definitions refer to J2EE Webservices book pg 258.]

The **find_service** operation will return a lightweight list of all the business Service entries that match the given categories, tModel keys, or both. [For schema definition for this method and its WSDL message and portType definitions refer to J2EE Webservices book pg 260.]

The **find_binding** operation returns a set of zero or more bindingTemplate entries whose tModels match those specified in the query. [For schema definition for this method and its WSDL message and portType definitions refer to J2EE Webservices book pg 261.]

The **find_tModel** operation finds all the tModels that match the names, identifiers, or categories listed in the request message. It returns a lightweight list of tModel keys. [For schema definition for this method and its WSDL message and portType definitions refer to J2EE Webservices book pg 263.]

GET Operations

The get operations allow you to request specific data structures by their unique identifiers. A get operation can return one or many of the same type of data structure, depending on how many unique identifiers you supply in the request message. There are five get operations:

Operation	Description
get_businessDetail	gets businessEntity entries.
get_businessDetailExt	gets businessEntityExt entries.
get_serviceDetail	gets businessService entries.
get_bindingDetail	gets bindingTemplate entries.
get_tModelDetail	gets tModel entries.

The SOAP request messages used by the get operations all have the same basic form. In the request message the XML fragment in the SOAP `Body` element contains a list of UUID keys:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <get_something generic="2.0" operator="operatorURL"
                   xmlns="urn:uddi-org:api_v2">
       <somethingKey>4C9D3FE0-2A16-11D7-9B59-000629DC0A53</somethingKey>
       <somethingKey>2B4C3DE0-23B4-84FE-7B22-000438FE0C22</somethingKey>
       ...
    </get_something>
  </Body>
</Envelope>
```

All the response messages are structured alike as well. They simply return a list of data structures appropriate to the get operation.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <somethingDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
       <some-data-structure>…<some-data-structure>
       <some-data-structure>…<some-data-structure>
       ...
    </somethingDetail>
  </Body>
</Envelope>
```

For example, the `get_businessDetail` operation returns a list of `business Entity` data structures. The `get_bindingDetail` returns a list of `bindingTemplate` data structures, and so on.

The **`get_businessDetail`** operation requests one or more `businessEntity` data structures by their unique business keys.

The **`get_businessDetailExt`** operation requests one or more `business EntityExt` data structures by the unique business keys assigned to the `business Entitys`.

The **`get_serviceDetail`** operation requests one or more `businessService` data structures by their unique service keys.

The **`get_bindingDetail`** operation requests one or more `bindingTemplate` data structures by their unique binding keys.

The **`get_tModelDetail`** operation requests one or more `tModel` data structures by their unique `tModel` keys.

Summary:
Access any UDDI public directory and try to find organizations by a specific category and you'll quickly discover that searching is difficult because organizations are not using categorizations consistently, as you would expect them to or are entering data improperly like specifying their WSDL URL in accessPoint element of businessService rather than in tModel where its supposed to be.

UDDI Publishing API

Organizations use the Publishing API to add, change, and delete information in a UDDI registry. This API allows organizations to save their own `businessEntity`, `businessService`, `bindingTemplate`, `tModel`, and `publisherAssertion` data structures in a UDDI registry, and to remove them when necessary. Publishing API is a full-fledged Web service based on SOAP's **Document/Literal mode of messaging**, and is described by a WSDL document. Unlike the Inquiry API, the **Publishing API requires that UDDI operators use HTTPS** (HTTP with SSL 3.0) for confidentiality and some form of authentication. The API supports **four kinds** of **operations**:

1. authorization operations,

2. save operations,
3. delete operations, and
4. get operations.

The authorization operations allow you to authenticate yourself, obtain an authorization token, and terminate a session and its authentication token. The save operations let you add or update the primary data structures. The delete operations let you remove primary data structures. The get operations let you view `publisherAssertions` and a summary of registered information.

Operation Definition and Payloads

The `dispositionReport` and `result` types are used for response messages to delete operations as well as for error messages. The `authInfo` type, which contains the authentication token, is included in every publisher request message except `get_authToken`.

Authorization Operations

Before you can publish anything to a UDDI registry, you have to enroll with a specific UDDI operator. Once you've enrolled, you can use your user-id and password to log in, then use the Publishing APIs to add, modify, and delete data in the UDDI registry. There are two authorization operations:

1. **get_authToken** logs you into the registry
2. **discard_authToken** logs you out of the registry.

If supported, the client must invoke **get_authToken** before starting a publishing session. To start a publishing session with a UDDI operator, you first establish an HTTPS connection with the UDDI registry and then send it a `get_authToken` SOAP message containing your login credentials (usually user-id and password). If your credentials pass authentication, then the reply message will contain an authorization token, which you must send back to the UDDI registry in all subsequent SOAP messages for the duration of your connection.

When you are finished accessing the UDDI Publishing endpoint, you can terminate your session by sending a **`discard_authToken`** message. The registry will then invalidate your authorization token. Further access with that token will cause an error, so your session is effectively ended. If the `discard_authToken` is never sent, the session will simply time out, which also invalidates the authorization token.

Save Operations

The save operations allow you to add or update information in the UDDI registry. Each of the five primary data structures has a corresponding save operation except `publisherAssertion`, which has special add and set operations:

Operation	Description
`save_business`	adds or updates one or more `businessEntity` entries.
`save_service`	adds or updates one or more `businessService` entries.
`save_binding`	adds or updates one or more `bindingTemplate` entries.
`save_tModel`	adds or updates one or more `tModel` entries.
`add_publisherAssertions`	adds one or more `publisherAssertion` entries.
`set_publisherAssertions`	updates one or more `publisherAssertion` entries.

The SOAP request and response messages used for the save operations all take the same basic form: The request message carries one or more primary data structures to be added or updated, while the response returns the data structures that were successfully updated or added. To update any part of a data structure, you have to submit the whole thing; you cannot update a single field.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <save_something generic="2.0" xmlns="urn:uddi-org:api_v2">
      <authInfo>...</authInfo>
      <some-data-structure>...<some-data-structure>
      <some-data-structure>...<some-data-structure>
```



```
      ...
    </save_something>
  </Body>
</Envelope>
```

The response message, provided it's not a SOAP fault, returns the data structures that were updated:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <somethingDetail generic="2.0" operator="operatorURI"
                     xmlns="urn:uddi-org:api_v2">
      <some-data-structure>...<some-data-structure>
      <some-data-structure>...<some-data-structure>
      ...
    </somethingDetail>
  </Body>
</Envelope>
```

The **`save_business`** operation adds or updates one or more `businessEntity` data structures in a UDDI registry.

The **`save_service`** operation adds or updates one or more `businessService` data structures in a UDDI registry.

The **`save_binding`** operation adds or updates one or more `bindingTemplate` data structures in a UDDI registry.

The **`save_tModel`** operation adds or updates one or more `tModel` data structures in a UDDI registry.

The **`add_publisherAssertions`** operation adds one or more `publisherAssertion` data structures to a UDDI registry. A `publisherAssertion` is not valid and visible until both parties submit assertions with the same data.

The **`set_publisherAssertions`** operation updates one or more existing `publisherAssertion` data structures to a UDDI registry. Because a `publisherAssertion` is not valid unless both parties assert the same relationship, they must perform complementary updates in order to keep the `publisherAssertion` visible.

Delete Operations

Operation	Description
`delete_business`	Deletes one or more `businessEntity` entries.
`delete_service`	Deletes one or more `businessService` entries.
`delete_binding`	Deletes one or more `bindingTemplate` entries.
`delete_tModel`	Deletes (or rather, hides) one or more `tModel` entries.
`delete_publisherAssertions`	Deletes `publisherAssertion` entries.

Structure of SOAP request message is similar to the save_xxx methods. Unlike the response message of a save operation, the delete operation's response doesn't contain the data in question. Unless it's reporting a fault, the response message contains a **`dispositionReport`** element.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <dispositionReport generic="2.0" operator="operatorURI"
                       xmlns="urn:uddi-org:api_v2">
      <result errno="0" >
         <errInfo errCode="E_success" />
      </result>
    </dispositionReport>
  </Body>
</Envelope>
```

If the operation is a success, the reply message for all delete operations will look exactly like the one above. The dispositionReport element is also used to report errors.

The **delete_business** operation deletes one or more businessEntity entries from a UDDI registry. In addition, all businessServices, bindingTemplates, and publisherAssertions owned by any deleted businessEntity will be permanently removed from the UDDI registry. The only exceptions are:

1. bindingTemplates that other bindingTemplates refer to as hosting redirectors.
2. projected references, data types referred to by the businessEntity being deleted, but owned by some other businessEntity
3. tModels, which must be deleted explicitly using the delete_tModel operation

The **delete_service** operation deletes one or more businessService data structures from a UDDI registry. All bindingTemplates owned by the deleted businessService will also be permanently removed from the UDDI registry. The exceptions are the same as for delete_business: projected references, hosting redirectors, and tModels.

The **delete_binding** operation deletes one or more bindingTemplate data structures from a UDDI registry. tModels referred to by bindingTemplates being deleted are not affected. They must be deleted explicitly using the delete_tModel operation.

The **delete_tModel** operation **does not actually delete tModel data structures from a UDDI registry**. **Instead the tModels are made invisible to find operations**—although you can still retrieve them using a get_tModelDetail message. To remove a tModel from a UDDI registry permanently, you must petition the UDDI operator to delete it.

The **delete_publisherAssertions** operation deletes one or more publisherAssertion data structures from a UDDI registry. This delete operation is different from others because a publisherAssociation doesn't have a UUID key; you delete a publisherAssociation using its to and from keys as a compound key.

Get Operations

Operation	Description
get_assertionStatusReport	gets a summary of publisherAssertion entries.
get_publisherAssertions	gets a list of publisherAssertion entries.
get_registeredInfo	gets an abbreviated list of businessEntity and tModel entries.

The **get_assertionStatusReport** operation allows you to see the status of any assertions that you have made, or that have been made about businessEntity entries that you control. You specify the assertion status you want to view using a code in the request message.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <get_assertionStatusReport generic="2.0" xmlns="urn:uddi-org:api_v2">
      <authInfo>...</authInfo>
      <completionStatus>status:complete</completionStatus>
    </get_assertionStatusReport>
  </Body>
</Envelope>
```

Other completionStatus values are: **status:toKey_incomplete** and **status_fromKeyIncomplete**.
The reply SOAP message will contain a list of zero or more assertion StatusItem elements. Each of these provides the same data as a publisherAssertion entry but also includes a keysOwned element. The keysOwned element indicates which of the keys (toKey or fromKey) is owned by the

person making the request—which makes it a little easier to figure out who is making the assertion, you or the other party.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <assertionStatusReport generic="2.0" operator="operatorURI"
                           xmlns="urn:uddi-org:api_v2" >

      <assertionStatusItem completionStatus="status:complete" >
        <fromKey>0207DE98-9C61-4138-A121-4B9E636B7649</fromKey>
        <toKey>1EE48BF0-9357-11D5-8838-002035229C64</toKey>
        <keyedReference
           keyName="corporate division"
           keyValue="parent-child"
           tModelKey="uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03"  />
        <keysOwned>
           <toKey>1EE48BF0-9357-11D5-8838-002035229C64</toKey>
        </keysOwned>
      </assertionStatusItem>
    </assertionStatusReport>
  </Body>
</Envelope>
```

The `get_publisherAssertions` operation returns all the `publisherAssertion` entries you have submitted to the UDDI registry, both visible and invisible.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <get_publisherAssertions generic="2.0" xmlns="urn:uddi-org:api_v2">
      <authInfo>...</authInfo>
    </get_publisherAssertions>
  </Body>
</Envelope>
```

The UDDI registry will look up all the `publisherAssertion` entries made by the party associated with the `authInfo` value. The SOAP response message will contain zero or more `publisherAssertion` data structures that you have submitted in the past, both visible (confirmed by the other party) and invisible (not confirmed by the other party).

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <publisherAssertions generic="2.0" authorizedName="yourName"
        operator="OperatorURL" xmlns="urn:uddi-org:api_v2" >

      <publisherAssertion>
        <fromKey>0207DE98-9C61-4138-A121-4B9E636B7649</fromKey>
        <toKey>1EE48BF0-9357-11D5-8838-002035229C64</toKey>
        <keyedReference
           keyName="Corporation"
           keyValue="corporation-has-division"
           tModelKey="uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03"  />
      </publisherAssertion>
      ...
    </publisherAssertions>
  </Body>
</Envelope>
```

The **`get_registeredInfo`** operation returns an abbreviated list of all the `businessEntity` and `tModel` structures controlled by the individual who made the request.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <get_registeredInfo generic="2.0" xmlns="urn:uddi-org:api_v2">
      <authInfo>...</authInfo>
    </get_registeredInfo >
  </Body>
</Envelope>
```


The response SOAP message contains a summary of the data controlled by the requester in the form of `businessInfos` and `tModelInfos` elements.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <registeredInfo generic="2.0" xmlns="urn:uddi-org:api_v2"
                    operator="OperatorURL">
      <businessInfos>
         <businessInfo businessKey="01B1FA80-2A15-11D7-9B58-000629DC0A53">
           <name>Monson-Haefel, Inc.</name>
           <serviceInfos>
             <serviceInfo businessKey="01B1FA80-2A15-11D7-9B58-000629DC0A53"
                          serviceKey="807A2C6A-EE22-470D-ADC7-E0424A337C03">
              <name>BookQuoteService</name>
             </serviceInfo>
           </serviceInfos>
         </businessInfo>
         ...
      </businessInfos>
      <tModelInfos>
        <tModelInfo tModelKey="UUID:4C9D3FE0-2A16-11D7-9B58-000629DC0A53">
           <name>Monson-Haefel:BookQuote</name>
        </tModelInfo>
        ...
      </tModelInfos>
    </registeredInfo>
  </Body>
</Envelope>
```

Fault Messages

When the SOAP message sent by the client is malformed, contains unknown elements, or requires understanding of a header block, or when the UDDI registry is having trouble processing the message because of some server-side malfunction. UDDI also specifies how UDDI-specific errors are reported using SOAP fault messages. If there is some inconsistency in the data (an unknown key value or an oversized message, for example), then the error will be reported using a SOAP fault that carries a `dispositionReport` element. The `dispositionReport` element is carried in the `detail` element of the fault message and contains an error number, an error code, and a description.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoaporg.org/soap/envelope/">
  <Body>
    <Fault>
      <faultcode>Client</faultcode>
      <faultstring>Client Error</faultstring>
      <detail>
        <dispositionReport generic="2.0" operator="OperatorURI"
                           xmlns="urn:uddi-org:api_v2" >
          <result errno="10210" >
            <errInfo errCode="E_invalidKeyPassed">
               The businesskey passed did not match with any known key values:
               InvalidKey=01B1FA80-2A15-11D7-9B58-000629DC0A43
            </errInfo>
          </result>
        </dispositionReport>
      </detail>
    </Fault>
  </Body>
</Envelope>
```

UDDI 2.0 uses 24 standard error codes [for details refer to the book J2EE Webservices pg 306]. A UDDI registry is required to report only the first error it encounters, so if a message has multiple errors it might take several attempts before you know about all of them. In addition to being used for fault messages the `dispositionReport` element is also used as the return data structure for a successful delete operation, in which case the error code is `"E_success"`, as covered earlier.

Summary:
When you create a new Web service, you may choose to list it in a public or private UDDI registry, at which time you'll use the Publishing API. You may need to update entries from time to time, but for the most part they'll be static. When you need to peruse a public or private UDDI registry to find out more

about a business partner or some freely available Web service, you'll use the Inquiry API—but those occasions will come along pretty seldom as well. Finding out the details about another Web service is pretty much a one-time deal (unless the interface or endpoint changes). **You look it up once, save the `tModel`, and build or generate your interface to the Web service**. Many of the UDDI registry products offer graphical user interfaces or web interfaces that allow you to query and publish information to a UDDI registry. So for the most part, it's the tools which are more likely to use the UDDI APIs than the Web services developers.

JAX-RPC 1.1

[RMH – JAX-RPC is core of Webservices in Java. Chapters 9-15 and Appendix G. RMH does not cover extensible type-mapping viz mechanism for defining custom translations between XML and Java so refer to J2EE server vendor's manual for that].
4.1 Explain the service description model, client connection types, interaction modes, transport mechanisms/protocols, and endpoint types as they relate to JAX-RPC.
4.2 Given a set of requirements for a Web service, such as transactional needs, and security requirements, design and develop Web service applications that use servlet-based endpoints and EJB based endpoints.
4.3 Given an set of requirements, design and develop a Web sevice client, such as a J2EE client and a stand-alone Java client, using the appropriate JAX-RPC client connection style.
4.4 Given a set of requirements, develop and configure a Web service client that accesses a stateful Web service.
4.5 Explain the advantages and disadvantages of a WSDL to Java vs. a Java to WSDL development approach.
4.6 Describe the advantages and disadvantages of web service applications that use either synchronous/request response, one-way RPC, or non-blocking RPC invocation modes.
4.7 Use the JAX-RPC Handler API to create a SOAP message handler, describe the function of a handler chain, and describe the role of SAAJ when creating a message handler.

JAX-RPC Overview (4.1)

There are essentially two sides to the JAX-RPC model: **client-side** and **server-side**. The client-side programming model allows you to access a remote Web service as if it were a local object, using methods that represent SOAP operations. The server-side programming model allows you to develop Web service endpoints as Java objects or Enterprise JavaBeans, which run on the J2EE platform. JAX-RPC 1.1 vendors must support SOAP 1.1 clients and endpoints and the WS-I Basic Profile 1.0. They are required to support RPC/Literal and Document/Literal messaging modes, and both One-Way and Request/Response operation styles, using HTTP. In addition to the BP-conformant Web services technologies, JAX-RPC vendors are required to support RPC/Encoded messaging and SOAP with Attachments.

Server-Side Programming Models

JAX-RPC defines two server-side programming models for creating J2EE Web service endpoints: JAX-RPC service endpoints and EJB endpoints. EJB endpoints allow stateless EJBs to act as endpoints. EJB endpoints are valuable if you want to expose existing EJBs to Web service clients, or need to exploit the transactional qualities of Enterprise JavaBeans.

Service Endpoints
A JAX-RPC service endpoint (JSE) is easy to develop because it's just a plain old Java object—sort of. JSE is a full-fledged J2EE component, which runs inside the J2EE Servlets container system. It has access to the same resources and contexts as a standard servlet, but instead of exposing HTTP streams, it marshals SOAP messages into method invocations defined by a remote interface. A JSE is composed of two parts:

1. **an implementation class** that does all the work,
2. and a `java.rmi.Remote` interface, called the **endpoint interface**, that declares the Web service's publicly accessible methods.

```
public interface BookQuote extends java.rmi.Remote {
    public float getBookPrice(String isbn)
     throws java.rmi.RemoteException;
```



```
}
public class BookQuote_Impl_1 implements BookQuote {

    // Given the ISBN of a book, get its wholesale price.
    public float getBookPrice(String isbn){
        return 24.99f;
    }
}
```

Once you have defined the endpoint interface and implementation objects, you can use them, along with other deployment files, to generate a WSDL document that provides a platform-independent description of the JSE.

EJB Endpoints

Because SOAP is a stateless messaging protocol, only stateless session beans can act as EJB endpoints. The J2EE Web service specification defines a new Web service endpoint interface for stateless session beans that is on a par with the remote and local interfaces already used by enterprise beans. The endpoint interface extends the `javax.ejb.Remote` interface directly. In addition to the endpoint interface, you'll need to define the stateless session bean that implements the interface.

```
public interface BookQuote extends java.rmi.Remote {
    public float getBookPrice(String isbn)
     throws java.rmi.RemoteException;
}
public class BookPriceWS implements javax.ejb.SessionBean, BookQuote  {
     public void setSessionContext(javax.ejb.SessionContext ctx){}
     public void ejbCreate(){}

     public float getBookPrice(String isbn){
        return 24.99f;
     }

     public void ejbRemove(){}
     public void ejbActivate(){}
     public void ejbPassivate(){}
}
```

EJB endpoints do not require a corresponding home interface. SOAP doesn't specify support for pass-by-reference, so you can't ask one Web service interface (a home interface) to pass you a reference to another (a remote interface). Furthermore**, you can't create or remove a Web service. For these reasons a home interface is unnecessary**. When developing a Web service as a stateless session bean, you'll define the endpoint interface first, then use it along with other deployment descriptors to generate the WSDL document. The WSDL document can be read and used by other Web service platforms to access the EJB endpoint.

Client-Side Programming Models

The JAX-RPC client-side programming models can be used by Java applications, servlets, JSPs, JAX-RPC service endpoints (JSEs), and EJBs to exchange SOAP messages with remote Web service endpoints on any platform. JAX-RPC defines **three** client-programming models:

1. generated stub,
2. dynamic proxy, and
3. DII (Dynamic Invocation Interface).

Generated Stubs – Stub is generated at compile time from the WSDL using JAX-RPC compiler. The stub is generated for all port types in WSDL at compile time.

For a client to use JAX-RPC to access a Web service endpoint, the endpoint must publish a WSDL document. The JAX-RPC compiler, which is provided by your J2EE vendor, generates Java remote interfaces and stubs that implement the endpoint operations described by the WSDL document. Once the stubs and interfaces are generated, you can bind them to the JNDI ENC (Environment Naming Context) of any J2EE component and use them to communicate with the endpoint. In this model, the stubs are generated at deployment time.

WSDL describes the interfaces to Web service endpoints using `portType` definitions; each `portType` may have one or more `operation` elements. WSDL `portType` and `operation` elements are

analogous to Java interfaces and methods, respectively. In fact, JAX-RPC defines a mapping between WSDL and Java that generates remote interfaces from ports, with methods that correspond to port operations.

```
<message name="getBookPrice">
  <part name="string" type="xsd:string"/>
</message>
<message name="getBookPriceResponse">
  <part name="result" type="xsd:float"/>
</message>
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input name="isbn" message="mh:getBookPrice"/>
    <output message="mh:getBookPriceResponse"/>
  </operation>
</portType>
```

At deployment time a JAX-RPC compiler converts the WSDL `portType` into a corresponding remote interface—the endpoint interface. The JAX-RPC compiler also generates a stub that implements the endpoint interface according to the `port` and `binding` definitions. It can also create a factory for accessing the stub that conforms to the WSDL `service` definition. The service interface generated by the JAX-RPC compiler is shown below:

```
public interface BookQuoteService extends javax.xml.rpc.Service{
    public BookQuote getBookQuotePort( ) throws java.rmi.RemoteException;
}
```

Once the endpoint interface and stub have been generated, they can be used at runtime to invoke operations on the Web service endpoint.

```
InitialContext jndiContext = new InitialContext ( );

BookQuoteService service = (BookQuoteService)
             jndiContext.lookup("java:comp/env/service/BookQuoteService");
BookQuote bookQuote = service.getBookQuotePort();

float price = bookQuote.getBookPrice( isbn );
```

The generated stubs can include methods with primitive argument types like `int` and `long`, primitive wrappers like `java.lang.Integer` and `java.lang.Long`, arrays, a few standard Java types such as `String` and `Date`, custom object types, and special **holder types**. Holder types allow the JAX-RPC stubs to model INOUT and OUT parameter modes.

Dynamic Proxies – Generation of stub happens at run-time based on the port and binding definitions in WSDL. Which port to use of the many ports in WSDL can be configured at runtime.

A dynamic proxy is used in the same way as a generated stub, except the remote interface's stub implementation is generated dynamically, at runtime.

```
InitialContext jndiContext = new InitialContext ( );

BookQuoteService service = (BookQuoteService)
                jndiContext.lookup("java:comp/env/service/DynamicService");
BookQuote BookQuote_port = (BookQuote)service.getPort(BookQuote.class);

float price = BookQuote_port.getBookPrice( isbn );
```

At runtime the **`getPort()`** method automatically maps the `BookQuote` interface to a corresponding `port` definition in the WSDL document, then generates a stub for the interface that implements the protocol defined by the associated `binding`.

DII – Dynamically configure the client to use certain port type's operation at runtime and use DII to generate the stub for that port type's operation at runtime.

DII allows the developer to assemble SOAP method calls dynamically at runtime. JAX-RPC DII is kind of like Java Reflection. It enables you to get a reference to an object that represents a Web service operation, in the form of a method, and to invoke that method without having to use a stub or a remote interface. For example, the following fragment of code demonstrates how a J2EE component uses JAX-RPC DII to get the wholesale price of a book from some Web service.

```
InitialContext jndiContext = new InitialContext ( );
javax.xml.rpc.Service service =  (javax.xml.rpc.Service)
             jndiContext.lookup("java:comp/env/service/DynamicService");

QName port = new QName("http://www.xyz.com/BookQuote ","BookQuote");
QName operation = new QName("http://www.xyz.com/BookQuote","getBookPrice");

Call callObject = service.createCall(port, operation);

Object []  parameters = new Object[1];
parameters[0] = isbn;

Float price = (Float) callObject.invoke( parameters );
```

You can actually configure—at runtime—the parameters, invocation style, encoding, and so on. Everything you can configure statically with WSDL you can configure dynamically with DII. In most cases, you will know in advance the Web service endpoints and specific operations you will be accessing, so the DII will not be necessary.

Other JAX-RPC Topics

Message handlers are an important extension of JAX-RPC. They allow you to manipulate SOAP header blocks as they flow in and out of JAX-RPC endpoint and client applications. One of the primary goals of the JAX-RPC specification is to define **mappings** from WSDL and XML to Java. Specifically, it details how Java endpoint interfaces used by JAX-RPC Web services are converted into WSDL and how WSDL documents are converted into JAX-RPC generated stubs and dynamic proxies. It also details how method calls to JAX-RPC client APIs—generated stubs, dynamic proxies, and DII—are converted into SOAP messages, and how SOAP messages are mapped to JAX-RPC service endpoint and EJB endpoint methods.

SAAJ (SOAP with Attachments API for Java) is used for constructing and manipulating SOAP messages.

JAX-RPC Service Endpoints (4.2)

The endpoint interface defines the Web service operations that will be publicly accessible—in other words, the operations that Web service clients can access using SOAP. The endpoint interface is required to extend (directly or indirectly) the `java.rmi.Remote` interface and to throw the `java.rmi.RemoteException` type from all of its methods.

JSEs are deployed into a J2EE servlet container, and have access to the same resources and context information a servlet has. When a JSE is deployed, it is embedded in a special JAX-RPC servlet provided by the vendor, which is responsible for responding to HTTP-based SOAP requests, parsing the SOAP messages, and invoking the corresponding methods of the JSE implementation object. When the JSE returns a value from the method invocation, the JAX-RPC servlet creates a SOAP message to hold the return value (or a SOAP fault if an exception is thrown) and sends that SOAP message back to the requesting client via an HTTP reply message.

Because the JSE is embedded in a servlet, it can access the same resources the servlet can, via the JNDI Environment Naming Context (ENC). JSEs can access JDBC drivers, JMS providers, EJBs, J2EE connectors, other Web services, environment variables, the `ServletContext`, the `HttpSession`, and anything else a servlet can access. In addition, the JSE is allowed access to the actual SOAP message to which it's responding—which can be useful for examining header blocks and other information that is not explicitly passed as method parameters.

J2EE defines a very simple directory service, called the JNDI Environment Naming Context (JNDI ENC), which allows developers to bind resources (JDBC and JMS connection factories, for example), enterprise beans, Web service endpoints, and environment variables into a directory structure. The JNDI ENC is a standard service available to all J2EE components, but each component has a unique view of the service. A component may access its own view of the JNDI ENC, but not the view of any other component. Every component's view of the JNDI ENC is fixed at deployment time and is immutable, which means that it's read-only. You cannot insert, remove, or modifiy values in the JNDI ENC at runtime. The `InitialContext` object represents the root of the JNDI ENC hierarchy. When a JSE creates a new `InitialContext`, the servlet container automatically intervenes—under the covers—to return the root of the JNDI ENC for that JSE component. Every JSE, just like every servlet, has its own view of the JNDI

ENC, which you configure before deploying the JSE. The root context of the JNDI ENC is always "java:comp/env", but you can define any path relative to the root context you want. There are, however, naming conventions that most J2EE projects follow: JDBC DataSource objects are placed under the "/jdbc" subcontext; references to EJB objects are placed under the "/ejb" subcontext; JavaMail uses the "/mail" subcontext; JMS uses the "/jms" subcontext; and **Web services use the "/services" subcontext**.

Looking up a resource factory and then using it to obtain resource connections is the normal pattern when using the JNDI ENC. For example, you can use the JNDI ENC to obtain a JMS ConnectionFactory, a JavaMail Session, or a J2EE ConnectionFactory.

A JSE can also access environment variables from the JNDI ENC. Environment variables are static; they cannot be changed. They can be of any Java primitive wrapper type (Byte, Character, Short, Integer, Long, Float, or Double) or a String value. Environment variables are often used for configuring variables used by a JSE.

```
javax.naming.InitialContext jndiEnc = new javax.naming.InitialContext();
Double maxValue = (Double)jndiEnc.lookup("java:comp/env/max_value");
Boolean flag = (Boolean)jndiEnc.lookup("java:comp/env/the_flag");
String name = (String)jndiEnc.lookup("java:comp/env/some_name");
```

You configure the JNDI ENC at deployment time, coding XML elements of the deployment descriptor to identify the resources, EJBs, Web services, and environment variables, and bind them to specific path names for a specific JSE.

```
<web-app>
  <display-name>BookQuoteJSE</display-name>
  ...
  <resource-ref>
    <res-ref-name>jdbc/BookDatabase</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <env-entry>
    <env-entry-name>max_value</env-entry-name>
    <env-entry-type>java.lang.Double</env-entry-type>
    <env-entry-value>3929.22</env-entry-value>
  </env-entry>
  <ejb-local-ref>
    <ejb-ref-name>/ejb/BookHomeLocal</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>jwsed1.support.BookHomeLocal</local-home>
    <local>jwsed1.support.BookLocal</local>
    <ejb-link>BookEJB</ejb-link>
  </ejb-local-ref>
  ...
</web-app>
```

When the JSE is deployed, the container will ensure that the resources, EJBs, and Web services declared in the deployment descriptor are live, working references. When you look up a JDBC DataSource, you will receive a live connection to a database. Similarly, when you look up an EJB or Web service, you will receive a working reference to an EJB or a Web service endpoint. The person who deploys the JSE into the J2EE application server is responsible for mapping the resource, EJB, and Web service references to actual resources, using tools provided by the J2EE vendor. In the case of a JDBC DataSource, for example, the deployer must map the DataSource object to an actual database using a specific JDBC driver.

ServletEndpointContext and ServiceLifeCycle

In addition to the JNDI ENC there is another API that a JSE can use to interact with its environment, the ServletEndpointContext. This is an interface to the servlet container system itself. It provides the JSE with access to the underlying ServletContext, SOAP messages, and other information.

To take advantage of the **ServletEndpointContext**, the JSE must implement the optional **javax.xml.rpc.server.ServiceLifecyle** interface, which defines two methods:

```
package javax.xml.rpc.server;
import javax.xml.rpc.ServiceException;
```

```
public interface ServiceLifecycle {
    public void init(Object context) throws ServiceException;
    public void destroy();
}
```

The `init()` method is called at the beginning of a JSE's life, just after it's instantiated by the container system, before it begins handling incoming SOAP requests. When the servlet container calls the `init()` method, the JSE is given a reference to its `ServletEndpointContext`. The servlet container calls the `destroy()` method at the end of the JSE's life, just before it is removed from service. The `init()` method is called only once in the life of a JSE instance. If the servlet container has more JSE instances than it needs for a particular Web service endpoint, it may choose to evict some of the instances from memory to conserve resources. Each JSE instance's `destroy()` method is called when the servlet container evicts it, when the JSE is removed from service, or when the J2EE server is shut down. Because `init()` is called at the beginning of the JSE instance's life, and `destroy()` is called at the end, it makes sense to use these methods to obtain and release resources that will be used for the entire life of the instance.

```
package com.jwsbook.jaxrpc;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.xml.rpc.server.ServletEndpointContext;
import javax.xml.rpc.server.ServiceLifecycle;
import javax.xml.rpc.ServiceException;

public class BookQuote_Impl_5 implements BookQuote, ServiceLifecycle {
  javax.sql.DataSource dataSource;
  ServletEndpointContext endPtCntxt;

  public void init(Object context) throws ServiceException{
    try{
      endPtCntxt = (ServletEndpointContext)context;
      InitialContext jndiEnc = new InitialContext();
      javax.sql.DataSource dataSource = (javax.sql.DataSource)
                        jndiEnc.lookup("java:comp/env/jdbc/BookDatabase");
    }catch(NamingException ne){
      throw new ServiceException("Cannot initialize JNDI ENC", ne);
    }
  }
  public float getBookPrice(String isbn){
    java.sql.Connection jdbcConnection = null;
    java.sql.Statement sqlStatement = null;
    java.sql.ResultSet resultSet;
    try {
      jdbcConnection = dataSource.getConnection();
      sqlStatement = jdbcConnection.createStatement();
      resultSet = sqlStatement.executeQuery(
        "SELECT wholesale FROM CATALOG WHERE isbn = \'"+isbn+"\'");

      if(resultSet.next()){
        float price = resultSet.getFloat("wholesale");
        return price;
      }
      return 0;// zero means it's not stocked.
    }catch (java.sql.SQLException se) {
      throw new RuntimeException("JDBC access failed");
    }
  }
  public void destroy( ){
    dataSource = null;
  }
}
```

The `ServiceLifecycle` interface is intended to be a general-purpose interface that can be used outside of J2EE and a servlet container system—it might be running in some other type of container. JAX-RPC defines the context parameter as an `Object` type, so the interface is compatible with other, unanticipated or proprietary, container systems. In J2EE Web services, however, the JAX-RPC service endpoint is the

only kind of component that implements the `ServiceLifecycle` interface, and because it runs in a servlet container, it naturally makes use of the `ServletEndpointContext`.

The `ServletEndpointContext` is the JSE's interface to its servlet container. It provides **access to the identity of the client making a request, session data, the servlet context, and the message context used by JAX-RPC handlers**.

```
package javax.xml.rpc.server;
public interface ServletEndpointContext {
    public java.security.Principal getUserPrincipal();
    public boolean isUserInRole(String role);
    public javax.xml.rpc.handler.MessageContext getMessageContext();
    public javax.Servlet.http.HttpSession getHttpSession()
      throws javax.xml.rpc.JAXRPCException;
    public javax.Servlet.ServletContext getServletContext();
}
```

Although a JSE instance maintains a reference to the same `ServletEndpointContext` object throughout its life, the values returned by its methods will change with every new SOAP request. The information obtained by the `ServletEndpointContext` is actually obtained via the JAX-RPC servlet that wraps the JSE instance. The `Principal` and `HttpSession` objects, returned by the `getUser Principal()` and `getHttpSession()` methods, are obtained from the servlet's `doPost()` method's `HttpServletRequest` parameter. Similarly, the `ServletContext`, returned by the `getServletContext()` method, is a reference to the JAX-RPC servlet's `ServletContext`. The JAX-RPC servlet also instantiates and manages the `MessageContext`, returned by `getMessageContext()`, which is used by the JAX-RPC handler chain.

The `ServiceEndpointContext.getUserPrincipal()` method returns the identity of the sender, the SOAP client that is making the request. The `java.security .Principal` object represents the identity of the application that is sending the SOAP message and is available only if the JSE was configured to use either HTTP Basic Authentication (BASIC-AUTH) or Symmetric HTTP. If no authentication is used, the `getUserPrincipal()` method returns `null`. Once a SOAP sender has been authenticated, using BASIC-AUTH or Symmetric HTTP, the same `Principal` object can be associated with the SOAP sender's subsequent messages by using session tracking, via SSL, cookies, or some other session-tracking method. The `Principal` object may be propagated (for authorization) to any resources or EJBs accessible by the JSE, depending on the `run-as` and `res-auth` deployment settings in the deployment descriptor. You can authenticate callers to see if they belong to a specific role—assuming they have been authenticated—by calling the `isUserInRole()` method.

An HTTP session is a continuing conversation between the SOAP sender and the servlet container. It's an identifier that allows the servlet container to associate multiple HTTP requests with a specific SOAP sender. Session tracking is not used with Web services very much today. **Cookies** are identifiers that are sent from the servlet container to the SOAP sender and are associated with a specific host, such as monson-haefel.com. Every time the SOAP sender makes an HTTP request, it includes the cookie, which allows the servlet container to match the request to a specific session. URL rewriting and session tracking based on hidden forms are not used in SOAP-based Web services because they require the exchange of HTML, which isn't used in SOAP communications.
A SOAP application can use the HTTPS for confidential communications with a JSE. SSL has session tracking built right into the protocol, so when SSL is used, you get session tracking as a bonus. Regardless of the mechanism used to track a session, the `HttpSession` object is used to represent the session itself. The `HttpSession` object allows the JSE to view and change information about the session, such as the session identifier, creation time, last access time, and when the session will time out. In addition, the JSE can associate custom attributes with a session, so that it can maintain state across invocations by the same client. An attribute is a name-value pair, where the value is any serializable object. If HTTP session tracking is not used, `ServletEndpoint Context.getHttpSession()` will return `null`.

In theory, Web services are supposed to be completely stateless, so the use of sessions and the storage of data associated with sessions are somewhat controversial. That said, there are circumstances in which

session tracking and session data are necessary to improve the usability or performance of a Web service. For example, a Web service might use session tracking to ensure that clients receive information in their native language or currency. Most clients in the United States use English and the dollar, while those in France use French and the euro. Initially, the Web service might fetch language and currency preferences from the database, but once these are obtained they can be cached in the `HttpSession` to avoid repeated access to the database for the same information.

Most JSEs will not need to use the `ServletContext` except for accessing initialization parameters and perhaps logging.

When a JSE is multi-threaded, all client requests access exactly the same JAX-RPC servlet and JSE instance. Each client request is associated with its own thread of execution, and many threads will access the same JSE instance simultaneously. This model allows a single instance to support hundreds of clients, but it also requires care when accessing instance or class-level variables. Because all threads see the same instance and class variables, access to these variables should be avoided or synchronized using synchronized methods or blocks.

When a JSE is single-threaded, the servlet container normally maintains a pool of JAX-RPC servlets, and therefore JSE instances, then plucks them from the pool to handle client requests. Each simultaneous client request accesses a different JSE instanceYou don't have to worry about instance variables; they can be accessed freely without using synchronization. You should avoid creating static variables, on the other hand, or synchronize access to them, just as in the multi-threaded model. The single-threaded programming model has turned out to work poorly in practice because it degrades performance and introduces resource-management problems the multi-threaded model doesn't suffer from. A JSE that implements the `SingleThreadModel` interface can be accessed by only one thread at a time—the assumption being that a pool of instances will be used to service requests.

Practicals:
Its easy to write a webservice using IDEs like Netbeans:

1. You create a web application first then
2. You create a webservice as New->Webservice (and give it some package name)
3. The IDE then takes care of generating the SEI (endpoint interface) and also provides you the wizard to add your business method declarations.
4. Then you can edit the XXXImpl.java file to add your business logic.
5. The IDE creates the webservices.xml and the Sun Java System App Server specific sun-web.xml where it adds the port component name and endpoint address URI.
6. Having writen your business logic just run the project and browse to the **http://localhost:8080/<Webapp Name>/<Webservice Name>?WSDL** to get the WSDL.

For the BookQuoteJSE described above the WSDL is:

```
<?xml version="1.0"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:BookQuoteJSE/wsdl"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns2="urn:BookQuoteJSE/types"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" name="BookQuoteJSE"
targetNamespace="urn:BookQuoteJSE/wsdl">
        <types>
          <schema xmlns="http://www.w3.org/2001/XMLSchema"
                  xmlns:tns="urn:BookQuoteJSE/types"
                  xmlns:soap11-enc=http://schemas.xmlsoap.org/soap/encoding/
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  targetNamespace="urn:BookQuoteJSE/types">
                  <complexType name="getBookPrice">
                    <sequence>
                      <element name="String_1" type="string" nillable="true"/>
                    </sequence>
                  </complexType>
                  <complexType name="getBookPriceResponse">
                    <sequence>
                      <element name="result" type="float"/>
                    </sequence>
```



```
            </complexType>
            <element name="getBookPrice" type="tns:getBookPrice"/>
            <element name="getBookPriceResponse" type="tns:getBookPriceResponse"/>
        </schema>
    </types>
    <message name="BookQuoteJSESEI_getBookPrice">
        <part name="parameters" element="ns2:getBookPrice"/>
    </message>
    <message name="BookQuoteJSESEI_getBookPriceResponse">
        <part name="result" element="ns2:getBookPriceResponse"/>
    </message>
    <portType name="BookQuoteJSESEI">
            <operation name="getBookPrice">
                    <input message="tns:BookQuoteJSESEI_getBookPrice"/>
                    <output message="tns:BookQuoteJSESEI_getBookPriceResponse"/>
            </operation>
    </portType>
    <binding name="BookQuoteJSESEIBinding" type="tns:BookQuoteJSESEI">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
            style="document"/>
            <operation name="getBookPrice">
                    <soap:operation soapAction=""/>
                    <input>
                            <soap:body use="literal"/>
                    </input>
                    <output>
                            <soap:body use="literal"/>
                    </output>
            </operation>
    </binding>
    <service name="BookQuoteJSE">
            <port name="BookQuoteJSESEIPort" binding="tns:BookQuoteJSESEIBinding">
                    <soap:address
                    location="http://aditidt164.aditi.tech:8080/staffing/BookQuoteJSE"
                    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"/>
            </port>
    </service>
</definitions>
```

To test the above Webservice with a client, use the IDEs built-in client generator:

1. Go to Runtime tab
2. Click on WebServices-> Add a Webservice
3. In the wizard, provide the above WSDL url for the BookQuoteJSE webservice.
4. It will list all operations supported by the BookQuoteJSE webservice. For us, there is only one business method there – getBookPrice.
5. Click on the adjacent Test Operation hyperlink near the method name.
6. It will open a window where you can pass some param to the business method.
7. Pass an ISBN number, it will return the value.

We can even modify the IDE generated BookQuoteJSEImpl class to implement ServiceLifeCycle interface and also try out the JNDI. Note: The target server cannot be Tomcat as Tomcat does not support JAX-RPC and so IDE will demand that you choose a Sun Java System App Server as target or some other JAX-RPC provider.

JAX-RPC EJB Endpoints (4.2)

Using an API's transaction methods works well if you are accessing only a single resource because the resource itself (usually a database) will take care of managing the transaction. If, however, you are accessing two different databases, or two different resources—JDBC and JMS, for example—and you want the operations performed by both resources to be part of the same transaction, then you have to use the Java Transaction API (JTA) to group all the operations into a single transaction.

```
javax.transaction.UserTransaction userTransaction =
             (javax.transaction.UserTransaction)
              jndiEnc.lookup("java:comp/UserTransaction");
userTransaction.begin();
try{

  Statement op_1 = jdbcConnection.createStatement( );
```



```
  op_1.executeUpdate("UPDATE account SET balance = balance-"
                          +transfer_amount+" WHERE   account_number = "
                          +accountA);
  Statement op_2 = jdbcConnection.createStatement( );
  op_2.executeUpdate("UPDATE account SET balance = balance+"
                          +transfer_amount+" WHERE   account_number = "
                          +accountB);
  Statement op_3 = jdbcConnection.createStatement( );
  op_3.executeUpdate("INSERT INTO transfer_audit ("
                          +accountA+","+accountB+","+transfer_amount+","
                          +new java.sql.Date(System.currentTimeMillis())+")");
```

```
  MessageProducer producer = jmsSession.createProducer(destinationA);
  TextMessage message =
  jmsSession.createTextMessage("$"+transfer_amount+" transferred from "
                          +accountA+" to "
                          +accountB);
  producer.send(message);
```

```
  userTransaction.commit();
}catch(SQLException sqle){
    userTransaction.rollback();
    // other error-handling code follows
}
```

The `UserTransaction.begin()` method marks the beginning of the transaction, and `UserTransaction.commit()` marks the end. Any JDBC and JMS operations performed between the `begin()` and `commit()` method calls are included in the same transaction. The `UserTransaction.commit()` method commits the changes made with the JDBC connection to the database and completes the delivery of the JMS message. If any of the operations throws a `SQLException`, the `User Transaction.rollback()` method rolls the transaction back—that is, abandons all the operations.

The use of JTA appears fairly simple, but it can be difficult to use correctly. It requires a good understanding of both transactions and the resources used by the application developer. To make life easier, Enterprise JavaBeans handles transaction demarcation implicitly. When using EJB, you don't need to use JTA or call the `begin()`, `commit()`, or `rollback()` methods—the EJB container system will take care of all these chores for you automatically. In addition, a transaction that is started by one EJB can be propagated to other EJBs automatically—a very powerful feature.

If your Web service needs to interact with CMP entity components, then using an EJB endpoint (a stateless session bean that acts as a Web service) can be beneficial.

It's considered a bad design practice, however, for any other type of bean (stateless, stateful, entity, or message-driven) to make calls on a stateful session bean, because it represents a client's session.

BMP is useful in situations where CMP won't work; for example, when an entity's state is derived from two or more resources. Entity beans will sometimes access stateless session beans, but most of the time they operate on their own state, or that of related entity beans.

```
javax.naming.InitialContext jndiEnc = new javax.naming.InitialContext();
Object remoteRef =  jndiEnc.lookup("java:comp/env/ejb/BookQuoteHome");
BookQuoteHomeRemote ejbHome = (BookQuoteHomeRemote)
 javax.rmi.PortableRemoteObject.narrow( remoteRef, BookQuoteHomeRemote.class);
```

```
BookQuoteRemote bookQuote = ejbHome.create( );
float price = bookQuote.getBookPrice( isbn );
```

Within the method itself the EJB can access JDBC drivers, JMS providers, or any other type of transactional resource, and be assured that all the operations performed will be parts of the same transaction, and will either succeed or fail together. Transaction propagation most often occurs when one EJB calls another to do some work. Propagating the transaction ensures that all the tasks performed by both EJBs are parts of the same transaction. Because transactions can propagate between EJBs automatically, you can have multiple EJBs collaborate to achieve a task, without having to worry about system failures creating inconsistent results.

```
package com.jwsbook.jaxrpc;
```

```
public interface BookQuote extends java.rmi.Remote {
  public float getBookPrice(String isbn) throws java.rmi.RemoteException;
}
```

Every business method defined in the endpoint interface must declare the `java.rmi.RemoteException` in its `throws` clause. The real purpose of the endpoint interface is to declare the Web service operations supported by the EJB endpoint, and any faults those operations may generate. At deployment time the J2EE application server's deployment tools can examine the endpoint interface, along with other information provided by deployment descriptors, and use it to generate the `portType` and `message` definitions of the WSDL document.
In any case, whether you choose to extend the endpoint interface explicitly or not, the bean class must implement methods that match the endpoint interface methods exactly. The only exception to this rule is the `Exception` types thrown by the bean class's methods. Each method in the implementation class must throw the same exceptions that the corresponding methods in the interface throw—except one: `java.rmi.RemoteException`. **The endpoint bean's methods must never declare `RemoteException`**.
Every stateless session bean, whether it's used as a Web service endpoint or not, must define a bean class that implements the `javax.ejb.SessionBean` interface. The `SessionBean` interface defines a number of callback methods the container uses to interact with the bean. In addition to the callback methods**, an EJB endpoint's bean class must declare an `ejbCreate()` method.** This may perform initialization work that cannot be done in the `setSessionContext()` method. Specifically, it can access the EJB object, EJB home, and `TimerService` objects that are not available in the `setSessionContext()` method. The **`ejbCreate()` method must not declare any parameters** because it's called by the container rather than a client, so there are no arguments to pass to it.
Fundamentally, there are at least **two basic deployment descriptors** you must create for an EJB endpoint:

1. The **`ejb-jar.xml`** deployment descriptor declares the bean class, the local and remote interfaces, and the endpoint interface. In addition, it declares the JNDI ENC references to other beans, resources, Web services, and environment variables. Finally, it declares the security and transaction attributes of the EJB.
2. The **`webservices.xml`** deployment descriptor describes each Web service endpoint and links these descriptions to an endpoint, either JSE or EJB. In either case `webservices.xml` declares the location of the WSDL document, the `port` definition associated with the endpoint, the JAX-RPC mapping file, the endpoint interface, and a link to the actual component.
3. In addition to `ejb-jar.xml` and `webservices.xml`, the deployment tool will generate a **JAX-RPC mapping file**, which defines more complex mappings, between Java objects used as parameters and return values on one hand, and the XML types used in SOAP and WSDL on the other.

```
package javax.ejb;
import java.security.Principal;
public interface SessionContext extends EJBContext {

    public EJBLocalObject getEJBLocalObject();
    public EJBObject getEJBObject();

    public EJBLocalHome getEJBLocalHome();
    public EJBHome getEJBHome();

    public Principal getCallerPrincipal();
    public boolean isCallerInRole(String roleName);

    public UserTransaction getUserTransaction();
    public boolean getRollbackOnly();
    public void setRollbackOnly();

    public MessageContext getMessageContext();
    public TimerService getTimerService();

}
```

In many cases, a Web service just performs simple reads or updates to a single database. In these cases, you want as light an implementation as possible, and the far simpler JSE may be preferable to an EJB endpoint—especially if the operations are read-only. If your Web service will be modifying data via resources like JDBC and JMS, or coordinating the activities of other EJBs that modify data, then it's a good idea to use an EJB endpoint.

JAX-RPC Client APIs (4.3)

Generated Stubs

JAX-RPC Compilers read a WSDL document and generate from it at least two Java class files, an endpoint interface and a generated stub class. The endpoint interface extends `java.rmi.Remote` and defines the methods that can be invoked on the Web service endpoint. The generated stub class implements the endpoint interface. At runtime an instance of the generated stub class converts each method invocation made on the remote interface into a SOAP 1.1 message, and sends it to the Web service endpoint. The generated stub also converts each SOAP reply message into a method return value, or throws an exception if the reply message is a SOAP fault.

A JAX-RPC compiler will read the WSDL document and generate an endpoint interface that represents the abstract `portType`, and a generated stub class that implements the network protocol and SOAP messaging style described by the `binding` and `port` definitions. The JAX-RPC specification defines a detailed mapping between WSDL and generated stubs. It includes rules for mapping the abstract `message`, `operation`, and `portType` definitions, as well as XML schema types (simple and complex), to JAX-RPC endpoint interfaces.

The endpoint interface is derived from the WSDL `portType` definition, and each of the interface's methods represents a WSDL `operation` definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- message elements describe the input and output parameters -->
  <message name="GetBookPriceRequest">
    <part name="isbn" type="xsd:string" />
  </message>
  <message name="GetBookPriceResponse">
    <part name="price" type="xsd:float" />
  </message>

  <!-- portType element describes the abstract interface of a Web service -->
  <portType name="BookQuote">
    <operation name="getBookPrice">
      <input name="isbn" message="mh:GetBookPriceRequest"/>
      <output name="price" message="mh:GetBookPriceResponse"/>
    </operation>
  </portType>

  <!-- binding tells us which protocols and encoding styles are used  -->
  <binding name="BookQuote_Binding" type="mh:BookQuote">
    <soapbind:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getBookPrice">
      <soapbind:operation style="rpc"
       soapAction="http://www.Monson-Haefel.com/jwsbook/BookQuote"/>
        <input>
          <soapbind:body use="literal"
           namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
        </input>
        <output>
          <soapbind:body use="literal"
           namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
        </output>
    </operation>
  </binding>

  <!-- service tells us the Internet address of a Web service -->
  <service name="BookQuoteService">
```



```
    <port name="BookQuotePort" binding="mh:BookPrice_Binding">
      <soapbind:address
       location="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
    </port>
  </service>

</definitions>
```

Here's what that gets generated from above WSDL as service endpoint interface:

```
package com.jwsbook.jaxrpc;

public interface BookQuote extends java.rmi.Remote {
  public float getBookPrice(String isbn) throws java.rmi.RemoteException;
}
```

The name of the endpoint interface comes from the `portType` name. Similarly, there is a one-to-one mapping between methods defined in the endpoint interface and `operation` definitions declared by the `portType`. The parameters and return value of each endpoint method are derived from `message` definitions that the `portType` refers to in the `input` and `output` elements of an `operation` element. Each parameter name and type derives from a `part` of the input `message`, and the return type derives from the `part` of the output `message`.

An instance of the generated stub class (routinely shortened to "the stub") is responsible for translating method calls into SOAP communications with the Web service endpoint. It will perform this translation according to the WSDL `binding` associated with the `portType`, which describes the style of messaging (`rpc` or `document`) and the type of encoding (always `Literal` in BP-conformant Web services). The stub will use the `soap:address` element specified by the WSDL `port` definition as the URL of the Web service.

In addition to implementing the endpoint interface, the stub class will also implement the **`javax.xml.rpc.Stub` interface, which defines methods for reading and setting properties relating to network communication and authentication. Using these, an application can set the endpoint address, user name, password, and so on at runtime.**

```
package javax.xml.rpc;
import java.util.Iterator;

public interface Stub {

    // Standard property: The Web service's Internet address.
    public static String ENDPOINT_ADDRESS_PROPERTY;
    // Standard property: Password for authentication.
    public static String PASSWORD_PROPERTY;
    // Standard property: User name for authentication.
    public static String USERNAME_PROPERTY;
    // Standard property: Boolean flag for maintaining an HTTP session.
    public static String SESSION_MAINTAIN_PROPERTY;

    // Given a property name, get its value.
    public Object _getProperty(java.lang.String name);
    // Get the names of all the properties the stub supports.
    public Iterator _getPropertyNames();
    // Configure a property on the stub.
    public void _setProperty(java.lang.String name, java.lang.Object value);

}
```

The JAX-RPC compiler can, as an option, generate a **service interface** representing the `service` definition from a WSDL document, and a class that implements this interface. A J2EE component can use an instance of the **service** interface, called a service, to **obtain a reference to a specific generated stub**. The service interface defines an accessor for every `port` declared by the `service` definition.

```
package com.jwsbook.jaxrpc;

public interface BookQuoteService extends javax.xml.rpc.Service{
    public BookQuote getBookQuotePort()
    throws javax.xml.rpc.ServiceException;
}
```

The name of the service interface, in this case `BookQuoteService`, comes from the `name` attribute of the WSDL `service` definition. The accessor method, `getBookQuotePort()`, gets its method name from the `name` attribute of the WSDL `port` element. The return type is the endpoint interface type that the JAX-RPC compiler generated for the corresponding `portType`, in this case the `BookQuote` endpoint interface. An instance of this class instantiates and preconfigures generated stubs requested by the client. In J2EE, the service is bound to the JNDI ENC (Environment Naming Context), and is used to obtain a reference to the generated stub. Think of the service as a factory for generated stubs. The service interface extends the `javax.xml.rpc.Service` interface, which defines methods normally used with the dynamic proxy and DII programming models.
The generated stub is "hard-wired" to use the protocol and network address described by the WSDL document, so all a client needs to do is obtain a reference to the stub from the service.

```
package com.jwsbook.jaxrpc;
import javax.servlet.http.*;
import javax.servlet.*;
import javax.naming.InitialContext;

public class BookQuoteServlet_1 extends javax.servlet.http.HttpServlet {
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException,java.io.IOException {
    try{
      String isbn = req.getParameter("isbn");

      InitialContext jndiContext = new InitialContext();

      BookQuoteService service = (BookQuoteService)
      jndiContext.lookup("java:comp/env/service/BookQuoteService");

      BookQuote bookQuote = service.getBookQuotePort();

      float price = bookQuote.getBookPrice( isbn );

      java.io.Writer outStream = resp.getWriter();
      outStream.write("<html><body>The wholesale price for ISBN:"+isbn+
                      " = "+price+"</body></html>");
    }catch(javax.naming.NamingException ne){
      throw new ServletException(ne);
    }catch(javax.xml.rpc.ServiceException se){
      throw new ServletException(se);
    }
  }
}
```

At deployment time the implementation class of the `BookQuoteService` interface is assigned to the JNDI namespace `"java:comp/env/service/BookQuoteService"` by a service reference element. A JAX-RPC Web service is declared in a `service-ref` element of the deployment descriptor of a J2EE component.

```
<service-ref xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
    <service-ref-name>service/BookQuoteService</service-ref-name>
    <service-interface>com.jwsbook.jaxrpc.BookQuoteService</service-interface>
    <wsdl-file>BookQuote.wsdl</wsdl-file>
    <service-qname>mh:BookQuoteService</service-qname>
</service-ref>
```

In fact, you should get better performance out of your J2EE component by maintaining a reference to a stub rather than fetching a new one with every request.
When any of the stub's methods is invoked it will convert the method call into a SOAP message and send that message to the Web service endpoint.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```


When the Web service replies, it sends a SOAP message back to the stub using an HTTP reply message.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPriceResponse>
          <price>29.99</price>
      </mh:getBookPriceResponse>
   </soap:Body>
</soap:Envelope>
```

When the stub receives the SOAP reply message, it will extract its return value from the message or, if the message contains a fault, it will throw the appropriate Java exception.

You can also use JAX-RPC from non-J2EE clients.

```
package com.jwsbook.jaxrpc;
import javax.naming.InitialContext;

public class JaxRpcExample_1 {
  public static void main(String [] args) throws Exception{
    String isbn = args[0];

    BookQuoteService service = (BookQuoteService)
    ServiceFactory.loadService(com.jwsbook.jaxrpc.BookQuoteService.class);

    BookQuote bookQuote = service.getBookQuotePort();

    float price = bookQuote.getBookPrice( isbn );

    System.out.println("The price is = "+price);
  }
}
```

`JaxRpcExample_1` uses the static method **`ServiceFactory.loadService()`** to obtain a reference to the service object. **This method can be used in non-J2EE clients only**; J2EE components must use the JNDI ENC instead.

BookQuote is a Request/Response Web service, but **generated stubs can also use One-Way messaging. If they do, their methods all return `void`.**

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://www.Monson-Haefel.com/jwsbook/Foo"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo">
  <message name="bar">
    <part name="value" type="xsd:string"/>
  </message>
  <portType name="Foo">
    <operation name="setBar">
      <input message="tns:bar"/>
    </operation>
  </portType>
  <binding name="FooBinding" type="tns:Foo">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="setBar">
      <soap:operation soapAction="" style="rpc"/>
      <input>
        <soap:body use="literal"
            namespace="http://www.Monson-Haefel.com/jwsbook/Foo"/>
      </input>
    </operation>
  </binding>
  ...
</definitions>

package com.jwsbook.jaxrpc;

public interface Foo extends java.rmi.Remote {
```

```
    public void setBar(String value) throws java.rmi.RemoteException;
}
```

Although the endpoint interface is semantically a one-way operation style, the underlying stub may in fact be using HTTP 1.1, which is a Request/Response protocol. In this case, the method returns after the stub receives the HTTP reply code (for example, `200` for OK) but it doesn't return any application data to the client.

Dynamic Proxies

Dynamic Proxy is created dynamically at runtime rather than generated statically at deployment time. Other than the method of accessing them, you use dynamic proxies to invoke Web service operations the same way you use generated stubs. The **benefits of using dynamic proxies instead of generated stubs are not clear**—it's probably best to stick with generated stubs.

```
package com.jwsbook.jaxrpc;
import javax.naming.InitialContext;

public class JaxRpcExample_2 {
  public static void main(String [] args) throws Exception{
    String isbn = args[0];

    InitialContext jndiContext = new InitialContext();

    javax.xml.rpc.Service service = (javax.xml.rpc.Service)
    jndiContext.lookup("java:comp/env/service/Service");

    BookQuote BookQuote_proxy = (BookQuote)
                                service.getPort(BookQuote.class);

    float price = BookQuote_proxy.getBookPrice( isbn );

    System.out.println("The price is = "+price);
  }
}

package javax.xml.rpc;

public interface Service {
    public java.rmi.Remote getPort(java.lang.Class endpointInterface)
    throws javax.xml.rpc.ServiceException;

    public java.rmi.Remote getPort(javax.xml.namespace.QName portName,
                                   java.lang.Class endpointInterface)
    throws javax.xml.rpc.ServiceException;

    ...
}
```

The whole point of dynamic proxies is that they are generated dynamically the first time you use them. At runtime the JAX-RPC provider (the vendor implementation) will read the WSDL document and generate a proxy to implement the endpoint interface at the moment the `getPort()` method is invoked. It may do so every time `getPort()` is invoked, but it will probably cache proxies for subsequent requests. The J2EE container generates the proxy by matching the endpoint interface passed into the `getPort()` method with a matching `portType` definition in the WSDL document, or by examining the JAX-RPC mapping file. If it finds a matching `portType`, it uses the `binding` and `port` definitions associated with that `portType` to create a class that implements the endpoint interface. When you deploy a J2EE component that uses a JAX-RPC dynamic proxy, you include a `service-ref` element in the deployment descriptor that tells the J2EE container which WSDL document you're using.

```
<service-ref xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
    <service-ref-name>service/Service</service-ref-name>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-file>BookQuote.wsdl</wsdl-file>
    <service-qname>mh:BookQuoteService</service-qname>
</service-ref>
```

You probably noticed that the `service-interface` is the generic `javax.xml.rpc.Service` interface customarily used with dynamic proxies. The `Service` interface is associated with a specific

service definition in the WSDL document via the service-qname element in the service-ref. That way if the WSDL document defines more than one service, the J2EE container system can tell which service definition this service-ref is referring to.
If there is only one port defined for the matching portType, the getPort(Class) method will create a dynamic proxy for that port definition.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote">
 ...
 <service name="BookQuote">
   <port name="BookQuoteEncodedPort" binding="mh:BookQuote_EncodedBinding">
     <soap:address location="http://www.Monson-Haefel/jwsbook/BookQuote/Encoded"/>
   </port>
   <port name="BookQuoteLiteralPort" binding="mh:BookQuote_LiteralBinding">
     <soap:address location="http://www.Monson-Haefel/jwsbook/BookQuote/Literal"/>
   </port>
 </service>
</definitions>
```

In this case the binding definition used to create the proxy depends on the **JAX-RPC mapping file**, which pinpoints the exact WSDL binding definition to use in cases where a WSDL service defines multiple ports. The portion of JAX-RPC mapping file which defines the exact WSDL binding to use for a certain port type:

```
<?xml version='1.0' encoding='UTF-8' ?>
<java-wsdl-mapping
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"...>
  ...
  <service-endpoint-interface-mapping>
    <service-endpoint-interface>com.jwsbook.jaxrpc.BookQuote
    </service-endpoint-interface>
    <wsdl-port-type>mh:BookQuote</wsdl-port-type>
    <wsdl-binding>mh:BookQuote_LiteralBinding</wsdl-binding>
    ...
  </service-endpoint-interface-mapping>
</java-wsdl-mapping>
```

As an alternative, you can use the Service.getPort(QName, Class) method to create a dynamic proxy for the correct WSDL port and binding.

```
package com.jwsbook.jaxrpc;
import javax.naming.InitialContext;
import javax.xml.namespace.QName;

public class JaxRpcExample_3 {
  public static void main(String [] args) throws Exception{
    String isbn = args[0];

    InitialContext jndiContext = new InitialContext();

    javax.xml.rpc.Service service = (javax.xml.rpc.Service)
    jndiContext.lookup("java:comp/env/service/Service");

    QName portName =
          new QName("http://www.Monson-Haefel/jwsbook/BookQuote",
          "BookQuoteLiteralPort");

    BookQuote BookQuote_proxy = (BookQuote)
                               service.getPort(portName, BookQuote.class);

    float price = BookQuote_proxy.getBookPrice( isbn );

    System.out.println("The price is = "+price);
  }
}
```

The QName class is defined in the javax.xml.namespace package. An instance of this class represents an XML name.


```
package javax.xml.namespace;

public class QName implements java.io.Serializable {
    public QName(String localPart) {...}
    public QName(String namespaceURI, String localPart){...}
    public String getNamespaceURI(){...}
    public String getLocalPart(){...}
    public static QName valueOf(String qualfiedName){...}
}
```

The `localPart` is equal to the name assigned to the port, and `namespaceURI` is the XML namespace of the port. The static **`QName.valueOf()`** method allows you to create an instance of the port's `QName` from a single `String` value formatted as **`"{namespaceURI}local Part".`** The XML namespace of the port is nested in braces to separate it from the local name. The following code snippet creates two identical `QNames` in different ways, using a constructor, and using the static `valueOf()` method.

```
// Use constructor
QName qname1 = new QName("http://www.Monson-Haefel/jwsbook/BookQuote",
                         "BookQuoteLiteralPort");

// Use static valueOf() method
String s = "{http://www.Monson-Haefel/jwsbook/BookQuote}BookQuoteLiteralPort";
QName qname2 = QName.valueOf(s);
```

Dynamic proxies are usually used with the generic `Service` interface. You can also obtain them from generated service implementations, because generated service interfaces, like `BookQuoteService.` The `Service.getPort()` methods may return a generated stub instead of a dynamic proxy. If you deploy a J2EE component that uses generated stubs and then make a call to the `Service.getPort()` method, you'll probably get back a stub rather than a dynamic proxy—it's difficult to tell the difference, because both dynamic proxies and generated stubs implement the `javax.xml.rpc.Stub` interface. The endpoint interface used to create a dynamic proxy can be generated (using a JAX-RPC compiler), or it can be hand-coded by the developer.

Beneath the surface, the J2EE container system uses the J2SE Reflection API defined by the `java.lang.reflect` package to generate dynamic proxies. Specifically, it uses the **`java.lang.reflect.Proxy`** class, and implementations of the `java.lang.reflect.InvocationHandler` interface to create JAX-RPC dynamic proxies.

DII

The Dynamic Invocation Interface (DII) defines an API for invoking operations on Web service endpoints. Unlike generated stubs and dynamic proxies, DII doesn't use an endpoint interface that is different for every `portType`. Instead, DII uses a fixed API that can be used to invoke operations on just about any endpoint. This makes the DII toolable, meaning **it can be used by applications that automatically discover and invoke Web service operations.**

DII can be used with or without a WSDL document. If a WSDL document is available, it's much simpler to use because the details about the Web service operation being invoked can be derived from the WSDL document. To use DII to call the getBookPrice operation of the BookQuote Web service is shown below:

```
package com.jwsbook.jaxrpc;
import javax.naming.InitialContext;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.namespace.QName;

public class JaxRpcExample_4 {
  public static void main(String [] args) throws Exception{
    String isbn = args[0];

    InitialContext jndiContext = new InitialContext();

    javax.xml.rpc.Service service = (javax.xml.rpc.Service)
    jndiContext.lookup("java:comp/env/service/Service");

    QName portName =
     new QName("http://www.Monson-Haefel.com/jwsbook/BookQuote",
```



```
                "BookQuotePort");
    QName operationName =
     new QName("http://www.Monson-Haefel.com/jwsbook/BookQuote",
                "getBookPrice");

    Call call = service.createCall(portName,operationName);

    Object [] inputParams = new Object[]{isbn};

    Float price = (Float)call.invoke(inputParams);

    System.out.println("The price is = "+price.floatValue());
  }
}
```

The `javax.xml.rpc.Call` interface represents a Web service operation that can be invoked or called using the `Call.invoke()` method. As with generated stubs and dynamic proxies, the `Service` type is bound to the JNDI ENC using a `service-ref` element in a deployment descriptor. The `service-ref` element associates the `Service` type with a specific WSDL `service` element for a specific WSDL document, in this case `BookQuote.wsdl`.

```
<service-ref xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
    <service-ref-name>service/Service</service-ref-name>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-file>META-INF/BookQuote.wsdl</wsdl-file>
    <service-qname>mh:BookQuoteService</service-qname>
</service-ref>
```

Using the `Service` object, you can create a `Call` object for a specific WSDL `port` and `operation`. The `Call` object is associated with a specific WSDL `operation` of a specific `port`, from which it derives the proper operation style (RPC or Document) and encoding (for example, Literal). The `Call` object uses this information to construct the SOAP message it sends to the Web service. Once the `Call` object is created, its `invoke()` method can be executed using the proper arguments.

The operation return value is always returned by the `invoke()` method. To retrieve the values of OUT and INOUT parameters, after the `invoke()` method is called you have to invoke the `Call.getOutputValues()` method.

```
java.util.List outputParams = call.getOutputValues();
```

The Basic Profile requires that Web service endpoints provide a WSDL document that describes the Web service. You may need to interact with a non-conformant Web service, however, in which case you can use **the DII's facilities for messaging without a WSDL document**.[This topic is outside the scope of exam.]

DII can be used to send One-Way messages as well as Request/Response messages. If you are sending an RPC/Literal message that does not require a reply message, you can simply use the `invokeOneWay()` method instead of `invoke()`. The only difference is that **`invokeOneWay`**`()` has no return value (or output values) and does not block, waiting for a reply (other than the HTTP reply with code `200`).

```
Object [] inputParams = new Object[]{value1, value2,...};
call.invokeOneWay(inputParams);
```

JAX-RPC defines the `javax.xml.rpc.NamespaceConstants` type, which contains a bunch of other SOAP 1.1 XML namespace URIs and prefix constants that are commonly used.

Constructing `QName` objects is kind of a pain, so JAX-RPC provides the `javax.xml.rpc.encoding.XMLType` class that defines a couple of dozen final static `QName` constant values for the most common XML schema and SOAP 1.1 types.

Message Handlers (4.7)

The primary **purpose of message handlers is to provide a mechanism for adding, reading, and manipulating header blocks in SOAP messages** that JAX-RPC clients and Web service endpoints send and receive. Message handlers operate behind the scenes, and are managed by the J2EE container system. When you send a SOAP message using a JAX-RPC client API, the JAX-RPC runtime will filter the outgoing SOAP message through a chain of message handlers before sending it across the network to the

Web services. Similarly, any SOAP reply messages received by a JAX-RPC client will be filtered through the same chain of message handlers before the result is returned to your application code.

Message Handler Example

To illustrate how message handlers work, we'll develop a very simple message handler that adds a `message-id` header block to a SOAP message that is produced by a generated stub. All message handlers must implement the `javax.xml.rpc.handler.Handler` interface, which provides methods for processing SOAP requests, replies, and fault messages.

```
package javax.xml.rpc.handler;

public interface Handler {
    public boolean handleRequest(MessageContext context);
    public boolean handleResponse(MessageContext context);
    public boolean handleFault(MessageContext context);
    public void init(HandlerInfo config) ;
    public void destroy();
    public QName[] getHeaders();
}
```

In many cases, you won't need to implement all the methods defined by the `Handler` interface, so the JAX-RPC API provides an abstract class, **`javax.xml.rpc.handler.GenericHandler`**, that "stubs out" all the methods of the `Handler` interface. Extending the `GenericHandler` class allows you to implement the methods you're interested in and ignore the rest.

```
package com.jwsbook.jaxrpc;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPHeaderElement;
import javax.xml.soap.Name;

public class MessageIDHandler extends javax.xml.rpc.handler.GenericHandler{

  QName [] headerNames = null;

  public QName [] getHeaders(){
    if(headerNames == null){
      QName myHeader =
        new QName("http://www.Monson-Haefel.com/jwsbook/message-id",
                  "message-id");
      headerNames = new QName[]{ myHeader };
    }
    return headerNames;
  }
  public boolean handleRequest(MessageContext context){
    String messageID = new java.rmi.dgc.VMID().toString();

    try{

      SOAPMessageContext soapCntxt = (SOAPMessageContext)context;
      SOAPMessage message = soapCntxt.getMessage();
      SOAPHeader header = message.getSOAPPart().getEnvelope().getHeader();

      Name blockName = SOAPFactory.newInstance().createName(
                        "message-id","mi",
```



```
                        "http://www.Monson-Haefel.com/jwsbook/message-id");
      SOAPHeaderElement headerBlock = header.addHeaderElement(blockName);
      headerBlock.setActor("http://www.Monson-Haefel.com/logger");
      headerBlock.addTextNode( messageID );
      return true;

    }catch(javax.xml.soap.SOAPException se){
          throw new javax.xml.rpc.JAXRPCException(se);
    }
  }
}
```

The first thing we need to do is to identify the header blocks that the `MessageID Handler` will process. Every `Handler` class must implement the **`getHeaders`**`()` method, which returns an array of header-block element names that the `Handler` should process. The JAX-RPC runtime will use this list of names to determine whether it has a message handler that matches a header block of an incoming message.

The real meat of `MessageIDHeader` is in its **`handleRequest`**`()` method, which adds a `message-id` header block to an outgoing SOAP message. When `handleRequest()` is done, it returns `true`, to indicate that it processed the message successfully. If the SAAJ API throws a `SOAPException`, the method catches it and throws a `javax.xml.rpc.JAXRPCException`, a kind of `java.lang.RuntimeException`, which causes the message delivery to fail.

When a JAX-RPC client invokes a method (for example, an endpoint method on a generated stub) to send a SOAP message, the JAX-RPC runtime will first marshal the arguments of the call into an XML SOAP message, then place that SOAP message in a SAAJ `SOAPMessage` object. Once the `SOAPMessage` object is constructed, it's passed into the chain of message handlers.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote">
  <types>
    <schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/message-id"
     xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="message-id" type="string"/>
    </schema>
  </types>

  <message name="Headers">
    <part name="message-id" element="mi:message-id"/>
  </message>
  <message name="getBookPrice">
    <part name="isbn" type="xsd:string"/>
  </message>
  <message name="getBookPriceResponse">
    <part name="result" type="xsd:float"/>
  </message>
  <portType name="BookQuote">
    <operation name="getBookPrice">
      <input message="mh:getBookPrice"/>
      <output message="mh:getBookPriceResponse"/>
    </operation>
  </portType>
  <binding name="BookQuoteSoapBinding" type="mh:BookQuote">
    <soapbind:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getBookPrice">
      <soapbind:operation soapAction="" style="rpc"/>
      <input>
        <soapbind:header message="mh:Headers" part="message-id"
                         use="literal"/>
        <soapbind:body use="literal"
          namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
      </input>
```

```
      <output>
        <soapbind:body use="literal"
          namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
      </output>
    </operation>
  </binding>

  <!-- service tells us the Internet address of a Web service -->
  <service name="BookPriceService">
    <port name="BookPrice_Port" binding="mh:BookQuoteSoapBinding">
      <soapbind:address
       location="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
    </port>
  </service>
</definitions>
```

We define the WSDL first which has the message-id header block definition. We use the WSDL to generate the endpoint ...

```
package com.jwsbook.jaxrpc;

public interface BookQuote extends java.rmi.Remote {
  public float getBookPrice(String isbn) throws java.rmi.RemoteException;
}
```

and service interfaces and their implementations…

```
package com.jwsbook.jaxrpc;

public interface BookQuoteService extends javax.xml.rpc.Service{
    public BookQuote getBookQuotePort()
    throws javax.xml.rpc.ServiceException;
}
```

There are no methods for accessing message handlers, nor is there a `message-id` parameter in the `getBookPrice()` method. The application of the `MessageIDHandler` by the JAX-RPC runtime is automatic and hidden. The `MessageIDHandler`'s `handleRequest()` method is invoked after `getBookPrice()` is called, but before the SOAP message is sent across the network.

There are basically two ways that you can **configure the JAX-RPC runtime to use message handlers:** In a J2EE environment you must configure them using Web services deployment descriptors. In standalone applications you can add them programmatically at runtime, using the JAX-RPC API.

```
<service-ref xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
             xmlns:mi="http://www.Monson-Haefel.com/jwsbook/messsage-id" >
  <service-ref-name>service/BookQuoteService</service-ref-name>
  <service-interface>com.jwsbook.jaxrpc.BookQuoteService
  </service-interface>
  <wsdl-file>META-INF/BookQuote.wsdl</wsdl-file>
  <jaxrpc-mapping-file>META-INF/mapping.xml</jaxrpc-mapping-file>
  <service-qname>mh:BookQuoteService</service-qname>
  <handler>
    <handler-name>MessageID</handler-name>
    <handler-class>com.jwsbook.jaxrpc.MessageIDHandler</handler-class>
    <soap-header>mi:message-id</soap-header>
    <soap-role>http://www.Monson-Haefel.com/logger</soap-role>
    <port-name>BookQuotePort</port-name>
  </handler>
</service-ref>
```

The `service-ref` element describes a reference to a JAX-RPC Web service endpoint. Basically, the `service-ref` element identifies the directory name of the service in the JNDI ENC, the endpoint and service interfaces, and the location of the WSDL document that describes the Web service. In addition, you can configure message handlers that are to be used in conjunction with the service reference.

Once you have developed one or more message handlers, defined the `service-ref` element, and generated the service implementation and endpoint stub, you are ready to deploy your J2EE component. **Remember:** The JAX-RPC runtime adds the `message-id` header block after the `getBookPrice()` method is invoked, but before the SOAP message is sent across the network to the BookQuote Web service.

Handler Chains and Oder of Processing

In some cases you will need to use several message handlers, each assigned to process a different header block. In these cases, the message handlers are chained: They process the SOAP message serially, one after another, in the order in which you declare them in the deployment descriptor.

For example, imagine that you use three message handlers: `MessageIDHandler`, `ProcessedByHandler`, and `ClientSecurityHandler`. The `MessageIDHandler` we've already discussed. The `ProcessedByHandler` is responsible for adding to the message the `processed-by` header block, which logs information about the nodes along the message path that process the message. This information is useful for debugging. The `ClientSecurityHandler` is responsible for adding an XML digital signature to the message. The digital signature reveals whether the `Body` of the message was tampered with en route. An XML digital signature doesn't actually encrypt the message; it simply generates a unique sequence of bytes from the data that can be reproduced only if the data has not changed. So the SOAP message processed by these 3 message handlers will look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Header>
      <mi:message-id soap:actor="http://www.Monson-Haefel.com/logger">
          deabc782dbbd11bf:29e357:f1ad93fdbf:-8000
      </mi:message-id>
      <proc:processed-by
        soap:actor="http://schemas.xmlsoap.org/soap/actor/next">
        <node>
          <time-in-millis>1013694684723</time-in-millis>
          <identity>http://www.client.com/JaxRpcExample_6</identity>
        </node>
      </proc:processed-by>
      <sec:Signature>
        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <ds:SignedInfo>
            ...
          </ds:SignedInfo>
          <ds:SignatureValue>CFFOMFCtVLrklR...</ds:SignatureValue>
        </ds:Signature>
      </sec:Signature>
   </soap:Header>
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

Here's how the 3 message handlers will be specified in the DD:

```
<service-ref>
  <service-ref-name>service/BookQuoteService</service-ref-name>
  ...
  <handler>
    <handler-name>MessageID</handler-name>
    <handler-class>com.jwsbook.jaxrpc.MessageIDHandler</handler-class>
    <soap-header>mi:message-id</soap-header>
    <soap-role>http://www.Monson-Haefel.com/logger</soap-role>
    <port-name>BookQuotePort</port-name>
  </handler>
  <handler>
    <handler-name>ProcessedBy</handler-name>
    <handler-class>com.jwsbook.jaxrpc.ProcessedByHandler</handler-class>
    <soap-header>proc:processed-by</soap-header>
    <soap-role>http://schemas.xmlsoap.org/soap/actor/next</soap-role>
    <port-name>BookQuotePort</port-name>
```



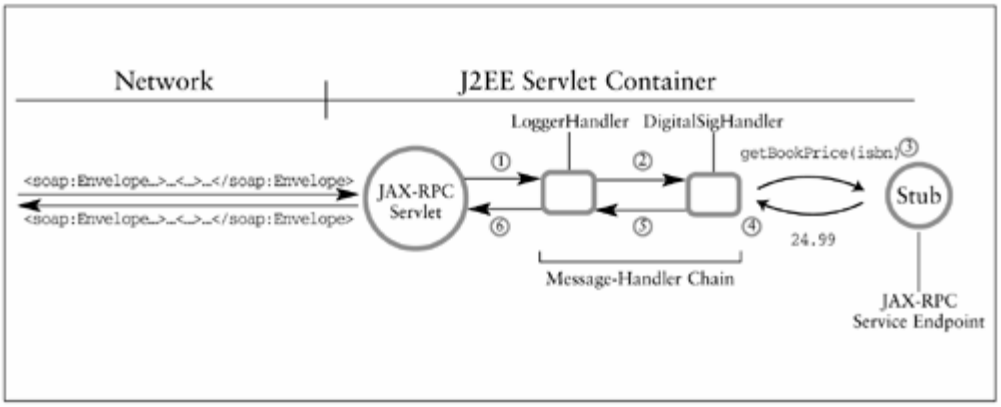
```
  </handler>
  <handler>
    <handler-name>ClientSecurityHandler</handler-name>
    <handler-class>com.jwsbook.jaxrpc.ClientSecurityHandler
    </handler-class>
    <soap-header>sec:Signature</soap-header>
    <port-name>BookQuotePort</port-name>
  </handler>
</service-ref>
```

Message handlers can **process both outgoing and incoming messages**. When a client application sends SOAP messages, the `handleRequest()` method of each message handler processes the outgoing messages, while the **`handleResponse()`** method processes incoming messages (the replies from the Web service). All the message handlers have an opportunity to process both outgoing and incoming messages, in the order in which they are configured. **When the Web service endpoint sends a reply back to the client, the message handlers process it in the reverse order that they processed the corresponding request**.

JAX-RPC Service Endpoints to use message handlers in the same way a J2EE client does, with one significant difference: The `handleRequest()` method of each message handler will process the incoming messages—those sent by the client—while the `handleResponse()` method will process the outgoing messages—the replies sent back to the client. In other words, SOAP requests are outgoing messages for clients but incoming messages for endpoints, and SOAP replies are outgoing messages for endpoints but incoming messages for clients.

Message handlers for Web service endpoints process messages in the order they are declared in the J2EE component's **`webservice.xml`** deployment descriptor. This file is a separate document that describes the Web service aspects of a J2EE component. When you develop a JSE or an EJB endpoint, you must define a `webservice.xml` deployment descriptor. The message handlers for JSEs and EJB endpoints are configured in the `webservices.xml` deployment descriptor in the same way that JAX-RPC clients' message handlers are configured in the `service-ref` element.

```
<webservices ...
  xmlns:ds="http://schemas.xmlsoap.org/soap/security/2000-12"
  xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by"
  xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id" ...>
  <webservice-description>
    <webservice-description-name>BookQuote</webservice-description-name>
    ...
    <port-component>
      <port-component-name>BookQuoteJSE</port-component-name>
      ...
      <handler>
        <handler-name>Logger</handler-name>
        <handler-class>com.jwsbook.jaxrpc.LoggerHandler</handler-class>
        <soap-header>mi:message-id</soap-header>
        <soap-header>proc:processed-by</soap-header>
        <soap-role>http://www.Monson-Haefel.com/logger</soap-role>
      </handler>
      <handler>
        <handler-name>ServerSecurityHandler</handler-name>
        <handler-class>com.jwsbook.jaxrpc.ServerSecurityHandler
        </handler-class>
        <soap-header>ds:Signature</soap-header>
      </handler>
    </port-component>
  </webservice-description>
</webservices>
```



One thing to keep in mind is that returning `false` from a `handleXXX()` method does not generate a SOAP fault message. A fault message is generated when a `handleXXX()` method throws an exception.

When a `handleRequest()` method on a JSE or EJB endpoint returns `false`, the JAX-RPC runtime immediately aborts the normal processing of the SOAP message and sends it back through the handler chain in the direction it came from, starting from the current message handler. Before a server-side handler's `handleRequest()` method returns `false`, it must change the SOAP request message into a SOAP reply message. That way, when the message goes back up the handler chain, it's in the form of a SOAP reply message, as the handlers who receive it expect.

If a client-side `handleRequest()` method returns `false`, the SOAP message is never sent over the network; it just reverses direction and is changed into a reply message.

When a server-side `handleResponse()` method returns `false`, the SOAP message does not reverse direction. Instead, the message skips the rest of the message handlers in the chain and the JAX-RPC servlet sends it straight back to the client. A handler would do this if it determined that a SOAP reply message needed no further processing.

Returning `false` from a `handleFault()` method on the server side has the same effect as returning `false` from a server-side `handleResponse()` method; the fault message just skips the rest of the handlers and is sent across the network to the client. On the client side, returning `false` from a `handleFault()` method has the same effect as returning `false` from a `handleResponse()` method: The fault message skips the rest of the handlers, and returns directly to the client as a `RemoteException` or some type of application exception (a SOAP fault exception defined by the developer).

While short-circuiting message-handler chains is discouraged for normal processing, it's a good approach to fault handling—if you do it the right way. In the event of a fault (for example, an erroneous header block), don't return `false` from the handler that detected the error, throw an exception. Only return `false` to avoid processing by subsequent headers.

There are basically two kinds of exceptions that a `handleXXX()` method can throw: **`javax.xml.rpc.soap.SOAPFaultException`** and **`javax.xml.rpc.JAXRPCException`**. A `SOAPFaultException` is used to indicate that a SOAP fault has occurred—that the SOAP message is incorrect or could not be processed. A `JAXRPCException` is used when some type of runtime error occurs that prevents the message handler from functioning properly.

1. A **`SOAPFaultException`** can be thrown only by the **`handleRequest()`** methods of a handler chain on the server side, when a message handler encounters a problem with the SOAP message itself. Perhaps the header block is structured improperly or the message contains invalid values.When a server-side handler throws a `SOAPFaultException` from its `handleRequest()` method, the JAX-RPC runtime will short-circuit the handler chain and reverse the direction of message flow back up the message chain, just as it does when the `handleRequest()` method returns `false`. As the message flows back up the handler chain, it's processed by the **`handleFault()`** methods of each handler in the chain, however, rather than by the `handle Response()` methods.
2. A message handler should throw a `JAXRPCException` whenever it encounters a problem processing a SOAP message that is not related to the message's contents—for example, when it

cannot do its own processing because a JDBC method throws a `SQLException`. On the server side a `JAXRPC Exception` always results in a SOAP fault being generated and sent back to the client (assuming Request/Response messaging). The SOAP fault message has a fault code of `"soap:Server"` to indicate that the problem is not related to the SOAP message itself, that it results from some server-side error. All handler methods on the server side (`handleRequest()`, `handleResponse()`, and `handleFault()`) behave the same way. on the client side a `JAXRPCException` indicates that the client-side JAX-RPC runtime encountered some error sending or receiving the SOAP message that's unrelated to the message contents. Perhaps there was a class loader problem or an array overflow. When a `JAXRPCException` is thrown, the JAX-RPC runtime simply returns a `java.rmi.RemoteException`, or one of its subtypes, to the client. Once a message handler throws a `JAXRPCException`, no other handlers will get to process that SOAP message. On the client side a `RemoteException` type is immediately thrown to the client application. On the server side a SOAP fault is immediately sent to the client. The `JAXRPCException` is a subtype of `java.lang.RuntimeException`.

Handler Runtime Environment

A message handler is itself a J2EE component, with its own behavior and life cycle. Message handlers are stateless objects. Message handlers that serve JSEs (depending on whether the JSE they serve are using single-thread or multi-thread model) and J2EE application clients may be single- or multi-threaded, but message handlers that serve EJB endpoints are always single-threaded (as are EJBs – always single threaded).

Although message handlers are stateless, it is possible to exchange information, or state, between message handlers, using the **`MessageContext`** object. In most cases, instance variables in message handlers maintain references to resources that are not associated with specific SOAP messages. A reference to a JDBC `DataSource` object, for example. You should ensure that any object an instance variable refers to can tolerate access from multiple threads, because it's possible that the message handler will be used in a multi-threaded environment. For example, maintaining a reference to a JDBC `DataSource` is fine, because the `javax.sql.DataSource` object can handle multiple threads—each thread receives its own JDBC `Connection` object. Similarly, the `java.security.KeyPair` is not a problem. It's not a good idea, however, to maintain a JDBC `Connection` reference. Although some JDBC vendors support concurrent use of a `Connection` object by multiple threads, others employ synchronization to serialize access to it. If many threads attempt to use the same `Connection` object, the JDBC driver that serializes access will become a major bottleneck.

Message handlers always execute in the context of the component they are serving, which means that they have access to the JNDI ENC of the component. A message handler has access to any resources, EJB references, Web service references, or environment variables that are configured for the J2EE component it's serving.

```
package com.jwsbook.jaxrpc;
import javax.xml.namespace.QName;
import javax.xml.rpc.handler.*;
import javax.xml.rpc.handler.soap.*;
import javax.xml.soap.*;
import java.util.*;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.Statement;
import javax.naming.InitialContext;
import com.jwsbook.saaj.SaajOutputter;

public class LoggerHandler extends javax.xml.rpc.handler.GenericHandler{
  QName [] headerNames = null;
  javax.sql.DataSource dataSource;

  public QName [] getHeaders(){...}

  public boolean handleRequest(MessageContext context){
```

```
    String messageID = null;

    try{
      SOAPMessageContext soapCntxt = (SOAPMessageContext)context;
      SOAPMessage message = soapCntxt.getMessage();
      SOAPHeader header = message.getSOAPPart().getEnvelope().getHeader();

      Iterator allHeaders = header.getChildElements();
      while(allHeaders.hasNext()){
        SOAPElement headerBlock = (SOAPElement)allHeaders.next();
        Name headerName = headerBlock.getElementName();
        if(headerName.getLocalName().equals("message_id")){
          messageID = header.getValue();
        }
      }
      if(dataSource == null){
        InitialContext jndiEnc = new InitialContext();
        dataSource = (DataSource)
                     jndiEnc.lookup("java:comp/env/jdbc/MyDatabase");
      }

      String soapMessage = SaajOutputter.writeToString(message);
      try{
        Connection conn = dataSource.getConnection();
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("INSERT INTO soap_log ("
                           +messageID+","
                           +System.currentTimeMillis()+","
                           +soapMessage+")");
        conn.close();
        return true;
      }catch(java.sql.SQLException sqe){
         throw new javax.xml.rpc.JAXRPCException(sqe);
      }
    }catch(javax.xml.soap.SOAPException se){
      throw new javax.xml.rpc.JAXRPCException(se);
    }catch(javax.naming.NamingException ne){
      throw new javax.xml.rpc.JAXRPCException(ne);
    }
  }
}
```

As seen above, subsystem (JDBC, JNDI, or SAAJ) exceptions are caught and rethrown as `javax.xml.rpc.JAXRPCException` types.

A message handler's life cycle is delimited by its **`init`**`()` and **`destroy`**`()` methods. Before a message-handler instance is allowed to process SOAP messages, the JAX-RPC runtime invokes its `init()` method. The message handler must include a no-argument constructor so that the J2EE container can instantiate it using the `Class.newInstance()` method. The `init()` method takes a single parameter, `HandlerInfo`, which provides the handler with configuration information that it may need to process messages. The `init()` method is invoked only once in the life of an instance.

The `javax.xml.rpc.handler.`**`HandlerInfo`** class provides a message handler with configuration information, in the form of a `java.util.Map` object, when it's initialized. The configuration `Map` is accessed via the `HandlerInfo` object's `getHandlerConfig()` method.

```
package javax.xml.rpc.handler;

public class HandlerInfo implements java.io.Serializable {
    public HandlerInfo() {...}
    public HandlerInfo(Class handlerClass, java.util.Map config,
                       QName[] headers) {...}

    public java.util.Map getHandlerConfig(){...}

    public Class getHandlerClass() {...}
    public QName[] getHeaders(){...}

    public void setHandlerClass(java.lang.Class handlerClass){...}
    public void setHandlerConfig(java.util.Map config){...}
```

```
    public void setHeaders(QName[] headers){...}
}
```

Because J2EE container takes care of constructing the handler chain and the `HandlerInfo` objects automatically, the `HandlerInfo` class's `setHandlerClass()`, `setHandlerConfig()`, and `setHeaders()` methods simply don't work in J2EE; they are ignored. (These methods are useful in standalone applications where deployment descriptors are not used.) In J2EE, the `Class` and configuration `Map` are defined in the deployment descriptor, and the header-block `QName` objects are defined by the message handler's implementation of the `getHeaders()` method.

```
<handler xmlns:ds="http://schemas.xmlsoap.org/soap/security/2000-12">
   <handler-name>ClientSecurityHandler</handler-name>
   <handler-class>com.jwsbook.jaxrpc.ServerSecurityHandler</handler-class>
   <init-param>
       <param-name>KeyAlgorithm</param-name>
       <param-value>DSA</param-value>
   </init-param>
   <init-param>
       <param-name>KeySize</param-name>
       <param-value>512</param-value>
   </init-param>
   <init-param>
       <param-name>SignatureAlgorithm</param-name>
       <param-value>SHA1</param-value>
   </init-param>
   <soap-header>ds:Signature</soap-header>
 </handler>
```

The `ClientSecurityHandler` creates a `KeyPair` object and a `Signature` object based on the initialization parameters.

```
package jwsed1.part5_jaxprc;
import java.security.*;
import javax.xml.namespace.QName;

public class ClientSecurityHandler extends javax.xml.rpc.handler.GenericHandler {
  java.security.KeyPair keyPair;
  java.security.Signature signature;
  QName [] headerNames = null;

  public QName [] getHeaders(){...}

  public void init(javax.xml.rpc.handler.HandlerInfo info ){
    try{
      java.util.Map map = info.getHandlerConfig();
      String keyAlgorithm = (String)map.get("KeyAlgorithm");
      String sigAlgorithm = (String)map.get("SignatureAlgorithm");
      int keySize = Integer.parseInt((String)map.get("KeySize"));

      KeyPairGenerator generator = KeyPairGenerator.getInstance(keyAlgorithm);
      generator.initialize(keySize);
      keyPair = generator.generateKeyPair();

      signature = Signature.getInstance(sigAlgorithm);
      signature.initSign(keyPair.getPrivate());
    }catch(java.security.GeneralSecurityException se){
      throw new javax.xml.rpc.JAXRPCException(se);
    }
  }
  ...
}
```

When a handler instance is no longer needed by the JAX-RPC runtime, its `Handler.destroy()` method is called just before it is discarded. This method gives the handler an opportunity to free up any resources it was using during its life (JDBC connections, JMS connections, and so forth).

MessageContext:

a `SOAPMessageContext` object is passed to each of the `handleXXX()` methods of the `Handler` interface. You also learned that `SOAPMessageContext` defines a method for accessing the

SOAPMessage object. In addition, the SOAPMessageContext defines a method named getRoles(). This method tells you which roles the entire message-handler chain is able to process on the SOAP message.

You can also take advantage of the property methods defined by the SOAPMessageContext type's supertype, **MessageContext**. During the course of a message handler's life it will be used to process many different SOAP requests, replies, and sometimes fault messages. As you've learned, message handlers are stateless, but in some cases you will want to pass information specific to a SOAP message from one message handler to the next in a handler chain. This is where the property methods of javax.xml.rpc.handler.MessageContext come in handy.

```
package javax.xml.rpc.handler.soap;
import javax.xml.soap.SOAPMessage;
import java.util.Iterator;

public interface SOAPMessageContext extends javax.xml.rpc.handler.MessageContext {
   public SOAPMessage getMessage();
   public void setMessage(SOAPMessage message);
   public String[] getRoles();

   public boolean containsProperty(String name);
   public Object getProperty(String name);
   public Iterator getPropertyNames();
   public void removeProperty(String name);
   public void setProperty(String name, Object value);
}
```

As a SOAP message passes from one message handler to the next in a handler chain, the SOAPMessageContext object that carries it can also carry properties. Any message handler in the chain can read, add, update, or remove these properties—and use them to pass information about the message to the next handler in the chain. Once the Web service consumes a SOAP message, the SOAPMessageContext object and its properties are discarded. **If you need to pass information between message handlers while processing a SOAP message, use the property methods defined in the MessageContext interface and inherited by the SOAPMessageContext.**

In most cases the only objects that need access to the SOAPMessageContext are the message-handler objects, but a JSE can also access the SOAPMessageContext via its ServiceEndpointContext.

```
public class BookQuoteImpl implements BookQuote {
    ...
    ServletEndpointContext endPtCntxt;

    public void init(Object context) throws ServiceException{
       try{
         endPtCntxt = (ServiceEndpointContext)context;
         ...
    }
    public float getBookPrice(String isbn) {

         SOAPMessageContext msgCntxt = endPtCntxt.getMessageContext();
         ...
    }
...
}
```

The JSE can use the SOAPMessageContext and its property methods to obtain information from its message handlers that may not be communicated in headers themselves. The JSE can also access the SOAPMessage object via the SOAPMessageContext.

In J2EE 1.4 an EJB endpoint can access the SOAPMessageContext object via the SessionContext reference, which is handed to the EJB endpoint at the beginning of its life cycle.

```
package com.jwsbook.jaxrpc;
import javax.ejb.SessionContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
```

```
public class BookQuoteBean implements javax.ejb.SessionBean {
    SessionContext sessionContext;

    public void setSessionContext(SessionContext cntxt){
        sessionContext = cntxt;
    }

    public void ejbCreate(){}

    public float getBookQuote(String isbn) {
        SOAPMessageContext msgCntxt = (SOAPMessageContext)
          sessionContext.getMessageContext();
        ...

    }

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove(){}
}
```

Given:

A Web service with two SOAP handlers: handlerA and handlerB. A client makes a SOAP request: handlerA.handleRequest() is invoked followed by handlerB.handleRequest().

What is the result if handlerB.HandleRequest() returns false?

A The service endpoint is invoked.
B handlerB.HandleResponse() is invoked.
C handlerB.HandleFault() is invoked.
D handlerA.HandleResponse() is invoked.
E handlerA.HandleFault() is invoked.

Ans: B.

Mapping Java to WSDL and XML (4.5)

Mapping WSDL to Java

WSDL names the operations that can be invoked and describes the exact format of SOAP messages, the Internet protocol used (for example, HTTP) and the Internet addresses of Web service endpoints. JAX-RPC defines how developers can use WSDL documents to generate endpoint and service interfaces, and the classes that implement them. JAX-RPC can also be used to generate WSDL documents from endpoint interface definitions and configuration files.

A WSDL document has both abstract definitions and implementation definitions. The abstract definitions are declared in the `types`, `message`, and `portType` elements. They're considered abstract because they define what the Web service looks like (operation names, data types, and the like) but not the message formats and protocols used to access an endpoint. The `binding` and `service` are considered implementation definitions because they tell us how the abstract definitions are bound to a specific SOAP messaging mode, Internet protocol, and Internet address.

There is a one-to-one mapping between a `portType` and a JAX-RPC endpoint interface. The mapping is not complex, and it applies only to One-Way and Request/Response operation styles; JAX-RPC doesn't provide a mapping for the Solicit/Response or Notification operation styles.

Mapping a Java package to an XML namespace requires a special configuration file called a **JAX-RPC mapping file**.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FooWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

  <message name="fooRequest">
    <part name="param1" type="xsd:string"/>
  </message>
  <message name="fooResponse">
    <part name="param2" type="xsd:float"/>
  </message>
  <portType name="FooBar">
    <operation name="foo" >
      <input message="mh:fooRequest"/>
      <output message="mh:fooResponse"/>
    </operation>
  </portType>
```

This `portType` definition describes a Web service operation that takes a single `xsd:string` type as input and returns an `xsd:float` type value as output. When a JAX-RPC compiler reads this `portType` definition, it will generate the endpoint interface.

```
public interface FooBar extends java.rmi.Remote {
    public float foo(java.lang.String param1)
    throws java.rmi.RemoteException;
}
```

The `portType` name is mapped directly to the endpoint interface name. Similarly, the `operation` names and the `part` names are mapped to method and parameter names of the endpoint interface. The mapping is always one-to-one, character-for-character, unless the name is illegal in Java, in which case the mapping is subject to certain name-collision rules. It's also possible to deviate from this strict one-to-one mapping of names by specifying different names for methods in a JAX-RPC mapping file.

When the input message declares multiple `part` elements, the corresponding endpoint method will have multiple parameters. It's also possible to define a `portType` whose output message has multiple parts.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FooWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<message name="fooRequest">
  <part name="param1" type="xsd:string"/>
</message>
<message name="barRequest">
  <part name="param2" type="xsd:float"/>
</message>
<message name="fooBarResponse">
  <part name="param3" type="xsd:int" />
</message>
<message name="fooRequest2">
  <part name="param1" type="xsd:string"/>
</message>

<message name="fooRequest1">
  <part name="param1" type="xsd:string"/>
  <part name="param2" type="xsd:int" />
  <part name="param3" type="xsd:boolean" />
</message>
<message name="fooResponse1">
  <part name="param4" type="xsd:float"/>
  <part name="param5" type="xsd:boolean" />
  <part name="param6" type="xsd:float"/>
</message>
```

```
<portType name="FooBar">
  <operation name="foo1" >
    <input message="mh:fooRequest1"/>
    <output message="mh:fooResponse1"/>
  </operation>
  <operation name="foo" >
    <input message="mh:fooRequest"/>
    <output message="mh:fooBarResponse"/>
  </operation>
  <operation name="bar" >
    <input message="mh:barRequest"/>
    <output message="mh:fooBarResponse"/>
  </operation>
  <operation name="foo2" >
    <input message="mh:fooRequest"/>
  </operation>

</portType>
```

```
public interface FooBar extends java.rmi.Remote {
    public void foo1(java.lang.String param1,
                     int param2,
                     boolean param3,
                     javax.xml.rpc.holders.IntHolder param4,
                     javax.xml.rpc.holders.BooleanHolder param5,
                     javax.xml.rpc.holders.FloatHolder param6)
    throws java.rmi.RemoteException;
    public int foo(String param1) throws java.rmi.RemoteException;
    public int bar(float param2) throws java.rmi.RemoteException;
    public void foo2(String param1) throws java.rmi.RemoteException;
}
```

Because WSDL is language-agnostic, it allows Web services to define INOUT and OUT parameters in addition to the IN parameters typically used in Java. JAX-RPC overcomes Java's limitations, and accommodates INOUT and OUT parameters through the use of special wrapper classes, called **holders**. Param1, param2 and param3 are normal IN parameters whereas, param4, param5 and param6 are OUT parameters are declared using **javax.xml.rpc.holders.XXXHolder** types. (foo1 in the above example).

A WSDL `portType` may define multiple operations, each of which maps to a different method in an endpoint interface. JAX-RPC compilers also support operation overloading in WSDL as long as the operations define different input messages, output messages, or both. **Operation overloading is not allowed by the WS-I Basic Profile 1.0**.

A `portType` indicates that it uses Request/Response messaging by listing an `input` message followed by an `output` message, and that it uses One-Way messaging by listing only an `input` message. One-Way endpoint interface methods always return `void`. (foo2 in the above example).

It's important to understand two things about One-Way messaging in JAX-RPC.

1. First, you cannot define faults. Conceptually, the Web service (receiver) is not supposed to respond to the client (sender).
2. Second, when using an HTTP SOAP binding, the underlying protocol is still Request/Response HTTP, but the reply is limited to HTTP success or failure codes. An HTTP success code (for example, `200 OK`) will result in a successful return of the method, while a failure code such as `500` will generally cause a `java.rmi.RemoteException` to be thrown.

Mapping XML Schema to Java

JAX-RPC Compiler may also generate custom JavaBeans components and special `javax.xml.rpc.holders.Holder` types for parameters and return values, depending on the complexity of the `message` definitions used by the `portType`.

Many of the XML schema built-in types map cleanly to Java primitive types. Other XML schema built-in types map to standard Java classes like `String`, `Calendar`, and `BigDecimal`, or to primitive byte arrays.

XML Schema Built-in Type	*Java Primitive or Class type*	*Java Class type if nillable=ture*	*Holder type*	*Holder type if nillable=true*
xsd:byte	byte	java.lang.Byte	ByteHolder	ByteWrapperHolder
xsd:boolean	boolean	java.lang.Boolean	BooleanHolder	BooleanWrapperHolder
xsd:short	short	java.lang.Short	ShortHolder	ShortWrapperHolder
xsd:int	int	java.lang.Integer	IntHolder	IntegerWrapperHolder
xsd:long	long	java.lang.Long	LongHolder	LongWrapperHolder
xsd:float	float	java.lang.Float	FloatHolder	FloatWrapperHolder
xsd:double	double	java.lang.Double	DoubleHolder	DoubleWrapperHolder
xsd:string	java.lang.String	-	StringHolder	-
xsd:dateTime	java.util.Calendar	-	CalendarHolder	-
xsd:integer	java.lang.BigInteger	-	BigIntegerHolder	-
xsd:decimal	java.lang.BigDecimal	-	BigDecimalHolder	-
xsd:Qname	java.xml.namespace.QName	-	QNameHolder	-
xsd:base64Binary	byte[]	-	ByteArrayHolder	-
xsd:hexBinary	byte[]	-	ByteArrayHolder	-

A message `part` definition can be based on an XML schema complex type.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
   <xsd:schema
    targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote">
    <xsd:simpleType name="USState">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="AK"/> <!-- Alaska   -->
        <xsd:enumeration value="AL"/> <!-- Alabama  -->
        <xsd:enumeration value="AR"/> <!-- Arkansas -->
        <!-- and so on ... -->
      </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="Book">
      <xsd:sequence>
        <xsd:element name="title"           type="xsd:string"/>
        <xsd:element name="isbn"            type="xsd:string"/>
        <xsd:element name="authors"         type="mh:Author"
                                            maxOccurs="unbounded"/>
        <xsd:element name="state"           type="mh:USState"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Author">
      <xsd:sequence>
        <xsd:element name="firstName"          type="xsd:string"/>
        <xsd:element name="lastName"           type="xsd:string"/>
        <xsd:element name="middleInitial"      type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
</types>
<message name="getBookPriceRequest">
  <part name="book" type="mh:Book"/>
</message>
<message name="getBookPriceResponse">
  <part name="result" type="xsd:float"/>
</message>
```

```
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input message="mh:getBookPriceRequest"/>
    <output message="mh:getBookPriceResponse"/>
  </operation>
</portType>
```

```
public interface BookQuote extends java.rmi.Remote {
    public float getBookQuote(Book book) throws java.rmi.RemoteException;
}
```

Complex types are mapped to JavaBeans class definitions, where the name of the bean class is the name of the XML schema complex type. Bean properties map to the elements of the complex type.

```
public class Book {
    private String title;
    private String isbn;
    private Author [] authors;
    private USState state;

    public void Book(){}

    public String getTitle() { return title;}
    public void setTitle(String title){ this.title = title;}

    public String getIsbn() {return isbn;}
    public void setIsbn(String isbn){this.isbn = isbn;}

    public Author [] getAuthors(){ return authors;}
    public void setAuthors(Author [] authors){this.authors = authors;}

    public USState getState() {return state;}
    public void setState(USState state){this.state = state;}

}
```

Anytime the type's `maxOccurs` attribute is greater than one, the JAX-RPC will map to an array type. In general, `minOccurs` attribute values other than `"0"` are ignored—there is no standard way to support `minOccurs` in Java.

You can even map complex types that include elements of other complex types into Java beans.

```
public class Author {
    private String firstName;
    private String lastName;
    private String middleInitial;

    public String getFirstName() { return firstName;}
    public void setFirstName(String firstName){this.firstName = firstName;}

    public String getLastName() { return lastName;}
    public void setLastName(String lastName){this.lastName = lastName;}

    public String getMiddleInitial() { return middleInitial;}
    public void setMiddleInitial(String middleInitial){
           this.middleInitial = middleInitial;
    }
}
```

There are several ways to declare arrays in a WSDL document, including the use of `maxOccurs` greater than one for elements in a structure, as just illustrated, SOAP RPC/Encoding arrays, and WSDL modifiers for RPC/Encoded arrays. JAX-RPC supports all of these, but the Basic Profile supports `maxOccurs` values greater than one only for elements in a structure. The other types of arrays are explicitly forbidden.

When a JAX-RPC compiler encounters an XML schema enumeration, it generates a Java class that models the enumeration appropriately. Probably this will change with J2SE 5.0 supporting enums.

```
public class USState {
    private String value;

    protected USState(String state) {
        value = state;
    }
```



```
    public static final String _AK = "AK";
    public static final String _AL = "AL";
    public static final String _AR = "AR";
    ...
    public static final USState AK = new USState(_AK);
    public static final USState AL = new USState(_AL);
    public static final USState AR = new USState(_AR);
    ...
    public String getValue() { return value;}

    public static USState fromValue(String other){
      if(other.equals(_AK)) return AK;
      else if(other.equals(_AL)) return AL;
      else if(other.equals(_AR)) return AR;
      ...
    }
    public boolean equals(Object obj) { ... }
    public int hashCode() { ... }
}
```

When the Document/Literal messaging mode is used, nodes exchange document fragments, rather than SOAP representations of RPC calls. An example is the PurchaseOrder Web service used by retailers to send purchase orders to Monson-Haefel Books. The PurchaseOrder Web service consumes SOAP messages that carry a `purchaseOrder` document fragment, based on the Purchase Order Markup Language.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
    xmlns:mh="http://www.Monson-Haefel.com/jwsbook/PurchaseOrder"
    targetNamespace="http://www.Monson-Haefel.com/jwsbook/PurchaseOrder">

  <types>
    <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO">
    <xsd:import namespace="http://www.Monson-Haefel.com/jwsbook/PO"
      schemaLocation="http://www.Monson-Haefel.com/jwsbook/po.xsd"/>
    </xsd:schema>
  </types>

  <message name="PurchaseOrderMessage">
    <part name="body" element="po:purchaseOrder"/>
  </message>
  <portType name="PurchaseOrder">
    <operation name="submitPurchaseOrder">
      <input message="mh:PurchaseOrderMessage"/>
    </operation>
  </portType>
  <binding name="PurchaseOrderBinding" type="mh:PurchaseOrder">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="submitPurchaseOrder">
      <input>
        <soap:body use="literal"/>
      </input>
    </operation>
  </binding>
  <service name="PuchaseOrderService">
    <port name="PurchaseOrderPort" binding="mh:PurchaseOrderBinding">
      <soap:address
          location="http://www.Monson-Haefel.com/jwsbook/PurchaseOrder"/>
    </port>
  </service>
</definitions>
```

When a JAX-RPC toolkit generates a stub and an endpoint interface from this WSDL definition, it will usually define a method with a single Java bean argument representing the document fragment and a `void` return type.

```
public interface PurchaseOrder extends java.rmi.Remote {
```



```
    public void submitPurchaseOrder(PurchaseOrder purchaseOrder)
    throws java.rmi.RemoteException;
}
```

The return type for the `submitPurchaseOrder()` method is `void` because the operation style is One-Way (specified by the absence of an `output` element in the `portType` definition). The `PurchaseOrder` parameter's type is a Java bean class, which represents the `purchaseOrder` document fragment.

Occasionally, however, you may bump into an XML schema type that is not supported by your JAX-RPC compiler, in which case the compiler should switch to using the SAAJ `SOAPElement` type as a parameter instead of a Java bean class.

```
public interface PurchaseOrder extends java.rmi.Remote {
    public void submitPurchaseOrder(javax.xml.soap.SOAPElement element)
    throws java.rmi.RemoteException;
}
```

The `SOAPElement` type represents the `purchaseOrder` document fragment that is sent to the PurchaseOrder Web service. To use this service, a client must construct a `SOAPElement` object and pass it to the endpoint stub when invoking the `submitPurchaseOrder()` endpoint method.

The WSDL document for the PurchaseOrder Web service imports the XML schema for Purchase Order Markup using the `types` element. Importing it rather than embedding it allows you to reuse the XML schema in other applications, and helps ensure that there is a single, uniform source for the `PurchaseOrder` type.

You should pay special attention to the use of the **`group`** type shown in bold, because **it's a JAX-RPC non-standard type**. Support for this type is optional. If your JAX-RPC compiler doesn't support the `group` type, then it should generate an endpoint that uses the `SOAPElement` as a parameter. The strategy of using the `SOAPElement` type allows you to exchange messages with Web services even when the JAX-RPC compiler doesn't recognize their XML schema types. The downside to this strategy is that you have to construct the XML document fragment by hand using SAAJ.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
    targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO">

  <element name="purchaseOrder" type="po:PurchaseOrder"/>

  <complexType name="PurchaseOrder">
    <sequence>
      <element name="accountName" type="string"/>
      <element name="accountNumber" type="short"/>
      <group ref="po:shipAndBill"/>
      <element name="book" type="po:Book"/>
      <element name="total" type="float"/>
    </sequence>
  </complexType>
  <group name="shipAndBill">
      <sequence>
          <element name="shipTo" type="po:USAddress" maxOccurs="0"/>
          <element name="billTo" type="po:USAddress"/>
      </sequence>
  </group>
  <complexType name="USAddress">
    <sequence>
      <element name="name" type="string"/>
      <element name="street" type="string"/>
      <element name="city" type="string"/>
      <element name="state" type="string"/>
      <element name="zip" type="string"/>
    </sequence>
  </complexType>
  <complexType name="Book">
    <sequence>
      <element name="title" type="string"/>
      <element name="quantity" type="short"/>
      <element name="wholesale-price" type="float"/>
```



```
    </sequence>
  </complexType>
</schema>
```

And this is how you will use SAAJ to create a valid document fragment and deliver it to WebService using JAX-RPC:

```
package com.jwsbook.jaxrpc;
import javax.naming.InitialContext;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.SOAPElement;

public class JaxRpcExample_7 {
  public static void main(String [] args) throws Exception{
    InitialContext jndiContext = new InitialContext();
    PurchaseOrderService service = (PurchaseOrderService)
      jndiContext.lookup("java:comp/env/service/PurchaseOrderService");
    PurchaseOrder po_stub = service.getPurchaseOrderPort();

    SOAPElement xml_fragment = getPurchaseOrderXML();

    po_stub.submitPurchaseOrder( xml_fragment);
  }
  public static SOAPElement getPurchaseOrderXML()
  throws javax.xml.soap.SOAPException{
    SOAPFactory factory = SOAPFactory.newInstance();
    SOAPElement purchaseOrder =
    factory.createElement("purchaseOrder","po",
                          "http://www.Monson-Haefel.com/jwsbook/PO");

    purchaseOrder.addChildElement("accountName").addTextNode("Amazon.com");
    purchaseOrder.addChildElement("accountNumber").addTextNode("923");

    SOAPElement billingAddress =
      purchaseOrder.addChildElement("billingAddress");
    billingAddress.addChildElement("name").addTextNode("Amazon.com");
    billingAddress.addChildElement("street").addTextNode("1516 2nd Ave");
    billingAddress.addChildElement("city").addTextNode("Seattle");
    billingAddress.addChildElement("state").addTextNode("WA");
    billingAddress.addChildElement("zip").addTextNode("90952");

    SOAPElement book = purchaseOrder.addChildElement("book");
    book.addChildElement("title").addTextNode("J2EE Web Services");
    book.addChildElement("quantity").addTextNode("3000");
    book.addChildElement("wholesale-price").addTextNode("24.99");

    return purchaseOrder;
  }
}
```

The main idea is that you pass the endpoint stub method a `SOAPElement`. When it sends a SOAP message to the PurchaseOrder Web service, the stub simply places the XML fragment in the body of the SOAP message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <po:purchaseOrder xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO">
      <accountName>Amazon.com</accountName>
      <accountNumber>923</accountNumber>
      <billingAddress>
        <name>Amazon.com</name>
        <street>1516 2nd Ave</street>
        <city>Seattle</city>
        <state>WA</state>
        <zip>90952</zip>
      </billingAddress>
      <book>
        <title>J2EE Web Services</title>
        <quantity>3000</quantity>
        <wholesale-price>24.99</wholesale-price>
      </book>
    </po:purchaseOrder>
```



```
  </soap:Body>
</soap:Envelope>
```

Corollary to the above example for one-way document/literal messaging, a request/response document/literal messaging will also behave similarly. Assuming that, like `PurchaseOrderRequest`, the `PurchaseOrderResponse` type is an XML document fragment that uses non-standard XML schema types, the toolkit would generate a `PurchaseOrder` endpoint interface whose method returns a `SOAPElement` instead of a `void` type:

```
import javax.xml.soap.SOAPElement;

public interface PurchaseOrder extends java.rmi.Remote {
    public SOAPElement submitPurchaseOrder( SOAPElement element)
    throws java.rmi.RemoteException;
}
```

When a WSDL `part` declares that its type uses an `xsd:any` element, the JAX-RPC compiler will use a **`SOAPElement`** to represent the **`xsd:any`** parameter or bean field.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FooWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
  <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo">
     <xsd:complexType name="xmlFragment">
      <xsd:sequence>
       <xsd:any namespace="##any"/>
      </xsd:sequence>
     </xsd:complexType>
     <xsd:complexType name="ArbitraryXML">
       <xsd:sequence>
         <xsd:element name="label"           type="xsd:string"/>
         <xsd:element name="xmlFragment"     type="mh:xmlFragment"/>
       </xsd:sequence>
     </xsd:complexType>
  </xsd:schema>
</types>
<message name="fooRequest">
  <part name="isbn" type="mh:ArbitraryXML"/>
</message>
<message name="fooResponse">
  <part name="result" type="mh:xmlFragment"/>
</message>
<portType name="foo">
  <operation name="bar">
    <input message="mh:fooRequest"/>
    <output message="mh:fooResponse"/>
  </operation>
</portType>

public interface FooBar extends java.rmi.Remote {
    public javax.xml.soap.SOAPElement foo(ArbitraryXML xmlDoc)
    throws java.rmi.RemoteException;
}
```

The return type, `mh:xmlFragment`, is mapped to the `SOAPElement` type, and the parameter is of the type `ArbitraryXML`, a bean class whose `xmlFragment` field is of type `SOAPElement`.

```
import javax.xml.soap.SOAPElement;

public class ArbitraryXML {
    private String label;
    private SOAPElement xmlFragment;

    public String getLabel() { return label;}
    public void setLabel(String label){ this.label = label;}
```

```
    public SOAPElement getXmlFragment() {return xmlFragment;}
    public void setXmlFragment(SOAPElement xmlFragment){
      this.xmlFragment = xmlFragment;
    }
}
```

A `part` can refer to an element that is **nillable**, meaning that the element can be empty without causing a validation error.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FooWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
  <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo">
       <xsd:element name="nillable_int" type="xsd:int" nillable="true" />
  </xsd:schema>
</types>
<message name="fooRequest">
  <part name="quantity" element="mh:nillable_int"/>
</message>
<portType name="Foo">
  <operation name="bar">
    <input message="mh:fooRequest"/>
  </operation>
</portType>
```

The implication is that the element in the SOAP message can be valid even when it's empty. An empty XML element maps to a `null` value in Java. Because primitive types cannot be `null`, nillable elements are mapped to Java primitive wrapper classes instead of to Java primitive types.

```
public interface Foo extends java.rmi.Remote {
    public void bar(java.lang.Integer quantity)
    throws java.rmi.RemoteException;
}
```

For each of the XML schema built-in types that map to a Java primitive, there is a corresponding Java primitive wrapper that can be used if a nillable element is specified.

Holders

To accommodate OUT and INOUT parameters JAX-RPC defines the `javax.xml.rpc.holders.Holder` interface, which is implemented by classes called **holders**, whose instances act as wrappers for OUT and INOUT parameters at runtime. Essentially, a `Holder` is an object that wraps around the argument you're passing. This mechanism allows the JAX-RPC runtime to change the value held by a `Holder` object before the method returns, so that you can extract the INOUT or OUT parameter value passed back from the Web service. Note that the value of a `Holder` type is always available via the public instance variable named **`value`**.

```
import javax.xml.rpc.holders.IntHolder;
import javax.xml.rpc.holders.CalendarHolder;

public interface FooBar extends java.rmi.Remote {
    public void foo(IntHolder param1, CalendarHolder param2)
    throws java.rmi.RemoteException;
}
```

Param1 – INOUT parameter
Param2 – OUT parameter
Here's how to invoke the above foo() web method:

```
IntHolder intHolder = new IntHolder( 10 );
CalendarHolder calendarHolder = new CalendarHolder();

FooBar fooBarStub = fooBarService.getFooBarPort();
```

```
fooBar.foo(intHolder, calendarHolder);

System.out.println("IntHolder = "+intHolder.value);
System.out.println("CalendarHolder = "+calendarHolder.value);
```

It's fairly easy to tell whether a WSDL operation uses INOUT or OUT parameters by examining the `parameterOrder` attribute of the `operation` and comparing the `part` elements defined by the `input` and `output` messages. Here's how the above end point interface will be described in the WSDL:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FooWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<message name="fooRequest">
  <part name="param1" type="xsd:int"/>
</message>
<message name="fooResponse">
  <part name="param1" type="xsd:int"/>
  <part name="param2" type="xsd:dateTime"/>
</message>
<portType name="FooBar">
  <operation name="foo" parameterOrder="param1 param2">
    <input message="mh:fooRequest"/>
    <output message="mh:fooResponse"/>
  </operation>
</portType>
```

Any time a `part` is declared in both `input` and `output` the JAX-RPC compiler knows it is an INOUT parameter, and will require a `Holder` type. In this case, the `part` specifies that `param1` is an INOUT parameter of type `xsd:int`, which maps to the `javax.xml.rpc.holders.IntHolder` type. The JAX-RPC compiler will also notice that one of the `part` elements is defined in the `output` message but not the `input` message, indicating that it is either a return value or an OUT parameter. If it should be an OUT parameter, then the `part` label will be listed in the `parameterOrder` attribute of the `operation` element. If the `part` should be a return value, it will not be listed in the `parameterOrder` attribute. Any `part` that is declared in the input message but not the output message is an IN parameter, and doesn't require a `Holder` class because Java automatically supports IN parameters.

Given:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FooWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<message name="fooRequest">
  <part name="param1" type="xsd:int"/>
  <part name="param2" type="xsd:double"/>
</message>
<message name="fooResponse">
  <part name="param2" type="xsd:double"/>
  <part name="param3" type="xsd:dateTime"/>
  <part name="param4" type="xsd:float" />
</message>
<portType name="FooBoo">
  <operation name="foo" parameterOrder="param1 param2 param3">
    <input message="mh:fooRequest"/>
    <output message="mh:fooResponse"/>
  </operation>
</portType>
```

Identify the IN, INOUT, OUT and return type parameters.

Ans:

```
import javax.xml.rpc.holders.DoubleHolder;
import javax.xml.rpc.holders.CalendarHolder;

public interface FooBoo extends java.rmi.Remote {
    public float foo(int param1, DoubleHolder param2, CalendarHolder param3)
    throws java.rmi.RemoteException;
}
```

The `parameterOrder` attribute is actually optional; if it's not declared, then all the parameters declared in the `input` message are listed first, followed by parameters declared by the `output` message. Unlisted INOUT parameters, those that are declared by both the `input` and `output` messages, are listed in the order they are declared in the `input` message only.

The standard holder types all follow the same design, which includes the definition of a no-argument constructor, a single-argument constructor, and a public instance field named `value`.

```
package javax.xml.rpc.holders;

public class IntHolder extends javax.xml.rpc.holders.Holder {
   public int value;

   public IntHolder(){
   }
   public IntHolder(int myint){
      value = myint;
  }
}
```

The **`Holder`** interface is purely a typing, or "marker," interface: It defines no methods.

```
package javax.xml.rpc.holders;

public interface Holder {
}
```

When the JAX-RPC compiler encounters an INOUT or OUT parameter of a type not already supported by the standard holder classes, it will generate a **custom `Holder` type** specifically for that parameter.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FooWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
  <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo">
    <xsd:complexType name="Book">
      <xsd:sequence>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element name="isbn" type="xsd:string"/>
        <xsd:element name="authors" type="mh:Author"
            maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Author">
       <xsd:sequence>
         <xsd:element name="firstName"          type="xsd:string"/>
         <xsd:element name="lastName"           type="xsd:string"/>
         <xsd:element name="middleInitial"      type="xsd:string"/>
       </xsd:sequence>
     </xsd:complexType>
  </xsd:schema>
</types>
<message name="fooRequest">
  <part name="book" type="mh:Book" />
</message>
<message name="fooResponse">
  <part name="book" type="mh:Book" />
</message>
<portType name="FooBar">
```

```
  <operation name="foo">
    <input message="mh:fooRequest"/>
    <output message="mh:fooResponse"/>
  </operation>
</portType>
```

The endpoint interface will be:

```
public interface FooBar extends java.rmi.Remote {
  public void foo(BookHolder book) throws java.rmi.RemoteException;
}
```

And the generated custom book holder class will be:

```
public class BookHolder implements javax.xml.rpc.holders.Holder {
  public Book value;

  public BookHolder(){
    // no-arg contructor is required.
  }
  public BookHolder(Book myBook) {
    value = myBook;
  }
}
```

The class name of a generated holder will be the type of the generated bean followed by the word "Holder." **Aggregated types do not need holders. The holder class is needed only for the containing type**. For example, if the `Book` Java bean refers to another Java bean type, `Author`, the JAX-RPC compiler will not generate an `AuthorHolder` class, only a `BookHolder` class, because `Book` is the containing type.

In addition to generating holders for complex types, the **JAX-RPC compiler will also generate holders for arrays that are used as INOUT or OUT parameters**.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="FooWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/Foo"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
  <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/Foo">
    <xsd:complexType name="StringArray">
     <xsd:sequence>
       <xsd:element name="item" maxOccurs="unbounded"/>
     </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
</types>
<message name="fooRequest">
  <part name="strings" type="mh:StringArray" />
</message>
<message name="fooResponse">
  <part name="strings" type="mh:StringArray" />
</message>
<portType name="FooBar">
  <operation name="foo">
    <input message="mh:fooRequest"/>
    <output message="mh:fooResponse"/>
  </operation>
</portType>
```

The endpoint interface will be:

```
public interface FooBar extends java.rmi.Remote {
    public void foo(StringArrayHolder strings)
    throws java.rmi.RemoteException;
}
```

The generated custom holder class for the array type will be:

```
public final class StringArrayHolder implements javax.xml.rpc.holders.Holder
{
  public java.lang.String[] value;
```

```
  public StringArrayHolder(){
    // no-arg contructor is required.
  }
  public StringArrayHolder(java.lang.String[] myStrings) {
    this.value = value;
  }
}
```

Faults and Java Exceptions

All endpoint interface methods must declare the `java.rmi.RemoteException` type in their `throws` clause. The stub uses the `RemoteException` to report network communication errors, including TCP/IP communication problems, and problems with higher-level protocols like HTTP (for example, the familiar `404 Not Found` error). The `RemoteException` is also used for standard SOAP faults: `Server`, `Client`, and so on.

In addition to the mandatory `RemoteException` type, **endpoint methods may declare application-specific exceptions that map to fault messages defined in the WSDL document**.

```
<message name="InvalidIsbnFault">
  <part name="InvalidIsbn" type="xsd:string" />
</message>
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input message="mh:getBookPriceRequest"/>
    <output message="mh:getBookPriceResponse"/>
    <fault name="InvalidIsbnFault" message="mh:InvalidIsbnFault"/>
  </operation>
</portType>
```

The message name, `"InvalidIsbnFault"`, is mapped to an exception class of the same name, which extends `java.lang.Exception`. A single accessor is defined for the `InvalidIsbnFault` exception class, `getInvalidIsbn()`—this method is based on the fault message's `part` definition.

```
public class InvalidIsbnFault extends java.lang.Exception {
  private String invalidIsbn;
  public InvalidIsbnFault(String invalidIsbn){
    super();
    this.invalidIsbn = invalidIsbn;
  }
  public String getInvalidIsbn(){
    return invalidIsbn;
  }
}
```

The endpoint interface will be:

```
public interface BookQuote extends java.rmi.Remote {
  public float getBookPrice(String isbn)
   throws java.rmi.RemoteException, InvalidIsbnFault;
}
```

Although **a fault message may have only a single `part`**, that part may refer to an XML schema complex type, which allows for a richer set of data to be associated with the exception.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
  <xsd:schema
    targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote">
    <xsd:complexType name="InvalidIsbnType">
       <xsd:sequence>
         <xsd:element name="offending-value" type="xsd:string"/>
         <xsd:element name="conformance-rules" type="xsd:string"/>
       </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
```

```
</types>
<message name="getBookPriceRequest">
  <part name="isbn" type="xsd:string"/>
</message>
<message name="getBookPriceResponse">
  <part name="result" type="xsd:float"/>
</message>
<message name="getBookPriceFault">
  <part name="fault" type="mh:InvalidIsbnType" />
</message>
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input message="mh:getBookPriceRequest"/>
    <output message="mh:getBookPriceResponse"/>
    <fault name="fault" message="mh:getBookPriceFault"/>
  </operation>
</portType>
```

When a JAX-RPC compiler generates an application exception class from a fault message that's based on an XML schema complex type, it will derive the exception's name from the name of the complex type, rather than from the fault message name.

```
public class InvalidIsbnType extends java.lang.Exception{
  private java.lang.String offendingValue;
  private java.lang.String conformanceRules;

  public InvalidIsbnType() {
    // no-arg constructor is required.
  }
  public InvalidIsbnType(String offendingValue,String conformanceRules){
    this.offendingValue = offendingValue;
    this.conformanceRules = conformanceRules;
  }
  public java.lang.String getOffendingValue() {
    return offendingValue;
  }
  public java.lang.String getConformanceRules() {
    return conformanceRules;
  }
}
```

And here's the endpoint interface:

```
public interface BookQuote extends java.rmi.Remote {
    public float getBookPrice(java.lang.String isbn)
    throws java.rmi.RemoteException, InvalidIsbnType;
}
```

If you use inheritance by extension, the JAX-RPC compiler will generate exception classes that represent both the base type and the extension type. [For example, refer book pg 511].

SOAP and XML Processing APIs

5.1 Describe the functions and capabilities of the APIs included within JAXP.
5.2 Given a scenario, select the proper mechanism for parsing and processing the information in an XML document.
5.3 Describe the functions and capabilities of JAXB, including the JAXB process flow, such as XML-to-Java and Java-to-XML, and the binding and validation mechanisms provided by JAXB.
5.4 Use the SAAJ APIs to create and manipulate a SOAP message.

JAXP 1.2 (5.1 and 5.2)

[RMH - Chapters 20 and 21 – SAX and DOM + Blueprints – Chapter 4]. TrAX is not covered in RMH.
JAXP 1.2 = SAX 2 + DOM 2.

SAX 2

The **Simple API for XML** (SAX) uses an event-based architecture for processing XML documents. To use SAX, you create a class that implements one of the SAX listener interfaces (there are three of them), and

register an instance of that class with a SAX parser at runtime. A SAX parser reads an XML document from a data stream sequentially, from beginning to end. As it reads the stream, the SAX parser sends events to the listener object that you registered. In other words, the parser invokes callback methods on the listener object as it encounters different parts of the XML document. For example, at the start of the XML document the parser invokes the listener's **`startDocument`**`()` method, when it reads the start tag of an element it invokes the listener's **`startElement`**`()` method, and when it reads the end tag of an element it invokes the listener's **`endElement`**`()` method.

Parsing with SAX: XMLReaderFactory and XMLReader

```
package com.rwatsh.jaxp.sax;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
import java.io.IOException;

public class SAXExample_1 {

    /** Creates a new instance of SAXExample_1 */
    public SAXExample_1() {}

    public static void main(String[] args) throws SAXException, IOException {
        String    fileName = "./xml/BookQuote.xml";
        System.setProperty("org.xml.sax.driver", "org.apache.xerces.parsers.SAXParser");
        XMLReader parser   = XMLReaderFactory.createXMLReader();
       ContentHandler contentHandler = new SimpleHandler();
        parser.setContentHandler( contentHandler);
        parser.setFeature("http://xml.org/sax/features/namespace-prefixes", true);
        parser.setFeature(
                "http://apache.org/xml/features/validation/schema-full-checking",true);
        parser.parse(fileName);
        System.out.println("No problems parsing the document");
    }
}

A Sample xml document ( a soap message) is:
<?xml version="1.0" encoding="UTF-8"?>

<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
          <isbn xsi:type="xsd:string">0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

You may need to set the `org.xml.sax.driver` system property, which tells JAXP exactly which SAX2 implementation you're using in your application. If you don't set the above property then the default is to use crimson parser's SAXParser class (**javax.xml.parsers.SAXParser**'s concrete implementation class which is created using the SAXParserFactory defined by system property **javax.xml.parsers.SAXParserFactory**) which comes bundled with the JDK (and hence is in the class path). In case you want to use Xerces_J (I used version 2.8.0) then you must add the Xerces_Impl.jar to your class path and also set the **org.xml.sax.driver** system property (as in the example above). The default behavior of the **createXMLReader**() method is to create an instance of SAXParser class from a SAX parser implementation found in the classpath in case the system property identifying a SAX parser implementation is not set. Note: Crimson does not support XML Schema validation but Xerces_J does.

The ContentHandler and DefaultHandler interfaces

The primary listener interface in SAX2 is the **`ContentHandler`**, which defines 11 methods, each of which corresponds to some part of an XML document, such as an element, namespace declaration,

document location, piece of text, white space, processing instruction, or DTD (Document Type Definition) entity.

```
package org.xml.sax;

public interface ContentHandler {

  public void startDocument() throws SAXException;
  public void endDocument() throws SAXException;
  public void setDocumentLocator(Locator locator);
  public void startElement(String namespaceURI, String localName,
                           String qName, Attributes atts) throws SAXException;
  public void endElement(String namespaceURI, String localName, String qName)
    throws SAXException;
  public void startPrefixMapping(String prefix, String uri) throws SAXException;
  public void endPrefixMapping(String prefix) throws SAXException;
  public void characters(char[] text, int start, int length) throws SAXException;
  public void ignorableWhitespace(char[] text, int start,
    int length) throws SAXException;
  public void processingInstruction(String target, String data) throws SAXException;
  public void skippedEntity(String name) throws SAXException;
}
```

In most cases, you'll need to implement only a subset of these methods. To make life easier, SAX2 provides an adapter class, **`org.xml.sax.helpers.DefaultHandler`**, which implements ContentHandler with empty methods. You can extend DefaultHandler, overriding only the event methods you want to implement, and ignore the rest.

Because the BP doesn't support processing instructions and DTD entities, events for these constructs will not be covered.

The **`namespace-prefixes`** property, ensures that the parser will report XML namespace assignments and prefixes—this property may not be turned on by default.

The parser invokes the **`startElement`**`()` and **`endElement`**`()` methods when it encounters the start and end tags (`<tag-name>` and `</tag-name>`) that mark the boundaries of elements in an XML document, in the order it finds them in the input stream. When the parser encounters an empty element (`<tag-name/>`), it invokes both methods, in sequence.

- **namespaceURI** – is the URI of the namespace to which the element belongs. For example, the namespace of the `soap:Envelope` element is `"http://schemas.xmlsoap.org/soap/envelope/"`.
- **localName** –is the name of the elment without the prefix. For example, BookQuote.xml has four elements, whose `localName` values are `Envelope`, `Body`, `getBookPrice`, and `isbn`.
- **qName** –is the fully qualified name of the element—that is, the prefix plus the `localName`. Examples from 20–2 include `soap:Envelope`, `soap:Body`, `mh:getBookPrice`, and `isbn`.
- **atts** - contains the attributes declared for that element.

```
package org.xml.sax;

public interface Attributes {
  public int    getLength ();
  public int    getIndex(String uri, String localPart);
  public int    getIndex(String qName);
  public String getLocalName(int index);
  public String getQName(int index);
  public String getURI(int index);
  public String getValue(String uri, String localName);
  public String getValue(String qName);
  public String getValue(int index);

  public String getType(String uri, String localName);
  public String getType(String qName);
  public String getType(int index);
}
```


The `Attributes` interface allows you to access an element's attributes by index (position), qualified name, or XML namespace and local name. The order of the attributes in the index is arbitrary; it may or may not be the same as the order in which the attributes were declared in the XML document.

```
import org.xml.sax.Attributes;

public class AttributeHandler extends org.xml.sax.helpers.DefaultHandler {
  public void startElement(String namespaceURI, String localName,
                           String qName, Attributes atts) {
    int length = atts.getLength();
    if(length > 0){
      System.out.println("Element <"+qName+"> has the following attributes");
      for(int i = 0; i < length; i++){
        System.out.println("   "+atts.getQName(i)+" = "+atts.getValue(i));
      }
    }
  }
}
```

Although the XML namespace of an element is available in the `namespaceURI` parameter of the `startElement()` and `endElement()` methods, and the XML namespace of an attribute is available in the `Attributes` parameter of `startElement()`, the XML namespaces of attribute and element values are not provided by these callback methods. Instead you must use the **`startPrefixMapping`**`()` and **`endPrefixMapping`**`()` methods to determine the XML namespace of a particular value. When processing Web service documents such as SOAP messages and WSDL documents, it's very common to use XML namespaces for typing values. For example, the `xsi:type` attribute usually refers to a qualified value, such as `"xsd:string"`. It's important to remember that the parser invokes `startPrefixMapping()`just before it invokes `startElement()` for each start tag, and that it invokes `endPrefixMapping()` immediately after it calls `endElement()` for any end tag.

```
import org.xml.sax.Attributes;
import java.util.HashMap;

public class PrefixHandler extends org.xml.sax.helpers.DefaultHandler {
  String XSI_NS = "http://www.w3.org/2001/XMLSchema-instance";
  HashMap prefixMap = new HashMap();

  public void startPrefixMapping(String prefix, String uri){
     prefixMap.put(prefix,uri);
  }
  public void startElement(String namespaceURI, String localName,
                           String qName, Attributes atts) {
     int length = atts.getLength();
     String attValue = atts.getValue(XSI_NS, "type");
     if(attValue != null){
       String prefix = attValue.substring(0, attValue.indexOf(":"));
       String uri = (String)prefixMap.get(prefix.trim());
       if(uri!=null){
          System.out.println("xsi:type = '"+attValue+
          "' xmlns:"+prefix+"='"+uri+"'");
       }
     }
  }
  public void endPrefixMapping(String prefix){
    prefixMap.remove(prefix);
  }
}
```

The `startPrefixMapping()` method in the `PrefixHandler` stores XML namespace declarations in a `HashMap`. The `startElement()` method checks every start tag to see if it includes declaration of an `xsi:type` attribute. If it does, then `startElement()` extracts the value of the attribute and maps its prefix to one of the namespaces in the `HashMap` cache.

In general, the parser invokes the **`characters`**`()` method when it encounters an element that contains text. This method may be called once, in which case it passes all the text contained by the element, or more than once, in which case it delivers the text in two or more blocks. The latter case is odd, and a nuisance,

but it does arise sometimes, as a result of parser limitations. For example, the Crimson parser has a limit of about 8K of text. If the text contained by an element is longer than 8K, the Crimson parser will invoke `characters()` on the content handler multiple times, until all the text is processed.

```
import org.xml.sax.Attributes;
import java.util.Stack;

public class CharactersHandler extends org.xml.sax.helpers.DefaultHandler {
  Stack stack = new Stack();

  public void startElement(String namespaceURI, String localName,
                           String qName, Attributes atts){
    StringBuffer buffy = new StringBuffer();
    stack.push(buffy);

  }
  public void characters(char[] chars, int start, int length){
     StringBuffer buffy = (StringBuffer)stack.peek();
     buffy.append(chars,start,length);
  }
  public void endElement(String namespaceURI, String localName,
                           String qName){
    StringBuffer buffy = (StringBuffer)stack.pop();

    System.out.println("The <"+qName+"> contains the following text:"+
                       "\""+buffy+"\"");
  }
}
```

Each time `CharactersHandler` is notified of a new start tag, `startElement()` pushes a new `StringBuffer` onto a `Stack` variable. If the `characters()` method is called, it will be for the element whose start tag was just processed by `startElement()`, so the `StringBuffer` for that element will be at the top of the stack. The `characters()` method peeks at the `StringBuffer` at the top of the stack and appends text to it. When `endElement()` is called, the `StringBuffer` for that element is popped off the stack and its contents are printed to the screen.

Other ContentHandler Methods:

- **setDocumentLocator** – is called before `startDocument()`. It provides the content handler with a reference to a `Locator` object, which any method can use to determine the exact position of the parser in the document, measured in terms of rows and columns. This object can be useful for debugging code or reporting errors.
- **ignorableWhiteSpace** – is called when the parser encounters white space between elements in the document. It's used in advanced scenarios where white space needs to be managed.
- **processingInstruction** –is called when the parser encounters a processing instruction in the XML document. Because the Basic Profile prohibits the use of processing instructions, you are unlikely to use this callback method in your Web services.
- **skippedEntity** – is invoked by the parser when it encounters "skipped entities." Because entities are related to DTDs, which are not used in BP-conformant Web services, we'll skip them too.

Other Listener interfaces:

You use the `org.xml.sax.`**`DTDHandler`** and `org.xml.sax.`**`EntityResolver`** interfaces when you are parsing documents that are associated with a DTD.

SAX uses the `org.xml.sax.`**`ErrorHandler`** to notify the application that an error occurred while parsing the XML document.

```
package org.xml.sax;

public interface ErrorHandler {
  public void error(SAXParseException exception)  throws SAXException;

  public void fatalError(SAXParseException exception)  throws SAXException;
```



```
  public void warning(SAXParseException exception) throws SAXException;
}
```

Validating with W3C XML Schema

The SAX2 programming model is designed for validation using DTDs, but not using XML schemas. SAX2 is simple and adaptable, though, so many parsers provide **custom configuration options** that allow you to validate an XML document against a W3C XML schema.

`setFeature``()` activates or deactivates a specific piece of functionality, in this case validation by XML schema. The **`setProperty`**`()` call configures some aspect of a feature. tells the parser to use specific files for XML schema validation. When you use the `external-schemaLocation` property, the parser will ignore `xsi:schemaLocation` attributes in favor of the XML schemas identified in the `setProperty()` call. Because a SOAP message doesn't usually include an `xsi:schemaLocation` attribute, using an `external-schemaLocation` property when validating SOAP messages makes sense—but validation of SOAP messages is not necessary in many cases when you're using JAX-RPC, because the marshalling from XML to JavaBeans components will detect many validity errors.

```
import org.xml.sax.ContentHandler;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import java.io.IOException;

public class JaxpExample_4  {

  public static void main(String [] args)
    throws SAXException, IOException, Exception{
    String contentHanderClassName = args[0];
    String fileName = args[1];

    ContentHandler contentHandler = (ContentHandler)
     Class.forName(contentHanderClassName).newInstance();

    XMLReader parser = XMLReaderFactory.createXMLReader();
    parser.setContentHandler( contentHandler);
    parser.setFeature("http://xml.org/sax/features/namespace-prefixes", true);

    // Xerces-specific features and properties
    parser.setFeature(
     "http://apache.org/xml/features/validation/schema",true);

    parser.setProperty(
     "http://apache.org/xml/properties/schema/external-schemaLocation",
     "http://schemas.xmlsoap.org/soap/envelope/ SOAP-1_1.xsd "+
     "http://www.Monson-Haefel.com/jwsbook/BookQuote bookquote.xsd");

    parser.parse(fileName);
  }
}
```

DOM 2

Document Object Model (DOM) is a set of interfaces and classes used to model XML documents as a tree of objects called **nodes**. The DOM interfaces and classes represent XML documents, elements, attributes, text values, and pretty much all the other constructs from the XML 1.0 language. You use the DOM programming API to examine existing XML documents, or to create new ones. When an implementation of DOM parses an XML document, it reads the XML text from some source (a file, network stream, or database, for example), then builds an object graph, called a tree, that mirrors the structure of the XML document.

Every element in the XML document is represented by a corresponding instance of the DOM **`Element`** type. Each of the other XML artifacts has a corresponding DOM type; attributes map to the type **`Attr`**, comments to `Comment`, text to **`Text`**, CDATA sections to **`CDATASection`**, and so on. Their common supertype is the **`org.w3c.dom.Node`** interface. When a DOM parser processes an XML document successfully, it returns an **`org.w3c.dom.Document`** object, which represents an XML document instance. The `Document` object provides access to the root **`Element`**, which in turn provides access to all the `Elements`, **`Text`**, **`Comments`** and other parts of the XML document.

Parsing with DOM: DocumentBuilderFactory and DocumentBuilder

Although DOM 2 defines a standard object model for XML documents, it doesn't define a standard way of instantiating a parser and parsing a document, which could affect the portability of your code. If you are using the Apache DOM parser (Xerces) and decide to switch to the Sun parser (Crimson), you'll have to change the code that instantiates the parser and supplies it an XML document. JAXP largely eliminates this portability problem by providing a single programming API for creating parsers and supplying them XML documents. JAXP defines two classes that are used to bootstrap a DOM 2 parser: `javax.xml.parsers.`**`DocumentBuilderFactory`** and `javax.xml.parsers.`**`DocumentBuilder`**. These two types provide vendor-agnostic abstractions for accessing DOM parsers and supplying them with XML documents to be parsed.

```
package com.rwatsh.jaxp.dom;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import java.io.FileInputStream;
//import com.jwsbook.jaxp.DomOutputter;

public class DomTest {
    public static void main(String[] args) throws Exception {

        String filePath = "./xml/po.xml";
        FileInputStream fileStream = new FileInputStream(filePath);

        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder parser = factory.newDocumentBuilder();
        Document xmlDoc = parser.parse(fileStream);

        //DomOutputter.write(xmlDoc, System.out);
    }
}
```

`DocumentBuilderFactory` is a factory for creating DOM parsers. You instantiate the factory class by calling its `newInstance()` method, then use the factory instance to create an instance of `DocumentBuilder`, which represents a DOM parser. When you invoke the `DocumentBuilderFactory.newInstance()` method, the JAXP runtime will follow a predefined routine for finding the proper parser. Basically, it will first check environment properties, then the Java runtime directory, then all the JAR files in the classpath. If it doesn't find a parser in any of these locations, it instantiates the default parser provided by the JAXP vendor.

Nodes

```
package org.w3c.dom;

public interface Node {
  // Constants
  public static final short ATTRIBUTE_NODE;
  public static final short CDATA_SECTION_NODE;
  public static final short COMMENT_NODE;
  public static final short DOCUMENT_FRAGMENT_NODE;
  public static final short DOCUMENT_NODE;
  public static final short DOCUMENT_TYPE_NODE;
  public static final short ELEMENT_NODE;
  public static final short ENTITY_NODE;
  public static final short ENTITY_REFERENCE_NODE;
  public static final short NOTATION_NODE;
```



```
  public static final short PROCESSING_INSTRUCTION_NODE;
  public static final short TEXT_NODE;

  // Type-dependent properties
  public short    getNodeType();
  public String   getNodeName();
  public String   getNodeValue() throws DOMException;
  public void     setNodeValue(String nodeValue) throws DOMException;

  // The XML names of this node
  public String   getLocalName();
  public String   getNamespaceURI();
  public String   getPrefix();
  public void     setPrefix(String prefix) throws DOMException;

  // The attributes of this node
  public boolean      hasAttributes();
  public NamedNodeMap getAttributes();

  // The document to which this node belongs
  public Document     getOwnerDocument();

  // The parent, children, and sibling navigation methods
  public Node         getParentNode();
  public boolean      hasChildNodes();
  public Node         getFirstChild();
  public Node         getLastChild();
  public NodeList     getChildNodes();
  public Node         getPreviousSibling();
  public Node         getNextSibling();

  // The child-management methods
  public Node appendChild(Node newChild) throws DOMException;
  public Node replaceChild(Node newChild,  Node oldChild) throws DOMException;
  public Node removeChild(Node oldChild) throws DOMException;
  public Node insertBefore(Node newChild, Node refChild) throws DOMException;

  // Other methods
  public Node cloneNode(boolean deep);
  public void normalize();
  public boolean isSupported(String feature, String version);
}
```

`Node` defines four methods that behave very differently depending on the type of `Node` being accessed: `getNodeType()`, `getNodeName()`, `getNodeValue()`, and `setNodeValue()`.

```
package com.rwatsh.jaxp.dom;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import java.io.FileInputStream;
import java.io.PrintWriter;

public class DomTest {
    private static final String DEFAULT_ELEMENT_NAME = "*";
    public static void main(String[] args) throws Exception {
        String filePath = "./xml/po.xml";
        String elementName = DEFAULT_ELEMENT_NAME;
        String attributeName = null;
        PrintWriter out = new PrintWriter(System.out);
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder parser = factory.newDocumentBuilder();
        Document xmlDoc = parser.parse(filePath);
        print(out, xmlDoc, elementName);
    }

    /** Prints the specified elements in the given document. */
```



```
    public static void print(PrintWriter out, Document document, String elementName) {

        // get elements that match
        NodeList elements = document.getElementsByTagName(elementName);

        // is there anything to do?
        if (elements == null) {
            return;
        }

        // print all elements - this is the part which executes as we don't pass
        // an attributeName in the main().

        int elementCount = elements.getLength();
        for (int i = 0; i < elementCount; i++) {
            Element element = (Element)elements.item(i);
            print(out, element, element.getAttributes());
        }
        return;
    } // print(PrintWriter,Document,String,String)

    /** Prints the specified element and its attributes (name/value pair). */
    protected static void print(PrintWriter out,
            Element element, NamedNodeMap attributes) {

        out.print('<');
        out.print(element.getNodeName());
        if (attributes != null) {
            int attributeCount = attributes.getLength();
            for (int i = 0; i < attributeCount; i++) {
                Attr attribute = (Attr)attributes.item(i);
                out.print(' ');
                out.print(attribute.getNodeName());
                out.print("=\"");
                out.print(attribute.getNodeValue()); // not shown is you may need to
                                                     // normalize the value as it may
                                                    // contain special chars
                                                    // refer to sample code.
                out.print('"');
            }
        }
        out.println('>');
        out.flush();

    } // print(PrintWriter,Element,NamedNodeMap)
```

The above code prints all the elements, attributes and the attribute values of the xml document. The code is excerpted from the GetElementsByTagName.java sample source code provided with Xerces_J 2.8.0. For iterating all elements in the document, we need to pass * for elementName pattern to the getElementByTagName() method which returns a NodeList.

Here's some more example:

```
package com.rwatsh.jaxp.dom;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.Attr;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;

public class DomTest {

    /** Creates a new instance of DomTest */
    public DomTest() {
    }
    public static void main(String[] args) throws Exception {
        String filePath = "./xml/po.xml";
        // Create an instance of a DOM parser and get the Document object.
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
```

```
        DocumentBuilder parser = factory.newDocumentBuilder();
        Document xmlAddressDoc = parser.parse(filePath);

        // Obtain the root element of the document and output its names.
        Element rootElement = xmlAddressDoc.getDocumentElement();
        printElementNames(rootElement);
        System.out.println("********");
        printAttributes(rootElement);
    }
    // Print out the local name, QName, and namespace URI of an element.
    public static void printElementNames(Node element){
        String prefix = element.getPrefix();
        if(prefix!=null){
            System.out.print(prefix+":");
        }
        String localName = element.getLocalName();
        System.out.print(localName);
        String uri = element.getNamespaceURI();
        if(uri!=null){
            System.out.print(" xmlns:"+prefix+"="+uri);
        }
        System.out.println();
        // Output the names of elements contained by the root element.
        if(element.getNodeType() == Node.ELEMENT_NODE){
            NodeList nodeList = element.getChildNodes();
            for(int i=0;i < nodeList.getLength();i++){
                Node childNode = nodeList.item(i);
                printElementNames(childNode);
            }
        }
    }
    // Print out the attributes of a node. This method is recursive.
    public static void printAttributes(Node node){
        if(node.hasAttributes()){
            NamedNodeMap attrMap = node.getAttributes();
            for(int i=0;i<attrMap.getLength();i++){
                Attr attribute = (Attr)attrMap.item(i);
                System.out.println(attribute.getName()+"="+attribute.getValue());
            }
        }
        if(node.getNodeType() == Node.ELEMENT_NODE){
            NodeList nodeList = node.getChildNodes();
            for(int i=0;i < nodeList.getLength();i++){
                Node childNode = nodeList.item(i);
                printAttributes(childNode);
            }
        }
    }
}
```

The **`getOwnerDocument`**`()` method returns a reference to the `Document` object to which the `Node` belongs. It's important to understand that `Nodes` are owned by specific `Documents`, and that you cannot take a `Node` from one `Document` and add it to another—even if you remove it from the source `Document`. You get the following exception if you move the nodes across documents:

```
Exception in thread "main" org.w3c.dom.DOMException: WRONG_DOCUMENT_ERR: A node is used
in a different document than the one that created it.
```

Although you cannot move `Nodes` between `Documents`, `Document.importNode()` method allows you to create a copy of a `Node`, and then assign to a different `Document`.

A `Node` may contain other `Nodes` (children), and it may be contained by some other `Node` (a parent). You can navigate up and down a DOM tree using `Node`'s navigation methods.

```
// The parent, children, and sibling navigation methods
public Node          getParentNode();
public boolean       hasChildNodes();
public Node          getFirstChild();
public Node          getLastChild();
public NodeList      getChildNodes();
public Node          getPreviousSibling();
```

```
public Node         getNextSibling();
```

getParent() - Most Nodes have a parent. Only Documents do not. The parent of an Element may be another Element or, in the case of the root element, a Document. The parent of an Attr is always an Element, as are the parents of Text, Comment, and CDATASection nodes.

getPreviousSibling or **getNextSibling** - Even if the parent Element does not contain mixed content, it may contain multiple Text nodes, because some parsers distribute a single logical string of text into several sibling Text nodes—for example, when its size exceeds some limit. The siblings of Attr type Nodes are always other Attrs. The Document and the root Element node never have siblings.

Document and Element types may contain children. The child of a Document is always an Element, which you can reach with **getDocumentElement()**. The children of an Element may be other Elements, Text, CDATASections, or Comments. You can reach the children of an Element using **getChildNodes()**, **getLastChild()**, and **getFirstChild()**.

The **Node** interface defines a set of methods for adding, removing, and replacing nodes in a DOM tree.
The **cloneNode()** method, as its name suggests, makes a copy of the Node on which the method is invoked. The copy may be shallow or deep, depending on the value of the boolean parameter. A deep copy will make an exact copy of the Node, its attributes (if it's an Element), and all of its children. The only thing that isn't copied is the reference to the parent Node. The clone is associated with the same Document object as the original, but it's not assigned as a child to any other Node on the tree.

The **normalize()** method removes all of the superfluous white space (spaces, tabs, carriage returns, and so on) from the node, reducing it to a leaner yet semantically equivalent version of itself. Invoking the normalize() method should consolidate adjacent Text nodes and delete superfluous white space. This method is especially helpful when using DOM with XPointer operations, which depend on succinct tree structures to operate properly.

The **isSupported()** method tests to see whether a Node supports a specific parsing feature, such as XML schema, DTDs, or XML namespaces.

Building a DOM Document

Using JAXP, it's fairly easy to construct an in-memory DOM representation of an XML document. For example, you can use DOM to construct a simple SOAP 1.1 message, complete with elements, namespaces, and attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
          <isbn xsi:type="xsd:string">0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

Here's how to build the above document:

```
package com.rwatsh.jaxp.dom;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.Text;
import org.w3c.dom.CDATASection;
import org.w3c.dom.Comment;
```



```
import java.io.*;

public class DomDocumentBuilder {
  // XML namespace constants
  public final static String SOAP_NS =
   "http://schemas.xmlsoap.org/soap/envelope/";
  public final static String MH_NS =
   "http://www.Monson-Haefel.com/jwsbook/BookQuote";
  public final static String XSD_NS =
   "http://www.w3c.org/2001/XMLSchema";
  public final static String XSI_NS =
   "http://www.w3c.org/2001/XMLSchema-instance";

  public static void main(String[] args) throws Exception {

    // Create a new Document object representing a SOAP Envelope.
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    DocumentBuilder builder = factory.newDocumentBuilder();
    DOMImplementation domImpl = builder.getDOMImplementation();
    Document xmlDoc = domImpl.createDocument(
      SOAP_NS,"soap:Envelope",null);

    // Add namespace declarations to the root element.
    Element root = xmlDoc.getDocumentElement();
    root.setAttribute("xmlns:soap",SOAP_NS);
    root.setAttribute("xmlns:mh",MH_NS);
    root.setAttribute("xmlns:xsd", XSD_NS);
    root.setAttribute("xmlns:xsi", XSI_NS);

    // Add the Body element.
    Element body = xmlDoc.createElementNS(SOAP_NS,"soap:Body");
    root.appendChild(body);

    // Add the getBookPrice and isbn elements.
    Element getBookPrice = xmlDoc.createElementNS(
     MH_NS, "mh:getBookPrice");
    body.appendChild(getBookPrice);

    Element isbn = xmlDoc.createElementNS(MH_NS, "isbn");
    body.appendChild(isbn);

    // Add the xsi:type attribute to the isbn element.
    Attr typeAttr = xmlDoc.createAttributeNS(XSI_NS, "xsi:type");
    typeAttr.setValue("xsd:string");
    isbn.setAttributeNodeNS(typeAttr);

    // Add the text contained by the isbn element.
    Text text = xmlDoc.createTextNode("0321146182");
    body.appendChild(text);

    //Write to stream
    DomWriter writer = new DomWriter();
    File f = new File("./soap_msg.xml");
    FileWriter fw = new java.io.FileWriter(f);
    writer.setOutput(fw);
    writer.write(xmlDoc);

    // Create a new Document object with a root element named example.
    Document xmlDoc2 = domImpl.createDocument(null,"example",null);
    Element root2 = xmlDoc2.getDocumentElement();

    // Add a label element to the root.
    Element label = xmlDoc2.createElement("label");
    Text text2 = xmlDoc2.createTextNode("Listing 21-9");
    label.appendChild(text2);
    root2.appendChild(label);

    // Create a code element.
    Element code = xmlDoc2.createElement("code");
```

```
    // Read an XML file and cache it in a String.
    FileReader fis = new FileReader("./soap_msg.xml");
    StringWriter sw = new StringWriter();
    for(int character = fis.read(); character!=-1; character = fis.read()){
      sw.write(character);
    }
    String soapMsg = sw.toString();

    // Create a new CDATASection with the XML document as its content.
    CDATASection cdata = xmlDoc2.createCDATASection(soapMsg);
    code.appendChild(cdata);
    root2.appendChild(code);

    // Add a comment to the Document, before the code element.
    Comment comment = xmlDoc2.createComment(
     "This is a SOAP message inside a CDATA section");
    root2.insertBefore(comment,code);
    writer.setOutput(System.out, null);
    writer.write(xmlDoc2);
  }
}
```

The **DomWriter** class is an utility class which is used to print the DOM document to a stream. This code has been excerpted from Xerces_J 2.8.0 samples program Writer.java. It has been trimmed to not require command line args and the support for cannonical and XML 1.1 version documents as BP only allows XML 1.0 to be used .

```
package com.rwatsh.jaxp.dom;

import java.io.*;
import org.w3c.dom.*;
import java.lang.reflect.Method;

public class DomWriter {
    /** Print writer. */
    protected PrintWriter fOut;

    /** Creates a new instance of DomWriter */
    public DomWriter() {
    }
    //
    // Public methods
    //

    /** Sets the output stream for printing. */
    public void setOutput(OutputStream stream, String encoding)
        throws UnsupportedEncodingException {

        if (encoding == null) {
            encoding = "UTF8";
        }

        java.io.Writer writer = new OutputStreamWriter(stream, encoding);
        fOut = new PrintWriter(writer);

    } // setOutput(OutputStream,String)

    /** Sets the output writer. */
    public void setOutput(java.io.Writer writer) {

        fOut = writer instanceof PrintWriter
             ? (PrintWriter)writer : new PrintWriter(writer);

    } // setOutput(java.io.Writer)

    /** Writes the specified node, recursively. */
    public void write(Node node) {

        // is there anything to do?
        if (node == null) {
            return;
```

```
        }

        short type = node.getNodeType();
        switch (type) {
            case Node.DOCUMENT_NODE: {
                Document document = (Document)node;

                fOut.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
                fOut.flush();
                write(document.getDoctype());

                write(document.getDocumentElement());
                break;
            }

            case Node.DOCUMENT_TYPE_NODE: {
                DocumentType doctype = (DocumentType)node;
                fOut.print("<!DOCTYPE ");
                fOut.print(doctype.getName());
                String publicId = doctype.getPublicId();
                String systemId = doctype.getSystemId();
                if (publicId != null) {
                    fOut.print(" PUBLIC '");
                    fOut.print(publicId);
                    fOut.print("' '");
                    fOut.print(systemId);
                    fOut.print('\'');
                }
                else if (systemId != null) {
                    fOut.print(" SYSTEM '");
                    fOut.print(systemId);
                    fOut.print('\'');
                }
                String internalSubset = doctype.getInternalSubset();
                if (internalSubset != null) {
                    fOut.println(" [");
                    fOut.print(internalSubset);
                    fOut.print(']');
                }
                fOut.println('>');
                break;
            }

            case Node.ELEMENT_NODE: {
                fOut.print('<');
                fOut.print(node.getNodeName());
                Attr attrs[] = sortAttributes(node.getAttributes());
                for (int i = 0; i < attrs.length; i++) {
                    Attr attr = attrs[i];
                    fOut.print(' ');
                    fOut.print(attr.getNodeName());
                    fOut.print("=\"");
                    normalizeAndPrint(attr.getNodeValue(), true);
                    fOut.print('"');
                }
                fOut.print('>');
                fOut.flush();

                Node child = node.getFirstChild();
                while (child != null) {
                    write(child);
                    child = child.getNextSibling();
                }
                break;
            }

            case Node.ENTITY_REFERENCE_NODE: {
                fOut.print('&');
                fOut.print(node.getNodeName());
                fOut.print(';');
                fOut.flush();
```

```
                break;
            }

            case Node.CDATA_SECTION_NODE: {
                fOut.print("<![CDATA[");
                fOut.print(node.getNodeValue());
                fOut.print("]]>");
                fOut.flush();
                break;
            }

            case Node.TEXT_NODE: {
                normalizeAndPrint(node.getNodeValue(), false);
                fOut.flush();
                break;
            }

            case Node.PROCESSING_INSTRUCTION_NODE: {
                fOut.print("<?");
                fOut.print(node.getNodeName());
                String data = node.getNodeValue();
                if (data != null && data.length() > 0) {
                    fOut.print(' ');
                    fOut.print(data);
                }
                fOut.print("?>");
                fOut.flush();
                break;
            }

            case Node.COMMENT_NODE: {
                fOut.print("<!--");
                String comment = node.getNodeValue();
                if (comment != null && comment.length() > 0) {
                    fOut.print(comment);
                }
                fOut.print("-->");
                fOut.flush();
            }
        }

        if (type == Node.ELEMENT_NODE) {
            fOut.print("</");
            fOut.print(node.getNodeName());
            fOut.print('>');
            fOut.flush();
        }

    } // write(Node)

    /** Returns a sorted list of attributes. */
    protected Attr[] sortAttributes(NamedNodeMap attrs) {

        int len = (attrs != null) ? attrs.getLength() : 0;
        Attr array[] = new Attr[len];
        for (int i = 0; i < len; i++) {
            array[i] = (Attr)attrs.item(i);
        }
        for (int i = 0; i < len - 1; i++) {
            String name = array[i].getNodeName();
            int index = i;
            for (int j = i + 1; j < len; j++) {
                String curName = array[j].getNodeName();
                if (curName.compareTo(name) < 0) {
                    name = curName;
                    index = j;
                }
            }
            if (index != i) {
                Attr temp = array[i];
```



```
                array[i] = array[index];
                array[index] = temp;
            }
        }

        return array;

    } // sortAttributes(NamedNodeMap):Attr[]

    //
    // Protected methods
    //

    /** Normalizes and prints the given string. */
    protected void normalizeAndPrint(String s, boolean isAttValue) {

        int len = (s != null) ? s.length() : 0;
        for (int i = 0; i < len; i++) {
            char c = s.charAt(i);
            normalizeAndPrint(c, isAttValue);
        }

    } // normalizeAndPrint(String,boolean)

    /** Normalizes and print the given character. */
    protected void normalizeAndPrint(char c, boolean isAttValue) {

        switch (c) {
            case '<': {
                fOut.print("&lt;");
                break;
            }
            case '>': {
                fOut.print("&gt;");
                break;
            }
            case '&': {
                fOut.print("&amp;");
                break;
            }
            case '"': {
                // A '"' that appears in character data
                // does not need to be escaped.
                if (isAttValue) {
                    fOut.print("&quot;");
                }
                else {
                    fOut.print("\"");
                }
                break;
            }
            case '\r': {
               // If CR is part of the document's content, it
               // must not be printed as a literal otherwise
               // it would be normalized to LF when the document
               // is reparsed.
               fOut.print("&#xD;");
               break;
            }
            case '\n': {
                // default print char
            }
            default: {
                fOut.print(c);
            }
        }
    } // normalizeAndPrint(char,boolean)

    /** Extracts the XML version from the Document. */
    protected String getVersion(Document document) {
        if (document == null) {
```



```
            return null;
        }
        String version = null;
        Method getXMLVersion = null;
        try {
            getXMLVersion = document.getClass().getMethod("getXmlVersion", new
Class[]{});
            // If Document class implements DOM L3, this method will exist.
            if (getXMLVersion != null) {
                version = (String) getXMLVersion.invoke(document, (Object[]) null);
            }
        }
        catch (Exception e) {
            // Either this locator object doesn't have
            // this method, or we're on an old JDK.
        }
        return version;
    } // getVersion(Document)
}
```

In addition to the `Element`, `Attr`, and `Text` types, you can also create and add `Comment`, `CDATASection`, and other `Node` types to a `Document`. The above example demonstrates it and following is the output document:

```
<?xml version="1.0" encoding="UTF-8"?>
<example>
  <label>Listing 21-9</label>
  <!--This is a SOAP message inside a CDATA section-->
  <code><![CDATA[
    <?xml version="1.0" encoding="UTF-8"?>
    <soap:Envelope
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
      <soap:Body>
        <mh:getBookPrice>
          <isbn xsi:type="xsd:string">0321146182</isbn>
        </mh:getBookPrice>
      </soap:Body>
    </soap:Envelope>
    ]]>
  </code>
</example>
```

The `Node` interface defines four methods for adding, removing, and replacing `Nodes` in a `Document`.

```
package org.w3c.dom;

public interface Node {
  ...
  // The child management methods
  public Node appendChild(Node newChild) throws DOMException;
  public Node replaceChild(Node newChild,  Node oldChild)
   throws DOMException;
  public Node removeChild(Node oldChild) throws DOMException;
  public Node insertBefore(Node newChild, Node refChild)
   throws DOMException;
  ...
}
```

It's important to understand that removed and replaced `Nodes` cannot subsequently be assigned to some other `Document` object. They're not visible in the tree, but they're still owned by the same `Document`.

Copying Nodes

In some cases you may want to copy a `Node` from one `Document` to another. This may be necessary if you need to transfer a portion of an XML document from one source to another. For example, you may want to copy the billing address information in an XML Purchase Order document to an XML Invoicing

document. Another reason for copying Nodes is reuse. If you have a Node of any kind that you use over and over, you may want to cache that Node in memory and make copies of it as needed.
In DOM 2, you can never truly move a Node from one Document to another, because DOM 2 specifies that the Document object that manufactures a Node owns that Node. So, if you want to use an existing Node in a different document you have to copy it. More accurately, you have to import the Node into the new Document, by invoking the Document.**importNode**() method.

```
public interface Document extends Node {
  ...
  public Node importNode(Node importedNode, boolean deep)
   throws DOMException;
  ...
}
```

Importing a Node consists of copying the data from an existing Node to a new Node that is associated with the Document on which you're invoking importNode().

```
package com.rwatsh.jaxp.dom;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.xml.sax.SAXException;
import java.io.IOException;


public class DomNodeImport {
  public static void main(String[] args) throws Exception {
    // Get a DOMImplementation object.
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    DOMImplementation domImpl = builder.getDOMImplementation();

    // Create a source document with a sourceElement with some text.
    Document sourceDoc = domImpl.createDocument(null,"source",null);
    Element sourceElement = sourceDoc.createElement("sourceElement");
    sourceElement.appendChild(sourceDoc.createTextNode("This is a test"));
    sourceDoc.getDocumentElement().appendChild(sourceElement);

    // Create the target document and import the sourceElement element.
    Document targetDoc =
      domImpl.createDocument(null,"target", null);
    Element targetElement = (Element)
      targetDoc.importNode(sourceElement,  true);
    targetDoc.getDocumentElement().appendChild(targetElement);

    System.out.println("**SOURCE DOCUMENT**");
    DomWriter writer = new DomWriter();
    writer.setOutput(System.out, null);
    writer.write(sourceDoc);

    System.out.println("\n\n**TARGET DOCUMENT**");
    writer.write(targetDoc);
  }
}

Following is the o/p:
**SOURCE DOCUMENT**
<?xml version="1.0" encoding="UTF-8"?>
<source><sourceElement>This is a test</sourceElement></source>

**TARGET DOCUMENT**
<?xml version="1.0" encoding="UTF-8"?>
<target><sourceElement>This is a test</sourceElement></target>
```

Alternative way to write an XML document to stream is:

```
        // Serialize the document onto System.out
```



```
        TransformerFactory xformFactory
                = TransformerFactory.newInstance();
        Transformer idTransform = xformFactory.newTransformer();
        Source input = new DOMSource(xmlDoc2);
        Result output = new StreamResult(System.out);
        idTransform.transform(input, output);
```

```
You will need to import the following:
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
```

XSLT and TrAX (Transformations API for XML)

[Elliotte Rusty Harold: Processing XML with Java: Chapter 17.2 XSLT - TrAX]

XSLT

Extensible Stylesheet Language Transformations (XSLT) is provably Turing-complete. That is, given enough memory an XSLT stylesheet can perform any calculation a program written in any other language can perform. But, XSLT is designed as a **templating language**.

An XSLT stylesheet describes how documents in one format are converted to documents in another format. Both input and output documents are represented by the XPath data model. XPath expressions select nodes from the input document for further processing. Templates containing XSLT instructions are applied to the selected nodes to generate new nodes that are added to the output document.

When designing an XSLT stylesheet, you concentrate on which input constructs map to which output constructs rather than on how or when the processor reads the input and generates the output. An XSLT stylesheet contains examples of what belongs in the output document, roughly one example for each significantly different construct that exists in the input documents. It also contains instructions telling the XSLT processor how to convert input nodes into the example output nodes. The XSLT processor uses those examples and instructions to convert nodes in the input documents to nodes in the output document.

Examples and instructions are written as *template rules*. Each template rule has a *pattern* and a *template*. The template rule is represented by an `xsl:template` element. The customary prefix `xsl` is bound to the namespace URI `http://www.w3.org/1999/XSL/Transform`. The pattern, a limited form of an XPath expression, is stored in this element's `match` attribute. The contents of the `xsl:template` element form the template. For example, this is a template rule that matches `methodCall` elements and responds with a template consisting of a single `methodResponse` element – **XSL stylesheet**:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="methodCall" xml:space="preserve">
          <methodResponse>
            <params>
              <param>
                <value>
                    <string>
                                <xsl:value-of select="params/param/value/int" />
                    </string>
                 </value>
              </param>
            </params>
          </methodResponse>
        </xsl:template>
</xsl:stylesheet>
```

(You can keep the white space by adding an **`xml:space="preserve"`** attribute to the `xsl:template` element if you want.) A template can also contain XSLT instructions that copy data from the input document to the output document or create new data algorithmically.

Each **xsl:value-of** element has a select attribute whose value contains an XPath expression identifying the object to take the value of.

```
<xsl:value-of select="/methodCall/params/value/int" />
```

This xsl:value-of element multiplies the number in the int element by ten and returns it:

```
<xsl:value-of select="10 * params/param/value/int" />
```

When xsl:value-of is used in a template, the context node is a node matched by the template and for which the template is being instantiated. The template below copies the string-value of the value element in the input document to the string element in the output document.

If the above XSL stylesheet is applied to the following XML document – **INPUT document**:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>calculateFibonacci</methodName>
  <params>
    <param>
      <value><int>10</int></value>
    </param>
  </params>
</methodCall>
```

Then the result will be the following XML document – **OUTPUT document**:

```
<?xml version="1.0" encoding="utf-8"?>
<methodResponse>
  <params>
    <param>
      <value>
        <string>
          10
        </string>
      </value>
    </param>
  </params>
</methodResponse>
```

The output is a combination of literal result data from the stylesheet and information read from the transformed XML document.

The **xsl:choose** instruction selects from multiple alternative templates. It contains one or more xsl:when elements, each of which contains a test attribute and a template. The first xsl:when element whose test attribute evaluates to true is instantiated. The others are ignored. There may also be an optional final xsl:otherwise element whose template is instantiated only if all the xsl:when elements are false.

For example, when an XML-RPC request is well-formed but syntactically incorrect, the server should respond with a fault document. This template tests for a number of possible problems with an XML-RPC request and only processes it if none of the problems arise. Otherwise it emits an error message:

```
<xsl:template match="/">
  <xsl:choose>
    <xsl:when test="not(/methodCall/methodName)">
       Missing methodName
    </xsl:when>
    <xsl:when test="count(/methodCall/methodName) &gt; 1">
      Multiple methodNames
    </xsl:when>
    <xsl:when test="count(/methodCall/params) &gt; 1">
       Multiple params elements
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates select="child::methodCall"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```


As well as matching a node in the input document, a template can be called by name using the **`xsl:call-template`** element. Parameters can be passed to such templates. Templates can even be called recursively.
For example, here's a template named `faultResponse` that generates a complete XML-RPC fault document when invoked:

```
<xsl:template name="faultResponse">
  <xsl:param name="err_code" select="0"/>
  <xsl:param name="err_message">Unspecified Error</xsl:param>
  <methodResponse>
    <fault>
      <value>
        <struct>
          <member>
            <name>faultCode</name>
            <value>
              <int><xsl:value-of select="$err_code"/></int>
            </value>
          </member>
          <member>
            <name>faultString</name>
            <value>
              <string>
                <xsl:value-of select="$err_message"/>
              </string>
            </value>
          </member>
        </struct>
      </value>
    </fault>
  </methodResponse>
</xsl:template>
```

The `xsl:call-template` element applies a named template to the context node.

```
<xsl:template match="/">
  <xsl:choose>
    <xsl:when test="not(/methodCall/methodName)">
      <xsl:call-template name="faultResponse">
        <xsl:with-param name="err_code" select="1" />
        <xsl:with-param name="err_message">
          Missing methodName
        </xsl:with-param>
      </xsl:call-template>
    </xsl:when>
    <xsl:when test="count(/methodCall/methodName) &gt; 1">
      <xsl:call-template name="faultResponse">
        <xsl:with-param name="err_code" select="1" />
        <xsl:with-param name="err_message">
          Multiple method names
        </xsl:with-param>
      </xsl:call-template>
    </xsl:when>
    <xsl:when test="count(/methodCall/params) &gt; 1">
      <xsl:call-template name="faultResponse">
        <xsl:with-param name="err_code" select="2" />
        <xsl:with-param name="err_message">
          Multiple params elements
        </xsl:with-param>
      </xsl:call-template>
    </xsl:when>
  <!-- etc. -->
    <xsl:otherwise>
      <xsl:apply-templates select="child::methodCall"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Named templates can factor out common code that's used in multiple places. Each template rule can have any number of parameters represented as **`xsl:param`** elements. These appear inside the `xsl:template` element before the template itself. Each `xsl:param` element has a `name` attribute and an optional **`select`** attribute. The `select` attribute provides a default value for that parameter when the

template is invoked but can be overridden. If the `select` attribute is omitted, then the default value for the parameter is set by the contents of the `xsl:param` element. (For a non-overridable variable, you can use a local `xsl:variable` element instead.) Most of the time writing a schema is easier. However, this technique can be used to verify things a schema can't. For example, it could test that a `value` element contains either an ASCII string or a type element such as `int`, but not a type element and an ASCII string.
XSLT is weakly typed. There is no `type` attribute on the `xsl:param` element. You can pass in pretty much any object as the value of one of these parameters. If you use such a variable in a place where an item of that type can't be used and can't be converted to the right type, then the processor will stop and report an error. The ability for a template to call itself (recursion) is the final ingredient of a fully Turing-complete language. [For a recursive factorial code, see the book.]
Functional languages like XSLT do not allow variables to change their values or permit side effects. This can seem a little strange to programmers accustomed to imperative languages like Java. **The key is to remember that almost any task a loop performs in Java, recursion performs in XSLT.**

```
for (int i=1; i <= 10; i++) {
  System.out.print(i);
}
```

```
<xsl:template name="CS101">
  <xsl:param name="index" select="1"/>
  <xsl:if test="$index &lt;= 10">
    <xsl:value-of select="$index"/>
    <xsl:call-template name="CS101">
      <xsl:with-param name="index" select="$index + 1"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
```

TrAX

TrAX, the Transformations API for XML, is a Java API for performing XSLT transforms. It is sufficiently parser-independent that it can work with many different XSLT processors including Xalan and SAXON. It is sufficiently model-independent that it can transform to and from XML streams, SAX event sequences, and DOM and JDOM trees. TrAX is standard part of JAXP (bundled with Java SE 1.4 or later).
4 main classes to use (all in javax.xml.transforms package):

1. **`Transformer`** - The class that represents the style sheet. It transforms a `Source` into a `Result`.
2. **`TransformerFactory`** – The class that represents the XSLT processor. This is a factory class that reads a stylesheet to produce a new `Transformer`.
3. **`Source`** – The interface that represents the input XML document to be transformed, whether presented as a DOM tree, an `InputStream`, or a SAX event sequence.
4. **`Result`** – The interface that represents the input XML document to be transformed, whether presented as a DOM tree, an `InputStream`, or a SAX event sequence.

To transform an input document into an output document follow these steps:

1. Load the `TransformerFactory` with the static **`TransformerFactory.newInstance()`** factory method.
2. Form a **`Source`** object from the **XSLT stylesheet**.
3. Pass this `Source` object to the factory's **`newTransformer()`** factory method to build a `Transformer` object.
4. Build a **`Source`** object from **the input XML document** you wish to transform.
5. Build a **`Result`** object for the **target of the transformation**.
6. Pass both the source and the result to the `Transformer` object's **`transform`**`()` method.

You can reuse the same `Transformer` object repeatedly in series, though you can't use it in multiple threads in parallel. (Steps 4-6 can be repeated).

```
try {
  TransformerFactory xformFactory = TransformerFactory.newInstance();
```



```
  Source xsl = new StreamSource("FibonacciXMLRPC.xsl");
  Transformer stylesheet = xformFactory.newTransformer(xsl);

  Source request  = new StreamSource(in);
  Result response = new StreamResult(out);
  stylesheet.transform(request, response);
}
catch (TransformerException e) {
  System.err.println(e);
}
```

Neither `TransformerFactory` nor `Transformer` is guaranteed to be thread-safe. If your program is multi-threaded, the simplest solution is just to give each separate thread its own `TransformerFactory` and `Transformer` objects. Here's a complete example using TrAX (**NOTE**: I used StylusStudio 2006 XML Enterprise Edition product build 501e. It's a useful product to try out for any XSLT and XML Schema work. It makes it easy to create XSLT from input XML document and then provides a WYSIWYG editor to edit the XSLT and create HTML/PDF/XML output from it. You can plug-n-play any XSLT processor stack with it and compare the results. It comes bundled with Saxon, Xalan_J etc.):

FibonacciXMLRPC.xsl:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:choose>
      <!-- Basic sanity check on the input -->
      <xsl:when
        test="count(methodCall/params/param/value/int) = 1">
        <xsl:apply-templates select="child::methodCall"/>
      </xsl:when>
      <xsl:otherwise>
        <!-- Sanity check failed -->
        <xsl:call-template name="faultResponse"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template match="methodCall">
    <methodResponse>
      <params>
        <param>
          <value>
            <xsl:apply-templates
              select="params/param/value/int"/>
          </value>
        </param>
      </params>
    </methodResponse>
  </xsl:template>

  <xsl:template match="int">
    <int>
      <xsl:call-template name="calculateFibonacci">
        <xsl:with-param name="index" select="number(.)"/>
      </xsl:call-template>
    </int>
  </xsl:template>

  <xsl:template name="calculateFibonacci">
    <xsl:param name="index"/>
    <xsl:param name="low"  select="1"/>
    <xsl:param name="high" select="1"/>
    <xsl:choose>
      <xsl:when test="$index &lt;= 1">
        <xsl:value-of select="$low"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:call-template name="calculateFibonacci">
          <xsl:with-param name="index" select="$index - 1"/>
          <xsl:with-param name="low"   select="$high"/>
```

```
          <xsl:with-param name="high"  select="$high + $low"/>
        </xsl:call-template>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template name="faultResponse">
    <xsl:param name="err_code"    select="0" />
    <xsl:param name="err_message" select="'Unspecified Error'"/>
    <methodResponse>
      <fault>
        <value>
          <struct>
            <member>
              <name>faultCode</name>
              <value>
                <int><xsl:value-of select="$err_code"/></int>
              </value>
            </member>
            <member>
              <name>faultString</name>
              <value>
                <string>
                  <xsl:value-of select="$err_message"/>
                </string>
              </value>
            </member>
          </struct>
        </value>
      </fault>
    </methodResponse>
  </xsl:template>

</xsl:stylesheet>
```

inputXMLDoc.xml:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>calculateFibonacci</methodName>
  <params>
    <param>
      <value><int>10</int></value>
    </param>
  </params>
</methodCall>
```

SimpleXMLRPCService.java:

```
package com.rwatsh.jaxp.transform;

import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;
public class SimpleXMLRPCService
{
        public SimpleXMLRPCService()
        {

        }

        public static void main(String args[])
        {
                try {
                  TransformerFactory xformFactory = TransformerFactory.newInstance();
                  Source xsl = new StreamSource("./FibonacciXMLRPC.xsl");
                  Transformer stylesheet = xformFactory.newTransformer(xsl);
                  File file = new File("./inputXMLRPCDoc.xml");
                  FileInputStream in = new FileInputStream(file);
                  Source request  = new StreamSource(in);
                  Result response = new StreamResult(System.out);
                  stylesheet.transform(request, response);
```

```
                }
                catch (TransformerException e) {
                  System.err.println(e);
                } catch (IOException ioe) {
                  ioe.printStackTrace();
                }

        }
}
```

O/P Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
        <params>
                <param>
                        <value>
                                <int>55</int>
                        </value>
                </param>
        </params>
</methodResponse>
```

The `javax.xml.transform.TransformerFactory` Java system property determines which XSLT engine TrAX uses. Its value is the fully qualified name of the implementation of the abstract `javax.xml.transform.TransformerFactory` class. Possible values could be: `net.sf.saxon.TransformerFactoryImpl`, `org.apache.xalan.processor.TransformerFactoryImpl etc.` You can set this property in one of the following ways:

1. `System.setProperty( "javax.xml.transform.TransformerFactory", "classname")`
2. The value specified at the command line using the `-Djavax.xml.transform.TransformerFactory=classname` option to the **java** interpreter
3. The class named in the `lib/jaxp.properties` properties file in the JRE directory
4. The class named in the `META-INF/services/javax.xml.transform.TransformerFactory` file in the JAR archives available to the runtime
5. Finally, if all of the above options fail, `TransformerFactory.newInstance()` returns a default implementation. In Sun's JDK 1.4, this is Xalan 2.2d10. Latest version of Xalan as of this writing is:2.7.0. I ran the above program using the default Xalan_J provided with Sun's JDK 1.5.0_06.

XML documents may contain an `xml-stylesheet` processing instruction in their prologs that specifies the stylesheet to apply to the XML document. At a minimum, this has an `href` pseudo-attribute specifying the location of the stylesheet to apply and a `type` pseudo-attribute specifying the MIME media type of the stylesheet. For XSLT stylesheets the proper type is application/xml.

```
<?xml-stylesheet href="docbook-xsl-1.50.0/html/docbook.xsl"
                 type="application/xml"
                 media="screen"
                 title="HTML"
                 encoding="UTF-8"
                 alternate="no"?>
<?xml-stylesheet href="docbook-xsl-1.50.0/fo/docbook.xsl"
                 type="application/xml"
                 media="print"
                 title="XSL-FO"
                 encoding="UTF-8"
                 alternate="no"?>
```

This processing instruction is a **hint**. It is only a hint. Programs are not required to use the stylesheet the document indicates. They are free to choose a different transform, multiple transforms, or no transform at all. Indeed, the purpose of this processing instruction is primarily browser display. Programs doing something other than loading the document into a browser for a human to read will likely want to use their own XSLT transforms for their own purposes.

The `TransformerFactory` class has a `getAssociatedStylesheet()` method that loads the stylesheet indicated by such a processing instruction.

```
public abstract Source getAssociatedStylesheet(Source xmlDocument, String media,
String title, String charset)
    throws TransformerConfigurationException;
```

A `TransformerConfigurationException` is thrown if there is no `xml-stylesheet` processing instruction pointing to an XSLT stylesheet matching the specified criteria.

Not all XSLT processors support exactly the same set of capabilities, even within the limits defined by XSLT 1.0. For example, some processors can only transform DOM trees, whereas others may require a sequence of SAX events, and still others may only be able to work with raw streams of text. TrAX uses URI-named features to indicate which of the TrAX classes any given implementation supports. It defines eight standard features as unresolvable URL strings, each of which is also available as a named constant in the relevant TrAX class:

TrAXFeatureTester.java:

```
package com.rwatsh.jaxp.transform;

import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.sax.*;


public class TrAXFeatureTester {

  public static void main(String[] args) {

    TransformerFactory xformFactory = TransformerFactory.newInstance();

    String name = xformFactory.getClass().getName();

    if (xformFactory.getFeature(DOMResult.FEATURE)) {
      System.out.println(name + " supports DOM output.");
    }
    else {
      System.out.println(name + " does not support DOM output.");
    }
    if (xformFactory.getFeature(DOMSource.FEATURE)) {
      System.out.println(name + " supports DOM input.");
    }
    else {
      System.out.println(name + " does not support DOM input.");
    }

    if (xformFactory.getFeature(SAXResult.FEATURE)) {
      System.out.println(name + " supports SAX output.");
    }
    else {
      System.out.println(name + " does not support SAX output.");
    }
    if (xformFactory.getFeature(SAXSource.FEATURE)) {
      System.out.println(name + " supports SAX input.");
    }
    else {
      System.out.println(name + " does not support SAX input.");
    }

    if (xformFactory.getFeature(StreamResult.FEATURE)) {
      System.out.println(name + " supports stream output.");
    }
    else {
      System.out.println(name + " does not support stream output.");
    }
    if (xformFactory.getFeature(StreamSource.FEATURE)) {
      System.out.println(name + " supports stream input.");
    }
    else {
```

```
      System.out.println(name + " does not support stream input.");
    }

    if (xformFactory.getFeature(SAXTransformerFactory.FEATURE)) {
      System.out.println(name + " returns SAXTransformerFactory "
       + "objects from TransformerFactory.newInstance().");
    }
    else {
      System.out.println(name
       + " does not use SAXTransformerFactory.");
    }
    if (xformFactory.getFeature(SAXTransformerFactory.FEATURE_XMLFILTER)) {
      System.out.println(
       name + " supports the newXMLFilter() methods.");
    }
    else {
      System.out.println(
       name + " does not support the newXMLFilter() methods.");
    }
  }
}
```

O/P: java com.rwatsh.jaxp.transform.TrAXFeatureTester
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl supports DOM output.
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl supports DOM input.
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl supports SAX output.
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl supports SAX input.
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl supports stream output.
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl supports stream input.
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl returns SAXTransformerFactory objects from TransformerFactory.newInstance().
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl supports the newXMLFilter() methods.
Xalan_J supports all 8 features:

- `StreamSource.FEATURE`: `http://javax.xml.transform.stream.StreamSource/feature`
- `StreamResult.FEATURE`: `http://javax.xml.transform.stream.StreamResult/feature`
- `DOMSource.FEATURE`: `http://javax.xml.transform.dom.DOMSource/feature`
- `DOMResult.FEATURE`: `http://javax.xml.transform.dom.DOMResult/feature`
- `SAXSource.FEATURE`: `http://javax.xml.transform.dom.SAXSource/feature`
- `SAXResult.FEATURE`: `http://javax.xml.transform.dom.SAXResult/feature`
- `SAXTransformerFactory.FEATURE`: `http://javax.xml.transform.sax.SAXTransformerFactory/feature`
- `SAXTransformerFactory.FEATURE_XMLFILTER`: `http://javax.xml.transform.sax.SAXTransformerFactory/feature/xmlfilter`

An XSLT stylesheet can use the `document()` function to load additional source documents for processing. It can also import or include additional stylesheets with the `xsl:import` and `xsl:include` instructions. In all three cases the document to load is identified by a URI.

Top-level `xsl:param` and `xsl:variable` elements both define variables by binding a name to a value. This variable can be dereferenced elsewhere in the stylesheet using the form `$name`. Once set, the value of an XSLT variable is fixed and cannot be changed. However if the variable is defined with a top-level `xsl:param` element instead of an `xsl:variable` element, then the default value can be changed

before the transformation begins. The initial (and thus final) value of any parameter can be changed inside your Java code using these three methods of the `Transformer` class.

```
public abstract void setParameter(String name, Object value);
public abstract Object getParameter(String name);
public abstract void clearParameters();
```

The `setParameter()` method provides a value for a parameter that overrides any value used in the stylesheet itself. The processor is responsible for converting the Java object type passed to a reasonable XSLT equivalent. This should work well enough for `String`, `Integer`, `Double`, and `Boolean` as well as DOM types like `Node` and `NodeList`. The `getParameter()` method returns the value of a parameter previously set by Java. It will not return any value from the stylesheet itself, even if it has not been overridden by the Java code. Finally, the `clearParameters()` method eliminates all Java mappings of parameters so that those variables are returned to whatever value is specified in the stylesheet.

```
<xsl:param name="fop.extensions">1</xsl:param>

transformer.setParameter("fop.extensions", "1");
```

XSLT is defined in terms of a transformation from one tree to a different tree, all of which takes place in memory. The actual conversion of that tree to a stream of bytes or a file is an optional step. If that step is taken, the `xsl:output` instruction controls the details of serialization. For example, it can specify XML, HTML, or plain text output. It can specify the encoding of the output, what the document type declaration points to, whether the elements should be indented, what the value of the standalone declaration is, where CDATA sections should be used, and more.

```
<xsl:output
  method="xml"
  encoding="UTF-16"
  indent="yes"
  media-type="text/xml"
  standalone="yes"
/>
```

You can also control these output properties from inside your Java programs using these four methods in the `Transformer` class.

```
public abstract void setOutputProperties(Properties outputFormat)
    throws IllegalArgumentException;
public abstract Properties getOutputProperties();
public abstract void setOutputProperty(String name, String value)
    throws IllegalArgumentException;
public abstract String getOutputProperty(String name);
```

For convenience, the `javax.xml.transform.OutputKeys` class provides named constants for all the property names.

```
package javax.xml.transform;

public class OutputKeys {

  private OutputKeys() {}

  public static final String METHOD = "method";
  public static final String VERSION = "version";
  public static final String ENCODING = "encoding";
  public static final String OMIT_XML_DECLARATION
   = "omit-xml-declaration";
  public static final String STANDALONE = "standalone";
  public static final String DOCTYPE_PUBLIC = "doctype-public";
  public static final String DOCTYPE_SYSTEM = "doctype-system";
  public static final String CDATA_SECTION_ELEMENTS
   = "cdata-section-elements";
  public static final String INDENT = "indent";
  public static final String MEDIA_TYPE = "media-type";
}

transformer.setOutputProperty(OutputKeys.INDENT, "yes");
```

The `Source` and `Result` interfaces abstract out the API dependent details of exactly how an XML document is represented. You can construct sources from DOM nodes, SAX event sequences, and raw

streams. You can target the result of a transform at a DOM `Node`, a SAX `ContentHandler`, or a stream-based target such as an `OutputStream`, `Writer`, `File`, or `String`.
It is important to set at least the **system IDs of your sources** because some parts of the stylesheet may rely on this.
In theory, you should be able to convert any DOM `Node` object into a `DOMSource` and transform it. In my tests, Xalan-J could transform all the nodes I threw at it. However, Saxon could only transform `Document` objects and `Element` objects that were part of a document tree.
A `DOMResult` is a wrapper around a DOM `Document`, `DocumentFragment`, or `Element Node` to which the output of the transform will be appended. If you specify a `Node` for the result, either via the constructor or by calling `setNode()`, then the output of the transform will be appended to that node's children. Otherwise, the transform output will be appended to a new `Document` or `DocumentFragment Node`. The `getNode()` method returns this `Node`.
The `SAXResult` class, receives output from the XSLT processor as a stream of SAX events fired at a specified `ContentHandler` and optional `LexicalHandler`.

```
public SAXResult(ContentHandler handler);
```

The `StreamSource` and `StreamResult` classes are used as sources and targets for transforms from sequences of bytes and characters. This includes streams, readers, writers, strings, and files.

```
public StreamSource(InputStream inputStream);
public StreamSource(InputStream inputStream, String systemID);
public StreamSource(Reader reader);
public StreamSource(Reader reader, String systemID);
public StreamSource(String systemID);
public StreamSource(File f);
```

You should not specify both an `InputStream` and a `Reader`. If you do, which one the processor reads from is implementation dependent. **If neither an `InputStream` nor a `Reader` is available, then the processor will attempt to open a connection to the URI specified by the system ID. You should set the system ID even if you do specify an `InputStream` or a `Reader` because this will be needed to resolve relative URLs that appear inside the stylesheet and input document**.

JAXB (5.3)

[WST – Chapters 1 and 2]
The Java™Architecture for XML Binding (JAXB) provides a fast and convenient way to bind XML schemas to Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for unmarshalling XML instance documents into Java content trees, and then marshalling Java content trees back into XML instance documents.

In the context of JAXB, an XML document containing data that is constrained by an XML schema is referred to as a *document instance*, and the structure and data within a document instance is referred to as a *content tree*. JAXB supports customizations and overrides to the default binding rules by means of *binding customizations* made either inline as annotations in a source schema, or as statements in an external binding customization file that is passed to the JAXB binding compiler (**xjc**). Note that custom JAXB binding customizations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings. XML content that is unmarshalled as input to the JAXB binding framework -- that is, an XML instance document, from which a Java representation in the form of a content tree is generated. In practice, the term "document" may not have the conventional meaning, as an XML instance document does not have to be a completely formed, selfstanding document file; it can instead take the form of streams of data passed between applications, or of sets of database fields, or of *XML infosets*, in which blocks of information contain just enough information to describe where they fit in the schema structure. JAXB can marshal XML data to XML documents, SAX content handlers, and DOM nodes.

The general steps in the JAXB data binding process are:
1. **Generate classes**. An XML schema is used as input to the JAXB binding compiler to generate JAXB classes based on that schema.
2. **Compile classes**. All of the generated classes, source files, and application code must be compiled.

3. **Unmarshal**. XML documents written according to the constraints in the source schema are unmarshalled by the JAXB binding framework. Note that JAXB also supports unmarshalling XML data from sources other than files/documents, such as DOM nodes, string buffers, SAX Sources, and so forth.
4. **Generate content tree**. The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes; this content tree represents the structure and content of the source XML documents.
5. **Validate (optional)**. The unmarshalling process optionally involves validation of the source XML documents before generating the content tree.
Note that if you modify the content tree in Step 6, below, you can also use the JAXB Validate operation to validate the changes before marshalling the content back to an XML document.
6. **Process content**. The client application can modify the XML data represented by the Java content tree by means of interfaces generated by the binding compiler.
7. **Marshal**. The processed content tree is marshalled out to one or more XML output documents. The content may be validated before marshalling.

Unmarshalling is not the only means by which a content tree may be created. Schema-derived content classes also support the **programmatic construction of content trees** by direct invocation of the appropriate factory methods. Once created, a content tree may be revalidated, either in whole or in part, at any time.

3 packages of JAXB binding framework:

1. javax.xml.bind – defines UnMarshaller, Marshaller,Validator and JAXBContext among others.
2. javax.xml.bind.util – utility classes to manage marshalling, validation and unmarshalling events.
3. javax.xml.bind.helper – partial implementation of abstract classes and interfaces provided in javax.xml.bind package. Classes not to be used directly by applications.

```
JAXBContext jc = JAXBContext.newInstance("com.acme.foo:com.acme.bar" );
```

The JAXBContext is entry point for an application. Parameter contains a list of package names that contain schema derived interfaces. By allowing for multiple Java packages to be specified, the JAXBContext instance allows for the management of multiple schemas at one time.

The JAXB provider should provide for

```
public static JAXBContext createContext( String contextPath,ClassLoader classLoader )
                                                    throws JAXBException;
```

The JAXB provider implementation must generate a jaxb.properties file in each package containing schema-derived classes. This property file must contain a property named javax.xml.bind.context.factory whose value is the name of the class that implements the createContext API.

A client application is able to unmarshal XML documents that are instances of any of the schemas listed in the contextPath;

```
JAXBContext jc = JAXBContext.newInstance("com.acme.foo:com.acme.bar" );
Unmarshaller u = jc.createUnmarshaller();
FooObject fooObj = (FooObject)u.unmarshal( new File( "foo.xml" ) ); // ok
BarObject barObj = (BarObject)u.unmarshal( new File( "bar.xml" ) ); // ok
BazObject bazObj = (BazObject)u.unmarshal( new File( "baz.xml" ) ); // error, "com.acme.baz" not in
                                                                    // contextPath
```

A client application may also generate Java content trees explicitly rather than unmarshalling existing XML data. For each schemaderived Java class, there will be a static factory method that produces objects of that type.

```
ObjectFactory objFactory = new ObjectFactory();
com.acme.foo.PurchaseOrder po = objFactory.createPurchaseOrder();
```

Note: If you have multiple packages on the contextPath, you should use the complete package name when referencing an ObjectFactory class in one of those packages. The JAXB provider implementation must

generate a class in each package that contains all of the necessary object factory methods for that package named **ObjectFactory** as well as the newInstance(javaContentInterface) method.

Once the client application has an instance of the schema-derived object, it can use the mutator methods to set content on it.
The Marshaller class in the javax.xml.bind package provides the client application the ability to convert a Java content tree back into XML data.

```
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );
// unmarshal from foo.xml
Unmarshaller u = jc.createUnmarshaller();
FooObject fooObj = (FooObject)u.unmarshal( new File( "foo.xml" ) );
// marshal to System.out
Marshaller m = jc.createMarshaller();
m.marshal( fooObj, System.out );
```

Client applications are not required to validate the Java content tree prior to calling one of the marshal APIs. There is also no requirement that the Java content tree be valid with respect to its original schema in order to marshal it back into XML data. Some JAXB providers will fully allow marshalling invalid content, others will fail on the first validation error.

The Validator class in the javax.xml.bind package is responsible for controlling the validation of content trees during runtime. When the unmarshalling process incorporates validation and it successfully completes without any validation errors, both the input document and the resulting content tree are guaranteed to be valid. By contrast, the marshalling process does not actually perform validation.
A JAXB client can perform two types of validation:

1. **Unmarshal-Time validation** enables a client application to receive information about validation errors and warnings detected while unmarshalling XML data into a Java content tree. To enable or disable it, use the Unmarshaller.setValidating method.
2. **On-Demand validation** enables a client application to receive information about validation errors and warnings detected in the Java content tree. At any point, client applications can call the Validator.validate method on the Java content tree (or any sub-tree of it).

If the client application does not set an event handler (using setEventHandler()) on its Validator, Unmarshaller, or Marshaller prior to calling the validate, unmarshal, or marshal methods, then a default event handler will receive notification of any errors or warnings encountered. The default event handler will cause the current operation to halt after encountering the first error or fatal error
Client applications that require sophisticated event processing can implement the **ValidationEventHandler** interface and register it with the Unmarshaller and/or Validator.

ValidationEventCollector utility: For convenience, a specialized event handler is provided that simply collects any **ValidationEvent** objects created during the unmarshal, validate, and marshal operations and returns them to the client application as a java.util.Collection.

XML to Java Bindings

A schema component using a simple type definition typically binds to a Java property. Schema type to Java type mapping table is here.
Default binding rules are summarized below:

XML namespace URI	java package
Named or anonymous complex types	java content interface
Named simple type with base type that derives from xsd:NCName and has enumeration facets.	java enum class
global element declaration	java Element interface
local element declaration	inserted into a general content list
attribute use, element reference or local element declaration	java property
repeating occurrence	a list content property

Customizing JAXB Bindings

JAXB uses default binding rules that can be customized by means of binding declarations made in either of two ways:

1. As inline annotations in a source XML schema
2. As declarations in an external binding customizations file that is passed to the JAXB compiler.

The binding compiler uses a general name-mapping algorithm to bind XML names to names that are acceptable in the Java programming language. However, if you want to use a different naming scheme for your classes, you can specify custom binding declarations to make the binding compiler generate different names. Customizations you can do are:

1. Name the package, derieved classes and methods
2. Assign types to the methods within the derived classes
3. Choose which elements to bind to classes
4. Decide how to bind each attribute and element declaration to a property in the appropriate content class
5. Choose the type of each attribute-value or content specification

In most cases, the default rules are robust enough that a usable binding can be produced with no custom binding declaration at all. Customizing JAXB Bindings is not needed for exam and henceforth will not be covered.

Using JAXB

Po.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="1" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice" type="xsd:decimal"/>
          <xsd:element ref="comment" minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
```

```
        <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Po.xml:

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Cambridge</city>
        <state>MA</state>
        <zip>12345</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Cambridge</city>
        <state>MA</state>
        <zip>12345</zip>
    </billTo>
    <items>
        <item partNum="242-NO" >
               <productName>Nosferatu - Special Edition (1929)</productName>
               <quantity>5</quantity>
               <USPrice>19.99</USPrice>
           </item>
        <item partNum="242-MU" >
               <productName>The Mummy (1959)</productName>
               <quantity>3</quantity>
               <USPrice>19.98</USPrice>
           </item>
        <item partNum="242-GZ" >
               <productName>Godzilla and Mothra: Battle for Earth/Godzilla vs. King
Ghidora</productName>
               <quantity>3</quantity>
               <USPrice>27.95</USPrice>
           </item>
        </items>
</purchaseOrder>
```

com.sun.tools.xjc.XJCTask (the JAXB compiler) is invoked with the po.xsd being passed to it as shown in the snippet from ant build.xml file:

```
<taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
  <classpath refid="classpath" />
 </taskdef>
<mkdir dir="gen-src" />
  <xjc schema="po.xsd" package="primer.po" target="gen-src">
    <produces dir="gen-src/primer.po" includes="**/*.java" />
  </xjc>
```

The above compilation generates jaxb.properties and bgm.ser files in the gen-src directory. Following is the description for generated .java files:

primer/po/Comment.java	Public interface extending javax.xml.bind.Element; binds to the global schema element named comment. Note that JAXB generates element

	interfaces for all global element declarations.
primer/po/Items.java	Public interface that binds to the schema complexType named Items.
primer/po/ObjectFactory.java	Public class extending com.sun.xml.bind.DefaultJAXBContextImpl; used to create instances of specified interfaces. For example, the ObjectFactory createComment() method instantiates a Comment object.
primer/po/PurchaseOrder.java	Public interface extending javax.xml.bind.Element, and PurchaseOrderType; binds to the global schema element named PurchaseOrder.
primer/po/PurchaseOrderType.java	Public interface that binds to the schema complexType named PurchaseOrderType.
primer/po/USAddress.java	Public interface that binds to the schema complexType named USAddress.

Then there are implementation classes for the above mentioned interfaces in the primer/po/impl package:

1. CommentImpl.java
2. ItemsImpl.java
3. PurchaseOrderImpl.java
4. PurchaseOrderTypeImpl.java
5. USAddressImpl.java

Note: You should never directly use the generated implementation classes—that is, *Impl.java in the *<packagename>*/impl directory. These classes are not directly referenceable because the class names in this directory are not standardized by the JAXB specification. The ObjectFactory method is the only portable means to create an instance of a schema-derived interface. There is also an ObjectFactory.newInstance(Class JAXBinterface) method that enables you to create instances of interfaces.

Schema to Java bindings:

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">	
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>	PurchaseOrder.java
<xsd:element name="comment" type="xsd:string"/>	Comment.java
<xsd:complexType name="PurchaseOrderType"> <xsd:sequence> <xsd:element name="shipTo" type="USAddress"/> <xsd:element name="billTo" type="USAddress"/> <xsd:element ref="comment" minOccurs="0"/> <xsd:element name="items" type="Items"/> </xsd:sequence> <xsd:attribute name="orderDate" type="xsd:date"/> </xsd:complexType>	PurchaseOrderType.java
<xsd:complexType name="USAddress"> <xsd:sequence> <xsd:element name="name" type="xsd:string"/> <xsd:element name="street" type="xsd:string"/> <xsd:element name="city" type="xsd:string"/> <xsd:element name="state" type="xsd:string"/> <xsd:element name="zip" type="xsd:decimal"/> </xsd:sequence> <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/> </xsd:complexType>	USAddress.java
<xsd:complexType name="Items"> <xsd:sequence> <xsd:element name="item" minOccurs="1" maxOccurs="unbounded"> <xsd:complexType> <xsd:sequence> <xsd:element name="productName" type="xsd:string"/> <xsd:element name="quantity"> <xsd:simpleType> <xsd:restriction base="xsd:positiveInteger"> <xsd:maxExclusive value="100"/> </xsd:restriction> </xsd:simpleType> </xsd:element> <xsd:element name="USPrice" type="xsd:decimal"/> <xsd:element ref="comment" minOccurs="0"/> <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/> </xsd:sequence> <xsd:attribute name="partNum" type="SKU" use="required"/>	Items.java **Items.ItemType.java** (Since item is a local element (and not global element like comment) so its defined as a nested interface of Items interface. Also as maxoccurs is unbounded so Items interface provides a getItems() method which returns a List.)

</xsd:complexType>	
<xsd:simpleType name="SKU"> <xsd:restriction base="xsd:string"> <xsd:pattern value="\d{3}-[A-Z]{2}"/> </xsd:restriction> </xsd:simpleType>	A Simple type is mapped to the base types Java equivalent (in this case String).

Then the generated classes are compiled as follows:

```
<mkdir dir="classes" />
   <javac destdir="classes" debug="on">
     <src path="src" />
     <src path="gen-src" />
     <classpath refid="classpath" />
   </javac>
   <copy todir="classes">
     <fileset dir="gen-src">
       <include name="**/*.properties" />
       <include name="**/bgm.ser" />
     </fileset>
   </copy>
```

Then run the Main.java client application.

```
<java classname="Main" fork="true">
     <classpath refid="classpath" />
   </java>
```

Main.java:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;

// import java content classes generated by binding compiler
import primer.po.*;


public class Main {

    // This sample application demonstrates how to unmarshal an instance
    // document into a Java content tree and access data contained within it.

    public static void main( String[] args ) {
        try {
            // create a JAXBContext capable of handling classes generated into
            // the primer.po package
            JAXBContext jc = JAXBContext.newInstance( "primer.po" );

            // create an Unmarshaller
            Unmarshaller u = jc.createUnmarshaller();

            // unmarshal a po instance document into a tree of Java content
            // objects composed of classes from the primer.po package.
            PurchaseOrder po =
                (PurchaseOrder)u.unmarshal( new FileInputStream( "po.xml" ) );

            // examine some of the content in the PurchaseOrder
            System.out.println( "Ship the following items to: " );

            // display the shipping address
            USAddress address = po.getShipTo();
            displayAddress( address );
```

```
            // display the items
            Items items = po.getItems();
            displayItems( items );

            // Modify the Content Tree and Marshall.
            // change the billto address
            USAddress address = po.getBillTo();
            address.setName( "John Bob" );
            address.setStreet( "242 Main Street" );
            address.setCity( "Beverly Hills" );
            address.setState( "CA" );
            address.setZip( new BigDecimal( "90210" ) );

            // create a Marshaller and marshal to a file
            Marshaller m = jc.createMarshaller();
            m.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE );
            m.marshal( po, System.out );

            // Create and Marshall Content Tree to XML.
            // create an ObjectFactory instance.
            // if the JAXBContext had been created with mutiple pacakge names,
            // we would have to explicitly use the correct package name when
            // creating the ObjectFactory.
            ObjectFactory objFactory = new ObjectFactory();

            // create an empty PurchaseOrder
            po = objFactory.createPurchaseOrder();

            // set the required orderDate attribute
            po.setOrderDate( Calendar.getInstance() );

            // create shipTo USAddress object
            USAddress shipTo = createUSAddress( objFactory,
                                                "Alice Smith",
                                                "123 Maple Street",
                                                "Cambridge",
                                                "MA",
                                                "12345" );

            // set the required shipTo address
            po.setShipTo( shipTo );

            // create billTo USAddress object
            USAddress billTo = createUSAddress( objFactory,
                                                "Robert Smith",
                                                "8 Oak Avenue",
                                                "Cambridge",
                                                "MA",
                                                "12345" );

            // set the requred billTo address
            po.setBillTo( billTo );

            // create an empty Items object
            items = objFactory.createItems();

            // get a reference to the ItemType list
            List itemList = items.getItem();

            // start adding ItemType objects into it
            itemList.add( createItemType( objFactory,
                                          "Nosferatu - Special Edition (1929)",
                                          new BigInteger( "5" ),
                                          new BigDecimal( "19.99" ),
                                          null,
                                          null,
                                          "242-NO" ) );
            itemList.add( createItemType( objFactory,
                                          "The Mummy (1959)",
                                          new BigInteger( "3" ),
```



```
                                            new BigDecimal( "19.98" ),
                                            null,
                                            null,
                                            "242-MU" ) );
            itemList.add( createItemType( objFactory,
                                            "Godzilla and Mothra: Battle for Earth/Godzilla
vs. King Ghidora",
                                            new BigInteger( "3" ),
                                            new BigDecimal( "27.95" ),
                                            null,
                                            null,
                                            "242-GZ" ) );

            // set the required Items list
            po.setItems( items );
            m.marshal( po, System.out );

            //Demo of on-demand validation.
            // get a reference to the first item in the po
            items = po.getItems();
            List itemTypeList = items.getItem();
            Items.ItemType item = (Items.ItemType)itemTypeList.get( 0 );

            // invalidate it by setting some bogus data
            item.setQuantity( new BigInteger( "-5" ) );

            // create a Validator
            Validator v = jc.createValidator();

            // validate the content tree
            System.out.println("NOTE:  This  sample  is  working  correctly  if  you  see
validation errors!!");
            boolean valid = v.validateRoot( po );
            System.out.println( valid );
        } catch( ValidationException ue ) {
            System.out.println( "Caught ValidationException" );
        } catch( JAXBException je ) {
            je.printStackTrace();
        } catch( IOException ioe ) {
            ioe.printStackTrace();
        }
    }

    public static void displayAddress( USAddress address ) {
        // display the address
        System.out.println( "\t" + address.getName() );
        System.out.println( "\t" + address.getStreet() );
        System.out.println( "\t" + address.getCity() +
                            ", " + address.getState() +
                            " "  + address.getZip() );
        System.out.println( "\t" + address.getCountry() + "\n");
    }

    public static void displayItems( Items items ) {
        // the items object contains a List of primer.po.ItemType objects
        List itemTypeList = items.getItem();


        // iterate over List
        for( Iterator iter = itemTypeList.iterator(); iter.hasNext(); ) {
            Items.ItemType item = (Items.ItemType)iter.next();
            System.out.println( "\t" + item.getQuantity() +
                                " copies of \"" + item.getProductName() +
                                "\"" );
        }
    }
}
```

The above code combines the 5 basic examples discussed in the WST 1.5.

SAAJ 1.2 (5.4)

[RMH – Chapters 13 & Appendix F. Infact read E, F and G in order to understand SwA.]

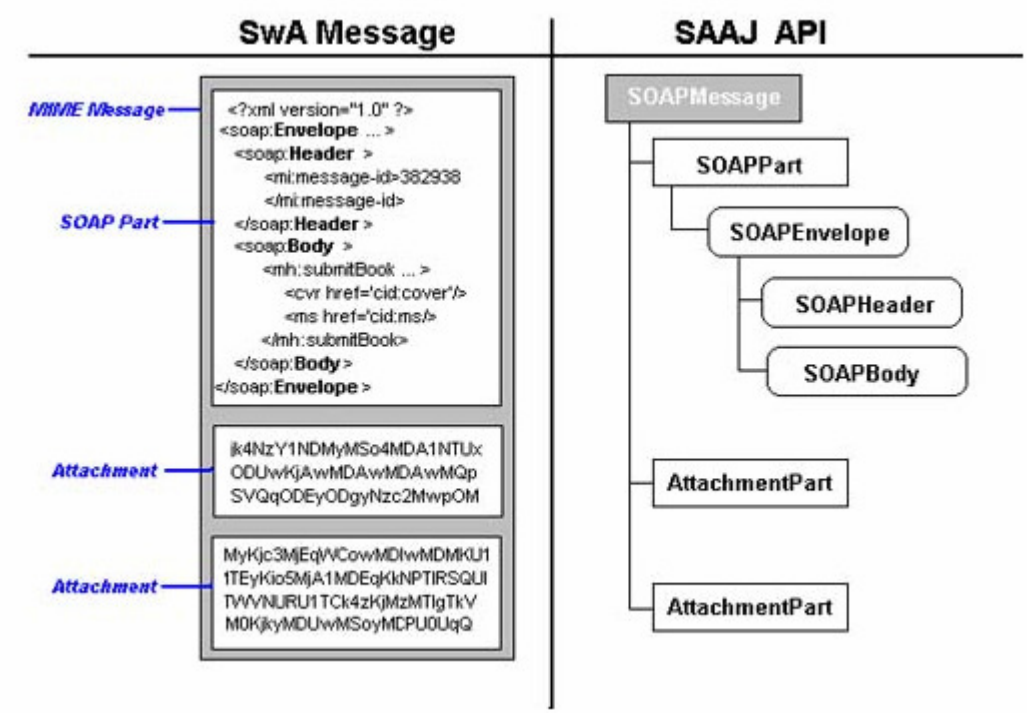
To read and manipulate SOAP header blocks in JAX-RPC you have to use message handlers which use SOAP with Attachments API for Java (SAAJ), version 1.2, to represent an XML SOAP message. SAAJ is an API-based SOAP toolkit, which means that it models the structure of SOAP messages. SAAJ models SOAP Messages with Attachments (SwA) in Java. SwA is the MIME message format for SOAP. While SAAJ models plain SOAP messages as well as SwA, the WS-I Basic Profile does not endorse SwA yet (BP 1.1 does).

SAAJ (rhymes with "page") is an API you can use to create, read, or modify SOAP messages using Java. It includes classes and interfaces that model the SOAP `Envelope`, `Body`, `Header`, and `Fault` elements, along with XML namespaces, elements, and attributes, and text nodes and MIME attachments. SAAJ is similar to JDBC, in that it's a hollow API. You can't use it by itself; you need a vendor implementation. Each J2EE vendor will provide its own SAAJ implementation.

You can use SAAJ to manipulate simple SOAP messages (just the XML without any attachments) or more complex SOAP messages with MIME attachments. SAAJ is used in combination with JAX-RPC (Java API for XML-based RPC), which is the J2EE standard API for sending and receiving SOAP messages. SAAJ can also be used independently of JAX-RPC and has its own, optional facilities for basic Request/ Response-style messaging over HTTP or other protocols.

Java developers can use SAAJ to work with SOAP messages within any SOAP application, including initial senders, intermediaries, and ultimate receivers. For example, you might develop a SOAP intermediary that processes a specific header block before sending the message on to the next receiver. Using SAAJ you can easily examine a SOAP message, extract the appropriate header block, then send the message along to the next node in the message path. Similarly, an ultimate receiver can use SAAJ to process the application-specific content of the SOAP body.

SAAJ is a family of types in which each type of object is manufactured by another type in the SAAJ family. The root of the Abstract Factory Pattern in SAAJ is the **`MessageFactory`** class. It's responsible for manufacturing an instance of itself, which can in turn be used to manufacture a **`SOAPMessage`**. A `SOAPMessage` contains a **`SOAPPart`**, which represents the SOAP document, and zero or more **`AttachmentPart`** objects, which represent MIME attachments (such as GIFs and PDFs). The `SOAPPart` contains a family of objects that model the SOAP document, including the **`Envelope`**, **`Header`**, and **`Body`** elements. You can obtain a clearer understanding of SAAJ by examining the basic structure of the SAAJ API alongside a diagram of an SwA message.



SAAJ 1.2 is also based, in part, on the W3C Document Object Model (DOM), version 2.

SAAJ Example

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

To create the above SOAP message using SAAJ:

```
package com.jwsbook.saaj;
import javax.xml.soap.*;

public class SaajExample_1 {
  public static void main(String [] args) throws SOAPException{

    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage message = msgFactory.createMessage();
    message.getSOAPHeader().detachNode();
    SOAPBody body = message.getSOAPBody();
    SOAPElement getBookPrice = body.addChildElement(
                            "getBookPrice",
                            "mh",
                            "http://www.Monson-Haefel.com/jwsbook/BookQuote");
    getBookPrice.setEncodingStyle(SOAPConstants.URI_NS_SOAP_ENCODING);
    SOAPElement isbn = getBookPrice.addChildElement("isbn");
    isbn.addTextNode("0321146182");

    SaajOutputter.writeToScreen(message);
  }
}
```

Creating a SOAP Message

To create a simple SOAP document, you obtain a new `SOAPMessage` object from a `MessageFactory` object.

```
MessageFactory msgFactory = MessageFactory.newInstance();
SOAPMessage message = msgFactory.createMessage();
```

SAAJ Attachments shows how to create an SwA message in which the SOAP document is treated as a MIME part, and accessed via the **`SOAPMessage.getSOAPPart()`** method.

The `MessageFactory` is the root factory of SAAJ. `MessageFactory` is an abstract class that contains three methods:

```
package javax.xml.soap;

public abstract class MessageFactory {

  private static final String DEFAULT_MESSAGE_FACTORY =
          "com.sun.xml.messaging.saaj.soap.MessageFactoryImpl";

  private static final String MESSAGE_FACTORY_PROPERTY =
          "javax.xml.soap.MessageFactory";

  public static MessageFactory newInstance() throws SOAPException;

  public SOAPMessage createMessage() throws SOAPException;

  public SOAPMessage createMessage(MimeHeaders headers,java.io.InputStream in)
  throws SOAPException;
}
```

Its `newInstance()` method creates an object that is actually a subtype of `MessageFactory`. By default, the instance is of a proprietary type provided by Sun Microsystems, when SAAJ is employed as a standalone API.

SOAPMessage message = msgFactory.**createMessage**();
creates a SOAP message from scratch and contains only a framework of a SOAP message as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header/>
  <soap:Body/>
</soap:Envelope>
```

Once the `SOAPMessage` is created, all you need to do is fill in the blanks using the SAAJ API. The `MessageFactory`'s second create method can construct a SAAJ representation of an existing SOAP message, instead of building a new one from scratch.

The `MimeHeaders` parameter holds one or more MIME headers. A MIME header is a name-value pair that describes the contents of a MIME block. For example, a SOAP document might have a MIME header with a name-value pair of `"Content-Type = text/xml"`. The `InputStream` parameter can be any kind of stream. For example it could be a network stream from a socket connection, an IO stream from a JDBC connection, or a simple file stream. The data obtained from the `InputStream` parameter must be a valid SOAP or SwA message. For example, suppose a file called `soap.xml` contains a SOAP document:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

Using a `FileInputStream`, the `MessageFactory` class can read the `soap.xml` file and generate a SAAJ object graph of the SOAP message it contains.

```
package com.jwsbook.saaj;
import java.io.FileInputStream;
```

```
import javax.xml.soap.*;

public class SaajExample_2 {
   public static void main(String [] args)
   throws SOAPException, java.io.IOException{

        MessageFactory msgFactory = MessageFactory.newInstance();

        MimeHeaders mimeHeaders = new MimeHeaders();
        mimeHeaders.addHeader("Content-Type","text/xml; charset=UTF-8");

        FileInputStream file = new FileInputStream("soap.xml");

        SOAPMessage message = msgFactory.createMessage(mimeHeaders, file);

        file.close();
        SaajOutputter.writeToScreen(message);
    }
}
```

`SOAPMessage.writeTo(java.io.OutputStream out)` method is used to write the SOAP message to an output stream. The SaajOutputter is a custom class in com.jwsbook.saaj which formats the output for SOAP messages without attachments and prints the SOAP message on the stream with new line delimiters. If the `SOAPMessage` has no attachments, `writeTo()` will write only the XML SOAP part of the message to the stream. The output from the `SOAPMessage.writeTo()` method is the default content of the SOAP message without line breaks.

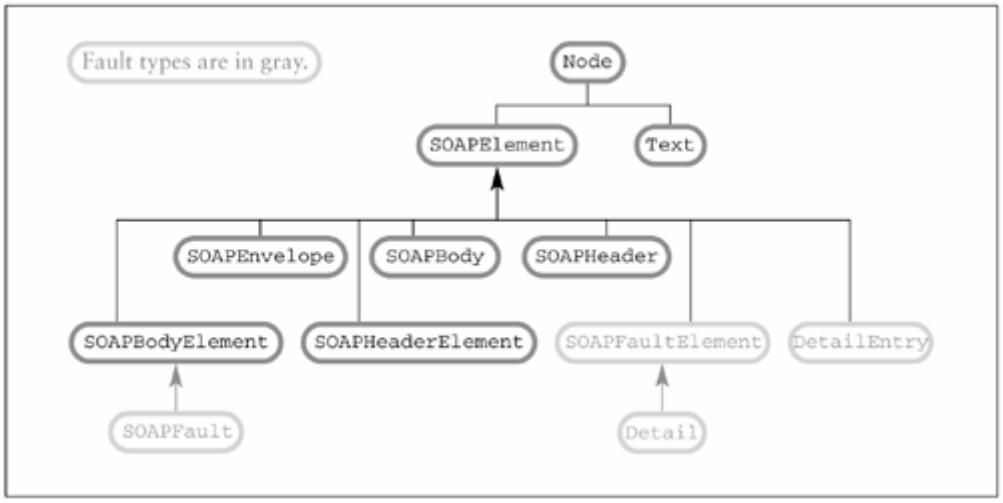
The `getSOAPPart()` method allows you to access the SOAP MIME part directly. When there are no attachments, you can simply use the `getSOAPBody()` and `getSOAPHeader()` methods to access those elements in the SOAP message directly (its not necessary to use getSOAPPart() for accessing header and body elements of a SOAP message).

```
MessageFactory msgFactory = MessageFactory.newInstance();
SOAPMessage message = msgFactory.createMessage();
message.getSOAPHeader().detachNode();
SOAPBody body = message.getSOAPBody();
```

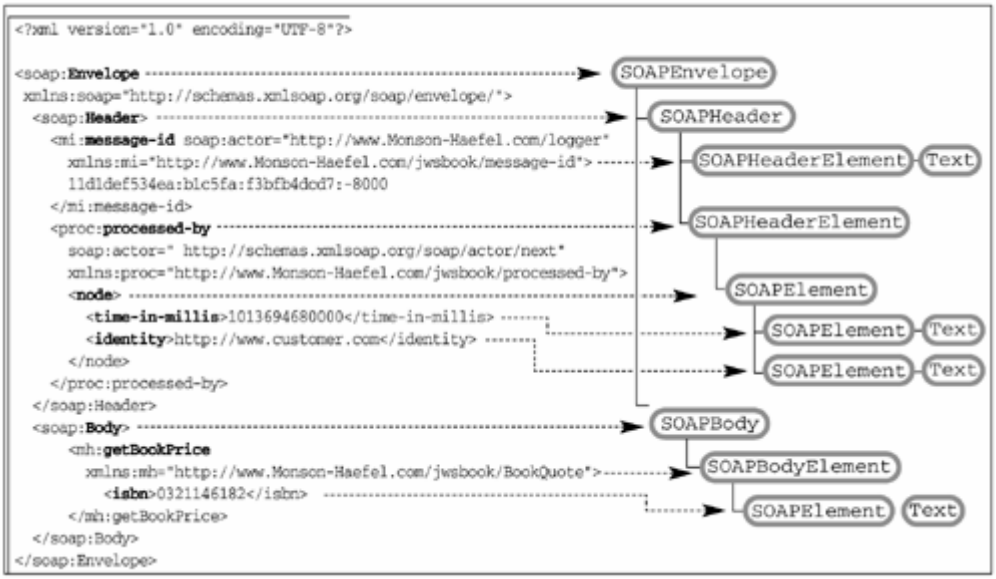
The `detachNode()` method simply removes the `Header` element from the SOAP message—SAAJ always includes the `Header` by default.

Working with SOAP Documents

An inheritance class diagram that shows all the SOAP document elements.



The `Envelope` is represented by `SOAPEnvelope`, the `Header` is represented by `SOAPHeader`, the `Body` by `SOAPBody`, and so on. The `SOAPElement` type is used for application-specific elements that don't belong to the SOAP 1.1 namespace.



The `SOAPPart` represents the root MIME part of an SwA message, which is always the SOAP XML document. You access the `SOAPPart` of an SwA message by invoking the `SOAPMessage.getSOAPPart()` method.

You can obtain a reference to the `SOAPEnvelope` by invoking the **`getEnvelope()`** method of a `SOAPPart`. The `SOAPEnvelope` represents the root of the XML SOAP document. It includes methods for accessing or creating the `SOAPHeader` and `SOAPBody`. Unless you're working with attachments, you won't usually need to deal with the `SOAPEnvelope` type, because SAAJ constructs the `SOAPEnvelope` automatically when you create a new `SOAPMessage` object.

The **`SOAPFactory`** provides two factory methods for creating `Name` type objects.The **`Name`** type is simply an abstraction of an XML qualified name.

```
package com.jwsbook.saaj;
import javax.xml.soap.*;

public class SaajExample_3 {
  public static void main(String [] args)
  throws SOAPException, java.io.IOException{

    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage message = msgFactory.createMessage();
    SOAPFactory soapFactory = SOAPFactory.newInstance();

    Name getBookPrice_Name = soapFactory.createName("getBookPrice","mh",
                        "http://www.Monson-Haefel.com/jwsbook/BookQuote");
    Name isbnName = soapFactory.createName("isbn");

    SOAPBody body = message.getSOAPBody();
    SOAPBodyElement getBookPrice_Element =
                      body.addBodyElement(getBookPrice_Name);
    getBookPrice_Element.addChildElement( isbnName );

    SaajOutputter.writeToScreen(message);
  }
}
```

The application-specific elements, those that are not part of the SOAP 1.1 XML namespace, are represented directly by objects of the `SOAPElement` type. It contains methods for accessing the child elements, attributes, namespace information, and so on. Just as an XML element may contain other XML elements, a `SOAPElement` may contain other `SOAPElement` objects.

```
package javax.xml.soap;
import java.util.Iterator;

public interface SOAPElement extends Node, org.w3c.dom.Element {
    public SOAPElement addAttribute(Name name, String value)
```

```
      throws SOAPException;
    public SOAPElement addChildElement(Name name) throws SOAPException;
    public SOAPElement addChildElement(SOAPElement element)
      throws SOAPException;
    public SOAPElement addChildElement(String localName) throws SOAPException;
    public SOAPElement addChildElement(String localName, String prefix)
      throws SOAPException;
    public SOAPElement addChildElement(String localName, String prefix,
                                       String uri) throws SOAPException;
    public SOAPElement addNamespaceDeclaration(String prefix, String uri)
      throws SOAPException;
    public SOAPElement addTextNode(String text);
    public Iterator getAllAttributes();
    public String getAttributeValue(Name name);
    public Iterator getChildElements();
    public Iterator getChildElements(Name name);
    public Name getElementName();
    public String getEncodingStyle();
    public Iterator getNamespacePrefixes();
    public String getNamespaceURI(String prefix);
    public Iterator getVisableNamespacePrefixes();
    public boolean removeAttribute(Name name);
    public boolean removeNamespaceDeclaration(String prefix);
    public boolean removeContents();
    public void setEncodingStyle(String encodingStyle);
}
```

For example: a SOAP message being created by an initial sender.

```
package com.jwsbook.saaj;
import javax.xml.soap.*;

public class SaajExample_4 {
  public static void main(String [] args) throws SOAPException {

    // Create SOAPMessage
    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage message = msgFactory.createMessage();
    SOAPElement header = message.getSOAPHeader();

    // Create message-id header block
    SOAPElement msgIdHeader = (SOAPElement)
      header.addChildElement("message-id","mi",
                             "http://www.Monson-Haefel.com/jwsbook/message-id");
    String uuid = new java.rmi.dgc.VMID().toString();
    msgIdHeader.addTextNode(uuid);


    // Create processed-by header block
    SOAPElement prcssdByHeader = (SOAPElement)
      header.addChildElement("processed-by","proc",
                            "http://www.Monson-Haefel.com/jwsbook/processed-by");
    SOAPElement node = prcssdByHeader.addChildElement("node");
    SOAPElement time = node.addChildElement("time-in-millis");
    long millis = System.currentTimeMillis();
    time.addTextNode(String.valueOf(millis));
    SOAPElement identity = node.addChildElement("identity");
    identity.addTextNode("SaajExample_4");

    // Create getBookPrice RPC call
    SOAPElement body = message.getSOAPBody();
    SOAPElement getBookPrice = body.addChildElement("getBookPrice","mh",
                              "http://www.Monson-Haefel.com/jwsbook/BookQuote");
    SOAPElement isbn = getBookPrice.addChildElement("isbn");
    isbn.addTextNode("0321146182");

    SaajOutputter.writeToScreen(message);
  }
}
```

It creates the header blocks using `SOAP Element.addChildElement()` instead of `SOAPHeader.addHeaderElement()` because it's easier (you don't have to create a `Name` object).

You should always use the **SOAPElement** returned by the **addChildElement()** method if you need to modify a SOAPElement object after you add it, thus:

```
SOAPElement child = // get SOAPElement from somewhere
child = element.addChildElement( child );
child.addAttribute( attribName, attribValue );
```

The **Node** interface provides a few useful methods for navigating through a hierarchical tree of elements, removing nodes from the tree, and marking nodes for "recycling."

```
package javax.xml.soap;

public interface Node extends org.w3c.dom.Node {
    public void detachNode();
    public SOAPElement getParentElement() throws java.lang.UnsupportedOperationException;
    public String getValue();
    public void setValue(String value) throws java.lang.IllegalStateException;
    public void recycleNode();
    public void setParentElement(SOAPElement parent) throws SOAPException;
}
```

The **detachNode()** method is frequently used to remove the SOAPHeader object from a newly created SOAP message. When a SOAPMessage is first created, it automatically contains Envelope, Header, and Body elements. If the SOAP message you are constructing will not be using any header blocks, then removing the Header element is a good idea.

```
Header header = message.getSOAPHeader();
header.detachNode();
header.recycleNode();
```

recycleNode() is supposed to help the underlying SAAJ implementation conserve resources.

SOAPHeader provides methods for adding, examining, and removing **SOAPHeaderElement** objects—effectively adding, examining, or removing header blocks from the SOAP document.

```
package com.jwsbook.saaj;
import javax.xml.soap.*;

public class SaajExample_5 {
  public static void main(String [] args) throws SOAPException {
    // Create SOAPMessage
    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage message = msgFactory.createMessage();
    SOAPHeader header = message.getSOAPHeader();

    // Create message-id header block
    SOAPHeaderElement msgId = (SOAPHeaderElement)
      header.addChildElement("message-id","mi",
                             "http://www.Monson-Haefel.com/jwsbook/message-id");
    String uuid = new java.rmi.dgc.VMID().toString();
    msgId.addTextNode(uuid);
    msgId.setActor("http://www.Monson-Haefel.com/logger");
    msgId.setMustUnderstand(false);


    // Create getBookPrice RPC call
    SOAPBody body = message.getSOAPBody();
    SOAPElement getBookPrice = body.addChildElement(
                              "getBookPrice",
                              "mh",
                              "http://www.Monson-Haefel.com/jwsbook/BookQuote");
    SOAPElement isbn = getBookPrice.addChildElement("isbn");
    isbn.addTextNode("0321146182");
    SaajOutputter.writeToScreen(message);
    }
}
```

Here's how the o/p will look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
  <soap:Header>
```



```
    <mi:message-id xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
     soap:actor="http://www.Monson-Haefel.com/logger"
     soap:mustUnderstand="0">
       11d1def534ea1be0:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
  </soap:Header>
  <soap:Body>
    <mh:getBookPrice>
      <isbn>0321146182</isbn>
    </mh:getBookPrice>
  </soap:Body>
</soap:Envelope>
```

The `SOAPHeader` class provides five methods for examining or extracting (accessing or removing) header blocks.

```
package javax.xml.soap;
import java.util.Iterator;

public interface SOAPHeader extends SOAPElement {
    public SOAPHeaderElement addHeaderElement( Name name ) throws SOAPException;
    public Iterator extractHeaderElements(String actor);
    public Iterator examineHeaderElements(String actor);
    public Iterator examineMustUnderstandHeaderElements(String actor);
    public Iterator examinAllHeaderElements();
    public Iterator extractAllHeaderElements();
}
```

All of these methods return a `java.util.Iterator` whose elements are `SOAPHeaderElement` objects. For example: given the following SOAP message:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header>
      <mi:message-id soap:actor="http://www.Monson-Haefel.com/logger"
         xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id">
           11d1def534ea1be0:b1c5fa:f3bfb4dcd7:-8000
      </mi:message-id>
      <proc:processed-by
         soap:actor="http://schemas.xmlsoap.org/soap/actor/next"
         xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
         <node>
            <time-in-millis>1013694684723</time-in-millis>
            <identity>http://local/SOAPClient2</identity>
         </node>
      </proc:processed-by>
   </soap:Header>
   <soap:Body>
       <!-- application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

Upon receiving this SOAP message, a SOAP node will request all header blocks that are associated with the standard **`next`** actor so that those headers can be processed.

```
SOAPHeader header = message.getSOAPHeader();
String actor = "http://schemas.xmlsoap.org/soap/actor/next";
Iterator headerBlocks = header.extractHeaderElements( actor );
while(headerBlocks.hasNext()){
    SOAPHeaderElement block = (SOAPHeaderElement)headerBlocks.next();
    if(block.getElementName().getLocalName().equals("processed-by")){
            SOAPElement node = block.addChildElement("node");

            // do something useful with header blocks and then discard them
    }
}
```

Intermediaries are required to remove any header block targeted to a role they play. The `extractHeaderElements()` method enables a receiver to fulfill that obligation in one operation. Some receivers will feign removal and insertion of the same header block, by simply modifying it. In such cases the receiver invokes `examineHeaderElements()` instead of `extract HeaderElements()`. This method allows the node to search for and access header blocks easily, without removing them.


```
package javax.xml.soap;

public interface SOAPHeaderElement extends SOAPElement {
    public String getActor();
    public boolean getMustUnderstand();
    public void setActor(String actorURI);
    public void setMustUnderstand(boolean flag);
}
```

In addition to the `actor` and `mustUnderstand` attributes, a `SOAPHeader Element` may also contain one or more `SOAPElement` objects, which represent the child elements of the header block.

```
package javax.xml.soap;

public interface SOAPBody extends SOAPElement {
  public SOAPBodyElement addBodyElement(Name name) throws SOAPException;
  public SOAPBodyElement addDocument(org.w3c.dom.Document doc)throws SOAPException;
  public SOAPFault addFault() throws SOAPException;
  public SOAPFault addFault(Name faultcode, String faultString,
                            java.util.Locale local) throws SOAPException;
  public SOAPFault addFault(Name faultcode, String faultString) throws SOAPException;
  public SOAPFault getFault();
  public boolean hasFault();
}
```

Example usage of SOAPBody:

```
Name getBookPrice_Name = soapFactory.createName("getBookPrice","mh",
                         "http://www.Monson-Haefel.com/jwsbook/BookQuote");
Name isbnName = soapFactory.createName("isbn");

SOAPBody body = message.getSOAPBody();
SOAPBodyElement getBookPrice_Element =
                body.addBodyElement(getBookPrice_Name);
getBookPrice_Element.addChildElement( isbnName );
```

`SOAPBodyElement` extends `SOAPElement` and doesn't add any methods of its own.

```
package javax.xml.soap;
public interface SOAPBodyElement extends SOAPElement {}
```

A `SOAPBodyElement` can be added to a `SOAPBody` object using a `Name` object, as shown above.

The **`Text`** type is an extension of `Node` that represents literal text contained by an element or a comment.

```
package javax.xml.soap;

public interface Text extends Node, org.w3c.dom.Text {
    public boolean isComment();
}
```

You have to use the methods `getValue()` and `setValue()` defined by the `Node` supertype to retrieve and set the contents of a `Text` object.

```
SOAPElement isbn = getBookPrice.addChildElement("isbn");
...
isbn.addTextNode("0321146182");
...
Text textValue = isbn.getValue();
```

`SOAPException` is used to report errors encountered by the SOAP toolkit while attempting to perform some operation. Many of the methods that manufacture objects throw this exception because they affect the structure of the SOAP message. Remember: A `SOAPException` does not represent a SOAP fault generated by the receiver.

The **`SOAPFactory`** class is provided for the developer's convenience. It's a nice thing to have around because it allows you to create a `SOAPElement` independent of context. In other words, you can use it to create detached instances of the `SOAPElement` type.

```
package javax.xml.soap;
```

```
public abstract class SOAPFactory {

    …
    public abstract  SOAPElement createElement(Name name) throws SOAPException{…}
    public abstract  SOAPElement createElement(String localName) throws SOAPException{…}
    public abstract  SOAPElement createElement(String localName,
                                               String prefix, String uri)
       throws SOAPException{…}
    …
}
```

`SOAPFactory` can be used to construct portions of a SOAP message independent of a `SOAPMessage` object. For example, you might use it to construct a specialized header block in one module, which can be added to a SOAP document in some other module.

```
package com.jwsbook.saaj;
import javax.xml.soap.*;

public class SaajExample_6 {
  public static void main(String [] args) throws SOAPException {

    // Create SOAPMessage
    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage message = msgFactory.createMessage();
    SOAPHeader header = message.getSOAPHeader();

    // Create message-id header block
    SOAPElement msgId =  MessageIDHeader.createHeaderBlock();
    SOAPHeaderElement msgId_header =
             (SOAPHeaderElement)header.addChildElement(msgId);
    msgId_header.setActor("http://www.Monson-Haefel.com/logger");

    SaajOutputter.writeToScreen(message);
  }
}

package com.jwsbook.saaj;
import javax.xml.soap.*;

public class MessageIDHeader {
  public static SOAPElement createHeaderBlock() throws SOAPException{

    SOAPFactory factory = SOAPFactory.newInstance();
    SOAPElement msgId = factory.createElement("message-id","mi",
                        "http://www.Monson-Haefel.com/jwsbook/message-id");

    String messageid = new java.rmi.dgc.VMID().toString();

    msgId.addTextNode( messageid );
    return msgId;
  }
}
```

`SOAPFactory` makes it fairly easy to modularize the construction of SOAP messages, especially SOAP headers, which are specialized and often used across a variety of SOAP applications. For example, at Monson-Haefel Books almost every Web service requires that a `message-id` header block be included for logging purposes. A class like `MessageIDHeader` can be reused throughout the system to generate the `message-id` header block.

Working with SOAP Faults

In SAAJ, SOAP fault messages are constructed in basically the same way as a plain SOAP message, except the `SOAPBody` object contains a `SOAPFault` instead of a `SOAPBodyElement`. `SOAPFault` is actually a subtype of `SOAPBodyElement` that specializes the behavior to support the structure of a SOAP `Fault` element.

Every instance of the **`SOAPFault`** type is contained by a `SOAPBody` element. It describes an error generated by the receiver while processing a SOAP message.

```
package javax.xml.soap;
import java.util.Locale;

public interface SOAPFault extends SOAPBodyElement {

    public Detail addDetail() throws SOAPException;
    public Detail getDetail();
    public String getFaultActor();
    public String getFaultCode();
    public Name getFaultCodeAsName();
    public String getFaultString();
    public Locale getFaultStringLocale();
    public void setFaultActor(String faultActor) throws SOAPException;
    public void setFaultCode(String faultCode) throws SOAPException;
    public void setFaultCode(Name faultCode) throws SOAPException;
    public void setFaultString(String faultString) throws SOAPException;
    public void setFaultString(String faultString, Locale local) throws SOAPException;
}
```

As an example, imagine that the ultimate receiver of the BookQuote SOAP message determines that the ISBN number declared in the `Body` of an incoming message is invalid. The receiver will generate a SOAP Fault message and deliver it to the sender immediately before it in the message path.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring>The ISBN contains an invalid character(s)</faultstring>
      <faultactor>
        http://www.Monson-Haefel.omc/BookQuote_WebService
      </faultactor>
      <detail>
        <mh:InvalidIsbnFaultDetail
              xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
          <offending-value>19318224-D</offending-value>
          <conformance-rules>
              The first nine characters must be digits. The last
              character may be a digit or the letter 'X'. Case is not
              important.
          </conformance-rules>
        </mh:InvalidIsbnFaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

This fault message can be constructed fairly easily using SAAJ, as shown below:

```
package com.jwsbook.saaj;
import javax.xml.soap.*;

public class SaajExample_7 {
  public static void main(String [] args) throws SOAPException {
    // Create SOAPMessage
    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage message = msgFactory.createMessage();
    message.getSOAPHeader().detachNode();

    // Create Fault message
    SOAPBody body = message.getSOAPBody();

    SOAPFault fault = body.addFault();
    fault.setFaultCode("soap:Client");
    fault.setFaultString("The ISBN contains an invalid character(s)");
    fault.setFaultActor("http://www.Monson-Haefel.org/BookQuote_WebService");

    Detail detail = fault.addDetail();
    SOAPFactory soapFactory = SOAPFactory.newInstance();
    Name errorName = soapFactory.createName(
                     "InvalidIsbnFaultDetail","mh",
                     "http://www.Monson-Haefel.com/jwsbook/BookQuote");

    DetailEntry detailEntry = detail.addDetailEntry(errorName);
```

```
    SOAPElement offendingValue =
      detailEntry.addChildElement("offending-value");
    offendingValue.addTextNode("19318224-D");
    SOAPElement conformanceRules =
      detailEntry.addChildElement("conformance-rules");
    conformanceRules.addTextNode(
       "The first nine characters must be digits.  The last character "+
       "may be a digit or the letter 'X'. Case is not important.");

    SaajOutputter.writeToScreen(message);
  }
}
```

Sending SOAP Messages with SAAJ

Usually SAAJ is used in combination with JAX-RPC, but it's not dependent on JAX-RPC. SAAJ comes with its own, fairly simple and limited, message-delivery system, which is a part of the basic API. Using SAAJ you can exchange Request/Response-style SOAP messages with a Web service over HTTP. You simply create a `SOAPConnection` and send the message.

```
package com.jwsbook.saaj;
import javax.xml.soap.*;
import java.net.URL;
import java.io.FileInputStream;

public class SaajExample_8 {
  public static void main(String [] args) throws SOAPException,
    java.io.IOException{

    // Build a SOAPMessage from a file
    MessageFactory msgFactory = MessageFactory.newInstance();
    MimeHeaders mimeHeaders = new MimeHeaders();
    mimeHeaders.addHeader("Content-Type","text/xml; charset=UTF-8");
    FileInputStream file = new FileInputStream("soap.xml");
    SOAPMessage requestMsg = msgFactory.createMessage(mimeHeaders, file);
    file.close();

    // Send the SOAP message to the BookQuote Web service
    SOAPConnectionFactory conFactry = SOAPConnectionFactory.newInstance();
    SOAPConnection connection = conFactry.createConnection();
    URL url = new URL(args[0]);
    SOAPMessage replyMsg =connection.call(requestMsg, url);

    // Print out the reply message
    SaajOutputter.writeToScreen(replyMsg);
  }
}
```

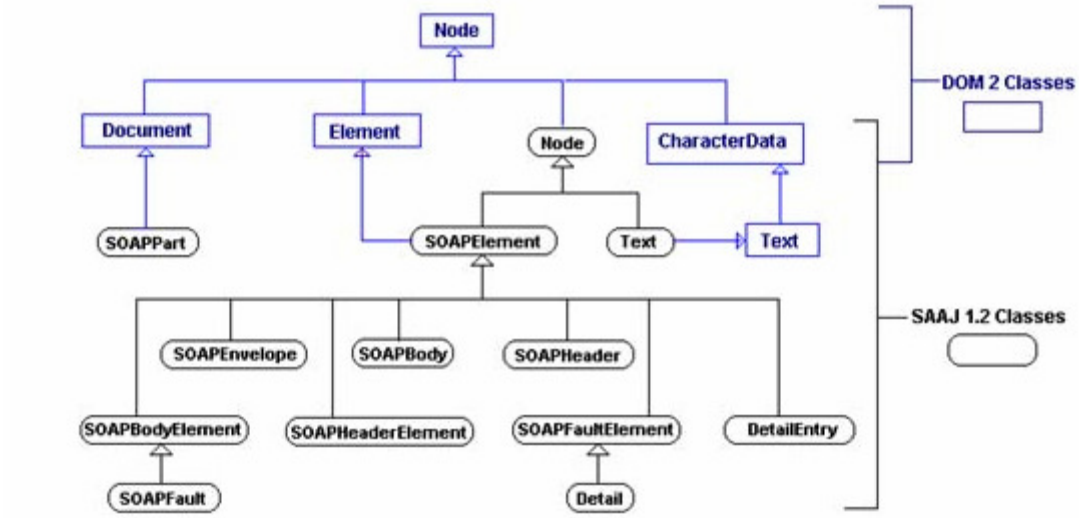
The `SOAPConnection` object creates an HTTP connection to the specified URL and sends the SOAP message to the Web service as an HTTP POST message. The HTTP reply that's sent from the Web service back to the `SOAPConnection` object will contain a SOAP message, a SOAP fault, or an HTTP error code; in the last case a `SOAPException` is thrown. The SOAP reply message is returned by the `SOAPConnection.call()` method as a `SOAPMessage` object, which can be accessed using the SAAJ API.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns0="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
  <soap:Body>
    <ns0:getBookPriceResponse>
      <result>24.99</result>
    </ns0:getBookPriceResponse>
  </soap:Body>
</soap:Envelope>
```

SAAJ 1.2 and DOM2

SAAJ 1.2 redefines the API so that it's an extension of the DOM 2 Java object model. SAAJ is now a lot more powerful, because you can use it to create and manipulate SOAP messages, but you can also take

advantage of low-level DOM 2 functionality as the need arises. In addition, you can import `Nodes` from a DOM 2 document into a SOAP message, which is useful when working with JAX-RPC message handlers and Document/Literal payloads.
In most cases interfaces were simply redefined to extend DOM 2 interface types like `Document`, `Element`, and `Text`.




```
package com.jwsbook.saaj;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import javax.xml.soap.*;

public class SaajExample_9 {
  public static void main(String [] args) throws Exception{

    // Read an XML document from a file into a DOM tree using JAXP.
    String filePath = args[0];
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder parser = factory.newDocumentBuilder();
    Document xmlDoc = parser.parse(filePath);

    // Create a SAAJ SOAPMessage object. Get the SOAPBody and SOAPPart.
    MessageFactory msgFactory = MessageFactory.newInstance();
    SOAPMessage message = msgFactory.createMessage();
    SOAPPart soapPart = message.getSOAPPart();
    SOAPBody soapBody = soapPart.getEnvelope().getBody();

    // Import the root element and append it to the body of the SOAP message.
    Node elementCopy =
      soapPart.importNode(xmlDoc.getDocumentElement(), true);
    soapBody.appendChild(elementCopy);
// Alternatively, soapBody.addDocument(xmlDoc);


    // Output SOAP message.
    SaajOutputter.writeToScreen(message);
  }
}
```

`SaajExample_9` reads an XML document from a file and has a DOM provider parse it into a `Document` object. It also creates a new `SOAPMessage` object. The application imports the root of the XML document into the `SOAPPart` (using the `Document.importNode()` method) and subsequently appends the imported copy to the `SOAPBody` (using the `Node.appendNode()` method).

SAAJ Attachments

[RMH – Appendix F]

JAXR 1.0

[RMH – Chapters 16-19]
6.1 Describe the function of JAXR in Web service architectural model, the two basic levels of business registry functionality supported by JAXR, and the function of the basic JAXR business objects and how they map to the UDDI data structures.
6.2 Use JAXR to connect to a UDDI business registry, execute queries to locate services that meet specific requirements, and publish or update information about a business service.

Getting Started with JAXR (6.1)

JAXR (Java API for XML Registries) is a client-side API for accessing different kinds of XML-based business registries, but is used predominantly for UDDI and ebXML registries. JAXR provides a vendor-neutral API for accessing any ebXML or UDDI registry. J2EE vendors will provide their own implementations of the JAXR API for accessing ebXML, UDDI, or other types of XML registry systems.
The JAXR API has two conformance levels: **Level 0** and **Level 1**. Level 1 is a richer API that is designed for accessing ebXML registries**. Level 0 provides fewer features and is intended for UDDI registries**, which are less flexible than ebXML but more popular. The WS-I Basic Profile 1.0 sanctions the use of UDDI, but not ebXML.
JAXR comes in **two interdependent packages**:

1. The Query and Life Cycle API (**`javax.xml.registry`**): The Query and Life Cycle API supports the UDDI Inquiry and Publishing APIs respectively.
2. the Information Model (**`javax.xml.registry.infomodel`**): The Information Model (infomodel for short) provides business- and technical-object views that correspond to UDDI data structures (`businessEntity`, `bindingTemplate`, `tModel`, and so on).

Uses of JAXR: If you need to build a UDDI browsing tool, JAXR is a great choice for supporting the actual SOAP communications between the browser and the UDDI registry. JAXR is useful for implementing failover mechanisms, to handle gracefully the problem of a Web service endpoint becoming unavailable. The JAXR API can be used to look up the Web service in UDDI and determine whether an alternative access point is offered.
A lot of the grunt work of communicating with a UDDI registry is hidden behind JAXR's APIs and infomodel. Essentially, you work with business objects and technical objects, which represent organizations, services, classification systems, and so on, without having to think in terms of SOAP messaging.

Connecting to a UDDI Registry

Connecting to a UDDI registry with JAXR requires that you obtain a `ConnectionFactory`, configure its connection properties, and request a `Connection` object. J2EE application servers may take care of some of this work, such as configuration, automatically—depending on the vendor. In a standalone application you have to configure the `ConnectionFactory` in your code.

```
package com.rwatsh.jaxr;
import javax.xml.registry.ConnectionFactory;
import javax.xml.registry.Connection;
import javax.xml.registry.RegistryService;
import javax.xml.registry.BusinessLifeCycleManager;
import javax.xml.registry.BusinessQueryManager;
import javax.xml.registry.infomodel.Organization;
import javax.xml.registry.infomodel.InternationalString;
import javax.xml.registry.CapabilityProfile;
import javax.xml.registry.BulkResponse;
import javax.xml.registry.JAXRResponse;
import javax.xml.registry.JAXRException;
import javax.xml.registry.infomodel.Organization;
import javax.xml.registry.infomodel.User;
```



```
import javax.xml.registry.infomodel.PersonName;
import javax.xml.registry.infomodel.EmailAddress;
import javax.xml.registry.infomodel.TelephoneNumber;
import javax.xml.registry.infomodel.PostalAddress;
import javax.xml.registry.infomodel.InternationalString;
import javax.xml.registry.infomodel.LocalizedString;
import javax.xml.registry.infomodel.Slot;
import java.net.PasswordAuthentication;
import java.util.Iterator;
import java.util.Properties;
import java.util.Set;
import java.util.Collection;
import java.util.HashSet;
import java.util.Locale;

public class JaxrExample_1 {
  private static String companyName = "Aditi";
  public static void main(String [] args) throws JAXRException {
String userName = "testuser";
    String password = "testuser";

    // Create a ConnectionFactory.
    ConnectionFactory factory = ConnectionFactory.newInstance();

    // Configure the ConnectionFactory.
    Properties props = new Properties();
    props.setProperty("javax.xml.registry.lifeCycleManagerURL",
                      "http://localhost:8080/RegistryServer/");
    props.setProperty("javax.xml.registry.queryManagerURL",
                      "http://localhost:8080/RegistryServer/");
    props.setProperty("javax.xml.registry.security.authenticationMethod",
                    "UDDI_GET_AUTHTOKEN");
    factory.setProperties(props);

    // Connect to UDDI test registry.
    Connection connection = factory.createConnection();
    System.out.println("Created connection to registry!");
    // Get registry service and managers
    RegistryService rs = connection.getRegistryService();
    // Get the capability profile
    CapabilityProfile capabilityProfile =  rs.getCapabilityProfile();
    if (capabilityProfile.getCapabilityLevel() == 0) {
    System.out.println("Capability Level 0, Business Focused API");
    }
    // Get manager capabilities from registry service
    BusinessQueryManager bqm = rs.getBusinessQueryManager();
    BusinessLifeCycleManager blcm = rs.getBusinessLifeCycleManager();
    System.out.println("Got registry service, query manager and lifecycle manager");

    PasswordAuthentication passwdAuth =
         new PasswordAuthentication(userName, password.toCharArray());
    Set creds = new HashSet();
    creds.add(passwdAuth);
    connection.setCredentials(creds);
    // Create an Organization object and assign it a name.
     Organization myOrganization =
                  blcm.createOrganization(companyName);


    // Create a new User object and add it to the Organization.
     User contact = blcm.createUser();
     myOrganization.addUser(contact);

     // Set the User's name.
     PersonName contactName = blcm.createPersonName(
                                                "Stanley Kubrick");
     contact.setPersonName(contactName);

     // Set the English and French descriptions of the User.
     LocalizedString  english = blcm.createLocalizedString(
                                Locale.ENGLISH,
```



```
                                "Web Services Technical Support");
    LocalizedString  french = blcm.createLocalizedString(
                                 Locale.FRENCH,
                                "Web Services le Soutien de Technial");
    InternationalString descript =
      blcm.createInternationalString();
    descript.addLocalizedString(english);
    descript.addLocalizedString(french);
    contact.setDescription(descript);

    // Set the User's e-mail addresses.
    EmailAddress email = blcm.createEmailAddress("Stanley.Kubrick@"+companyName+".com");
    Collection emails = new HashSet();
    emails.add(email);
    contact.setEmailAddresses(emails);

    // Set the User's telephone numbers.
    TelephoneNumber phone = blcm.createTelephoneNumber();
    phone.setNumber("01-555-222-4000");
    phone.setType("Voice");
    Collection phones = new HashSet();
    phones.add(phone);
    contact.setTelephoneNumbers(phones);

    // Set the User's postal address.
    PostalAddress address = blcm.createPostalAddress(
                                       "2001","Odyssey Ave.",
                                       "Galaxy","CA","USA","91223",
                                       "TechSupport");
    Collection addresses = new HashSet();
    addresses.add(address);
    contact.setPostalAddresses(addresses);

   connection.setSynchronous(true);

   // Save the Organization to the UDDI directory.
   Set organizationSet = new HashSet();
   organizationSet.add(myOrganization);
   BulkResponse response =
                blcm.saveOrganizations(organizationSet);

   // Check for registry exceptions.
   doExceptions(response);

   // Close connection.
   connection.close();
  }
  public static void doExceptions(BulkResponse rspns) throws JAXRException {
    if(rspns.getStatus()==JAXRResponse.STATUS_SUCCESS){
      System.out.println("\nProgram Complete: No problems reported!");
    }else {
      Iterator exceptions = rspns.getExceptions().iterator();
      while(exceptions.hasNext()){
        Exception je = (Exception)exceptions.next();
        System.out.println("\n***** BulkResponse Exceptions *****\n\n");
        je.printStackTrace();
        System.out.println("\n*********************************");
      }
    }
  }
}
```

Following is the o/p:

```
Created connection to registry!
Capability Level 0, Business Focused API
Got registry service, query manager and lifecycle manager
Jul 6, 2006 5:08:00 PM com.sun.xml.registry.uddi.UDDIMapper
postalAddressEquivalence2Address
WARNING: JAXR.UDDI.104: No PostalAddressMapping
```

```
Program Complete: No problems reported!
```

Different vendors offer their own implementations of the `ConnectionFactory`, so if you're not using the JAXR RI, you'll need to specify the following system property, in any of several ways.

```
javax.xml.registry.ConnectionFactoryClass =vendors_connectionfactory_class
```

To use the JAXR `ConnectionFactory` you have to configure its Inquiry and Publishing URLs so that the JAXR provider knows where to send Inquiry and Publishing SOAP messages. Table below lists the JAXR standard configuration properties:

JAXR Property Name	*Description*
`javax.xml.registry.`**`lifeCycleManagerURL`**	The URL of the UDDI's Publishing Web service.
`javax.xml.registry.`**`queryManagerURL`**	The URL of the UDDI's Inquiry Web service.
`javax.xml.registry.security.`**`authenticationMethod`**	The method of authentication used—the value allowed for UDDI is `UDDI_GET_AUTHTOKEN`
`javax.xml.registry.uddi.`**`maxRows`**	The maximum number of rows to be returned by find operations.
`javax.xml.registry.`**`postalAddressScheme`**	The id of the `ClassificationScheme` (a `tModel`) that is used as the default address scheme for this connection

Once you've configured the `ConnectionFactory`, you can use it to create a `Connection` object, which represents a virtual connection to the UDDI directory. UDDI SOAP messages use HTTP for inquiry and HTTPS for publishing—the inquiry SOAP messages are sent to the Inquiry URL while the publishing messages are sent to the Publishing URL. You can also create a **`FederatedConnection`**, which can be used to query several different registries at the same time. To create a `FederatedConnection` you would first create two or more regular JAXR connections as above, then use those `Connection` objects to create a `Federated Connection`.

```
Set connections = HashSet();
Connection con1 = factory1.createConnection();
connections.add(con1);
Connection con2 = factory2.createConnection();
connections.add(con2);
FederatedConnection federatedCon = factory3.createFederatedConnection( connections);
```

Once you have a FederatedConnection, search operations will be executed against all of the registries represented by the FederatedConnection—you'll be searching multiple registries at once.

In order to execute publishing operations, which include adding, updating, and removing information in the UDDI directory, you'll need to authenticate (log in) to the registry. The `PasswordAuthentication` class is part of the Java Networking package (`java.net`). It provides a wrapper for a user name and password. Private UDDI registries may require a different kind of authentication. For example, if a registry mandates use of X.509 certificates, you must pass an instance of the `javax.security.auth.x500.X500PrivateCredential` class, which is part of the JAAS API.

```
// Authenticate using X.509 certificate
KeyStore ks = KeyStore.getInstance("JKS");
X509Certificate certificate = (X509Certificate)
                keyStore.getCertificate( userName );
PrivateKey privateKey = (PrivateKey)
                keyStore.getKey( userName, password.toCharArray());
X500PrivateCredential x500Credential =
                new X500PrivateCredential(certificate, privateKey);
Set credentials = new HashSet();
credentials.add(x500Credential);
connection.setCredentials(credentials);
```

The following code snippet shows how a J2EE component would access a JAXR ConnectionFactory from the JNDI ENC.

```
InitialContext jndiEnc = new InitialContext();
ConnectionFactory factory = (ConnectionFactory)
   jndiEnc.lookup("java:comp/env/jaxr/UddiRegistry");
Connection connection = factory.createConnection();
...
```

To use JAXR in J2EE you'll have to configure it as a resource in your component's deployment descriptor.

```
<web-app>
    <display-name>SomeTypeOfJSE</display-name>
    ...
    <resource-ref>
         <res-ref-name>jaxr/UddiRegistry</res-ref-name>
         <res-type>javax.xml.registry.ConnectionFactory</res-type>
         <res-auth>Container</res-auth>
    </resource-ref>
```

At deployment time the deployer will map the configuration properties for authentication and access to the Publishing and Inquiry APIs. The J2EE container will authenticate you automatically each time you use JAXR to access the Publishing API of a UDDI registry.

Once the connection is established and you are authenticated to the UDDI registry, you can obtain the **`RegistryService`** object, which represents the UDDI registry. You can then use the `RegistryService` object to get a reference to the **`BusinessLifeCycleManager`** object, which is used to add, modify, and delete information in the registry.

The **`RegistryService`** is the principal interface in the JAXR API. The `RegistryService` represents the entire UDDI registry. The `RegistryService` defines methods you can use to access different capabilities offered by a registry. Some of these methods can be used with UDDI and some cannot.

```
package javax.xml.registry;
import java.util.*;
import javax.xml.registry.infomodel.*;

public interface RegistryService {
   // Returns the CapabilityProfile (JAXR version and level 0/1) for the JAXR provider.
   CapabilityProfile getCapabilityProfile() throws JAXRException;

   // Returns the BusinessLifeCycleManager object - used for adding/updating to registry.
   BusinessLifeCycleManager getBusinessLifeCycleManager()
     throws JAXRException;

   // Returns the BusinessQueryManager object - used for querying the registry.
   BusinessQueryManager getBusinessQueryManager() throws JAXRException;

   // This method is not applicable to UDDI.
   BulkResponse getBulkResponse(String requestId)
     throws InvalidRequestException, JAXRException;

   // Get the default postal scheme for PostalAddress.
   public ClassificationScheme getDefaultPostalScheme()  throws JAXRException;

   // This method sends an XML request in a registry-specific format.
   public String makeRegistrySpecificRequest(String request)
     throws JAXRException;
}
```

Many of the methods defined by **`LifeCycleManager`** are used to manufacture in-memory instances of JAXR information objects. The methods defined by the **`BusinessLifeCycleManager`** subtype are used to save, update, and delete information in the UDDI registry. The `LifeCycleManager` interface defines factory methods for creating instances of every type of information object, from `Organization` (createOrganization in the above example) to `PersonName`. When an information object (such as an `Organization`, `Service`, or `User`) is created by one of the `LifeCycleManager` factory methods, it is not automatically added to the UDDI registry; the factory methods simply instantiate the object type. An information object is not saved to the registry until you use the appropriate **`saveXXX()`** operation defined by the `BusinessLifeCycleManager` interface thus only one call to the registry is made, when the `saveXXX()` method is invoked to save the entire object graph created using JAXR APIs.

```
package javax.xml.registry;
import java.util.*;
import javax.xml.registry.infomodel.*;
```



```
public interface LifeCycleManager {

  // Create primary information objects.
  public Organization createOrganization(String name)
   throws JAXRException;
  public Service createService(String name) throws JAXRException;
  public ServiceBinding createServiceBinding() throws JAXRException;
  public Concept createConcept(RegistryObject parent,
                               InternationalString name,
                               String value) throws JAXRException;
  public Association createAssociation(RegistryObject targetObject,
                                       Concept associationType)
    throws JAXRException;

  // Create demographic information objects.
  public InternationalString createInternationalString(String s)
    throws JAXRException;
  public LocalizedString createLocalizedString(Locale l,String s)
    throws JAXRException;
  public User createUser() throws JAXRException;
  public PersonName createPersonName(String fullName)
   throws JAXRException;
  public EmailAddress createEmailAddress(String address)
   throws JAXRException;
  public TelephoneNumber createTelephoneNumber() throws JAXRException;
  public PostalAddress createPostalAddress(String streetNumber,
                                           String street,
                                           String city,
                                           String stateOrProvince,
                                           String country,
                                           String postalCode,
                                           String type)
   throws JAXRException;

  // Creates taxonomy information objects.
  public Classification createClassification(
                                        ClassificationScheme scheme,
                                         String name,String value)
   throws JAXRException;
  public ClassificationScheme createClassificationScheme(
                                                    String name,
                                                    String description)
   throws JAXRException, InvalidRequestException;
  public ExternalIdentifier createExternalIdentifier(
                           ClassificationScheme identificationScheme,
                           String name,String value)
   throws JAXRException;
  public ExternalLink createExternalLink(String externalURI,
                                         String description)
   throws JAXRException;
  public Slot createSlot(String name,String value,String slotType)
   throws JAXRException;
  public SpecificationLink createSpecificationLink()
   throws JAXRException;

  // General-purpose create, save, and delete methods.
  public Object createObject(String interfaceName)
   throws JAXRException, InvalidRequestException,
          UnsupportedCapabilityException;
  BulkResponse saveObjects(Collection objects) throws JAXRException;
  BulkResponse deleteObjects(Collection keys) throws JAXRException;

  // Get a reference to the Registry Service - that created this LifeCycleManager
instance.
  RegistryService getRegistryService() throws JAXRException;
}
```


Following is the BusinessLifeCycleManager interface: The methods provided by this interface can be grouped into three general categories by their use: to add or update information objects; to delete information objects; to confirm and undo associations.

```
package javax.xml.registry;
import java.util.*;
import javax.xml.registry.infomodel.*;

public interface BusinessLifeCycleManager extends LifeCycleManager {

   // Add or update information objects in the UDDI registry.
   BulkResponse saveOrganizations(Collection organizations)
     throws JAXRException;
   BulkResponse saveServices(Collection services) throws JAXRException;
   BulkResponse saveServiceBindings(Collection bindings)
    throws JAXRException;
   BulkResponse saveConcepts(Collection concepts) throws JAXRException;
   BulkResponse saveClassificationSchemes(Collection schemes)
     throws JAXRException;
   BulkResponse saveAssociations(Collection associations,
                                 boolean replace)
     throws JAXRException;

   // Delete information objects from a UDDI registry.
   BulkResponse deleteOrganizations(Collection organizationKeys)
     throws JAXRException;
   BulkResponse deleteServices(Collection serviceKeys)
     throws JAXRException;
   BulkResponse deleteServiceBindings(Collection bindingKeys)
     throws JAXRException;
   BulkResponse deleteConcepts(Collection conceptKeys)
     throws JAXRException;
   BulkResponse deleteClassificationSchemes(Collection schemeKeys)
     throws JAXRException;
   BulkResponse deleteAssociations(Collection associationKeys)
     throws JAXRException;

   // Confirm/undo an association.
   public void confirmAssociation(Association assoc)
     throws JAXRException, InvalidRequestException;
   public void unConfirmAssociation(Association assoc)
     throws JAXRException, InvalidRequestException;
}
```

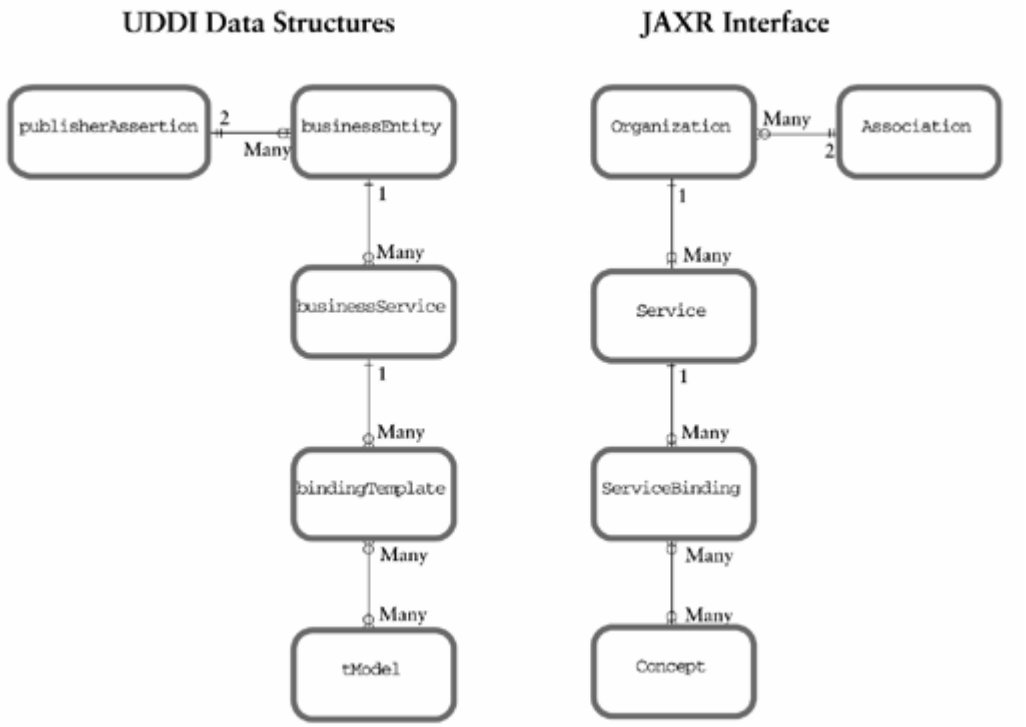
All of the `BusinessLifeCycleManager` save and delete methods (`saveOrganizations()`, `deleteOrganizations()`, and so on) return the **`BulkResponse`** type. So do most of the `BusinessQueryManager` type's `findXXX()` methods. The `BulkResponse` type is designed to **carry a `Collection` of arbitrary values, the type of which depends on the method invoked**. For example, the `BulkResponse` returned by the `BusinessQueryManager.findOrganizations()` method will contain a `Collection` of `Organization` objects. The `BulkResponse` returned by the `BusinessLifeCycleManager.saveOrganizations()` method, on the other hand, will contain a `Collection` of `Key` objects, one UUID key for each `Organization` that was saved.

```
package javax.xml.registry;
import java.util.*;
import javax.xml.registry.infomodel.*;

public interface BulkResponse extends JAXRResponse {
   // Collection of objects returned as a response of a bulk operation.
   public Collection getCollection() throws JAXRException;

   // Collection of RegistryException instances for partial commit.
   public Collection getExceptions() throws JAXRException;

   // Determines this is a partial response due to large result set - typically when a
   // BusinessQueryManager's findXXX() method returns a large result set.
   public boolean isPartialResponse() throws JAXRException;
}
```

The `getExceptions()` method returns JAXR `RegistryException` types when the UDDI registry itself encounters an error. The **`RegistryException`** types contain codes that correspond to the UDDI SOAP fault message types. There are three `RegistryException` subtypes: **`SaveException`**, **`DeleteException`**, and **`FindException`**. You can use the `getStatus()` method of the `BulkResponse` (declared by its supertype `JAXRResponse`) to determine whether the method performed successfully (possible values for UDDI registry are: JAXRResponse.STATUS_SUCCESS and JARResponse.STATUS_FAILURE).

The JAXR API has two mechanisms for returning exceptions. A method can throw them like any other exception, or return them in a `BulkResponse` object. Most of the JAXR `BusinessLifeCycleManager` and `BusinessQueryManager` return multiple results from a method. For example, `saveOrganizations()` returns a `Collection` of UUID `Key` objects, one for each `Organization` saved. When a JAXR Inquiry or Publishing method returns multiple results, it may return UDDI registry exceptions in a `Collection`, rather than just throw a single exception. These are accessed by calling `BulkResponse.getExceptions()`. If, however, an abnormal condition is encountered by the client-side runtime of the JAXR implementation (for example, a `NullPointerException`), the method will throw a `JAXRException` instead of returning it in the `BulkResponse` object. In a nutshell, **only SOAP faults generated by the UDDI registry are returned as exceptions in a `BulkResponse` object**. If the method returns only a single value (something other than `BulkResponse`), then all exceptions, even UDDI registry exceptions, are thrown directly from the method.

JAXR Business Objects (6.1)

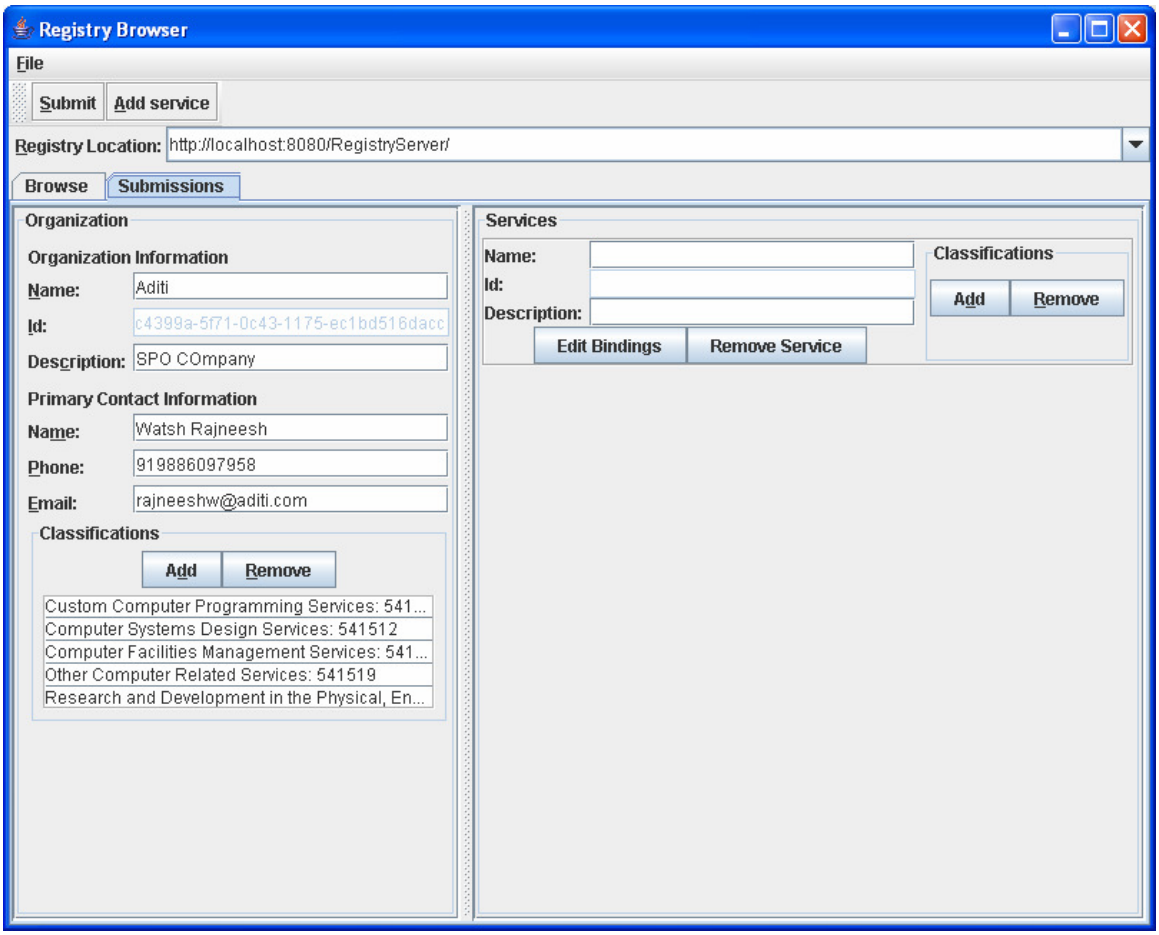


Most of the interfaces defined in the JAXR information model extend the **`javax.xml.registry.infomodel.RegistryObject`** interface. The `RegistryObject` methods that can be used with UDDI are typically get and set methods for accessing properties common to information objects, such as UUID, name, description, classification, external identifiers, and URLs.

```
package javax.xml.registry.infomodel;
import java.io.*;
import java.net.*;
import javax.xml.registry.*;
```

```
import java.util.*;

public interface RegistryObject extends ExtensibleObject  {

   //The universally unique ID (UUID) for this object.
   public Key getKey() throws JAXRException;
   public void setKey(Key key) throws JAXRException;

   //The name of this object.
   public InternationalString getName() throws JAXRException;
   public void setName(InternationalString name) throws JAXRException;

   //The text description for this object.
   public InternationalString getDescription() throws JAXRException;
   public void setDescription(InternationalString description)
    throws JAXRException;

   //The collection of Classifications for this object.
   public void setClassifications(Collection classifications)
    throws JAXRException;
   public Collection getClassifications() throws JAXRException;

   //The collection of ExternalIdentifiers for this object.
   public void setExternalIdentifiers(Collection externalIdentifiers)
    throws JAXRException;
   public Collection getExternalIdentifiers() throws JAXRException;

   //The collection of ExternalLinks for this object.
   public void setExternalLinks(Collection externalLinks)
    throws JAXRException;
   public Collection getExternalLinks() throws JAXRException;

   //The collection of Associations for this object.
   public void setAssociations(Collection associations)
    throws JAXRException;
   public Collection getAssociations() throws JAXRException;

   //The Organization that submitted this object.
   public Organization getSubmittingOrganization() throws JAXRException;

   //The LifeCycleManager that created this object.
   public LifeCycleManager getLifeCycleManager() throws JAXRException;
}
```

Note: In order to run the example code, install J2EE SDK and it comes bundled with the reference implementation of the UDDI Registry (implemented as a RegistryServer servlet). By default this may not be available in which case, download the JWSDP 1.5 and install it integrated with your Sun AS or Tomcat server. In case you need to integrate with some other instance of Sun or Tomcat AS post then installation then refer to the jwsdp/shared/bin path and run the applicable jwsdpon<yourserver>.bat file. The best way to verify that your installation has worked is to see the list of web applications hosted by the server (using AdminConsole of SunAS for example) and if there is one RegistryServer listed then your integration was successful and you can access the UDDI v2 registry JSE with the URL http://localhost:<port>/RegistryServer/ (the end slash in the URL is important) for both the publish and query webservice. Theres a Registry Browser installed with the JWSDP 1.5 installation which can be used to test the connection with the UDDI registry service.



The first time you save `RegistryObjects` of certain types (for example, `Organization` and `Concept`) to a UDDI registry, they will be assigned a UUID (Universally Unique Identifier) key.The JAXR object types that have a UUID key in UDDI are:

1. Organization
2. Service
3. ServiceBinding
4. Concept
5. ClassificationScheme

The `Concept` and `ClassificationScheme` objects, which represent UDDI `tModels`, prefix their UUID values with the characters **`uuid:`**.

JAXR requires every subtype of `RegistryObject` to return a key value when `getKey()` is invoked, but not all have genuine UUID keys they can return so the following JAXR objects return faux keys:

1. Association
2. Classification
3. ExternalIdentifier
4. ExternalLink
5. SpecificationLink
6. User

The **`Organization`** object is the root of a UDDI business entry. It represents an instance of the UDDI `businessEntity` data structure. An `Organization` object contains a business name and description, contact information, industry categorizations and identification values, and a collection of zero or more `BusinessService` objects, each of which represents some type of electronic service—usually a Web

service or Web site. The `Organization` interface defines about two dozen methods for adding information to and removing information from an `Organization` object.

```
package javax.xml.registry.infomodel;
import javax.xml.registry.*;
import java.util.*;

public interface Organization extends RegistryObject {

    // User methods.
    public void addUser(User user) throws JAXRException;
    public void addUsers(Collection users) throws JAXRException;
    public void removeUser(User user) throws JAXRException;
    public void removeUsers(Collection users) throws JAXRException;
    public Collection getUsers() throws JAXRException;
    public User getPrimaryContact() throws JAXRException;
    public void setPrimaryContact(User primaryContact)
     throws JAXRException;

    // Service methods.
    public void addService(Service service) throws JAXRException;
    public void addServices(Collection services) throws JAXRException;
    public void removeService(Service service) throws JAXRException;
    public void removeServices(Collection services)
     throws JAXRException;
    public Collection getServices() throws JAXRException;
}
```

User methods allow you to add, access, and remove **`javax.xml.registry.infomodel.User`** information objects to an `Organization` object. A `User` object represents a UDDI **`contacts`** element contained by a UDDI `businessEntity` data structure. The service methods enable you to add, access, and remove **`javax.xml.registry.infomodel.Service`** objects. A `Service` object represents a UDDI **`businessService`** data structure.

```
<schema targetNamespace="urn:uddi-org:api_v2"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2"
            version="2.03" id="uddi">
...
<element name="businessEntity" type="uddi:businessEntity"/>
<complexType name="businessEntity">
   <sequence>
      <element ref="uddi:discoveryURLs" minOccurs="0"/>
      <element ref="uddi:name" maxOccurs="unbounded"/>
      <element ref="uddi:description" minOccurs="0"
                                      maxOccurs="unbounded"/>
      <element ref="uddi:contacts" minOccurs="0"/>
      <element ref="uddi:businessServices" minOccurs="0"/>
      <element ref="uddi:identifierBag" minOccurs="0"/>
      <element ref="uddi:categoryBag" minOccurs="0"/>
   </sequence>
   <attribute name="businessKey" type="uddi:businessKey"
                  use="required"/>
   <attribute name="operator" type="string" use="optional"/>
   <attribute name="authorizedName" type="string" use="optional"/>
</complexType>
```

The **`externalLinks`** property is a collection of `ExternalLink` objects, which wrap around URLs that point to information (resources) that is associated with the `Organization`, but outside the registry. UDDI operators are required to provide at least one `ExternalLink`, which points to the raw XML data that describes the underlying `businessEntity`. Methods for adding, modifying, removing, and accessing `ExternalLink` objects are defined by `Organization`'s supertype, the `RegistryObject` interface.

```
public interface RegistryObject extends ExtensibleObject  {
   ...
   //The collection of ExternalLinks for this object.
   public void addExternalLink(ExternalLink externalLink) throws JAXRException;
   public void addExternalLinks(Collection externalLinks) throws JAXRException;
```



```
   public void removeExternalLink(ExternalLink externalLink) throws JAXRException;
   public void removeExternalLinks(Collection externalLinks) throws JAXRException;
   public void setExternalLinks(Collection externalLinks) throws JAXRException;
   public Collection getExternalLinks() throws JAXRException;
   //The name of this object.
   public InternationalString getName() throws JAXRException;
   public void setName(InternationalString name) throws JAXRException;

   //The text description for this object.
   public InternationalString getDescription() throws JAXRException;
   public void setDescription(InternationalString description) throws JAXRException;

   //The collection of Classifications for this object.
   public void setClassifications(Collection classifications) throws JAXRException;
   public Collection getClassifications() throws JAXRException;
   public void addClassification(Classification classification) throws JAXRException;
   public void addClassifications(Collection classifications) throws JAXRException;
   public void removeClassification(Classification classification) throws JAXRException;
   public void removeClassifications(Collection classifications) throws JAXRException;

   ...
}
```

A UDDI `Organization` object may refer to basically two types of `ExternalLink` objects: a required **`"businessEntity"`** link and any number of optional **`"businessEntityExt"`** links. A `"businessEntity"` **`ExternalLink`** is generated every time you update an information object in the UDDI registry. You cannot remove or modify this `ExternalLink`—only the UDDI registry can alter its value. You can however, add other `ExternalLink` objects, labeled `"businessEntityExt"`, which point to non-UDDI documents that you believe should be associated with your `Organization`. For example, you might define an `ExternalLink` that points to a PDF document that describes your business. You can add a `"businessEntityExt"` external link to an organization by creating an `ExternalLink` object, then adding it to the collection of external links.

```
ExternalLink myLink = lifeCycleManager.createExternalLink(
    "http://www.Monson-Haefel.com/jwsbook/MonsonHaefelBooks.doc","");
organization.addExternalLink( myLink );
```

You cannot remove, modify, or replace the `ExternalLink` labeled `"businessEntity"` because it is maintained by the UDDI registry—attempts to modify it will be ignored or generate an exception.

An `Organization` may have multiple names and descriptions in different languages. The **`InternationalString`** type, which is the parameter or return type of the methods that operate on names and descriptions, can represent several different language-specific text values.

An organization may have zero or more **`java.xml.registry.infomodel.User`** objects, which represent people that can be contacted by postal mail, phone, or e-mail. These people might provide technical support, sales information, administrative services, or any other service an organization wants to make accessible. In a UDDI registry, the collection of `User` objects is represented by a **`contacts`** element.

```
package javax.xml.registry.infomodel;
import java.net.*;
import javax.xml.registry.*;
import java.util.*;

public interface User extends RegistryObject {
   // Gets the Organization that this User is affiliated with.
   Organization getOrganization() throws JAXRException;

   // Get|Set the name of this User.
   public PersonName getPersonName() throws JAXRException;
   public void setPersonName(PersonName personName)
    throws JAXRException;

   // Get|Set the postal address for this User.
   public Collection getPostalAddresses() throws JAXRException;
   public void setPostalAddresses(Collection addresses)
```



```
    throws JAXRException;

   // Get|Set telephone numbers of the specified type.
   public Collection getTelephoneNumbers(String phoneType)
     throws JAXRException;
   public void setTelephoneNumbers(Collection phoneNumbers)
     throws JAXRException;

   // Get|Set the e-mail addresses for this User.
   public Collection getEmailAddresses() throws JAXRException;
   public void setEmailAddresses(Collection emailAddresses)
     throws JAXRException;

   // Get|Set the type for this User.
   public String getType() throws JAXRException;
   public void setType(String type) throws JAXRException;
}
```

The `personName` property of the `User` type doesn't support the concept of multiple language-specific names. [See the main example of the previous section].

The `RegistryObject` defines six methods for setting, getting, adding, and removing one or more **`Classification`** objects. In registry systems entries can also be classified with some type of taxonomy or system of categorization. There are three standard classification systems used in UDDI to categorize industries (NAICS), products and services (UNSPSC), and geographic locations (ISO 3166).

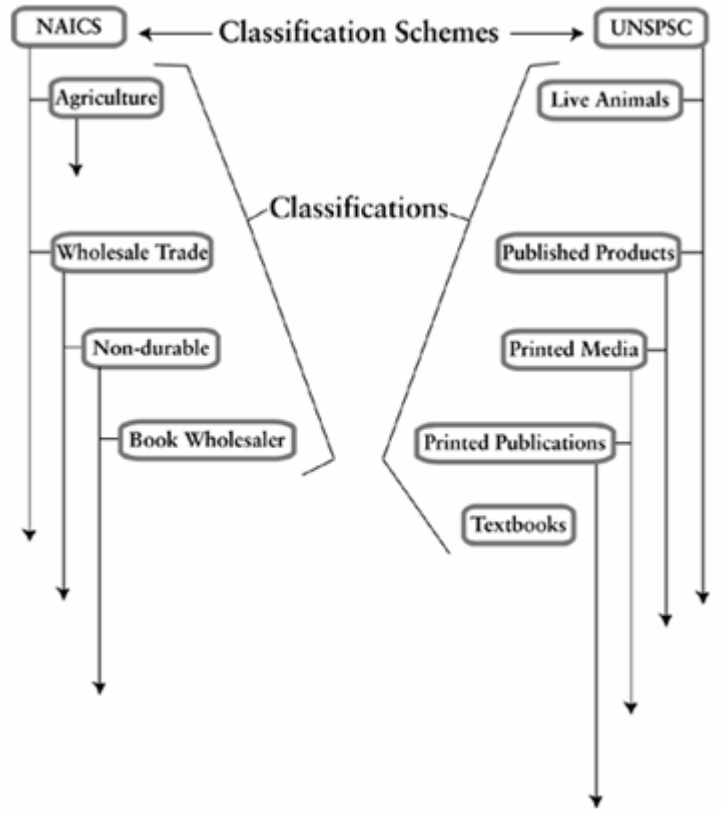
The **`BusinessQueryManager`** provides a Web service interface to the UDDI Inquiry API, which all UDDI registries are required to support. You can use the `BusinessQueryManager` to find existing `Organization`, `Association`, `Service`, `ServiceBinding`, `ClassificationScheme`, and `Concept` objects in a UDDI registry. Searches can be based on the object's UUID key, name, and other criteria. The `getRegistryObject()` method will find any data structure in the UDDI registry by its key and type. The **`getRegistryObject`**`()` is actually defined by the `BusinessQueryManager`'s supertype, the **`QueryManager`**`.` You can also use any of several `BusinessQueryManager.`**`findXXX`**`()` methods to locate UDDI data structures by their names as well as other criteria.
Adding one or more of the standard classifications to your `Organization` will help other people find it and identify the type of business you're in. Adding a standard classification requires that you first look up the classification scheme. Each of the standard classifications has a fixed name (as well as a UUID) that is the same across all UDDI registries. Because all UDDI operators are required to support, at a minimum, the same three classification systems, and under the same names, you'll always know how to find them. Using the fixed names, we can look up the `ClassificationScheme` for each of the standard classifications and create a `Classification` object based on that scheme.

```
// Create a Classification object for a NAICS value.
ClassificationScheme naics_Scheme =
   queryMngr.findClassificationSchemeByName(null, "ntis-gov:naics");
Classification naics_BookWhslrClass =
   lifeCycleMngr.createClassification(naics_Scheme,
                                      "Book Wholesaler", "42292");
```

The `Classification` object represents some code value in a specific `ClassificationScheme`. For example, in the NAICS classification system, the code `"42292"` represents a `"Book Wholesaler"`.
In addition to the three standard classifications, UDDI registries can choose to support their own classification systems. For example, the IBM UBR supports a classification system called WAND that's not supported by the other UBR operators. WAND is an on-line classification system with over 65,000 codes for various products and services.

The **`Classification`** interface, which represents a categorization of something, represents a `keyedReference` element contained by a UDDI `categoryBag` element. The **`ClassificationScheme`** interface represents the **`tModel`** referred to by a `keyedReference`

element contained by a UDDI `categoryBag` or `identifierBag` element. The `ClassificationScheme` represents the taxonomy or system of categorization used by a `Classification` object. It is the root of a classification system. For example **a NAICS `Classification` object (for example, `"Book Wholesaler"`), will refer to the NAICS `ClassificationScheme` object.**



It's the `ClassificationScheme` that gives a `Classification` object its context. In other words, if you don't know what taxonomy a `Classification` is based on, then the code and description provided by the `Classification` have no context and therefore no meaning.

```
package javax.xml.registry.infomodel;
import javax.xml.registry.*;
import java.util.*;

public interface ClassificationScheme extends RegistryEntry {

  // Manage child Concepts.
  public void addChildConcept(Concept concept) throws JAXRException;
  public void addChildConcepts(Collection concepts)
   throws JAXRException;
  public void removeChildConcept(Concept concept) throws JAXRException;
  public void removeChildConcepts(Collection concepts)
   throws JAXRException;
  public int getChildConceptCount() throws JAXRException;
  public Collection getChildrenConcepts() throws JAXRException;
  public Collection getDescendantConcepts() throws JAXRException;

  // Determine whether this ClassificationScheme is an external one.
  public boolean isExternal() throws JAXRException;

}

package javax.xml.registry.infomodel;
import javax.xml.registry.*;

public interface Classification extends RegistryObject {
```

```
  // Get|Set the Concept that is classifying the object.
  public Concept getConcept() throws JAXRException;
  public void setConcept(Concept concept) throws JAXRException;

  // Get|Set the ClassificationScheme used to classify the object.
  public ClassificationScheme getClassificationScheme()
   throws JAXRException;
  public void setClassificationScheme(ClassificationScheme
                                classificationScheme)
   throws JAXRException;

  // Get|Set the taxonomy value (the code) for this Classification.
  public String getValue() throws JAXRException;
  public void setValue(String value) throws JAXRException;

  // Get|Set the Object that is being classified.
  public RegistryObject getClassifiedObject() throws JAXRException;
  public void setClassifiedObject(RegistryObject classifiedObject)
   throws JAXRException;

  // Return true if this is an external classification.
  public boolean isExternal() throws JAXRException;
}
```

JAXR requires that vendors provide a full internal taxonomy for NAICS, UNSPSC, and ISO 3166. You can browse these taxonomies and discover specific taxonomy values. This helps when you write a UDDI browser. Here's **how to iterate the Classifications across ClassificationSchemes associated with an Organization:**

```
String companyKey = "10c4399a-5f71-0c43-1175-ec1bd516dacc"; // hardcoded actual UUID.
Organization myOrganization = (Organization)bqm.getRegistryObject(companyKey.trim(),
                                  LifeCycleManager.ORGANIZATION);
Iterator categories = myOrganization.getClassifications().iterator();
while(categories.hasNext()){
   Classification classif = (Classification)categories.next();
   ClassificationScheme scheme = classif.getClassificationScheme();
   printInternationalString(scheme.getName());
   System.out.println("  Classification Value = "+classif.getValue());
   System.out.print("  Classification Name = ");
   printInternationalString(classif.getName());
   System.out.println();
}
...
public static void printInternationalString(InternationalString interString)
throws JAXRException{
  Iterator names = interString.getLocalizedStrings().iterator();
  while(names.hasNext()){
     LocalizedString name = (LocalizedString)names.next();
     System.out.println(name.getValue());
  }
}
```

And following is the o/p:

```
ntis-gov:naics:1997
  Classification Value = 541511
  Classification Name = Custom Computer Programming Services

ntis-gov:naics:1997
  Classification Value = 541519
  Classification Name = Other Computer Related Services

ntis-gov:naics:1997
  Classification Value = 541512
  Classification Name = Computer Systems Design Services

ntis-gov:naics:1997
  Classification Value = 54171
  Classification Name = Research and Development in the Physical, Engineering, and Life
Sciences
```



```
ntis-gov:naics:1997
  Classification Value = 541513
  Classification Name = Computer Facilities Management Services
```

When you add a new `Organization` object to a UDDI registry, the registry will automatically generate a UUID key and assign it to the new `Organization` object. The UUID serves as the unique identifier for the object within the UDDI context, but there are other identifiers outside of UDDI that an `Organization` can have, some created by governments (such as U.S. Federal Tax IDs), some by private organizations. You can assign as many external identifiers to an `Organization` (as well as some other JAXR types) as you want. UDDI officially recognizes and supports two identification systems:

1. **D-U-N-S identification system** maintained by Dun & Bradstreet (D&B), which tracks the credit histories of 75 million companies in 214 countries. Every company listed by D&B has a unique identifier.
2. **Thomas Register Supplier Identifier Code system** is an on-line registry for products produced by over 189,000 U.S. and Canadian manufacturing companies.

```
    String companyKey = "10c4399a-5f71-0c43-1175-ec1bd516dacc"; // hardcoded actual UUID.
    Organization myOrganization = (Organization)bqm.getRegistryObject(companyKey.trim(),
                                   LifeCycleManager.ORGANIZATION);
    // Create a D-U-N-S identifier.
    ClassificationScheme duns_Scheme =
      bqm.findClassificationSchemeByName(null, "dnb-com:D-U-N-S");
    ExternalIdentifier dunsNumber =
      blcm.createExternalIdentifier(duns_Scheme,
                                          companyName, "038924499");
    // Add it to my Organization as an external identifier.
    myOrganization.addExternalIdentifier(dunsNumber);

    // Save the Organization in the UDDI directory.
    Set organizationSet = new HashSet();
    organizationSet.add(myOrganization);
    BulkResponse response =
                  blcm.saveOrganizations(organizationSet);
```

You can create a new `ExternalIdentifier` object using one of two overloaded `LifeCycleManager.createExternalIdentifier()` methods. [Note: the above code did not work for me and neither does the JAXR Registry Browser application provides a way to add an external identifier, probably the Sun AS RI UDDI v2 RegistryServer does not support this feature (SunAS 8.2). I tried creating the D-U-N-S classificationScheme also but it does not give any error too.].

```
package javax.xml.registry.infomodel;
import javax.xml.registry.*;

public interface ExternalIdentifier extends RegistryObject {

  // Get the parent RegistryObject for this ExternalIdentifier.
  RegistryObject getRegistryObject() throws JAXRException;

  // Get|Set the value of an ExternalIdentifier.
  public String getValue() throws JAXRExcetion;
  public void setValue(String value) throws JAXRException;

  // Get the ClassificationScheme used as the identification scheme.
  public ClassificationScheme getIdentificationScheme()
    throws JAXRException;
  public void setIdentificationScheme(ClassificationScheme
                                      identifierScheme)
    throws JAXRException;
}
```

The `Organization` interface defines a few methods for accessing, adding, and removing `Service` objects. A **`Service`** object represents a logical grouping of Web services, and an `Organization` may have one or more of them.

```
package javax.xml.registry.infomodel;
import javax.xml.registry.*;
```

```
import java.util.*;

public interface Organization extends RegistryObject {
    ...
    // Service methods.
    public void addService(Service service) throws JAXRException;
    public void addServices(Collection services) throws JAXRException;
    public void removeService(Service service) throws JAXRException;
    public void removeServices(Collection services) throws JAXRException;
    public Collection getServices() throws JAXRException;
}
```

JAXR Technical Objects (6.1)

The JAXR **`Service`** interface represents the UDDI **`businessService`** data structure, which you can use to group one or more Web services together. Organizations that have many different Web services may group those services under different `Service` objects according to their purpose, or their `Classification`, or other criteria. A `Service` represents a logical service accessible in multiple modes (options, protocols, access points, and so on). Each `ServiceBinding` represents a different mode of access to a `Service`. A **`ServiceBinding`** object corresponds to the **`bindingTemplate`** data structure in UDDI, which identifies the electronic address of a Web service and optionally refers to a set of specifications that describe that service.

```
    String companyKey = "10c4399a-5f71-0c43-1175-ec1bd516dacc";
    Organization myOrganization = (Organization)bqm.getRegistryObject(companyKey.trim(),
                                   LifeCycleManager.ORGANIZATION);

    // Create and add a new Service to my Organization.
    Service service = blcm.createService(companyName+" Web Site");
    InternationalString desc = blcm.createInternationalString(
                               "The main HTML Web site for "+companyName);
    service.setDescription(desc);
    myOrganization.addService(service);

    // Create, add, and configure a new ServiceBinding.
    ServiceBinding binding = blcm.createServiceBinding();
    service.addServiceBinding(binding);
    binding.setValidateURI(false);
    binding.setAccessURI("http://www."+companyName+".com/index.html");


    // Save the Organization in the UDDI directory.
    Set organizationSet = new HashSet();
    organizationSet.add(myOrganization);
    BulkResponse response =
                blcm.saveOrganizations(organizationSet);
```

createService() is from the **LifeCycleManager** interface. You can access, add, and remove `Service` objects relating to an `Organization` object using methods defined by the `Organization` interface.
createServiceBinding() is from the LifeCycleManager interface. The **`ServiceBinding`** object does not have a name, but it will almost always have an `accessURI`, which is the electronic address of the service. When a `ServiceBinding` is saved in the UDDI registry, the registry will verify that the `accessURI` is a valid URL—assuming it is a URL. This behavior can be overridden by setValidateURI(false).

```
package javax.xml.registry.infomodel;
import javax.xml.registry.*;
import java.util.*;

public interface Service extends RegistryEntry {

  // Get|Set the Organization that provides this service.
  public Organization getProvidingOrganization() throws JAXRException;
  public void setProvidingOrganization(Organization
                                       providingOrganization)
   throws JAXRException;

  // Manage the collection of ServiceBindings.
  public Collection getServiceBindings() throws JAXRException;
```



```
  public void addServiceBinding(ServiceBinding serviceBinding)
   throws JAXRException;
  public void addServiceBindings(Collection serviceBindings)
   throws JAXRException;
  public void removeServiceBinding(ServiceBinding serviceBinding)
   throws JAXRException;
  public void removeServiceBindings(Collection serviceBindings)
   throws JAXRException;
}
```

Service interface simply provides accessors to a collection of `ServiceBinding` objects, each of which represents a different Web service. A `Service` has a name, a description, and a reference to its owning `Organization`, and can be associated with one or more `Classification` objects. You use the **`getName()`** and **`getDescription()`** methods that `Service` inherited from `RegistryObject` to access the `Service`'s `name` and `description` properties. You can give a `Service` any name you like, but you should choose a name that accurately describes the kind of Web services it contains. You can **access a `Service` object's UUID key**, which was assigned by the UDDI registry when it was created, using the **`RegistryObject.getKey()`** method. A `Service` object can be associated with `Classification` objects, just as an `Organization` can. **You add `Classification` objects to a `Service` using the `Classification` management methods defined by the `RegistryObject` interface**.

`ServiceBinding` declares the electronic address of the Web service and associates one or more `SpecificationLink` objects with the Web service.

```
package javax.xml.registry.infomodel;
import javax.xml.registry.*;
import java.util.*;

public interface ServiceBinding extends RegistryObject, URIValidator {

  // Get|Set the URL of this ServiceBinding.
  public String getAccessURI() throws JAXRException;
  public void setAccessURI(String uri) throws JAXRException;

  // Get|Set a redirection ServiceBinding.- disallowed by BP - this maps to
  // hostingRedirector element of UDDI bindingTemplate data structure.
  public ServiceBinding getTargetBinding() throws JAXRException;
  public void setTargetBinding(ServiceBinding binding)
   throws JAXRException;

  // Get the parent Service that contains this ServiceBinding.
  public Service getService() throws JAXRException;

  // Manage the collection of SpecificationLink objects.
  public void addSpecificationLink(SpecificationLink specificationLink)
   throws JAXRException;
  public void addSpecificationLinks(Collection specificationLinks)
   throws JAXRException;
  public void removeSpecificationLink(SpecificationLink
                                      specificationLink)
   throws JAXRException;
  public void removeSpecificationLinks(Collection specificationLinks)
   throws JAXRException;
  public Collection getSpecificationLinks() throws JAXRException;

}
```

In UDDI the **`tModel`** data structure can be used to describe the technical specifications of a Web service. A technical `tModel` (also called a "`tModel` fingerprint") can refer to a WSDL document, an XML schema document, or some other specification. In the case of J2EE Web services, a technical `tModel` **usually refers to a WSDL document that describes the Web service**. In JAXR the **`Concept`** object represents a technical `tModel`. A `concept` object is created independently of an `Organization` and referred to by one or more Web services. This is especially useful when a technical `tModel` refers to an industry-standard WSDL document. For example, the wholesale book industry might create a standard Web service, with a standard WSDL document, for requesting quotes of book prices from a specific

company. All of the the organization's members that supported this Web service would refer to the same WSDL document. While sharing a tModel can be useful, most organizations will create their own tModels for a specific custom Web service. Here's how you create a tModel (or Concept):

```
    // Create the WSDL Concept object.
    Concept wsdlConcept = lifeCycleMngr.createConcept(null,"","");

    // Set the Concept name.
    InternationalString conceptName =
      lifeCycleMngr.createInternationalString(domainName+":BookQuote");
    wsdlConcept.setName(conceptName);

    // Set the URL to WSDL binding.
    ExternalLink overviewDoc = lifeCycleMngr.createExternalLink(
      "http://www."+domainName+"/jwsbook/BookQuote.wsdl"+
        "#xmlns(wsdl=http://schemas.xmlsoap.org/wsdl/) "+
        "xpointer(/wsdl:definitions/wsdl:portType["+
        "@name=\"BookQuoteBinding\"])",
      "The WSDL <binding> for this Web service");

    overviewDoc.setValidateURI(false);
    wsdlConcept.addExternalLink(overviewDoc);

    // Add the wsdlSpec Classification to the WSDL Concept.
    ClassificationScheme uddi_types =
      queryMngr.findClassificationSchemeByName(null, "uddi-org:types");
    Classification wsdlSpec_Class = lifeCycleMngr.createClassification(
      uddi_types,"WSDL Document","wsdlSpec");
    wsdlConcept.addClassification(wsdlSpec_Class);

    // Add the WS-I Conformance Classification to the WSDL Concept.
    ClassificationScheme wsI_types =
     queryMngr.findClassificationSchemeByName(
        null,
        "ws-i-org:conformsTo:2002_12");
    Classification wsIconform_Class =
     lifeCycleMngr.createClassification(
        wsI_types,
        "Conforms to WS-I Basic Profile 1.0",
        "http://ws-i.org/profiles/basic/1.0");
    wsdlConcept.addClassification(wsIconform_Class);

    // Save the Concept object.
    Set conceptSet = new HashSet();
    conceptSet.add(wsdlConcept);
    BulkResponse response = lifeCycleMngr.saveConcepts(conceptSet);
```

1. In UDDI the Concept object can have only one name. A Concept can have only one name (a LocalizedString value), but it may be language-specific.
2. The **ExternalLink** of a Concept object corresponds to the **overviewDoc** element of a UDDI tModel. This element will contain a URL and an optional description. The Basic Profile requires that the overviewDoc element use the XPointer reference system to identify a specific binding in a specific WSDL document.
3. When the Concept object is saved in the UDDI registry, the JAXR provider will automatically validate the URL by checking to see if the named document is accessible. If not, it will throw a JAXRException. You can prevent against this check by using **setValidateURI**(false).
4. The **uddi-org:types** category **"wsdlSpec"** is assigned to the Concept after it is created. The BP requires that tModels representing Web services be categorized using the "wsdlSpec" UDDI type.
5. The Basic Profile 1.0 defines a categorization that asserts that a WSDL tModel is compliant with the BP. This **conformance categorization** is assigned to the WSDL tModel. This categorization can aid in searches for BP conformant web services.

```
package javax.xml.registry.infomodel;
import java.util.*;
import javax.xml.registry.*;

public interface Concept extends RegistryObject {
```



```
  // Get the taxonomy value of this Concept.
  public String getValue() throws JAXRException;
  public void setValue(String value) throws JAXRException;

  // Taxonomy browsing methods.
  public Collection getDescendantConcepts() throws JAXRException;
  public Collection getChildrenConcepts() throws JAXRException;
  public int getChildConceptCount() throws JAXRException;
  public void removeChildConcepts(Collection concepts) throws JAXRException;
  public void removeChildConcept(Concept concept) throws JAXRException;
  public void addChildConcepts(Collection concepts) throws JAXRException;
  public void addChildConcept(Concept concept) throws JAXRException;

  // Get the parent of the Concept.
  public RegistryObject getParent() throws JAXRException;
  public Concept getParentConcept()  throws JAXRException;
  public ClassificationScheme getClassificationScheme() throws JAXRException;

  // Get the canonical path representation for this Concept.
  public String getPath() throws JAXRException;
}
```

If the `Concept` object represents a WSDL `tModel`, then you cannot make use of the methods defined by the `Concept` interface; you must use the methods defined by its supertype, `RegistryObject`. If, however, the `Concept` object represents a taxonomy value, then all its methods may be of use to you.

JAXR defines **two types of taxonomies**, **internal** and **external**. An internal taxonomy is one that is maintained by the JAXR provider and can be browsed using the `Concept` methods as an object graph. JAXR requires that all JAXR providers (vendors) include internal taxonomies for NAICS, UNSPSC, and ISO 3166. As a consequence, the JAXR provider you use in your client code will maintain an object graph representing all of the values in each of these taxonomies.

```
    String taxonomyName = args[0];

    // Find the ClassificationScheme.
    ClassificationScheme taxonomy =
      queryMngr.findClassificationSchemeByName(null, taxonomyName);

    // Get first-level children.
    Collection children = taxonomy.getChildrenConcepts();
    // List all descendants.
    list(3,children);
```

```
  /* This method is recursive. It lists all the children of a Concept.*/
  public static void list(int indent, Collection children)
  throws JAXRException{
    Iterator concepts = children.iterator();
    while(concepts.hasNext()){
      Concept concept = (Concept)concepts.next();
      for(int i = 0; i < indent;i++)System.out.print(" ");
      System.out.println(concept.getValue()+
                         " "+concept.getName().getValue());
      list(indent+3,concept.getChildrenConcepts());
    }
  }
```

`taxonomyName can be ntis-gov:naics:1997 or unspsc-org:unspsc or uddi-org:iso-ch:3166-1999.`

This is how the o/p looks: (U see how easy it is to create a tree representation of the internal taxonomy in a UDDI client browser.)

```
11 Agriculture, Forestry, Fishing and Hunting
      111 Crop Production
         1111 Oilseed and Grain Farming
            11111 Soybean Farming
            11112 Oilseed (except Soybean) Farming
            11113 Dry Pea and Bean Farming
            11114 Wheat Farming
            11115 Corn Farming
```



```
            11116 Rice Farming
            11119 Other Grain Farming
               111191 Oilseed and Grain Combination Farming
…
```

An **external taxonomy** is one that is maintained by the UDDI registry or some other source, and cannot be browsed using JAXR. One example is the WAND taxonomy used by the IBM UBR registry. You can browse the WAND taxonomy only by using the external WAND Web site, or the IBM UBR Web interface.

The UUIDs for `tModel`s are unique in that they always start with a **`uddi`** prefix.

The Basic Profile requires that a technical `tModel` for a WSDL document be classified as **`wsdlSpec`** and **`ws-i-org:conformsTo:2002_12`**.

The **`ExternalLink`** methods allow you to access the UDDI **`overviewDoc`** elements associated with the underlying `tModel`. A `Concept` may have many **`ExternalLink`** objects, but will usually have only one.

You can associate a `Concept` object with your `ServiceBinding` using a **`SpecificationLink`**. A `ServiceBinding` may contain references to one or more `SpecificationLink` objects, which are maintained in a `Collection`. A `SpecificationLink` represents the UDDI **`tModelInstanceInfo`** and **`tModelinstanceDetails`** elements of the **`bindingTemplate.`**

```
    // Find my Organization by its Key.
    Organization myOrganization = (Organization)
       queryMngr.getRegistryObject(companyKey.trim(),
                                   LifeCycleManager.ORGANIZATION);

    String orgName = myOrganization.getName().getValue();

    // Create and add a new Service to my Organization.
    Service service = lifeCycleMngr.createService("BookQuoteService");
    InternationalString desc = lifeCycleMngr.createInternationalString(
                       "This Web service provides "+orgName+
                       "'s current prices of Books");
    service.setDescription(desc);
    myOrganization.addService(service);

    // Create, add, and configure a new ServiceBinding.
    ServiceBinding binding = lifeCycleMngr.createServiceBinding();
    service.addServiceBinding(binding);
    binding.setValidateURI(false);
    binding.setAccessURI("http://www."+orgName+".com/BookQuote");

    // Create a SpecificationLink and add it to the ServiceBinding.
    SpecificationLink specLink =
                    lifeCycleMngr.createSpecificationLink();
    binding.addSpecificationLink(specLink);

    // Locate and set the Concept for the SpecificationLink.
    Concept wsdlConcept = (Concept)
            queryMngr.getRegistryObject(tModelKey.trim(),
                                        LifeCycleManager.CONCEPT);
    specLink.setSpecificationObject(wsdlConcept);

    // Save the Organization in the UDDI directory.
    Set organizationSet = new HashSet();
    organizationSet.add(myOrganization);
    BulkResponse response =
                  lifeCycleMngr.saveOrganizations(organizationSet);
```

`Steps:`

1. To associate a `Concept` with an `Organization`, you need to create a `Service` and a `ServiceBinding`. A `Service` represents a logical grouping of electronic services. Each `Service` may contain many `ServiceBinding` objects, which assign an `accessURI` to a Web service and, optionally, associate one or more `Concept` objects with it.

2. You create SpecificationLinks invoking the **createSpecificationLink**() method. Once a **SpecificationLink** is created, you can add it to the **ServiceBinding** and set it to refer to the **Concept** that represents the WSDL document.

```
// Create a SpecificationLink and add it to the ServiceBinding.
SpecificationLink specLink = lifeCycleMngr.createSpecificationLink();
binding.addSpecificationLink(specLink);

// Locate and set the Concept for the SpecificationLink.
Concept wsdlConcept = (Concept)
  queryMngr.getRegistryObject(tModelKey.trim(),
                              LifeCycleManager.CONCEPT);
specLink.setSpecificationObject(wsdlConcept);
```

3. The Concept object must be created and saved independent of the Organization, Service, ServiceBinding, and SpecificationLink. That's because a SpecificationLink can be saved only if it refers to an existing tModel UUID key.

```
package javax.xml.registry.infomodel;
import javax.xml.registry.*;
import java.util.*;

public interface SpecificationLink extends RegistryObject {

  // Get|Set the Concept for this link.
  public RegistryObject getSpecificationObject() throws JAXRException;
  public void setSpecificationObject(RegistryObject obj)
   throws JAXRException;

  // Get|Set the description of the usage parameters.
  public InternationalString getUsageDescription() throws JAXRException;
  public void setUsageDescription(InternationalString usageDescription)
   throws JAXRException;

  // Get|Set any usage parameters.
  public Collection getUsageParameters() throws JAXRException;
  public void setUsageParameters(Collection usageParameters)
   throws JAXRException;

  // Get the parent ServiceBinding for this SpecificationLink.
  public ServiceBinding getServiceBinding() throws JAXRException;
}
```

The **Association** object represents a categorized relationship between two Organization objects and corresponds to the UDDI **publisherAssertion** data structure. Although **createAssociation**() is a general-purpose method in the JAXR API, when working with UDDI it is used only to assert a relationship between businessEntity data types—Organization objects.
All you need to do is obtain a reference to each Organization, create an Association, and save the Association in the UDDI registry.

```
  // Find my Organization by its Key.
  Organization sourceOrg = (Organization)
     queryMngr.getRegistryObject(sourceOrgKey.trim(),
                                 LifeCycleManager.ORGANIZATION);

  // Find the other Organization by its Key.
  Organization targetOrg = (Organization)
     queryMngr.getRegistryObject(targetOrgKey.trim(),
                                 LifeCycleManager.ORGANIZATION);
  // Find the AssociationType Concept object.
  Concept associationType  =
     queryMngr.findConceptByPath("/AssociationType/RelatedTo");

  // Create an Association object and add it to my Organization.
  Association association =
     lifeCycleMngr.createAssociation(targetOrg, associationType);
  sourceOrg.addAssociation(association);

  // Save the Association in the UDDI directory.
```

```
  Set assocSet = new HashSet();
  assocSet.add(association);
  BulkResponse response = lifeCycleMngr.saveAssociations(assocSet,true);
```

To create a new Association you need a Concept object that represents the association type, which represents the nature of the relationship between two organizations. JAXR implementations are required to support several different association types, which are predefined Concept objects that can be accessed by a simple path name. The paths for predefined Concepts that can be used for Association objects in UDDI are described here:

- `"/AssociationType/RelatedTo"` represents a UDDI **peer-to-peer** relationship between two `Organization`s like a manufacturer and a supplier, or a retailer and a wholesaler.
- `"/AssociationType/HasChild"` represents a UDDI **parent-child** relationship in which the source organization owns the target organization. For example, the source is a company and the target is one of the company's divisions.
- `"/AssociationType/EquivalentTo"` represents a UDDI **identity** relationship, in which the source and target are actually the same entity. This might be used if a company has more than one name and lists itself under both in different entries in the UDDI directory.
- `"/AssociationType/HasMember"` has no corresponding type in UDDI. It represents the relationship between an industry organization and its member companies.
- `"/AssociationType/HasParent"` has no corresponding type in UDDI. It represents the relationship in which the source is owned by the target organization

You can define any type of association `Concept` object you want. You may want to use the UDDI standard relationship types (peer-peer, parent-child, and identity).

```
// Create and save a new association-type Concept object.
Concept associationType  = lifeCycleMngr.createConcept(null,
                                            "uddi-org:types",
                                            "relationship");
InternationalString conceptName  =
     lifeCycleMngr.createInternationalString("supplier-buyer");
associationType.setName(conceptName);
Collection conceptSet = new HashSet();
conceptSet.add(associationType);
lifeCycleMngr.saveConcepts(conceptSet);
```

Although you must save the `Association` object, it's not necessary to save the `Organization` object too. In UDDI the `publisherAssertion`, which corresponds to the JAXR `Association` object, is independent of the `businessEntity` entries it represents. The `publisherAssertion` maintains a reference to the `businessEntity` entries, but the `businessEntity` entries do not have references to any `publisherAssertion` entries.

Some of the JAXR infomodel types can be assigned a designator that describes their usage. For example, you can designate a `TelephoneNumber` object as "home phone," "beeper," "fax," and so on. As a convenience, the JAXR specification requires that vendors support a set of **predefined enumeration types** that represent these designators. There are **two predefined enumeration sets** that are of interest when working with a UDDI registry: **`AssociationType`** and **`URLType`**.

`ExtensibleObject` is the supertype of the `RegistryObject` interface. It defines methods for manipulating `Slot` objects. A **`Slot`** object represents arbitrary meta-data that can be attached to any infomodel object that implements the `javax.xml.registry.infomodel.ExtensibleObject` interface.

JAXR Inquiry and Publishing APIs (6.2)

Mapping JAXR to the UDDI Inquiry API

The `BusinessQueryManager` interface defines several `findXXX()` methods that correspond closely to the find operations in the UDDI Inquiry API.

BusinessQueryManager Method	*UDDI Operation*	*Description*
`findOrganizations`	`find_business`	Finds matching `businessEntity` entries.
`findAssociations`	`find_relatedBusiness`	Finds matching `publisherAssertion` entries.
`findCallerAssociations`	`get_publisherAssertions` `get_assertionStatus Report`	Finds caller's matching `publisherAssertion` entries. This is the only findXX() method which is executed against the publishing URL and hence requires authentication.
`findServices`	`find_service`	Finds matching `businessService` entries
`findServiceBindings`	`find_binding`	Finds matching `bindingTemplate` entries
`findClassificationSchemes`	`find_tModel`	Finds matching `tModel` entries
`findClassificationScheme ByName`	`find_tModel`	Finds `tModel` entry with matching name
`findConcepts`	`find_tModel`	Finds matching `tModel` entries
`findConceptByPath`	`find_tModel`	Finds `tModel` entry with matching name

Most of the findXXX() methods return a BulkResponse object which contains the result of the search.

```
// Finds all Organization objects that match criteria.
   public BulkResponse findOrganizations(Collection findQualifiers,
                                         Collection namePatterns,
                                         Collection classifications,
                                         Collection specifications,
                                         Collection externalIdentifiers,
                                         Collection externalLinks)
   throws JAXRException;
```

The `findOrganizations()` method is a prototypical example of the `findXXX()` methods in general. Most of the others use a subset of the same criteria used by `findOrganizations()`.

```
    // Access the BusinessQueryManager object.
    Connection connection = connectToRegistry(); // here we don't authenticate.
    RegistryService registry = connection.getRegistryService();
    BusinessQueryManager queryMngr = registry.getBusinessQueryManager();

    // Create name-pattern search criteria.
    Collection namePatterns = new HashSet();
    namePatterns.add("IBM");

    // Find Organizations that meet search criteria.
    BulkResponse response = queryMngr.findOrganizations(null, namePattern,
                                                        null,null,null,null);
    // Output results of query.
    Iterator results = response.getCollection().iterator();
    while(results.hasNext()){
       Organization org = (Organization)results.next();
       String orgName = org.getName().getValue();
       System.out.println(orgName);
    }

    // Check for registry exceptions.
    doExceptions(response);
```

Above example executes a fairly simple search for all `Organization` objects whose names start with "IBM". Once you have a reference to the `BusinessQueryManager`, you have the access you need to

make queries, but before you can execute a `findXXX()` method you'll need to create one or more **criteria objects**. You must, pass in at least one criteria object, or the method will throw a `JAXRException`.

Name criteria:

```
// Create name-pattern search criteria.
Collection namePatterns = new HashSet();
namePatterns.add("Titan");
namePatterns.add("Addison");

// Find Organizations that meet search criteria.
BulkResponse response = queryMngr.findOrganizations(null, namePatterns,
                                                    null,null,null,null);
```

By default, a name search is an **ORed** search. In the above example the results will contain all the `Organization` objects whose name starts with either "Titan" or "Addison." You can use a **wild-card character (`%`)** in `namePattern` searches. By default, a wild card is assumed to be at the end of each name.

Classification criteria:

You can search for `Organization`, `Service`, `ServiceBinding`, `ClassificationScheme`, and `Concept` objects using the `classifications` criteria.

```
// Create a NAICS Classification.
ClassificationScheme naics_Scheme =
   queryMngr.findClassificationSchemeByName(null, "ntis-gov:naics");
Classification naics_BookWhslrClass =
   lifeCycleMngr.createClassification(naics_Scheme,
                                      "Book Wholesaler", "42292");
// Create a UNSPSC Classification.
ClassificationScheme unspsc_Scheme =
   queryMngr.findClassificationSchemeByName(null, "unspsc-org:unspsc");
Classification unspsc_TextBookClass =
   lifeCycleMngr.createClassification(unspsc_Scheme,
                                      "TextBook", "55.10.15.09");

// Place Classification criteria into a Collection.
Collection classifications = new HashSet();
classifications.add(naics_BookWhslrClass);
classifications.add(unspsc_TextBookClass);

// Find Organization by search criteria.
BulkResponse response = queryMngr.findOrganizations(null, null,
                                                    classifications,
                                                    null,null,null);
```

The `classifications` search parameter corresponds to the UDDI **`categoryBag`** search element. When multiple `Classification` objects are used in the criteria, the search will be conducted as an AND search

`External Identifier criteria`:

You can search for `Organization` and `Concept` objects using the `externalIdentifiers` criteria.

```
// Create a D-U-N-S ExternallIdentifier.
ClassificationScheme duns_Scheme =
    queryMngr.findClassificationSchemeByName(null, "dnb-com:D-U-N-S");
ExternalIdentifier dunsNumber =
   lifeCycleMngr.createExternalIdentifier(duns_Scheme,
                                          "Monson-Haefel, Inc.", "038924499");

// Place ExternallIdentifier criteria into a Collection.
Collection identifiers = new HashSet();
identifiers.add(dunsNumber);

// Find Organizations that meet search criteria.
BulkResponse response = queryMngr.findOrganizations(null, null, null,
                                                    null, identifiers, null);
```

Search behavior is similar to Classifications criteria. The `externalIdentifiers` search parameter corresponds to the UDDI **`identifierBag`** search element.

Specifications criteria:
You can search for `Organization`, `Service`, and `ServiceBinding` objects using the `specifications` criteria. The `specifications` criteria take the form of a `Collection` of `Concept` objects that represent `tModel`s.

```
// Find the Concept object for a specific tModel.
Concept bookQuote_concept = (Concept)queryMngr.getRegistryObject(
             "UUID:2E802060-3CF2-11D7-9590-000629DC0A53",
             LifeCycleManager.CONCEPT);


// Place the Concept into a Collection object.
Collection specifications = new HashSet();
specifications.add(bookQuote_concept);

// Find Organizations that meet search criteria.
BulkResponse response = queryMngr.findOrganizations(null, null, null,
                                                    specifications,
                                                    null, null);
```

A search by `specifications` criteria is conducted as an AND search. The `specifications` search parameter corresponds to the UDDI **`tModelBag`** search element.

External links criteria:

```
// Create ExternalLink criteria.
String url = "http://sometypeofURL.com/somefile";
ExternalLink link = lifeCycleMngr.createExternalLink(url, "");

// Place the criteria in a Collection object.
Collection externalLinks = new HashSet();
externalLinks.add(link);

// Find Organizations that meet search criteria.
BulkResponse response = queryMngr.findOrganizations(null, null,
                                                    null,null,null,
                                                    externalLinks);
```

The `ExternalLink` you are searching for is a UDDI **`discoveryURL`** element of a `businessEntity` data structure. It's not necessary to set the `description` parameter when creating the `ExternalLink`; the search is based on the `URL` attribute of the `ExternalLink`, not the `description`. When multiple `ExternalLink` objects are used in the criteria parameter, the search will be conducted as an AND search.

Find Qualifiers criteria:
The `findQualifiers` criteria allow you to modify the default behavior of a `findXXX()` method. You can use them to specify ordering of results, the Boolean evaluation of other criteria, and wild-card searches on names.

```
// Create a NAICS and UNSPSC Classification Collection.
Classification naics_BookWhslrClass =
   lifeCycleMngr.createClassification(naics_Scheme,
                                      "Book Wholesaler", "42292");
Classification unspsc_TextBookClass =
   lifeCycleMngr.createClassification(unspsc_Scheme,
                                     "TextBook", "55.10.15.09");
Collection classifications = new HashSet();
classifications.add(naics_BookWhslrClass);
classifications.add(unspsc_TextBookClass);

// Create a Collection of FindQualifiers.
Collection findQualifiers = new HashSet();
findQualifiers.add(FindQualifier.OR_ALL_KEYS);
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);

// Find Organizations that meet search criteria.
BulkResponse response = queryMngr.findOrganizations(findQualifiers, null,
                                                    classifications,
```

```
                                                    null,null,null);
```

Here's a list of all findXXX() methods:

```
public BulkResponse findAssociations(Collection findQualifiers,
                                     String sourceObjectId,
                                     String targetObjectId,
                                     Collection associationTypes)
  throws JAXRException;

// Find the AssociationType Concept object and add it to a Collection.
Concept relatedTo_Type  =
   queryMngr.findConceptByPath("/AssociationType/RelatedTo");
Collection associationTypes = new HashSet();
associationTypes.add(relatedTo_Type);

BulkResponse response = queryMngr.findAssociations(null,
                                "9C508570-393D-11D7-9F18-000629DC0A53",
                null,associationTypes);

public BulkResponse findCallerAssociations(Collection findQualifiers,
                                           Boolean confirmedByCaller,
                                           Boolean confirmedByOtherParty,
                                           Collection associationTypes)

BulkResponse response =
    queryMngr.findCallerAssociations(null,Boolean.TRUE,Boolean.FALSE,null);

public BulkResponse findServices(Key orgKey,
                                 Collection findQualifiers,
                                 Collection namePatterns,
                                 Collection classifications,
                                 Collection specifications)
  throws JAXRException;

// Create an organization Key object.
Key key = lifeCycleMngr.createKey("9C508570-393D-11D7-9F18-000629DC0A53");

// Create name-pattern search criteria.
Collection namePatterns = new HashSet();
namePatterns.add("TextBook");

// Find Organization by search criteria.
BulkResponse response = queryMngr.findServices(key, null, namePatterns,
                                               null,null);

public BulkResponse findServiceBindings(Key serviceKey,
                                        Collection findQualifiers,
                                        Collection classifications,
                                        Collection specifications)
 throws JAXRException;

public BulkResponse findClassificationSchemes(Collection findQualifiers,
                                              Collection namePatterns,
                                              Collection classifications,
                                              Collection externalLinks)
  throws JAXRException;

public ClassificationScheme findClassificationSchemeByName
                                              (Collection findQualifiers,
                                              String namePattern)
  throws JAXRException;

public BulkResponse findConcepts(Collection findQualifiers,
                                 Collection namePatterns,
                                 Collection classifications,
                                 Collection externalIdentifiers,
                                 Collection externalLinks)
  throws JAXRException;

public Concept findConceptByPath(String path) throws JAXRException;
```

Mapping JAXR to the UDDI Publishing API

The UDDI specification includes a Publishing API that is used to save, modify, and remove data from the registry. Nearly all of the SOAP operations defined by the UDDI Publishing APImap to similarly named methods in the `BusinessLifeCycleManager` interface.

In many cases you don't need to save an infomodel object to the UDDI registry explicitly. If one object is contained by another, saving the containing object will often automatically save the contained object. For example, if you create a new `Organization` and add a `Service` and `ServiceBinding` to it, you can save all three just by invoking `saveOrganizations()` using the `Organization` object—its children (the `Service` and `ServiceBinding` objects) will be saved too. The same is not true of new and updated `Concept` and `ClassificationScheme` objects. These must be saved explicitly before any other infomodel object can refer to them.

BusinessLifeCycleManager Method	*UDDI Operation*	*Description*

`saveAssociations`
 	
`confirmAssociations`
	`add_publisherAssertions`	Adds a new `publisherAssertion` or confirms a relationship declared by a `publisherAssertion`

`deleteServiceBindings`	`delete_binding`	Removes one or more `templateBinding` entries
`deleteOrganizations`	`delete_business`	Removes one or more `businessEntity` entries
`deleteAssociations`	`delete_publisherAssertions`	Removes one or more `publisherAssertion` entries
`deleteClassificationSchemes`	`delete_tModel`	In UDDI `delete_tModel` does not delete the `tModel`. It simply hides it from `find_tModel` calls. The `QueryManager.getRegistryObject` calls will still return the deleted `tModel` after a `deleteConcepts` or `deleteClassification Schemes` call
`saveServiceBindings`	`save_binding`	Adds or modifies one or more `bindingTemplate` entries
`saveOrganizations`	`save_business`	Adds or modifies one or more `businessEntity` entries
`saveServices`	`save_service`	Adds or modifies one or more `businessService` entries

`saveClassificationSchemes`
 	
`saveConcepts`
	`save_tModel`	Adds or modifies one or more `tModel` entries

`saveAssociations`	`set_publisherAssertions`	Modifies one or more existing `publisherAssertion` entries

Developing Web Services

[RMH – Chapters 22-24]

7.1 Identify the characteristics of and the services and APIs included in the J2EE platform.

7.2 Explain the benefits of using the J2EE platform for creating and deploying Web service applications.

7.3 Describe the functions and capabilities of the JAXP, DOM, SAX, JAXR, JAX-RPC, and SAAJ in the J2EE platform.

7.4 Describe the **role of the WS-I Basic Profile** when designing J2EE Web services.

9.1 Describe the steps required to **configure, package, and deploy J2EE Web services and service clients**, including a description of the packaging formats, such as .ear, .war, .jar, **deployment descriptor settings, the associated Web services description file, RPC mapping files, and service reference elements** used for EJB and servlet endpoints.

9.2 Given a set of requirements, develop code to process XML files using the SAX, DOM, XSLT, and JAXB APIs.
9.3 Given an **XML schema for a document style Web service create a WSDL file** that describes the service and generate a service implementation.
9.4 Given a set of requirements, develop code to create an XML-based, document style, Web service using the JAX-RPC APIs.
9.5 Implement **a SOAP logging mechanism** for testing and debugging a Web service application using J2EE Web Service APIs.
9.6 Given a set of requirements, develop code to **handle system and service exceptions and faults** received by a Web services **client**.

J2EE Deployment (9.1)

A JAR file packages the class files and XML deployment descriptor of a J2EE component into a single file, which can be easily transferred or stored on a disk. When you deploy the component, the J2EE application server's deployment tool inspects the JAR file, reads the deployment descriptor, extracts the class files and other files (properties, images, and so on), and then instantiates and launches the component into the server. During this deployment process, you are usually given an opportunity to tune the runtime attributes (enterprise attributes like security restrictions, transactional behavior, instance pooling, and threading) of the J2EE component to meet your specific needs. A JAR file that contains Web components is called a Web ARchive (**WAR**) file. A JAR file that contains Enterprise JavaBeans is called an **EJB JAR** file. Another type of JAR file is a Resource ARchive (**RAR**), which contains J2EE connectors. A J2EE connector might be a JDBC driver, a JMS provider, a JDO provider, or an API that connects to a proprietary resource like SAP, PeopleSoft, CICS, or IMS. An Enterprise Archive (**EAR**) file contains a general XML deployment descriptor for the entire application, as well as all the J2EE components and connectors, packaged into their own EJB JAR, WAR, and RAR files.
JSEs go into WAR files and EJB endpoints go into EJB JAR files. The JAX-RPC specification requires that every JSE or EJB endpoint have a corresponding WSDL document that provides a platform-independent definition of the J2EE endpoint.

Starting with a J2EE Endpoint

```
public interface BookQuote extends java.rmi.Remote {
    public float getBookPrice(String isbn)
    throws java.rmi.RemoteException, InvalidIsbnException;
}
```

And following is the generated WSDL from above SEI:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuote"
   targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
   xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

   <message name="getBookPriceRequest">
       <part name="isbn" type="xsd:string"/>
   </message>
   <message name="getBookPriceResponse">
       <part name="result" type="xsd:float"/>
   </message>
   <message name="InvalidIsbnException">
       <part name="message" type="xsd:string"/>
   </message>
   <portType name="BookQuote">
       <operation name="getBookPrice" parameterOrder="isbn">
          <input message="mh:getBookPriceRequest"/>
          <output message="mh:getBookPriceResponse"/>
          <fault name="InvalidIsbnException"
                 message="mh:InvalidIsbnException"/>
       </operation>
   </portType>
   <binding name="BookQuoteBinding" type="mh:BookQuote">
```

```
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="rpc"/>
    <operation name="getBookPrice">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"
        namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"/>
      </input>
      <output>
        <soap:body use="literal"
        namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"/>
      </output>
      <fault name="InvalidIsbnException">
        <soap:fault name="InvalidIsbnException" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="BookQuoteService">
    <port name="BookQuotePort" binding="mh:BookQuoteBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
  </service>
</definitions>
```

While the Java-to-WSDL mapping is very specific, it's impossible to create a complete WSDL document from the endpoint interface alone. The definitions of **`binding`** and **`service`** elements elements cannot be derived from the endpoint interface, because the elements describe implementation-specific details. For example, the `binding` element tells us which protocol and messaging mode are to be used, and the `service` element identifies the URL of the Web service endpoint. This information is provided in webservice.xml (a special deployment descriptor).

Starting with a WSDL

By starting with a WSDL document instead of a service implementation, you have an opportunity to leave the baggage of a specific programming language behind and embrace a more neutral Web service perspective. For example, Java natively supports only IN parameters and methods that return only a single value, so we tend to design Java interfaces with several parameters and a single return type. By contrast, in programming languages like C++ and C#, the use of INOUT and OUT parameters is commonplace, and the avoidance of these parameter modes in Java is alien to people who work in C++ and C#.

There are a couple reasons you need a JAX-RPC mapping file.

1. First, it associates a J2EE endpoint with an exact WSDL `port` definition. Remember that a WSDL document can define several different `port` and `binding` definitions that may share a common `portType`. Without a JAX-RPC mapping file it can be difficult to determine which WSDL `port` a J2EE endpoint is associated with.
2. A JAX-RPC mapping file allows you to use naming conventions in the WSDL file that are different from those of the corresponding J2EE endpoint. The native Java class, method, and exception names can be mapped to platform-agnostic WSDL names. For example, your endpoint interface can define an `InvalidIsbnException` that the JAX-RPC mapping file maps to a corresponding WSDL `message` named `InvalidIsbnFault`.

JSEs are packaged in a WAR file along with a `web.xml` deployment descriptor, which defines all kinds of deployment properties related to security, threading, resources, and other runtime concerns. The **`web.xml`** deployment descriptor is used with servlets and JSPs, but has been adapted for use with JSE. Similarly, an EJB endpoint, which is a stateless session bean, is packaged in an EJB JAR file and described, in part, by an **`ejb-jar.xml`** deployment descriptor commonly used for EJB components.

Deploying JSEs

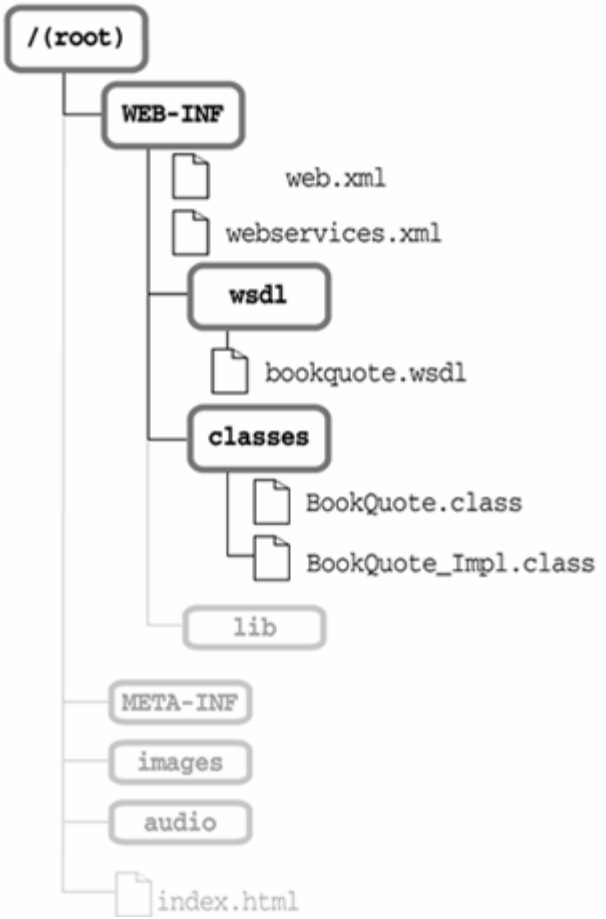


Figure 1 - WAR Directory Structure

Although the contents of the root and its subdirectories are accessible as URLs (once they're deployed), the contents of the `META-INF` and `WEB-INF` subdirectories are not publicly accessible—they may be accessed only by the container system and the Web components themselves. The `META-INF` directory contains meta-data about the JAR file and is generated automatically when you create the JAR file. The `WEB-INF` subdirectory is the most important directory for JSE deployments. It contains the deployment descriptors and the JSE endpoint and implementation class files, as well as any other classes or libraries that your JSE code depends on. Specifically, deployment descriptors are located directly in `WEB-INF` itself, while the Java classes and supporting libraries (usually other JAR files) are located in the `WEB-INF/classes` and `WEB-INF/lib` subdirectories. WSDL documents are placed in `WEB-INF/wsdl`. At runtime, the `wsdl` directory can be accessible to Web service clients directly. For example, you could access the `bookquote.wsdl` at `http://www.Monson-Haefel.com/jwsbook/wsdl/bookquote.wsdl`. In addition, WSDL and XSD documents imported by `bookquote.wsdl` must be stored in the `/META-INF/wsdl` directory, or its subdirectory, if they are imported using relative references—as opposed to explicit URLs.

The usual role of a servlet deployment descriptor—always named **`web.xml`**—is to describe the runtime attributes of a servlet or JSP component. Because JSEs are actually embedded in a servlet at runtime, however, the servlet deployment descriptor —the `web.xml` file—has been co-opted to describe a JSE deployment. Actually, you can deploy servlets, JSPs, and JSEs all in the same WAR file, with the same `web.xml` file. Here's how we identify a JSE in a web.xml:

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
```

```
                          http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <servlet>
        <servlet-name>BookQuoteJSE</servlet-name>
        <servlet-class>com.jwsbook.jaxprc.BookQuote_Impl</servlet-class>
    </servlet>

</web-app>
```

The Web services deployment descriptor, **`webservices.xml`**, tells the container which "servlets" in the `web.xml` are actually JSEs. It does this by identifying a JSE and then referring to the servlet name that appears in the `web.xml` file.

```
<webservices ...>
   <webservice-description>
       <webservice-description-name>BookQuote</webservice-description-name>
       <wsdl-file>/WEB-INF/wsdl/bookquote.wsdl</wsdl-file>
       <jaxrpc-mapping-file>/WEB-INF/bookquote.map</jaxrpc-mapping-file>
       <port-component>
          <port-component-name>BookQuoteJSE</port-component-name>
          <wsdl-port>
              <!-- the fully qualified name of the WSDL port goes here -->
          </wsdl-port>
          <service-endpoint-interface>com.jwsbook.jaxprc.BookQuote
          </service-endpoint-interface>
          <service-impl-bean>
             <servlet-link>BookQuoteJSE</servlet-link>
          </service-impl-bean>
       </port-component>
    </webservice-description>
</webservices>
```

The `webservices.xml` file provides information about the JSE that is not provided by the `web.xml` deployment descriptor. The same `webservices.xml` file is used for both JSEs and EJBs.
The JSE should be able to access the initialization parameters set by the `context-param` element in web.xml at any time by invoking the `ServletContext.getInitParameter()` method.
You also use `web.xml` to set the JNDI ENC entries used by the JSE.

A filter is similar in purpose to a JAX-RPC handler, except it operates on the raw data stream rather than a SAAJ `SOAP Message` object. A filter class implements the `javax.servlet.Filter` interface. It's possible to use filter objects in conjunction with handlers or instead of handlers to provide more control over the entire incoming and outgoing data stream, including HTTP headers.

```
<servlet-mapping>
  <servlet-name>BookQuoteJSE</servlet-name>
  <url-pattern>/BookQuote</url-pattern>

</servlet-mapping>
```

A `servlet-mapping` element assigns a JSE to handle requests for a specific URL. Unlike servlets and JSPs, JSEs are not allowed to use wild-card characters—the URL specified must be an exact URL. In addition, there may be only one URL specified for a JSE. When a JSE is deployed, the deployment tool will use this element to set the endpoint address of the JSE in the `port` element of the WSDL file.

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Monson-Haefel</realm-name>
</login-config>
```

The `login-config` element indicates the type of authentication used and the security realm that the client is to be authenticated in. The `auth-method` element may have one of four values (`BASIC`, `CLIENT-CERT`, `DIGEST`, or `FORM`), but **only two of them must be supported for Web services:**

BASIC, which corresponds to HTTP Basic Authentication and CLIENT-CERT, which corresponds to Symmetric HTTP.

```
<servlet>
  <servlet-name>BookQuoteJSE</servlet-name>
  <servlet-class>
    com.jwsbook.jaxprc.BookQuote_Impl
  </servlet-class>
  <init-param>
    <param-name>limit</param-name>
    <param-value>1000</param-value>
  </init-param>
</servlet>
```

The `init-param` elements, which are nested under the `servlet` element, are similar to the `context-param` elements, in that they provide initialization information to an individual servlet. **There is no way for a JSE to access values configured in the `init-param` element**s, however, so they are essentially useless to you as a Web service developer. Use the `context-param` elements they set context-wide initialization parameters that can be accessed via the `ServletContext as shown below:`

```
  public class InternationalJSE_Impl implements InternationalJSE, javax.xml.rpc.server
  .ServiceLifecycle {
  javax.sql.DataSource dataSource;
  ServletEndpointContext servletEndpointContext;

  public void init(Object context) throws ServiceException{
    ...
  }
  public void destroy(){
    ...
  }
  public void someMethod( ) {
    HttpSession httpSession = servletEndpointContext.getHttpSession();
    Principal principal = servletEndpointContext.getUserPrincipal();
    String param1 = servletEndpointContext.getInitParameter("param1");
  }
}
```

```
<servlet>
  <servlet-name>BookQuoteJSE</servlet-name>
  <servlet-class>
    com.jwsbook.jaxprc.BookQuote_Impl
  </servlet-class>
  <run-as>
    <role-name>MHApplication</role-name>
  <run-as>
</servlet>
```

The `run-as` element, which is nested in the `servlet` element, determines the logical security identity under which the JSE will execute while processing requests. When the JSE accesses resources (JDBC, JMS, and so on) or EJBs, it propagates this identity as its own, for authentication and authorization purposes. The `run-as` element contains exactly one `role-name` element, whose value is a logical security name, which must be mapped to a real security identity in the target environment at deployment time.

Deploying EJB Endpoints

Of all the different transactional attributes for EJBs (Requires, RequiresNew, Supports, NotSupported, Never and Mandatory) only Mandatory is not supported by an EJB endpoint Webservice. Although transaction propagation is important in EJB-to-EJB interactions—when one EJB calls a method on another EJB—it's less important when a SOAP client calls a method on an EJB endpoint. **There is no widely accepted protocol for propagating transactions between SOAP senders and receivers, and no standard way for a SOAP client to propagate its transaction to an EJB endpoint and this is the reason why Mandatory attribute is not supported as it requires the client to start a transaction.** For this reason, you will usually choose **`RequiresNew`** or **`Requires`**, both of which start a new transaction when a method is invoked. If you find you do not need transactions at all, and are considering using

Supports, **Never**, or **NotSupported**, then you may want to consider using a JSE instead. They are easier to develop, and can usually access any resource and perform the same tasks that EJB endpoints can.
The transaction attributes assigned to EJB methods are set in the assembly section of the `ejb-jar.xml` deployment descriptor.

```
<ejb-jar ...>
  ...
  <container-transaction>
      <method>
          <ejb-name>BookQuoteEJB</ejb-name>
          <method-name>getBookPrice</method-name>
      </method>
      <trans-attribute>RequiresNew</trans-attribute>
  </container-transaction>
  ...
</ejb-jar>
```

Just as you can declare transaction attributes at a method level, you can also declare security authorization at a method level. Specifically, you can declare in the deployment descriptor of an EJB endpoint which security roles can access which endpoint methods.

```
<ejb-jar ...>
  ...
    <method-permission>
           <role-name>RetailCustomer</role-name>
           <role-name>MHSalesperson</role-name>
           <method>
               <ejb-name>BookQuoteEJB</ejb-name>
               <method-name>getBookPrice</method-name>
           </method>
    </method-permission>
  ...
</ejb-jar>
```

Any role not specified in the `method-permission` element will not be allowed to call the `getBookPrice()` method on the BookQuote EJB.
The **`unchecked`** element signifies that no security-role authorization check is necessary; that is, any client, even anonymous ones, can invoke the method specified. The asterisk in the `method-name` element is a wild card that indicates that the method permission applies to all methods of the specified EJB endpoint. In many cases you will not be authenticating SOAP clients before handling their requests, so you'll want to use the **`unchecked`** method permission.

```
<ejb-jar ...>
  ...
    <method-permission>
           <unchecked/>
           <method>
               <ejb-name>BookQuoteEJB</ejb-name>
               <method-name>*</method-name>
           </method>
    </method-permission>
  ...
</ejb-jar>
```

In addition to specifying the roles that can access an EJB endpoint method, you can also specify the security identity under which the EJB endpoint will attempt to access other enterprise beans and resources. When one EJB calls another, or when an EJB accesses a resource like a JDBC connection, the EJB container will automatically authorize the calling EJB using its **`run-as`** security identity. The `run-as` security identity is the `java.security.Principal` object that the EJB endpoint presents when it attempts to access other EJBs or resources. In some cases, the EJB endpoint will simply propagate the security identity of the caller. For example, if an authenticated SOAP client calls the BookQuote EJB, the BookQuote EJB may use that caller's `Principal` to access the JDBC connection it uses to fetch wholesale prices from the database. It's also possible to specify that the EJB will execute under its own security identity, without regard to the security identity of the SOAP client. This approach can be beneficial for a couple of reasons: First, the SOAP client may not be authenticated, in which case there is no security identity to propagate. Second, you may want to limit access to resources and other enterprise beans to a specific security identity, and assign that identity to the EJB endpoint. You specify the `run-as` security identity for an EJB endpoint in the `ejb-jar.xml` deployment descriptor:

```
<ejb-jar ...>
    <enterprise-beans>
         <entity>
               <ejb-name>BookQuoteEJB</ejb-name>
               ...
               <security-identity>
                     <run-as>
                           <role-name>Administrator</role-name>
                     </run-as>
               </security-identity>
               ...
         </entity>
    </enterprise-beans>
    ...
</ejb-jar>
```

The above code indicates that no matter who the EJB endpoint's client is, the EJB endpoint will run under the `Administrator` principal and will use that security identity to access other EJBs and resources. If you wanted to specify instead that the EJB endpoint always propagates the security identity of the caller, you would write the `security-identity` element as shown below:

```
<ejb-jar ...>
    <enterprise-beans>
         <entity>
               <ejb-name>BookQuoteEJB</ejb-name>
               ...
               <security-identity>
                     <user-caller-identity/>
               </security-identity>
               ...
         </entity>
    </enterprise-beans>
    ...
</ejb-jar>
```

Service References

JAX-RPC client APIs are accessed via the JNDI ENC.

```
InitialContext jndiContext = new InitialContext();
BookQuoteService service = (BookQuoteService)
      jndiContext.lookup("java:comp/env/service/BookQuoteService");
BookQuote BookQuote_stub = service.getBookQuotePort();
float price = BookQuote_stub.getBookPrice(isbn);
```

All J2EE components that access a Web service using JAX-RPC must declare the Web service reference using a **`service-ref`** element in the ejb-jar.xml/web.xml.

```
<?xml version='1.0' encoding='UTF-8'?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:mh="http://www.Monson-Haefel.org/jwsbook/BookQuote"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                       http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
   version="2.1">
  <enterprise-beans>
    <entity>
      <ejb-name>HypotheticalEJB</ejb-name>
      ...
      <service-ref>
        <service-ref-name>service/BookQuoteService</service-ref-name>
        <service-interface>com.jwsbook.jaxrpc.BookQuoteService
        </service-interface>
        <wsdl-file>META-INF/wsdl/BookQuote.wsdl
        </wsdl-file>
        <jaxrpc-mapping-file>META-INF/mapping.xml</jaxrpc-mapping-file>
        <service-qname>mh:BookQuoteService</service-qname>
      </service-ref>
      ...
    </entity>
  </enterprise-beans>
  ...
</ejb-jar>
```


Here's how to configure the service-ref element in the web.xml:

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mh="http://www.Monson-Haefel.org/jwsbook/BookQuote"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <servlet>
    <servlet-name>HypotheticalServlet</servlet-name>
    <servlet-class>com.jwsbook.jaxrpc.HypotheticalServlet</servlet-class>
  </servlet>
  ...
  <service-ref>
    <service-ref-name>service/BookQuoteService</service-ref-name>
    <service-interface>com.jwsbook.jaxrpc.BookQuoteService
    </service-interface>
    <wsdl-file>WEB-INF/wsdl/BookQuote.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
    <service-qname>mh:BookQuoteService</service-qname>
  </service-ref>
  ...
</web-app>
```

The purpose of the `service-ref` is to describe a specific Web service in detail so that the J2EE container can provide a reference to a JAX-RPC service object, which is used to access that Web service at runtime. A `service-ref` element can be used to access a Web service endpoint using one of the JAX-RPC client APIs. A `service-ref` element contains both mandatory and optional elements, which help describe a Web service.

```
<service-ref>
  xmlns:mh="http://www.Monson-Haefel.org/jwsbook/BookQuote/"BookQuote">
  <service-ref-name>service/BookQuoteService</service-ref-name>
  <service-interface>com.jwsbook.jaxrpc.BookQuoteService</service-interface>
  <wsdl-file>WEB-INF/wsdl/bookquote.wsdl</wsdl-file>
  <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
  <service-qname>mh:BookQuoteService</service-qname>
  <port-component-ref>
    <service-endpoint-interface>com.jwsbook.jaxrpc.BookQuote
    </service-endpoint-interface>
    <port-component-link>BookQuoteJSE</port-component-link>
  </port-component-ref>
  <handler>
    <handler-name>MessageID</handler-name>
    <handler-class>com.jwsbook.jaxrpc.sec06.MessageIDHandler</handler-class>
    <soap-header>mi:message-id</soap-header>
    <soap-role>http://www.Monson-Haefel.com/logger</soap-role>
    <port-name>BookQuotePort</port-name>
  </handler>
  <handler>
    <handler-name>ClientSecurityHandler</handler-name>
    <handler-class>com.jwsbook.jaxrpc.sec09.ClientSecurityHandler
    </handler-class>
    <init-param>
      <param-name>KeyAlgorithm</param-name>
      <param-value>DSA</param-value>
    </init-param>
    <init-param>
      <param-name>KeySize</param-name>
      <param-value>512</param-value>
    </init-param>
    <init-param>
      <param-name>SignatureAlgorithm</param-name>
      <param-value>SHA1</param-value>
    </init-param>
    <soap-header>ds:Signature</soap-header>
    <soap-role>Security</soap-role>
    <port-name>BookQuotePort</port-name>
  </handler>
```



```
</service-ref>
```

1. The `service-ref-name` element declares the JNDI ENC lookup name that a J2EE component will use to obtain a JAX-RPC service reference at runtime.
2. The `service-interface` element provides the fully qualified class name of the JAX-RPC service interface that the JNDI ENC should return. When using JAX-RPC generated stubs, the service interface used is usually the one generated at deployment time. At runtime an implementation of this interface will be returned to the J2EE component via the JNDI ENC.

```
package com.jwsbook.jaxrpc;

public interface BookQuoteService extends javax.xml.rpc.Service{
    public BookQuote getBookQuotePort()
    throws javax.xml.rpc.ServiceException;
}
```

   If, however, the J2EE client uses JAX-RPC dynamic proxies or the DII, the interface named by the `service-interface` element will probably be the general-purpose interface, `javax.xml.rpc.Service`, which is defined by the JAX-RPC API.

```
<service-ref>
  <service-ref-name>service/GeneralService</service-ref-name>
  <service-interface>javax.xml.rpc.Service</service-interface>
  ...
</service-ref>

// A J2EE component using the DII
InitialContext jndiContext = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service)
jndiContext.lookup("java:comp/env/service/GeneralService");
...
Call call = service.createCall(portName,operationName);
```

3. The `wsdl-file` element identifies the WSDL document that describes the Web service endpoint you are referring to. The `service-qname` identifies the specific WSDL `service` you are accessing. The WSDL file must be located in the same JAR or WAR as the J2EE component; you can't point to a WSDL document in some other archive file even if it's in the same EAR (Enterprise ARchive) file. The deployment tool needs to know the exact WSDL `service` definition so it can generate the skeleton code used to marshal SOAP messages to and from the Web service endpoint.
4. The `jaxrpc-mapping-file` element declares the location of the JAX-RPC mapping file. This file defines the XML-to-Java mapping between the WSDL file and the arguments used with the JAX-RPC API.
5. The `port-component-ref` element is declared when the J2EE client will be using the `Service.getPort(Class)` method to obtain a JAX-RPC reference (Only needed for Dynamic Proxy approach). The `port-component-ref` element maps an endpoint interface type to a specific `service-ref` element, so that the JAX-RPC runtime knows which `service-ref` to use for a specific class name.
6. Although JAX-RPC is usually used to access Web service endpoints on other platforms, occasionally a J2EE component may need to access a J2EE Web service (JSE or EJB endpoint) that is deployed in the same J2EE application (the same EAR file). In this case you can declare a `port-component-link` element, which links the JAX-RPC service reference directly to a J2EE endpoint. If the port component were an EJB endpoint located in an EJB JAR in the same EAR, you would use a more qualified path in `port-component-link`

```
<service-ref>
  <service-ref-name>service/BookQuoteService</service-ref-name>
  <service-interface>com.jwsbook.jaxrpc.BookQuoteService</service-interface>
  <wsdl-file>/WEB-INF/wsdl/bookquote.wsdl</wsdl-file>
  <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
  <service-qname>mh:BookQuoteService</service-qname>
  <port-component-ref>
    <service-endpoint-interface>com.jwsbook.jaxrpc.BookQuote
    </service-endpoint-interface>
    <port-component-link>../ejb/BookQuote.jar#BookQuoteJSE
    </port-component-link>
```



```
  </port-component-ref>
  ...
</service-ref>
```

The part of the path before the pound symbol (#) specifies the path to the EJB JAR file within the EAR file. This path is relative to the location of the archive file that contains this `service-ref` element. The value after the pound symbol provides the unique component name within the `ejb-jar.xml` deployment descriptor of that EJB JAR file.

7. The `service-ref` element may optionally declare a set of descriptive elements (`description`, `display-name`, `small-icon`, and `large-icon`) for display in visual deployment tools.
8. The `handler` element used in the `service-ref` element is basically the same as the one used in the `webservices.xml` file, except that it's oriented to SOAP senders instead of SOAP receivers. (ie a J2EE component using a Web service reference, rather than a J2EE endpoint). The message handlers employed by service references filter outgoing and incoming messages. They provide clients with a mechanism to access and modify SOAP messages just before they are sent across the network to the Web service. They also allow clients to modify reply messages received from Web services before the J2EE component gets the reply. Message handlers can be configured in the `service-ref` element as a chain of handlers, which will process request and reply messages in the order that the handlers are declared.
    a. The `handler-class` element identifies the class that implements the handler. The handler class must implement the `javax.xml.rpc.handler.Handler` interface.
    b. The `init-param` elements are optional. They describe the initialization parameters that can be passed to the message handler at the begining of its life cycle. The parameters described by `init-param` are accessed collectively via the `java.util.Map` object returned by `HandlerInfo.getHandlerConfig()` method.
    c. A `handler` element may contain one or more `soap-header` elements, each of which declares the XML qualified names of the SOAP header blocks that the handler is supposed to process. The JAX-RPC runtime uses the qualified name of the header block, as specified by the `soap-header` element, to identify header blocks in the reply message that should be processed by the handler chain. Whether or not a header block is supposed to be processed by the handler chain also depends on the values of the `soap-role` elements.
    d. Remember that the SOAP message will indicate the roles that are supposed to process each header block. If the `actor` attribute is specified, then only nodes that play the designated role may process the header block. If the `actor` attribute is not specified, only the ultimate receiver may process the header block.

```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12" >
  <soap:Header>
      <sec:Signature actor="Security">
           <!-- digital-security elements go here -->
      </sec:Signature>
   </soap:Header>
   <soap:Body>
      <mh:getBookPrice >
          <mh:price>29.95</mh:price>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

The `ClientSecurityHandler` is responsible for processing the digital-security header block because the SOAP reply message declares the correct header-block name (`sec:Signature`) and, just as important, the correct role (`Security`) as identifier by `soap-role`.

e. A JAX-RPC service can support multiple Web service ports, but a message handler is associated with only one WSDL `port`. The `port-name` element identifies the exact WSDL `port` definition that the message handler is associated with. Only messages that are sent or received from the Web service that implements that WSDL `port` definition will use that handler.

When a J2EE component uses JAX-RPC to send a SOAP request message, the handlers declared in the `service-ref` element will process the message before it is sent across the network to the Web service endpoint. When processing a SOAP request message for a J2EE client, the handlers are expected to add header blocks to the SOAP message. For example, the `MessageIDHandler` will add a unique `message-id` header, and the `ClientSecurityHandler` will add a `Signature` header block to the SOAP request. The same message handlers that add header blocks to outgoing request messages can also be used to process header blocks of incoming reply messages. When the J2EE component receives a SOAP reply message, the JAX-RPC runtime will query the entire handler chain to see if any of its handlers process headers in the reply. If so, the handlers will process the message in the reverse of the order that they are declared in the `service-ref` element. Each message handler in the chain gets access to the entire SOAP message, not just the header block it's responsible for. Although message handlers should process only the header blocks they're identified with, there is nothing stopping them from processing other parts of the SOAP message as well**.** If a SOAP reply message contains a header block with the `mustUnderstand` attribute set to `true`, then there must be a message handler associated with the name and role of that header block. If not, the JAX-RPC runtime will throw an exception to the client.

Web Service Descriptors (9.1)

J2EE Web Services requires that you include a Web service deployment descriptor named `webservices.xml` with any archive file that contains a JSE or EJB endpoint. You place `webservices.xml` in the `META-INF` directory of the EJB JAR file for EJB endpoints, and in the `WEB-INF` directory of a WAR file for JSEs.

```
<webservices
   xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:mh="http://www.Monson-Haefel.org/jwsbook/BookQuote/BookQuote"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                  http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
   version="1.1">

      <webservice-description>
         <webservice-description-name>BookQuote</webservice-description-name>
         <wsdl-file>/WEB-INF/wsdl/bookquote.wsdl</wsdl-file>
         <jaxrpc-mapping-file>/WEB-INF/bookquote.map</jaxrpc-mapping-file>
         <port-component>
            <port-component-name>BookQuoteJSE</port-component-name>
            <wsdl-port>mh:BookQuotePort</wsdl-port>
            <service-endpoint-interface>com.jwsbook.jaxrpc.BookQuote
            </service-endpoint-interface>
            <service-impl-bean>
               <servlet-link>BookQuoteJSE</servlet-link>
            </service-impl-bean>
         </port-component>
      </webservice-description>
</webservices>
```

1. The `webservices` element declare one or more `webservice-description` elements.
2. The `webservice-description` element describes a collection of JSEs or EJB endpoints that use the same WSDL file. In other words, there must be a separate `webservice-description` element for each WSDL file in an archive file. The function of the `webservice-description` element is to bind J2EE endpoints to their WSDL `port` definition, implementation deployment descriptor, JAX-RPC mapping file, and endpoint interface.
3. The `wsdl-file` element identifies the location of a WSDL document.
4. The `port-component`, in turn, maps a specific JSE or EJB endpoint to a specific `port` element in that WSDL document. The `wsdl-port` element provides the qualified name of the exact WSDL `port` that corresponds to the J2EE endpoint.

5. For JSEs the WSDL `port` definition and its corresponding `binding` are used to generate the JAX-RPC servlet that will host the JSE. For EJB endpoints the `port` and `binding` definitions are used to generate skeleton code used by the EJB container to marshal incoming SOAP messages into method calls that can be dispatched to the proper stateless bean instance.
6. The `port-component-name` element provides a name to identify a particular JSE or EJB endpoint. This name must be unique within the `webservices.xml` file. Deployment tools display it to identify the J2EE endpoint, and the JAX-RPC mapping file uses it to correlate a mapping with a specific J2EE endpoint.
7. The `web.xml` deployment descriptor used by a JSE does not provide an element for declaring the type of the endpoint interface, so you must declare a fully qualified name for it in `webservices.xml`—specifically, in the `service-endpoint-interface`. You must also declare the interface type for EJB endpoints, but be aware that this type must be the same as the type declared by the `service-endpoint` element in the `ejb-jar.xml` file. This duplication in case of EJB endpoints is required for backward compatibility with J2EE 1.3 Webservices.
8. The `service-impl-bean` element tells the deployment tool which implementation definition is associated with the J2EE endpoint. Specifically, for a JSE it points to a specific `servlet` definition in the `web.xml` file, and for an EJB endpoint it points to a `session-bean` definition in the `ejb-jar.xml` file. The link must point to an implementation definition in the archive file that contains the related `webservices.xml` file. For a JSE the link is declared in a `servlet-link` element, as shown above. The value of the `servlet-link` element must match the value of a `servlet-name` element in the local `web.xml` file. For an EJB endpoint the link is declared using an `ejb-link` element as shown below:

```
<webservices
   xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:mh="http://www.Monson-Haefel.org/jwsbook/BookQuote/BookQuote"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
               http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
   version="1.1">
   <webservice-description>
       <webservice-description-name>BookQuote</webservice-description-name>
       <wsdl-file>/WEB-INF/wsdl/bookquote.wsdl</wsdl-file>
       <jaxrpc-mapping-file>/WEB-INF/bookquote.map</jaxrpc-mapping-file>
       <port-component>
          <port-component-name>BookQuoteEJB</port-component-name>
          <wsdl-port>mh:BookQuotePort</wsdl-port>
          <service-endpoint-interface>com.jwsbook.jaxrpc.BookQuote
          </service-endpoint-interface>
          <service-impl-bean>
             <ejb-link>BookQuoteEJB</ejb-link>
          </service-impl-bean>
       </port-component>
    </webservice-description>
</webservices>
```

   The value of the `ejb-link` element must match the value of an `ejb-name` element in the local `ejb-jar.xml` file.
9. The `jaxrpc-mapping-file` element declares the location of the JAX-RPC mapping file, which defines the mapping between the WSDL file and the J2EE endpoint.
10. In the `webservices.xml` file you can configure a chain of message handlers, which will process incoming messages in the order they are declared, and outgoing messages in the opposite order. This is essentially same as handler element in the service-ref above.

JAX-RPC Mapping Files (9.1)

For every WSDL document, there must be exactly one JAX-RPC mapping file. JAX-RPC mapping files come in two forms. A **lightweight mapping file** is short and simple; it only declares the package mapping. A **heavyweight mapping file** is verbose and complete; it covers every aspect of the WSDL-to-Java mapping.

Lightweight JAX-RPC Mapping Files

If the WSDL document meets certain conditions, then a lightweight mapping will suffice, but if any of the conditions aren't met, you must provide a heavyweight mapping file. Perhaps the most important criteria for a lightweight mapping is the use of RPC/Encoded messaging (which is not what you will have as messaging mode for your BP conformant Webservice).
You may use a lightweight JAX-RPC mapping file if your WSDL document meets the following conditions (only important ones are mentioned here – for full list refer the book pg 750.):

1. The `binding` definition uses the RPC messaging style (`style="rpc"`) and SOAP 1.1 Encoding (`encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"`) for all parts of all input, output, and fault messages.
2. There is only one `service` element, which contains one `port` element.

One detail of mapping for the WSDL (refer to BookQuote.wsdl file) cannot be automatically derived, is: the package name. A lightweight mapping file is required, because we have to specify the package declared by endpoint interfaces, service interfaces, JavaBeans components, JAX-RPC stubs, and other products generated by the JAX-RPC compiler.

```
<?xml version='1.0' encoding='UTF-8' ?>
<java-wsdl-mapping
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
            http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
  version="1.1">
  <package-mapping>
    <package-type>com.jwsbook.jaxrpc</package-type>
    <namespaceURI>
      http://www.Monson-Haefel.com/jwsbook/BookQuote
    </namespaceURI>
  </package-mapping>
</java-wsdl-mapping>
```

```
And the generated SEI is shown below (for BookQuote.wsdl with the above BookQuote-
mapping.xml):

package com.jwsbook.jaxrpc;

public interface BookQuote extends java.rmi.Remote {
    public float getBookPrice(String isbn)
    throws java.rmi.RemoteException, InvalidIsbnFault;
}
```

The `InvalidIsbnFault` exception class, the service interface, its implementation, and the endpoint stub would all be generated with the same package name. In a lightweight JAX-RPC mapping file, all you need to declare is the `package-mapping` element. The JAX-RPC mapping file is stored in the J2EE component's archive file, usually in the same directory as the WSDL document it pertains to (ie WEB-INF/wsdl for JSEs and META-INF/wsdl for EJB Endpoints).

Heavyweight JAX-RPC Mapping Files

When any of the conditions for a lightweight mapping file are violated, you must provide a complete JAX-RPC mapping file that details the mapping between:

- XML complex types and Java beans
- Fault messages and exception classes
- The WSDL `portType` definition and the endpoint interface
- The WSDL `service` definition and the service interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuote"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
```



```
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

<message name="BookQuote_getBookPrice">
  <part name="isbn" type="xsd:string"/>
</message>
<message name="BookQuote_getBookPriceResponse">
  <part name="result" type="xsd:float"/>
</message>
<message name="InvalidIsbnFault" >
  <part name="message" type="xsd:string"/>
</message>
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input message="mh:BookQuote_getBookPrice"/>
    <output message="mh:BookQuote_getBookPriceResponse"/>
    <fault name="InvalidIsbnFault" message="mh:InvalidIsbnFault" />
  </operation>
</portType>
<binding name="BookQuoteBinding" type="mh:BookQuote">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                style="document"/>
   <operation name="getBookPrice">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
    <fault name="InvalidIsbnFault">
      <soap:fault name="InvalidIsbnFault" use="literal" />
    </fault>
  </operation>
</binding>
<service name="BookQuoteService">
  <port name="BookQuotePort" binding="mh:BookQuoteBinding">
    <soap:address
     location="http://www.Monson-Haefel.com/jwsbook/BookQuoteService"/>
  </port>
</service>
</definitions>
```

In the above WSDL file, the binding specifies `document` message style and `literal` encoding. For lightweight mapping a WSDL document must specify the RPC messaging style and SOAP 1.1 Encoding.

```
<?xml version='1.0' encoding='UTF-8' ?>
<java-wsdl-mapping
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
            http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
  version="1.1">

  <package-mapping>
    <package-type>com.jwsbook.jaxrpc</package-type>
    <namespaceURI>
      http://www.Monson-Haefel.com/jwsbook/PurchaseOrder</namespaceURI>
  </package-mapping>
  <exception-mapping>
    <exception-type>com.jwsbook.jaxrpc.InvalidIsbnException</exception-type>
    <wsdl-message>mh:InvalidIsbnFault</wsdl-message>
  </exception-mapping>
  <service-interface-mapping>
    <service-interface>com.jwsbook.jaxrpc.BookQuoteService</service-interface>
    <wsdl-service-name>mh:BookQuoteService</wsdl-service-name>
    <port-mapping>
      <port-name>mh:BookQuotePort</port-name>
      <java-port-name>BookQuotePort</java-port-name>
    </port-mapping>
```

```
  </service-interface-mapping>
  <service-endpoint-interface-mapping>
    <service-endpoint-interface>com.jwsbook.jaxrpc.BookQuote
    </service-endpoint-interface>
    <wsdl-port-type>mh:BookQuote</wsdl-port-type>
    <wsdl-binding>mh:BookQuoteBinding</wsdl-binding>
    <service-endpoint-method-mapping>
      <java-method-name>getBookPrice</java-method-name>
      <wsdl-operation>mh:getBookPrice</wsdl-operation>
      <method-param-parts-mapping>
        <param-position>0</param-position>
        <param-type>java.lang.String</param-type>
        <wsdl-message-mapping>
          <wsdl-message>mh:BookQuote_getBookPriceRequest
          </wsdl-message>
          <wsdl-message-part-name>isbn</wsdl-message-part-name>
          <parameter-mode>IN</parameter-mode>
        </wsdl-message-mapping>
      </method-param-parts-mapping>
      <wsdl-return-value-mapping>
        <method-return-value>float</method-return-value>
        <wsdl-message>mh:BookQuote_getBookPriceResponse</wsdl-message>
        <wsdl-message-part-name>result</wsdl-message-part-name>
      </wsdl-return-value-mapping>
    </service-endpoint-method-mapping>
  </service-endpoint-interface-mapping>
</java-wsdl-mapping>
```

Anatomy of a Mapping File

Following is the general structure of the mapping file:

```
<java-wsdl-mapping...>
  <package-mapping/>
  <java-xml-type-mapping/>
  <exception-mapping/>
  <service-interface-mapping/>
  <service-endpoint-interface-mapping>
      ...
      <service-endpoint-method-mapping/>
  </service-endpoint-interface-mapping>
</java-wsdl-mapping>
```

1. The `java-wsdl-mapping` element is the root of the JAX-RPC mapping file.
2. When you use generated stubs and dynamic proxies, a JAX-RPC compiler will use the `package-mapping` element to generate Java class and interface definitions for a variety of types defined in the WSDL document: (Each of these classes and interfaces needs to be placed in one Java package or another—the `package-mapping` element dictates which.) You map the **package-type** (a java package name) to the **namespaceURI** (the targetNamespace for the WSDL).
    a. An endpoint interface for the WSDL `portType` element
    b. An endpoint stub that implements the endpoint interface (when you're using JAX-RPC generated stubs)
    c. A service interface for the WSDL `service` element
    d. A service implementation for the service interface (when you're using JAX-RPC generated stubs)
    e. A bean or plain Java class for each complex or simple type declared in the `types` element
    f. Holder types for INOUT and OUT parameters of `operation` elements
3. If the WSDL document uses more than one namespace for generated types, you must declare a separate `package-mapping` element for each XML namespace.

```
<?xml version='1.0' encoding='UTF-8' ?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:mh="http://www.Monson-Haefel.com/jwsbook/PurchaseOrder"
    targetNamespace="http://www.Monson-Haefel.com/jwsbook/PurchaseOrder"
    xmlns:soap="http://schemas.xmlsoap.org.wsdl/soap/"
```

```
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
    xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO" >
  <types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.Monson-Haefel.com/PO">
      <element name="purchaseOrder" type="po:PurchaseOrder"/>
      <complexType name="PurchaseOrder">
        <sequence>
          <element name="accountName" type="xsd:string"/>
          <element name="accountNumber" type="xsd:short"/>
          <element name="shipAddress" type="po:USAddress"/>
          <element name="billAddress" type="po:USAddress"/>
        </sequence>
        <attribute name="orderDate" type="xsd:date"/>
      </complexType>
      <complexType name="USAddress">
        <sequence>
          <element name="name" type="xsd:string"/>
          <element name="street" type="xsd:string"/>
          <element name="city" type="xsd:string"/>
          <element name="state" type="xsd:string"/>
          <element name="zip" type="xsd:string"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  <message name="PurchaseOrderMessage">
    <part name="body" element="po:purchaseOrder"/>
  </message>
  <portType name="PurchaseOrder">
    <operation name="submitPurchaseOrder">
      <input message="mh:PurchaseOrderMessage"/>
    </operation>
  </portType>
  ...
</definitions>
```

In this case the XML namespace used for the WSDL definitions (`message`, `portType`, `binding`, `service`, and `port`) is different from the XML namespace used to define the XML schema types (`PurchaseOrder` and `USAddress`). Because the JAX-RPC compiler will create interfaces and classes for types from both namespaces, the mapping file must have a `package-mapping` element for each namespace:

```
<java-wsdl-mapping ...>
  <package-mapping>
    <package-type>com.jwsbook.jaxrpc</package-type>
    <namespaceURI>http://www.Monson-Haefel.com/jwsbook/PurchaseOrder
    </namespaceURI>
  </package-mapping>
  <package-mapping>
    <package-type>com.jwsbook.jaxrpc.types</package-type>
    <namespaceURI>http://www.Monson-Haefel.com/jwsbook/PO</namespaceURI>
  </package-mapping>
</java-wsdl-mapping>
```

You could map both namespaces to the same Java package, which is often more convenient.

4. The `java-xml-type-mapping` element is necessary when you are using complex or simple types, whether defined in the `types` element or imported from another XML document—with one exception. This element is not necessary if you're using standard XML schema built-in types with standard mapping to Java.

```
<?xml version='1.0' encoding='UTF-8' ?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:po="http://www.Monson-Haefel.com/1ed/PurchaseOrder"
    xmlns:soap="http://schemas.xmlsoap.org.wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
    xmlns:mh="http://www.Monson-Haefel.com/jwsbook/PurchaseOrder"
    targetNamespace="http://www.Monson-Haefel.com/jwsbook/PurchaseOrder">
  <types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.Monson-Haefel.com/1ed/PurchaseOrder">
```

```
      <element name="purchaseOrder" type="po:PurchaseOrder"/>
      <complexType name="PurchaseOrder">
        <sequence>
          <element name="AcctName" type="xsd:string"/>
          <element name="ShipAddr" type="po:USAddress"/>
          <element name="ISBN" type="xsd:string" maxOccurs="unbounded"/>
        </sequence>
      </complexType>
      <complexType name="USAddress">
        <sequence>
          <element name="Street" type="xsd:string"/>
          <element name="City" type="xsd:string"/>
          <element name="State" type="xsd:string"/>
          <element name="Zip" type="xsd:string"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  <message name="PurchaseOrderMessage">
    <part name="body" element="po:purchaseOrder"/>
  </message>
  <portType name="PurchaseOrder">
    <operation name="submitPurchaseOrder">
      <input message="mh:PurchaseOrderMessage"/>
    </operation>
  </portType>
  ...
</definitions>
```

In this case the mapping file will need a `java-xml-type-mapping` element for each complex type, one for `PurchaseOrder` and one for `USAddress`.

```
<java-to-xml-mapping>
   <class-type>com.jwsbook.jaxrpc.PurchaseOrder</class-type>
   <root-type-qname>po:PurchaseOrder</root-type-qname>
   <qname-scope>element</qname-scope>
   <variable-mapping>
      <java-variable-name>accountName</java-variable-name>
      <xml-element-name>AcctName</xml-element-name>
   </variable-mapping>
   <variable-mapping>
      <java-variable-name>shipAddress</java-variable-name>
      <xml-element-name>ShipAddr</xml-element-name>
   </variable-mapping>
   <variable-mapping>
      <java-variable-name>isbn</java-variable-name>
      <xml-element-name>ISBN</xml-element-name>
   </variable-mapping>
</java-to-xml-mapping>

<java-to-xml-mapping>
<class-type>com.jwsbook.jaxrpc.USAddress</class-type>
   <root-type-qname>po:USAddress</root-type-qname>
   <qname-scope>complexType</qname-scope>
   <variable-mapping>
      <java-variable-name>street</java-variable-name>
      <xml-element-name>Street</xml-element-name>
   </variable-mapping>
   <variable-mapping>
      <java-variable-name>city</java-variable-name>
      <xml-element-name>City</xml-element-name>
   </variable-mapping>
   <variable-mapping>
      <java-variable-name>state</java-variable-name>
      <xml-element-name>State</xml-element-name>
   </variable-mapping>
   <variable-mapping>
      <java-variable-name>zip</java-variable-name>
      <xml-element-name>Zip</xml-element-name>
   </variable-mapping>
</java-to-xml-mapping>
```

5. An `exception-mapping` element maps a WSDL fault message to a Java exception class.

6. The `service-interface-mapping` element maps a WSDL `service` definition to a custom JAX-RPC service interface type. It also specifies the names and types of the `getPortName()` methods, which return references to generated endpoint stubs at runtime.
7. The `service-endpoint-interface-mapping` element maps a JAX-RPC endpoint interface to a specific set of WSDL `portType` and `binding` definitions. This element helps the JAX-RPC compiler generate the proper endpoint stub and endpoint interfaces. It also details how WSDL `operation` and `message part` definitions map to endpoint methods.

Web service Security

[MZ Notes + XYZWS.COM study guide + WST chapters 3 and 4 + Chapter 7 of Blueprints]
8.1 Explain basic security mechanisms including: transport level security, such as basic and mutual authentication and SSL, message level security, XML encryption, XML Digital Signature, and federated identity and trust.
8.2 Identify the purpose and benefits of Web services security oriented initiatives and standards such as Username Token Profile, SAML, XACML, XKMS, WS-Security, and the Liberty Project.
8.3 Given a scenario, implement J2EE based web service web-tier and/or EJB-tier basic security mechanisms, such as mutual authentication, SSL, and access control.
8.4 Describe factors that impact the security requirements of a Web service, such as the relationship between the client and service provider, the type of data being exchanged, the message format, and the transport mechanism.
There are two ways with which we can ensure security with Web Services. They are:

1. Security at Transport level
2. Security at XML level

Security at Transport Level

Transport level security is based on Secure Sockets Layer (SSL) or Transport Layer Security (TLS) that runs beneath HTTP. SSL and TLS provide security features including authentication, data protection, and cryptographic token support for secure HTTP connections. SSL is the Industry accepted standard protocol for secured encrypted communications over TCP/IP. In this model, a Web Service client will use SSL to open a secure socket to a Web Service. The client then sends and receives SOAP messages over this secured socket using HTTPS. The SSL implementation takes care of ensuring privacy by encrypting all the network traffic on the socket. SSL can also authenticate the Web Service to the client using the PKI infrastructure.

HTTPS provides encryption, which ensures privacy and message integrity. HTTPS also authenticates through the use of certificates, which can be used on the server side, the client side, or both. HTTPS with server-side certificates is the most common configuration on the Web today. In this configuration, clients can authenticate servers, but servers cannot authenticate clients. However, HTTPS can also be used in conjunction with basic or digest authentication, which provides a weaker form of authentication for clients.

Security at XML Level

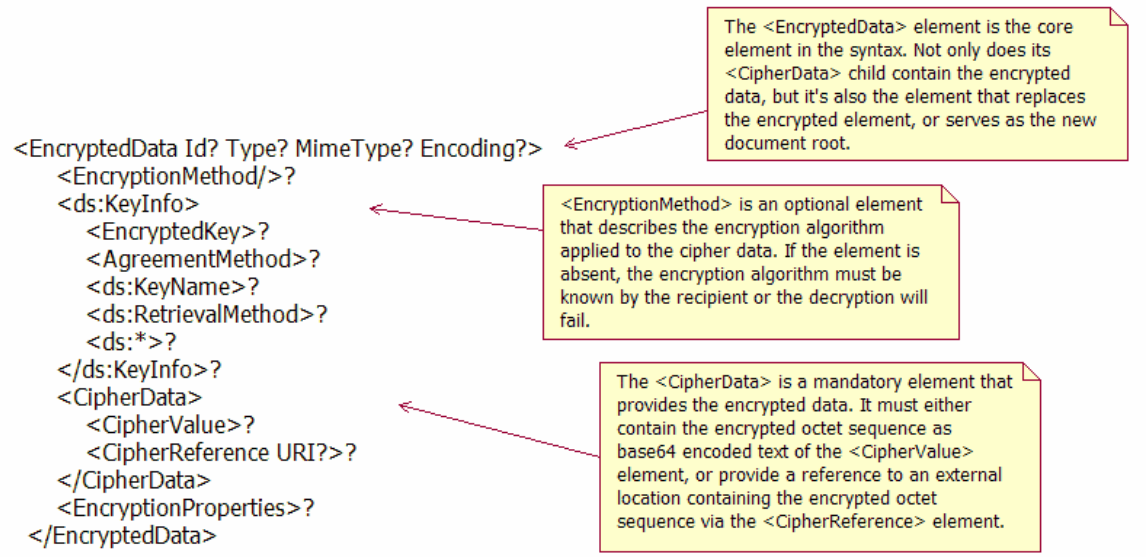
There are some standards available for securing Web Services at XML level.

XML Encryption

The W3C is coordinating XML Encryption. Its goal is to develop XML syntax for representing encrypted data and to establish procedures for encrypting and decrypting such data. Unlike SSL, with XML Encryption, you can encrypt only the data that needs to be encrypted, for example, only the credit card information in a purchase order XML document:

```
<purchaseOrder>
  <name>Mikalai Zaikin</name>
  <address> ... </address>

  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
                 xmlns='http://www.w3.org/2000/11/temp-xmlenc'>
```

```
    <EncryptionMethod Algorithm="urn:nist-gov:tripledes-ede-cbc">
       <s0:someMethod xmlns:s0='http://somens'>ABCD</s0:someMethod>
    </EncryptionMethod>
    <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
      <KeyName>SharedKey</KeyName>
    </KeyInfo>
    <CipherData>
        <CipherValue>A23B45C564562347e23e</CipherValue>
    </CipherData>
  </EncryptedData>

  <prodNumber>8a32gh19908</prodNumber>
  <quantity>1</quantity>
</purchaseOrder>
```



In XML Encryption, your plaintext is either an element or that element's content (that's the finest granularity you get—you can't encrypt, say, half an element's content). After encryption, you get an XML element called EncryptedData, containing the ciphertext in Base64-encoded format. That EncryptedData element replaces your plaintext. That is, if you encrypt element bar in this snippet below:

```
<foo>
  <bar>secret text</bar>
</foo>
```

you'll get back something like this:

```
<foo>
  <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
                 xmlns="http://www.w3.org/2001/04/xmlenc#"...>
    <!-- some info, including the ciphertext -->
  </EncryptedData>
</foo>
```

Whereas if you encrypt element bar's content, the result will look similar to this:

```
<foo>
  <bar>
    <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Content" xmlns=...>
      <!-- some info, including the ciphertext -->
    </EncryptedData>
  </bar>
</foo>
```

From looking at the **type** attribute, we can tell immediately whether the plaintext is an element or just its content.

If the application scenario requires all of the information to be encrypted, the whole document is encrypted as an octet sequence. This applies to arbitrary data including XML documents:

```
<?xml version='1.0'?>
<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#' MimeType='text/xml'>
  <CipherData>
    <CipherValue>A23B45C56</CipherValue>
  </CipherData>
</EncryptedData>
```

An XML document may contain zero or more EncryptedData elements. EncryptedData cannot be the parent or child of another EncryptedData element. However, the actual data encrypted can be anything, including EncryptedData and EncryptedKey elements (i.e., **super-encryption**). During super-encryption of an EncryptedData or EncryptedKey element, one must encrypt the entire element.

```
<pay:PaymentInfo xmlns:pay='http://example.org/paymentv2'>
  <EncryptedData Id='ED1' xmlns='http://www.w3.org/2001/04/xmlenc#'
            Type='http://www.w3.org/2001/04/xmlenc#Element'>
    <CipherData>
      <CipherValue>originalEncryptedData</CipherValue>
    </CipherData>
  </EncryptedData>
</pay:PaymentInfo>
```

A valid super-encryption of "//xenc:EncryptedData[@Id='ED1']" would be:

```
<pay:PaymentInfo xmlns:pay='http://example.org/paymentv2'>
  <EncryptedData Id='ED2' xmlns='http://www.w3.org/2001/04/xmlenc#'
            Type='http://www.w3.org/2001/04/xmlenc#Element'>
    <CipherData>
      <CipherValue>newEncryptedData</CipherValue>
    </CipherData>
  </EncryptedData>
</pay:PaymentInfo>
```

where the CipherValue content of 'newEncryptedData' is the base64 encoding of the encrypted octet sequence resulting from encrypting the EncryptedData element with Id='ED1'.

XML Digital Signature

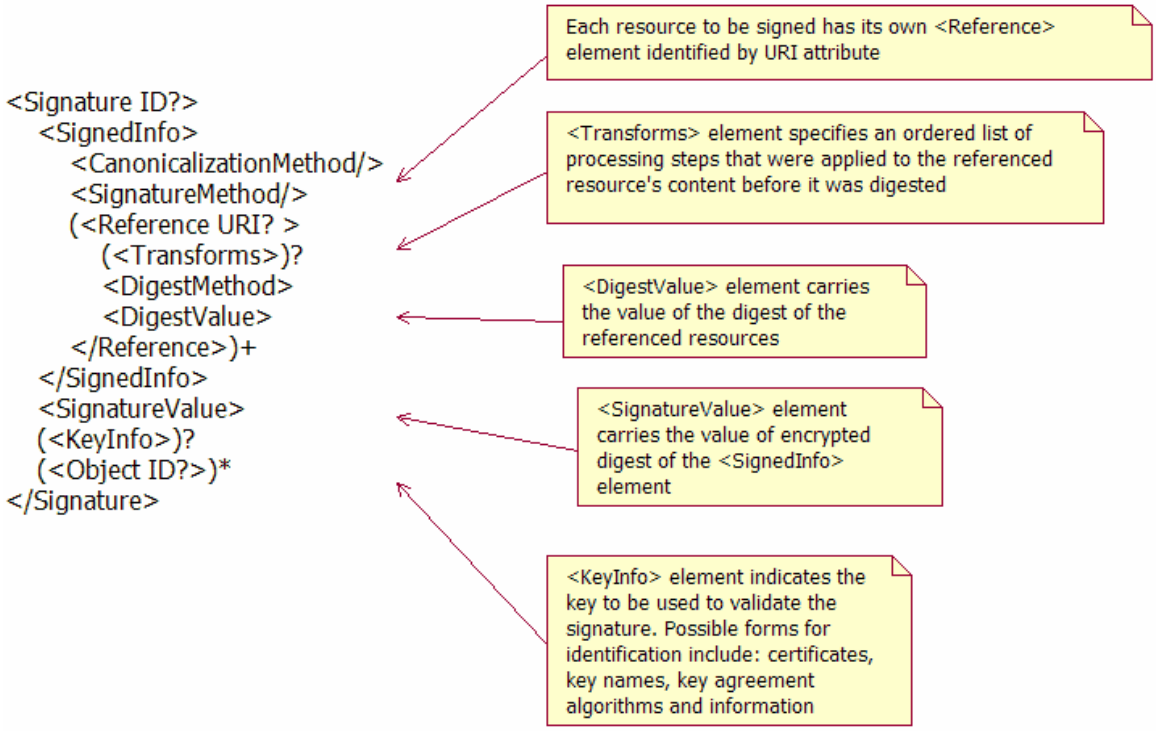
XML Digital Signature, like any other digital signing technology, provides authentication, data integrity (tamper-proofing), and **nonrepudiation**. Of all the XML-based security initiatives, the XML digital signature effort is the furthest along. The project aims to develop XML syntax for representing digital signatures over any data type. The XML digital signature specification also defines procedures for computing and verifying such signatures. Another important area that XML digital signature addresses is the **canonicalization of XML documents**. **Canonicalization enables the generation of the identical message digest and thus identical digital signatures for XML documents that are syntactically equivalent but different in appearance due to, for example, a different number of white spaces present in the documents**.

Benefits:

1. XML Digital Signature provides a flexible means of signing and supports diverse sets of Internet transaction models.
    a. you can sign individual items or multiple items of an XML document.
    b. The document you sign can be local or even a remote object, as long as those objects can be referenced through a URI (Uniform Resource Identifier).
    c. You can sign not only XML data, but also non-XML data.
    d. A signature can be either **enveloped** or **enveloping**, which means the signature can be either embedded in a document being signed or reside outside the document.

e. XML digital signature also allows multiple signing levels for the same content, thus allowing flexible signing semantics. For example, the same content can be semantically signed, cosigned, witnessed, and notarized by different people.

2. It supports signing complete XML documents, parts of XML documents and even non-XML documents. The resulting signature is a well-formed XML fragment that can either be used on its own (a standalone XML document) or embedded within a more complex XML document.



Enveloped XML Digital Signature:

An enveloped signature is useful when you have a simple XML document which you to guarantee the integrity of. For example, XKMS messages can use enveloped signatures to convey "trustable" answers from a server back to a client. The signature is over the XML content that contains the signature as an element. The content provides the root XML document element. Signature is enveloped within the content been signed:

```
<doc Id="myID">
        <myElement>
                ...
        </myElement>
        <Signature>
                ...
                <Reference URI="#myID"/>
                ...
        </Signature>
</doc>
```

Enveloping XML Digital Signature:

An enveloping signature is useful when the signing facility wants to add its own metadata (such as a timestamp) to a signature - it doesn't have to modify the source document, but can include additional data covered by the signature within the signature document it generates. (An XML Digital Signature can sign multiple objects at once, so enveloping is usually combined with another format). The signature is over content found within an Object element of the signature itself. The Object (or its content) is identified via a Reference (via a URI fragment identifier or transform). Signature envelopes the contents to be signed:

```
<Signature>
        ...
        <Reference URI="#myRefObjectID">
```

```
                ...
        </Reference>

        <Object Id="myRefObjectID">
                <doc>
                        <myElement>
                                ...
                        </myElement>
                        ...
                </doc>
        </Object>
</Signature>
```

Detached:
A detached signature is useful when you can't modify the source; the downside is that it requires two XML documents - the source and its signature - to be carried together. In other words, it requires a packaging format - enter SOAP headers.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Header>
      <mi:message-id soap:actor="http://www.Monson-Haefel.com/logger">
          deabc782dbbd11bf:29e357:f1ad93fdbf:-8000
      </mi:message-id>
      <proc:processed-by
        soap:actor="http://schemas.xmlsoap.org/soap/actor/next">
        <node>
          <time-in-millis>1013694684723</time-in-millis>
          <identity>http://www.client.com/JaxRpcExample_6</identity>
        </node>
      </proc:processed-by>
      <sec:Signature>
        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <ds:SignedInfo>
            ...
          </ds:SignedInfo>
          <ds:SignatureValue>CFFOMFCtVLrklR...</ds:SignatureValue>
        </ds:Signature>
      </sec:Signature>
   </soap:Header>
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

The signature is over content external to the Signature element, and can be identified via a URI or transform. Consequently, the signature is "detached" from the content it signs. This definition typically applies to separate data objects, but it also includes the instance where the Signature and data object reside within the same XML document but are SIBLING elements.

```
<Signature>
        ...
        <Reference URI="http://www.buy.com/books/purchaseWS"/>
        ...
</Signature>
```

XML Signatures are applied to arbitrary digital content (data objects) via an indirection [through **Reference** element]. Data objects are digested, the resulting value is placed in an element (with other information) [**SignedInfo** element] and that element is then digested and cryptographically signed [this is the value of **SignatureValue** element].

```
<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo Id="foobar">
    <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />

    <Reference URI="http://www.abccompany.com/news/2000/03_27_00.htm">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
    </Reference>

    <Reference          URI="http://www.w3.org/TR/2000/WD-xmldsig-core-20000228/signature-
example.xml">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>UrXLDLBIta6skoV5/A8Q38GEw44=</DigestValue>
    </Reference>

  </SignedInfo>

  <SignatureValue>MC0E~LE=</SignatureValue>

  <KeyInfo>
    <X509Data>
      <X509SubjectName>CN=Ed Simon,O=XMLSec Inc.,ST=OTTAWA,C=CA</X509SubjectName>
      <X509Certificate>
        MIID5jCCA0+gA...lVN
      </X509Certificate>
    </X509Data>
  </KeyInfo>
```

</Signature>The **CanonicalizationMethod** is the algorithm that is used to canonicalize the SignedInfo element before it is digested as part of the signature operation. The **SignatureMethod** is the algorithm that is used to convert the canonicalized SignedInfo into the SignatureValue. The algorithm names [Algorithm attribute] are signed to resist attacks based on substituting a weaker algorithm. To promote application interoperability we specify a set of signature algorithms that MUST be implemented, though their use is at the discretion of the signature creator. We specify additional algorithms as RECOMMENDED or OPTIONAL for implementation; the design also permits arbitrary user specified algorithms. Each **Reference** element includes the digest method and resulting digest value calculated over the identified data object. It also may include transformations that produced the input to the digest operation. A data object is signed by computing its digest value and a signature over that value. The signature is later checked via reference and signature validation. **KeyInfo** indicates the key to be used to validate the signature. Possible forms for identification include certificates, key names, and key agreement algorithms and information - we define only a few. KeyInfo is **optional** for two reasons. First, the signer may not wish to reveal key information to all document processing parties. Second, the information may be known within the application's context and need not be represented explicitly. Since KeyInfo is outside of SignedInfo, if the signer wishes to bind the keying information to the signature, a Reference can easily identify and include the KeyInfo as part of the signature.

To verify the signature of SignedInfo element: calculate the digest of SignedInfo using the digest algorithm mentioned in SignatureMethod and compare it with SignatureValue. If this stage passes, then calculate the digest value for references contained in SignedInfo using DigestMethod algorithm and match them with corresponding DigestValue.

Federated Identity and Trust:
There are two possible identity management architectures, one based on a centralized model and the other, on a federated model.

1. **Centralized Model:** In the centralized model, a single operator performs authentication and authorization by owning and controlling all the identity information. In the federated model, both authentication and authorization tasks are distributed among federated communities.
   Pros:
   a. because a single operator owns and controls everything, constructing and managing the identity network could be easier than with the federated model.

Cons:

b. the dangerous potential for the single operator becoming a tollgate for all transactions over the Internet.
c. the single operator could represent a single point of security failure or hacker attack.
d. a single operator can take away the most important business asset—that is, customer identity and profile information—from an organization. That results in a serious threat to businesses such as banks and brokerage houses whose success depends on their customer information.

2. **Federated Model:** The goal of the **Liberty Alliance Project** (driver of the federated model) is to create an open standard for identity, authentication, and authorization, which will lower e-commerce costs and accelerate organizations' commercial opportunities, while at the same time increasing customer satisfaction. In a Liberty architecture, organizations can maintain their own customer/employee data while sharing identity data with partners based on their business objectives and customer preferences. In the federated identity management architecture scheme, three roles could exist.
    a. **Consumer:** As a consumer, you can have multiple identity profiles, and you can ask different identity providers to maintain these profiles.
    b. **Identity providers** maintain user profile information and can interoperate among themselves as long as they have permission to do so from the profile's owner, the consumer.
    c. the **service provider**, the merchant who has services to offer consumers. Service providers can customize their services to each consumer by retrieving relevant identity profiles from the identity providers. For example, your travel agent might discover your travel and dining preferences from the identity provider you designated to maintain your travel preference.

In the phase with no federation (separate login for each site), a consumer must log in separately to each site. This phase will then evolve into an environment where multiple identity networks exist. Within a single identity network, single sign-on can be achieved. However, no network-to-network identity propagation is available at this stage. Eventually, these individually constructed and operating identity networks will work together by exchanging their consumers' identity information, thus providing a truly seamless, global-scale identity network, the Liberty Alliance Project's ultimate goal. The ATM network serves as an analogy for the federated network. Initially, individual banks issued their own ATM cards, and different banks did not interoperate. At this stage, you could not use your ATM card in an ATM machine owned and operated by another bank. These days, you can use your credit card or ATM card in any ATM machine, as long as the bank that owns the machine and your bank are members of the same affiliation network. In the not too distant future, it is not a stretch to think about a single global network to which all banks directly or indirectly belong. The identity network should evolve similarly. One possible challenge of the federated identity network model is that because there are many parties involved, the standard has to be defined in an unambiguous manner. The Liberty Alliance Project addresses that challenge.

Federated Identity allows users to link identity information between accounts without centrally storing personal information. Also, the user can control when and how their accounts and attributes are linked and shared between domains and Service Providers, allowing for greater control over their personal data. In practice, this means that users can be authenticated by one company or website and be recognized and delivered personalized content and services in other locations without having to re-authenticate or sign on with a separate username and password.

"Circle of Trust" is enabled through federated identity and is defined as a group of service providers that share linked identities and have pertinent business agreements in place regarding how to do business and interact with identities. Once a user has been authenticated by a Circle of Trust identity provider, that individual can be easily recognized and take part in targeted services from other service providers within that Circle of Trust.

SAML (Security Assertions Markup Language)

In a nutshell, SAML is an XML-based framework for exchanging security information (authentication and authorization information). As a framework, it deals with three things.

1. First, it defines syntax and semantics of XML-encoded assertion messages.
2. Second, it defines request and response protocols between requesting and asserting parties for exchanging security information.
3. Third, it defines rules for using assertions with standard transport and message frameworks.

The security information for exchanging is expressed in the form of **assertions about subjects**, where a subject is an entity (either human or computer) that has an identity in some security domain. A typical example of a subject is a person, identified by his or her email address in a particular Internet DNS domain.

Assertions can convey information about authentication acts performed by subjects, attributes of subjects, and authorization decisions about whether subjects are allowed to access certain resources. Assertions are represented as XML constructs and have a nested structure, whereby a single assertion might contain several different internal statements about authentication, authorization, and attributes. Note that assertions containing authentication statements merely describe acts of authentication that happened previously.

Assertions are issued by SAML authorities, namely, authentication authorities, attribute authorities, and policy decision points. SAML defines a protocol by which clients can request assertions from SAML authorities and get a response from them. This protocol, consisting of XML-based request and response message formats, can be bound to many different underlying communications and transport protocols; SAML currently defines one binding, to SOAP over HTTP.
SAML authorities can use various sources of information, such as external policy stores and assertions that were received as input in requests, in creating their responses. Thus, while clients always consume assertions, SAML authorities can be both producers and consumers of assertions.

One major design goal for SAML is Single Sign-On (SSO), the ability of a user to authenticate in one domain and use resources in other domains without re-authenticating. However, SAML can be used in various configurations to support additional scenarios as well. Several profiles of SAML are currently being defined that support different styles of SSO and the securing of SOAP payloads.

XACML (Extensible Access Control Markup Language)

Its primary goal is to standardize access control language in XML syntax. A standard access control language results in lower costs because there is no need to develop an application-specific access control language or write the access control policy in multiple languages. Plus, system administrators need to understand only one language. With XACML, it is also possible to compose access control policies from the ones created by different parties.
XACML is an OASIS standard that describes both a policy language and an access control decision request/response language (both written in XML). The policy language is used to describe general access control requirements, and has standard extension points for defining new functions, data types, combining logic, etc. The request/response language lets you form a query to ask whether or not a given action should be allowed, and interpret the result. The response always includes an answer about whether the request should be allowed using one of four values: Permit, Deny, Indeterminate (an error occurred or some required value was missing, so a decision cannot be made) or Not Applicable (the request can't be answered by this service). The typical setup is that someone wants to take some action on a resource. They will make a request to whatever actually protects that resource (like a filesystem or a web server), which is called a Policy Enforcement Point (PEP). The PEP will form a request based on the requester's attributes, the resource in question, the action, and other information pertaining to the request. The PEP will then send this request to a Policy Decision Point (PDP), which will look at the request and some policy that applies to the request, and come up with an answer about whether access should be granted. That answer is returned to the PEP, which can then allow or deny access to the requester. XACML also includes an access decision language used to represent the runtime request for a resource. When a policy is located which protects a resource, functions compare attributes in the request against attributes contained in the policy rules ultimately yielding a permit or deny decision. When a client makes a resource request upon a server, the

entity charged with access control by enforcing authorization is called the Policy Enforcement Point. In order to enforce policy, this entity will formalize attributes describing the requester at the Policy Information Point and delegate the authorization decision to the Policy Decision Point. Applicable policies are located in a policy store and evaluated at the Policy Decision Point, which then returns the authorization decision. Using this information, the Policy Enforcement Point can deliver the appropriate response to the client. An administrator creates policies in the XACML language. The key top-level element is the **PolicySet** which aggregates other **PolicySet** elements or **Policy** elements. The Policy element is composed principally of **Target**, **Rule** and **Obligation** elements and is evaluated at the Policy Decision Point to yield and access decision. Since multiple policies may be found applicable to an access decision, (and since a single policy can contain multiple Rules) Combining Algorithms are used to reconcile multiple outcomes into a single decision. The **Target** element is used to associate a requested resource to an applicable Policy. It contains conditions that the requesting **Subject**, **Resource**, or **Action** must meet for a Policy Set, Policy, or Rule to be applicable to the resource. The Target includes a build-in scheme for efficient indexing/lookup of Policies. **Rules** provide the conditions which test the relevant attributes within a Policy. Any number of Rule elements may be used each of which generates a true or false outcome. Combining these outcomes yields a single decision for the Policy, which may be "Permit", "Deny", "Indeterminate", or a "NotApplicable" decision.

XKMS (XML Key Management Specification)

It consists of two parts: **XKISS** (XML Key Information Service Specification) and **XKRSS** (XML Key Registration Service Specification). XKISS defines a protocol for resolving or validating public keys contained in signed and encrypted XML documents, while XKRSS defines a protocol for public key registration, revocation, and recovery. The key aspect of XKMS is that it serves as a protocol specification between an XKMS client and an XKMS server in which the XKMS server provides trust services to its clients (in the form of Web services) by performing various PKI (public key infrastructure) operations, such as public key validation, registration, recovery, and revocation on behalf of the clients.

One of the obstacles to PKI's wide adoption is that PKI operations such as public key validation, registration, recovery, and revocation are complex and require large amounts of computing resources, which prevents some applications and small devices such as cell phones from participating in PKI-based e-commerce or Web services transactions. XKMS enables an XKMS server to perform these PKI operations. In other words, applications and small devices, by sending XKMS messages over SOAP (Simple Object Access Protocol), can ask the XKMS server to perform the PKI operations. In this regard, the XKMS server provides trust services to its clients in the form of Web services.

XKMS defines a Web services interface to a public key infrastructure. This makes it easy for applications to interface with key-related services, like registration and revocation, and location and validation. Most developers will only ever need to worry about implementing XKMS clients. XKMS server components are mostly implemented by providers of public key infrasructure (PKI) providers, such as Entrust, Baltimore and VeriSign. VeriSign, for example, provides an XKMS responder that can be used to register and query VeriSign's certificate store. Even SSL server ID's can be validated in real-time using the XKMS interface. When combined with WS-Security, XKMS makes it easier than ever for developers to deploy enterprise applications in the form of secure Web services available to business partners beyond the firewall.
An XKMS-compliant service supports the following operations:

1. **Register**: XKMS services can be used to register key pairs for escrow services. Generation of the public key pair may be performed by either the client or the registration service. Once keys are registered, the XKMS-compliant service manages the revocation and recovery of registered keys, whether client- or server-generated. Additional functions are reissue, revoke, and recover.
2. **Locate**: The Locate service is used to retrieve a public key registered with an XKMS-compliant service. The public key can in turn be used to encrypt a document or verify a signature.
3. **Validate**: The Validate service is used to ensure that a public key registered with an XKMS-compliant service is valid, and has not expired or been revoked. The validation service can also be used to check attributes against a public key.

WS-Security

The goal of WS-Security is to enable applications to construct secure SOAP message exchanges.

The WS-Security (Web Services Security) specification defines a set of SOAP header extensions for end-to-end SOAP messaging security. It supports message integrity and confidentiality by allowing communicating partners to exchange signed and encrypted messages in a Web services environment. Because it is based on XML digital signature and XML Encryption standards, you can digitally sign and encrypt any combination of message parts. WS-Security supports multiple security models, such as username/password-based and certificate-based models. It also supports multiple security technologies, including Kerberos, PKI, SAML, and so on.

```
(001) <?xml version="1.0" encoding="utf-8"?>
(002)  <S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
                   xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
(003)    <S:Header>
(004)       <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
(005)         <m:action>http://fabrikam123.com/getQuote</m:action>
(006)         <m:to>http://fabrikam123.com/stocks</m:to>
(007)         <m:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</m:id>
(008)       </m:path>
(009)       <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
(010)         <wsse:UsernameToken Id="MyID">
(011)             <wsse:Username>Mikalai</wsse:Username>
(012)         </wsse:UsernameToken>
(013)         <ds:Signature>
(014)            <ds:SignedInfo>
(015)               <ds:CanonicalizationMethod  Algorithm="..."/>
(016)               <ds:SignatureMethod  Algorithm="..."/>
(017)               <ds:Reference URI="#MsgBody">
(018)                  <ds:DigestMethod Algorithm="..."/>
(019)                  <ds:DigestValue>LyLsF0Pi4wPU...</ds:DigestValue>
(020)               </ds:Reference>
(021)            </ds:SignedInfo>
(022)            <ds:SignatureValue>DJbchm5gK...</ds:SignatureValue>
(023)            <ds:KeyInfo>
(024)                <wsse:SecurityTokenReference>
(025)                 <wsse:Reference URI="#MyID"/>
(026)                </wsse:SecurityTokenReference>
(027)            </ds:KeyInfo>
(028)         </ds:Signature>
(029)      </wsse:Security>
(030)   </S:Header>
(031)   <S:Body Id="MsgBody">
(032)     <tru:StockSymbol xmlns:tru="http://fabrikam123.com/payloads">
             IBA-USA
          </tru:StockSymbol>
(033)   </S:Body>
(034) </S:Envelope>
```

WS-Security specification provides a means to protect a message by encrypting and/or digitally signing a body, a header, an attachment, or any combination of them (or parts of them). Message integrity is provided by leveraging XML Signature in conjunction with security tokens to ensure that messages are transmitted without modifications. The integrity mechanisms are designed to support multiple signatures, potentially by multiple actors, and to be extensible to support additional signature formats. Message confidentiality leverages XML Encryption in conjunction with security tokens to keep portions of a SOAP message confidential. The encryption mechanisms are designed to support additional encryption processes and operations by multiple actors.

The Security header block provides a mechanism for attaching security-related information targeted at a specific receiver (SOAP actor). This MAY be either the ultimate receiver of the message or an intermediary. Consequently, this header block MAY be present multiple times in a SOAP message. An intermediary on the message path MAY add one or more new sub-elements to an existing Security header block if they are targeted for the same SOAP node or it MAY add one or more new headers for additional targets. As stated, a message MAY have multiple Security header blocks if they are targeted for separate receivers. However, only one Security header block can omit the S:actor attribute and no two Security header blocks can have the same value for S:actor. Message security information targeted for different

receivers MUST appear in different Security header blocks. The Security header block without a specified S:actor can be consumed by anyone, but MUST NOT be removed prior to the final destination as determined by WS-Routing. As elements are added to the Security header block, they should be prepended to the existing elements. As such, the Security header block represents the signing and encryption steps the message sender took to create the message. This prepending rule ensures that the receiving application MAY process sub-elements in the order they appear in the Security header block, because there will be no forward dependency among the sub-elements. Note that WS-Security specification does not impose any specific order of processing the sub-elements. The receiving application can use whatever policy is needed. When a sub-element refers to a key carried in another sub-element (for example, a signature sub-element that refers to a binary security token sub-element that contains the X.509 certificate used for the signature), the key-bearing security token SHOULD be prepended subsequent to the key-using sub-element being added, so that the key material appears before the key-using sub-element.
The following sample message illustrates the use of security tokens, signatures, and encryption.

```
(001) <?xml version="1.0" encoding="utf-8"?>
(002) <S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
           xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
           xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
           xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
(003)    <S:Header>
(004)       <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
(005)          <m:action>http://fabrikam123.com/getQuote</m:action>
(006)          <m:to>http://fabrikam123.com/stocks</m:to>
(007)          <m:from>mailto:johnsmith@fabrikam123.com</m:from>
(008)          <m:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</m:id>
(009)       </m:path>
(010)       <wsse:Security>
(011)          <wsse:BinarySecurityToken
                      ValueType="wsse:X509v3"
                      Id="X509Token"
                      EncodingType="wsse:Base64Binary">
(012)          MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
(013)          </wsse:BinarySecurityToken>
(014)          <xenc:EncryptedKey>
(015)              <xenc:EncryptionMethod Algorithm=
                        "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
(016)              <ds:KeyInfo>
(017)                <ds:KeyName>CN=Hiroshi Maruyama, C=JP</ds:KeyName>
(018)              </ds:KeyInfo>
(019)              <xenc:CipherData>
(020)                 <xenc:CipherValue>d2FpbmdvbGRfE01m4byV0...
(021)                 </xenc:CipherValue>
(022)              </xenc:CipherData>
(023)              <xenc:ReferenceList>
(024)                  <xenc:DataReference URI="#enc1"/>
(025)              </xenc:ReferenceList>
(026)          </xenc:EncryptedKey>
(027)          <ds:Signature>
(028)             <ds:SignedInfo>
(029)                <ds:CanonicalizationMethod
                    Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
(030)                <ds:SignatureMethod
                  Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
(031)                <ds:Reference>
(032)                   <ds:Transforms>
(033)                      <ds:Transform
                              Algorithm="http://...#RoutingTransform"/>
(034)                      <ds:Transform
                    Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
(035)                   </ds:Transforms>
(036)                   <ds:DigestMethod
                     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
(037)                   <ds:DigestValue>LyLsF094hPi4wPU...
(038)                    </ds:DigestValue>
(039)                </ds:Reference>
(040)             </ds:SignedInfo>
(041)             <ds:SignatureValue>
(042)                   Hp1ZkmFZ/2kQLXDJbchm5gK...
(043)             </ds:SignatureValue>
```



```
(044)            <ds:KeyInfo>
(045)                <wsse:SecurityTokenReference>
(046)                    <wsse:Reference URI="#X509Token"/>
(047)                </wsse:SecurityTokenReference>
(048)            </ds:KeyInfo>
(049)         </ds:Signature>
(050)      </wsse:Security>
(051)   </S:Header>
(052)   <S:Body>
(053)      <xenc:EncryptedData
                Type="http://www.w3.org/2001/04/xmlenc#Element"
                Id="enc1">
(054)         <xenc:EncryptionMethod
              Algorithm="http://www.w3.org/2001/04/xmlenc#3des-cbc"/>
(055)         <xenc:CipherData>
(056)            <xenc:CipherValue>d2FpbmdvbGRfE0lm4byV0...
(057)            </xenc:CipherValue>
(058)         </xenc:CipherData>
(059)      </xenc:EncryptedData>
(060)   </S:Body>
(061) </S:Envelope>
```

Lines (004)-(009) specify the message routing information (as define in WS-Routing). In this case we are sending the message to the http://fabrikam123.com/stocks service requesting the "getQuote" action.
Lines (011)-(013) specify a security token that is associated with the message. In this case, it specifies an X.509 certificate that is encoded as Base64. Line (012) specifies the actual Base64 encoding of the certificate.
Lines (014)-(026) specify the key that is used to encrypt the body of the message. Since this is a symmetric key, it is passed in an encrypted form. Line (015) defines the algorithm used to encrypt the key. Lines (016)-(018) specify the name of the key that was used to encrypt the symmetric key. Lines (019)-(022) specify the actual encrypted form of the symmetric key. Lines (023)-(025) identify the encryption block in the message that uses this symmetric key. In this case it is only used to encrypt the body (Id="enc1").
Lines (027)-(049) specify the digital signature. In this example, the signature is based on the X.509 certificate. Lines (028)-(040) indicate what is being signed. Specifically, Line (029) indicates the canonicalization algorithm (exclusive in this example). Line (030) indicate the signature algorithm (rsa over sha1 in this case).
Lines (031)-(039) identify the parts of the message that are being signed. Specifically, Line (033) identifies a "transform". This fictitious transforms selects the immutable portions of the routing header and the message body. Line (034) specifies the canonicalization algorithm to use on the selected message parts from line (033). Line (036) indicates the digest algorithm use on the canonicalized data. Line (037) specifies the digest value resulting from the specified algorithm on the canonicalized data.
Lines (044)-(048) indicate the key that was used for the signature. In this case, it is the X.509 certificate included in the message. Line (046) provides a URI link to the Lines (011)-(013).
Lines (053)-(059) represent the encrypted metadata and form of the body using XML Encryption. Line (053) indicates that the "element value" is being replaced and identifies this encryption. Line (054) specifies the encryption algorithm - Triple-DES in this case. Lines (055)-(058) contain the actual cipher text (i.e., the result of the encryption). Note that we don't include a reference to the key as the key references this encryption - Line (024).

J2EE Webservice Security Implementation

JAX-RPC implementation has to support HTTP Basic authentication. JAX-RPC specifciation does not require JAX-RPC implementation to support certificate based mutual authentication using HTTP/S (HTTP over SSL).

HTTP Basic Authentication

Add the appropriate security elements to the web.xml deployment descriptor:

```
<?xml version="1.0"?>

<web-app version="2.4" ...>
       <display-name>Basic Authentication Security Example</display-name>
```

```
        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>SecureHello</web-resource-name>
                        <url-pattern>/hello</url-pattern>
                        <http-method>GET</http-method>
                        <http-method>POST</http-method>
                </web-resource-collection>

                <auth-constraint>
                        <role-name>admin</role-name>
                </auth-constraint>

                <user-data-constraint>
                        <transport-guarantee>NONE</transport-guarantee>
                </user-data-constraint>
        </security-constraint>

        <login-config>
                <auth-method>BASIC</auth-method>
        </login-config>

        <security-role>
                <role-name>admin</role-name>
        </security-role>
</web-app>
```

Set security properties in client code:

```
try {
        Stub stub = createProxy();
        stub._setProperty(javax.xml.rpc.Stub.USERNAME_PROPERTY, username);
        stub._setProperty(javax.xml.rpc.Stub.PASSWORD_PROPERTY, password);
        stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, endpointAddress);

        HelloIF hello = (HelloIF)stub;
        System.out.println(hello.sayHello(" Duke (secure)" ));
} catch (Exception ex) {
        ex.printStackTrace();
}
```

Mutual Authentication

Configure SSL connector

Add the appropriate security elements to the web.xml deployment descriptor:

```
<?xml version="1.0"?>
<web-app version="2.4" ...>
        <display-name>Secure Mutual Authentication Example</display-name>

        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>SecureHello</web-resource-name>
                        <url-pattern>/hello</url-pattern>
                        <http-method>GET</http-method>
                        <http-method>POST</http-method>
                </web-resource-collection>

                <user-data-constraint>
                        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
                </user-data-constraint>
        </security-constraint>

        <login-config>
                <auth-method>CLIENT-CERT</auth-method>
        </login-config>
</web-app>
```

Set Security Properties in client code:

```
try {
        Stub stub = createProxy();
        System.setProperty("javax.net.ssl.keyStore", keyStore);
        System.setProperty("javax.net.ssl.keyStorePassword", keyStorePassword);
```

```
        System.setProperty("javax.net.ssl.trustStore", trustStore);
        System.setProperty("javax.net.ssl.trustStorePassword", trustStorePassword);
        stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,endpointAddress);

        HelloIF hello = (HelloIF)stub;
        System.out.println(hello.sayHello(" Duke! secure!"));
} catch (Exception ex) {
        ex.printStackTrace();
}
```

Client and Service Endpoint Design and Architecture

[Blueprints – 5]
10.1 Describe the characteristics of a service oriented architecture and how Web services fits to this model.
10.2Given a scenario, design a J2EE service using the business delegate, service locator, and/or proxy client-side design patterns and the adapter, command, Web service broker, and/or faade server-side patterns.
10.3 Describe alternatives for dealing with issues that impact the quality of service provided by a Web service and methods to improve the system reliability, maintainability, security, and performance of a service.
10.4 Describe how to handle the various types of return values, faults, errors, and exceptions that can occur during a Web service interaction.
10.5 Describe the role that Web services play when integrating data, application functions, or business processes in a J2EE application.
10.6 Describe how to design a stateless Web service that exposes the functionality of a stateful business process.
[Blueprints – 3, 8]
11.1 Given a scenario, design Web service applications using information models that are either procedure-style or document-style.
11.2 Describe the function of the service interaction and processing layers in a Web service.
11.3 Describe the tasks performed by each phase of an XML-based, document oriented, Web service application, including the consumption, business processing, and production phases.
11.4 Design a Web service for an asynchronous, document-style process and describe how to refactor a Web service from a synchronous to an asynchronous model.
11.5 Describe how the characteristics, such as resource utilization, conversational capabilities, and operational modes, of the various types of Web service clients impact the design of a Web service or determine the type of client that might interact with a particular service.
[Below is the documentation excerpted from Blueprints book – Designing Webservices using J2EE 1.4.]

What is SOA?

A Web service enables a ***service-oriented architecture***, which is an architectural style that promotes software reusability by creating reusable services. Traditional object-oriented architectures promote reusability by reusing classes or objects. However, objects are often too fine grained for effective reuse. Hence, component-oriented architectures emerged that use software components as reusable entities. These components consist of a set of related classes, their resources, and configuration information. Component-oriented architectures remain a powerful way to design software systems; however, they do not address the additional issues arising from current day enterprise environments. Today, enterprise environments are quite complex due to the use of a variety of software and hardware platforms, Internet-based distributed communication, enterprise application integration, and so on. The service-oriented architectures address these issues by using a service as a reusable entity. The services are typically coarser grained than components, and they focus on the functionality provided by their interfaces. These services communicate with each other and with end-user clients through well-defined and well-known interfaces. The communication can range from a simple passing of messages between the services to a complex scenario where a set of services together coordinate with each other to achieve a common goal. These architectures allow clients, over the network, to invoke a Web service's functionality through the service's exposed interfaces.
So the parts of SOA are:

1. A **service** that implements the business logic and exposes this business logic through well-defined interfaces.
2. A **registry** where the service publishes its interfaces to enable clients to discover the service.
3. **Clients** (including clients that may be services themselves) who discover the service using the registries and access the service directly through the exposed interfaces.

An important **advantage of a service-oriented architecture** is that it **enables development of loosely-coupled applications that can be distributed** and are accessible across a network.

Benefits of Webservices

- Interoperability in a hetrogeneous environment
- Business services through web – B2B : an enterprise might make its product catalog and inventory available to its vendors through a Web service to achieve better supply chain management.
- Integration with existing systems – A2A or EAI.

Challenges to Webservices

1. Enterprises often use Web services as a means to distribute data or information. In addition, many businesses use services to conduct business transactions. Such business transactions may require a service to access other services; in a sense, to perform a global transaction. Although in the process of being defined, currently there are no universally accepted standards for such global transactions.
2. Interoperability is a continuous challenge. WS-I is up against this.
3. Coordination of multiple services for processing business logic. Often, what appears to the end user as a single business process is really implemented as a series of stages in a workflow, and each stage of the workflow might be implemented as a separate service. In such cases, all the services must coordinate with each other during the various steps of the business logic processing to achieve the desired goal. Standards are necessary for coordination among services. Such standards are in the process of being defined, but none has yet been universally accepted.
4. WS Security.

Typical Webservice scenarios

1. interactions between business partners, supply chain management, inventory management, and even simple services (specialized converters, calculators, and so forth).
2. Interoperability is required, different EISs need to be integrated, different client types need to be supported.

In the case of Adventure Builder application, which sells adventure packages to vacationers. It provides:

1. **Catalog** of adventure packages.
2. Allows for customers to plan the their trip (choosing accommodations, transportation, and scheduling various adventure activities, and finally booking the trip) – **Order workflow**.
3. **Order status tracking** by customers for their orders.

Other non-web-based activities are:

4. Keep track of client preferences and update the clients regarding the status of their orders. – CRM
5. Verify and obtain approval of client payments. – interact with financial institutions.
6. Interact with business partners—airlines, hotels, adventure or activity providers—so that the enterprise can provide a complete adventure package to the client.

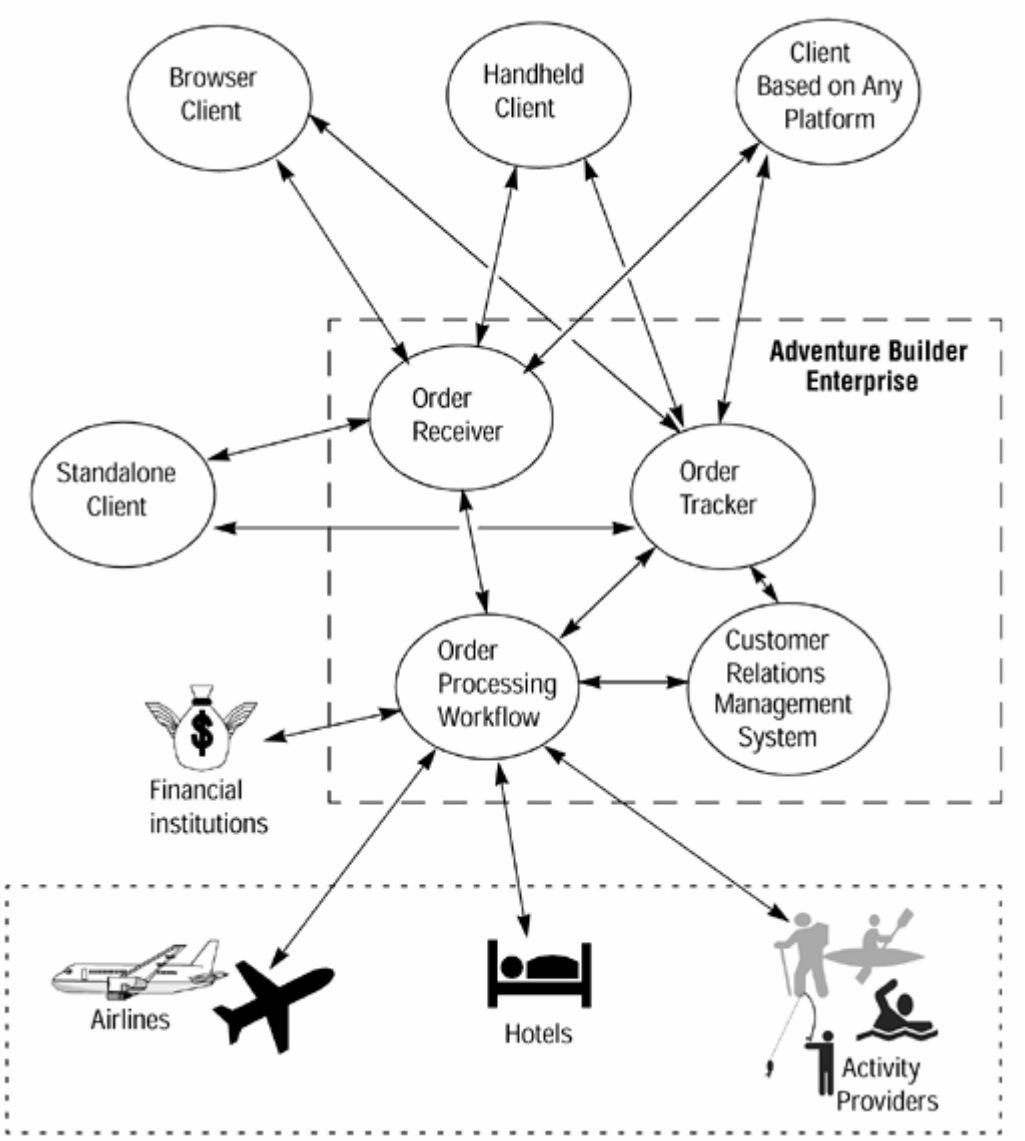


Figure 2 - Adventure Builder Enterprise Modules

Following are the scenarios in the adventure builder implementation where webservices will add value:

- Web services can be an ideal way to integrate the enterprise with multiple partners for several reasons. Web services are more cost effective than electronic data interchange (EDI), currently the most common solution available for business partner interaction. EDI requires a significant up-front investment by all parties. While its large business partners may have an extensive EDI infrastructure in place, the adventure builder enterprise cannot expect its many small business partners to expend the resources to implement such an EDI infrastructure. Web services-based interaction is a good solution for small businesses that have no investment in an EDI infrastructure and more cost effective for larger enterprises with existing EDI infrastructures.
- Most businesses consider it more cost effective to evolve and enhance their existing information systems. As a result, enterprises are looking for solutions that let them not only stay current with the times but also evolve their infrastructure and integrate their existing systems with new systems. Webservices are a natural integration layer. It is now possible to provide software to expose an existing EIS as a Web service. This makes it feasible for users who require access to that EIS to do so through its Web service.
- Web services make it possible for an enterprise to make its functionality available to various types of clients.

- To provide a complete adventure package to its customers, adventure builder must obtain from its partners such information as the details of modes of travel, accommodations, and activities or adventures. This information is dynamic in nature, and the adventure builder enterprise needs these updates. For example, adventure builder needs to know which seats for a particular airline are available or which hotels have rooms available, and at what price. This information changes frequently, and it is important for the adventure builder enterprise to keep its data current. So, adventure builder not only builds its catalog of adventure packages based on the services available from its various partners, but it may also periodically update its catalog of adventures, particularly when changes occur with its partners. W**eb services enable a truly dynamic way to build and maintain this catalog of information**. Furthermore, by incorporating its Web services with **registry services**, adventure builder can expand its network of suppliers and allow any number of suppliers or partners to participate at their own choosing. **Once suppliers signal their participation through the registry, adventure builder can dynamically build its own catalog from the suppliers' offerings**. Keep in mind, however, that a truly dynamic arrangement requires a set of legal and financial agreements to be in place beforehand.

Some notes on Webservices standards : (excerpted from chapter 2 of Blueprints book).

1. QName = Local Name + URI
2. Note that UDDI is a specification for a registry, not a repository. As a registry it functions like a catalog, allowing requestors to find available services. **A registry is not a repository** because it does not contain the services itself.
3. WSDL specifies a grammar that describes Web services as a collection of communication endpoints, called **ports**. The data being exchanged are specified as **part** of **messages**. Every type of action allowed at an endpoint is considered an **operation**. Collections of operations possible on an endpoint are grouped together into **port types**. The messages, operations, and port types are all **abstract definitions**, which means the definitions do not carry deployment-specific details to enable their reuse.
4. The protocol (rpc/document) and data format (literal/encoded) specifications for a particular port type are specified as a **binding**. A **port** is defined by associating a network address with a **reusable binding**, and a collection of ports define a **service**.
5. JAXP processes XML documents using the SAX or DOM models, and it permits use of XSLT engines during the document processing.
6. JAX-RPC supports 3 modes of operation:
    - Synchronous Request/Response
    - One-way RPC
    - Non-blocking RPC - A client invokes a remote procedure and continues in its thread without blocking. Later, the client processes the remote method return by performing a blocked receive call or by polling for the return value
7. A JAXR provider consists of two parts: a **registry-specific JAXR provider**, which provides a registry-specific implementation of the API, and a **JAXR pluggable provider**, which implements those features of the API that are independent of the type of registry. The pluggable provider hides the details of registry-specific providers from clients.

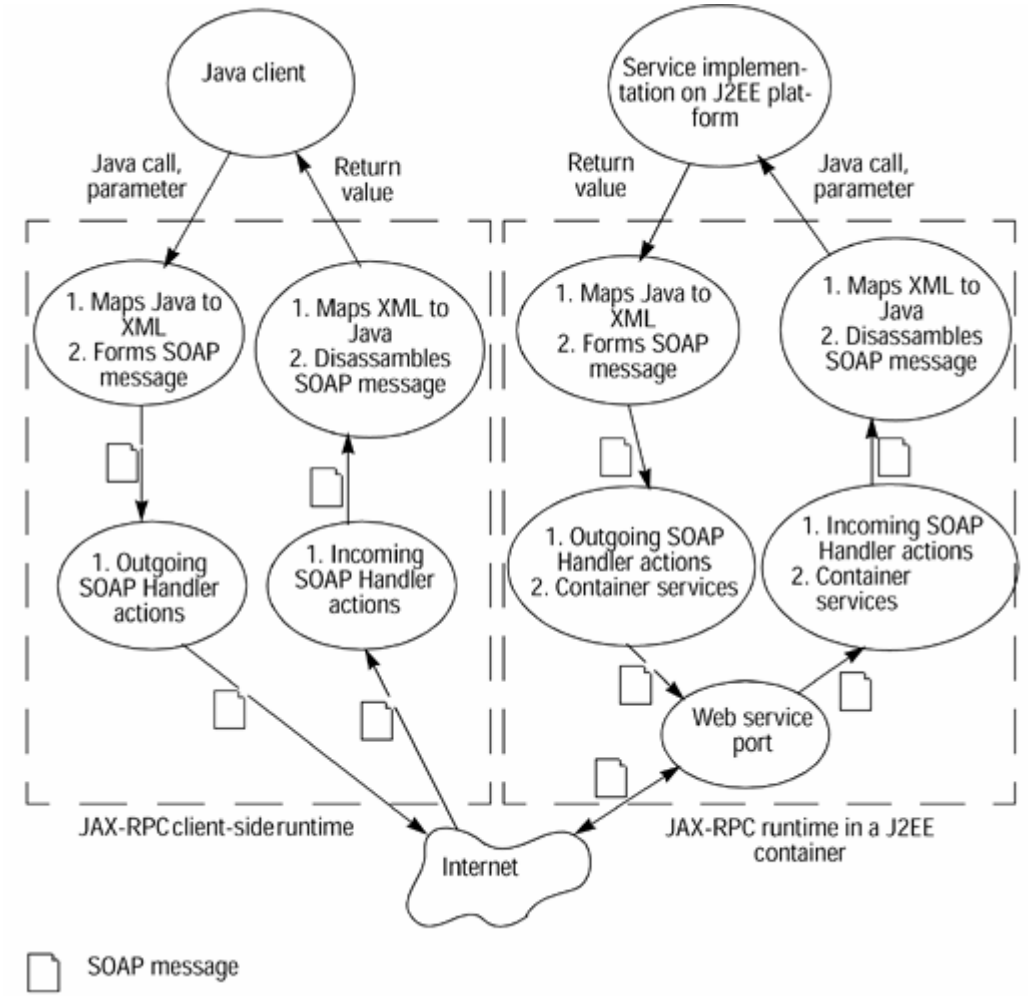
Service Endpoint Design

Following are some of the types of webservices we will be discussing in detail:

1. An **informational Web service** serving data that is more often read than updated—clients read the information much more than they might update it. In our adventure builder example, a good scenario is a Web service that provides interested clients with travel-related information, such as **weather forecasts**, for a given city.
2. A Web service that **concurrently completes client requests** while dealing with a high proportion of shared data that is updated frequently and hence requires heavy use of EIS or database transactions. The **airline reservation system** partner to adventure builder is a good example of this type of Web service. Many clients can simultaneously send details of desired airline reservations, and the Web service concurrently handles and conducts these reservations.
3. A business process Web service whose processing of a client request includes starting a series of long-running business and workflow processes. Adventure builder enterprise's decision to build a

service interface to partner travel agencies is a good example of this type of Web service. Through this service interface, partner agencies can offer their customers the same services offered in adventure builder's Web site. The partner agencies use adventure builder's business logic to fulfill their customer orders. A service such as this receives the details of a travel plan request from a partner agency, and then the service initiates a series of processes to reserve airlines, hotels, rental cars, and so forth for the specified dates.

Flow of a Webservice Call



The client's request reaches the service through a **port**, since a port provides access over a specific protocol and data format at a network endpoint consisting of a host name and port number. Before the port passes the request to the endpoint, it ensures that the J2EE container applies its declarative services (such as security checks) to the SOAP request. After that, any developer-written SOAP **handlers** in place are applied to the request.

Webservices Design Decisions

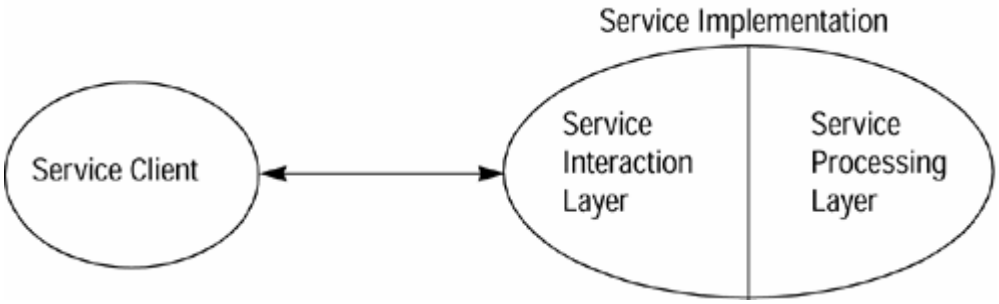
When you design a Web service interface for an application, you must consider those issues that pertain specifically to interoperability and Web services—and not to the business logic—and you make your design decisions based on these issues.

Typical steps for designing a Webservice:

1. Decide on the interface to be exposed to the client.

You should consider the type of endpoints you want to use—EJB service endpoints or JAX-RPC service endpoints—and when to use them. You must also decide whether you are going to use SOAP handlers. Next, you decide whether you want to publish the service interface, and, if so, how to publish it. Publishing a service makes it available to clients. You can restrict the service's availability to clients you have personally notified about the service, or you can make your service completely public and register it with a public registry.

2. Determine how to receive and preprocess requests.
   You should consider whether you need to introduce any message handlers for pre- or post-processing of SOAP messages.
3. Determine how to delegate the request to business logic.
   Once a request has been received and preprocessed, then you are ready to delegate it to the service's business logic.
4. Decide how to process a request.
   If the webservice provides an interface to an existing business logic then we may only need to determine how to use the business logic interfaces.
5. Determine how to formulate and send a response.
6. Determine how to report problems.



A partner travel agency uses adventure builder enterprise's Web service to build a travel itinerary for its clients. Through the service interface it exposes to these travel agencies, adventure builder enterprise receives business documents (in XML format) containing all required details for travel itinerary requests. Adventure builder uses its existing workflow systems to process and satisfy these partner requests. The **interaction layer** of adventure builder's exposed Web service interface **validates these incoming business documents, then converts the incoming XML documents to its internal format or maps document content to Java objects.** Once the conversion is finished, control passes to the workflow mechanisms in the processing layer where travel requests are completed. The interaction layer generates responses for completed travel requests, converts responses to XML documents or other appropriate formats, and ensures that responses are relayed to the partner agencies.
The weather service scenario is one such service that might benefit from merging the interaction and processing layers into a single layer. Since incoming requests require no preprocessing, a layered view of the weather service only complicates what otherwise should be a simple service.

WSDL2Java or Java2WSDL?

With the Java-to-WSDL approach, it may be hard to evolve the service interface without forcing a change in the corresponding WSDL document, and changing the WSDL might require rewriting the service's clients. These changes, and the accompanying instability, can affect the interoperability of the service itself. Since achieving interoperability is a prime reason to use Web services, the instability of the Java-to-WSDL approach is a major drawback. Also, keep in mind that different tools may use different interpretations for certain Java types (for example, `java.util.Date` might be interpreted as `java.util.Calendar`), resulting in different representations in the WSDL file. While not common, these representation variations may result in some semantic surprises.
On the other hand, the WSDL-to-Java approach gives you a powerful way to expose a stable service interface that you can evolve with relative ease. Not only does it give you greater design flexibility, the WSDL-to-Java approach also provides an ideal way for you to finalize all service details—from method call types and fault types to the schemas representing exchanged business documents—before you even

start a service or client implementation. Although a good knowledge of WSDL and the WS-I Basic Profile is required to properly describe these Web services details, using available tools helps address these issues.

JSE or EJB Endpoint?

1. When you develop a new Web service that does not use existing business logic, choosing the endpoint type to use for the Web service interface is straightforward. The endpoint type choice depends on the nature of your business logic—whether the business logic of the service is completely contained within either the Web tier or the EJB tier.
2. When you add a Web service interface for an existing application, choose an endpoint type suited for the tier on which the preprocessing logic occurs in the existing application. Use a JAX-RPC service endpoint when the preprocessing occurs on the Web tier of the existing application and an EJB service endpoint when preprocessing occurs on the EJB tier. If the existing application or service does not require preprocessing of the incoming request, choose the appropriate endpoint that is present in the same tier as the existing business logic.
3. **Multithreading consideration -** A JAX-RPC service endpoint has to handle concurrent client access on its own, whereas the EJB container takes care of concurrent client access for an **EJB service endpoint.**
4. **Transaction consideration -** If the Web service's business logic requires using transactions (and the service has a JAX-RPC service endpoint), you must implement the transaction-handling logic using JTA or some other similar facility. If your service uses an **EJB service endpoint**, you can use the container's declarative transaction services. By doing so, the container is responsible for handling transactions according to the setting of the deployment descriptor element `container-transaction`.
5. **Method-level access permission consideration -** When you want to control service access at the individual method level, consider using an **EJB service endpoint** rather than a JAX-RPC service endpoint. A JAX-RPC service endpoint, on the other hand, does not have a facility for declaring method-level access constraints, requiring you to do this programmatically.
6. **HTTP Session access consideration -** A **JAX-RPC service endpoint,** because it runs in the Web container, has complete access to an `HttpSession` object. Access to an `HttpSession` object, which can be used to embed cookies and store client state, may help you build session-aware clients. An EJB service endpoint, which runs in the EJB container, has no such access to Web container state. However, generally HTTP session support is appropriate for short duration conversational interactions, whereas Web services often represent business processes with longer durations and hence need additional mechanisms.

Service Operations Granularity

If you are planning to expose existing stateless session beans as Web service endpoints, remember that such beans may not have been designed with Web services in mind. Hence, they may be too fine grained to be good Web service endpoints. You should consider consolidating related operations into a single Web service operation. Good design for our airline reservation Web service, for example, is to expect the service's clients to send all information required for a reservation—destination, preferred departure and arrival times, preferred airline, and so forth—in one invocation to the service, that is, as one large message. This is far more preferable than to have a client invoke a separate method for each piece of information comprising the reservation.

```
public interface AirlineTicketsIntf extends Remote {

   public String submitReservationRequest(

          AirReservationDetails details) throws RemoteException;

}
```

combines logically-related data into one large message for a more efficient client interaction with the service. This is preferable to receiving the data with individual method calls like submitFlightInfo and submitPreferredDates etc.

Coarse-grained services involve transferring large amounts of data. If you opt for more coarse-grained service operations, **it is more efficient to cache data on the client side to reduce the number of round trips** between the client and the server.

Parameter types for Webservice Operations

Points to consider when you use Java objects with standard type mappings as parameters:

1. Utilities for handling lists, such as `ArrayList` and `Collection`, to name a few, are not supported standard types. Instead, Java arrays provide equivalent functionality, and have a standard mapping provided by the platform.
2. JAX-RPC value types are user-defined Java classes with some restrictions. They have constructors and may have fields that are public, private, protected, static, or transient. JAX-RPC value types may also have methods, including set and get methods for setting and getting Java class fields. However, when mapping JAX-RPC value types to and from XML, there is no standard way to retain the order of the parameters to the constructors and other methods. Hence, avoid setting the JAX-RPC value type fields through the constructor. Using the get and set methods to retrieve or set value type fields avoids this mapping problem and ensures portability and interoperability.
3. The J2EE platform supports nested JAX-RPC value types; that is, JAX-RPC value types that reference other JAX-RPC value types within themselves. For clarity, it is preferable to use this feature and embed value type references rather than to use a single flat, large JAX-RPC value type class.
4. Since the J2EE container automatically handles mappings based on the Java types, using these Java-MIME mappings frees you from the intricacies of sending and retrieving documents and images as part of a service's request and response handling.

MIME Type	Java Type
`image/gif`	`java.awt.Image`
`image/jpeg`	`java.awt.Image`
`text/plain`	`java.lang.String`
`multipart/*`	`javax.mail.internet.MimeMultipart`
`text/xml` or `application/xml`	`javax.xml.transform.Source`

A better way is to pass the JAX-RPC non-standard type parameters as SOAP document fragments represented as a DOM subtree in the service endpoint interface. If so, you should consider binding (either manually or using JAXB) the SOAP fragments to Java objects before passing them to the processing layer to avoid tightly coupling the business logic with the document fragment.

You may want to avoid using overloaded methods in your Java interface altogether if you have only intuitive method names in the WSDL (like getWeatherByCity and getWeatherByZip instead of using two overloaded getWeather methods which take city code or zip as param).

Service-specific exceptions like `CityNotFoundException`, which are thrown by the Web service to indicate application-specific error conditions, must be checked exceptions that directly or indirectly extend `java.lang.Exception`. They cannot be unchecked exceptions. On the service side, keep in mind how to include exceptions in the service interface and how to throw them. Generally, you want to do the following:

1. Convert application-specific errors and other Java exceptions into meaningful service-specific exceptions and throw these service-specific exceptions to the clients. Although they promote interoperability among heterogeneous platforms, Web service standards cannot address every type of exception thrown by different platforms. For example, the standards do not specify how Java exceptions such as `java.io.IOException` and `javax.ejb.EJBException` should be returned to the client. As a consequence, it is important for a Web service—from the service's interoperability point of view—to not expose Java-specific exceptions (such as those just mentioned) in the Web service interface. Instead, throw a service-specific exception.
2. You cannot throw nonserializable exceptions to a client through the Web service endpoint.

3. For example, suppose your service encounters a `javax.ejb.FinderException` exception while processing a client request. The service should catch the `FinderException` exception, and then, rather than throwing this exception as is back to the client, the service should instead throw a service-specific exception that has more meaning for the client (for example, InvalidKeyException).
4. Exception inheritances are lost when you throw a service-specific exception. The exception stack trace is not passed to the client. You should avoid defining service-specific exceptions that inherit or extend other exceptions. For example, if `CityNotFoundException` extends another exception, such as `RootException`, then when the service throws `CityNotFoundException`, methods and properties inherited from `RootException` are not passed to the client.
5. A SOAP `fault` defines system-level exceptions, such as `RemoteException`, which are irrecoverable errors. The WSDL `fault` denotes service-specific exceptions, such as `CityNotFoundException`, and these are recoverable application error conditions. Since the WSDL `fault` denotes a recoverable error condition, the platform can pass it as part of the SOAP response message. Thus, the standards provide a way to exchange fault messages and map these messages to operations on the endpoint.

Use of Handlers

Handlers are particular to a Web service and are associated with the specific port of the service. As a result of this association, the handler's logic applies to all SOAP requests and responses that pass through a service's port. Following are some guidelines on their use:

1. It is not advisable to put in a handler business logic or processing particular to specific requests and responses. You cannot store client-specific state in a handler: A handler's logic acts on all requests and responses that pass through an endpoint. However, you may use the handler to store port-specific state, which is state common to all method calls on that service interface. Note also that handlers execute in the context of the component in which they are present.
2. Do not store client-specific state in a handler.
3. Use of handlers can result in a significant performance impact for the service as a whole. Using handlers makes sense primarily for writing system services such as auditing, logging, and so forth.

Interoperability

For maximum interoperability, you should keep these three points in mind:

1. Because the WS-I Basic Profile 1.0, to which J2EE1.4 platform conforms, supports only literal bindings, you should avoid encoded bindings.
2. Literal bindings cannot represent complex types, such as objects with circular references, in a standard way.
3. Since the WS-I Basic Profile 1.0 specification does not address attachments, a Web service using the Java-MIME mappings provided by the J2EE platform is not guaranteed to be interoperable.
4. When using handlers, you must be careful not to change a SOAP message to the degree that the message no longer complies with WS-I specifications, thereby endangering the interoperability of your service.

Receiving Request

1. Before delegating these incoming client requests to the Web service business logic, you should perform any required security validation, transformation of parameters, and other required preprocessing of parameters.
2. Web service calls are basically method calls whose parameters are passed as either Java objects, XML documents (`javax.xml.transform.Source` objects), or even SOAP document fragments (`javax.xml.soap.SOAPElement` objects).
3. For parameters that are passed as Java objects (such as `String, int`, JAX-RPC value types, and so forth), do the application-specific parameter validation and map the incoming objects to

domain-specific objects in the interaction layer before delegating the request to the processing layer.

You may have to undertake additional steps to handle XML documents that are passed as parameters (in case of document-literal messaging style):

4. The service endpoint should validate the incoming XML document against its schema.
5. When the processing layer deals with objects but the service interface receives XML documents, then, as part of the interaction layer, map the incoming XML documents to domain objects before delegating the request to the processing layer.
6. When the service's processing layer and business logic are designed to deal with XML documents, you should transform the XML document to an internally supported schema, if the schema for the XML document differs from the internal schema, before passing the document to the processing layer.
7. It is important that these three steps—validation of incoming parameters or XML documents, translation of XML documents to internal supported schemas, and mapping documents to domain objects—be performed as close to the service endpoint as possible, and certainly in the service interaction layer.
8. It is generally advisable to do all common processing—such as security checks, logging, auditing, input validation, and so forth—for requests at the interaction layer as soon as a request is received and before passing it to the processing layer.

Delegating requests to processing layer

Requests from client can be synchronous and asynchronous type. The travel agency service is a good example of an asynchronous interaction between a client and a service. A client requests arrangements for a particular trip by sending the travel service all pertinent information (most likely in an XML document). Based on the document's content, the service performs such steps as verifying the user's account, checking and getting authorization for the credit card, checking accommodations and transportation availability, building an itinerary, purchasing tickets, and so forth. Since the travel service must perform a series of often time-consuming steps in its normal workflow, the client cannot afford to pause and wait for these steps to complete.

The **recommended approach to handle an asynchronous request** is: the client sends a request to the service endpoint. The service endpoint **validates** the incoming request in the interaction layer and then delegates the client's request to the appropriate processing layer of the service. It does so by sending the request as a **JMS message** to a JMS queue or topic specifically designated for this type of request. After the request is successfully delegated to the processing layer, the service endpoint may return a **correlation identifier** to the client. This correlation identifier is for the client's future reference and may help the client associate a response that corresponds to its previous request. If the business logic is implemented using enterprise beans, message-driven beans in the EJB tier read the request and initiate processing so that a response can ultimately be formulated.

The service may make the result of the client's request available in one of two ways:

1. The client that invoked the service periodically checks the status of the request using the correlation identifier that was provided at the time the request was submitted. This is also known as **polling**.
2. Or, if the client itself is a Web service peer, the service calls back the client's service with the result. The client may use the correlation identifier to relate the response with the original request.

Response generation

1. You should perform response generation, which is simply constructing the method call return values and output parameters, on the interaction layer, as close as possible to the service endpoint.
2. This permits having a common location for response assembly and XML document transformations, particularly if the document you return to the caller must conform to a different schema from the internal schema. Keeping this functionality near the endpoint lets you implement data caching and avoid extra trips to the processing layer.
3. Consider response generation from the weather information service's point-of-view. The weather information service may be used by a variety of client types, from browsers to rich clients to

handheld devices. A well-designed weather information service would render its responses in formats suitable for these different client types. it is better to design a common business logic for all client types. Then, in the interaction layer, transform the results per client type for rendering.

Publishing a Web Service

1. You may want to register Web services for general public consumption on a well-known public registry. Keep in mind that the public registry holds the Web service description, which consists not only of the service's WSDL description but also any XML schemas referenced by the service description. In short, your Web service must publish its public XML schemas and any additional schemas defined in the context of the service. You also must publish on the same public registry XML schemas referred to by the Web service description.
2. When a Web service is strictly for intra-enterprise use, you may publish a Web service description on a corporate registry within the enterprise.
3. You do not need to use a registry if all the customers of your Web services are dedicated partners and there is an agreement among the partners on the use of the services. When this is the case, you can publish your Web service description—the WSDL and referenced XML schemas—at a well-known location with the proper access protections.

Registry concepts

1. **Public registries are not repositories**. Rather than containing complete details on services, public registries contain only details about what services are available and how to access these services. For example, a service selling adventure packages cannot register its complete catalog of products. A registry can only store the type of service, its location, and information required to access the service.
2. **Register a service under the proper taxonomy**. When you want to publish your service on a registry, either a public or corporate registry, you must do so against a taxonomy that correctly classifies or categorizes your Web service. Using existing, well-known taxonomies gives clients of your Web service a standard base from which to search for your service, making it easy for clients to find your service. For example, suppose your travel business provides South Sea island-related adventure packages as well as alpine or mountaineering adventures. Rather than create your own taxonomy to categorize your service, clients can more easily find your service if you publish your service description using two different standard taxonomies: one taxonomy for island adventures and another for alpine and mountaineering adventures.

Handling XML documents in a Web service

There are additional considerations when a Web service implementation expects to receive an XML document containing all the information from a client, and which the service uses to start a business process to handle the request. There are several reasons why it is appropriate to exchange documents:

1. Documents, especially business documents, may be very large, and as such, they are often sent as a batch of related information. They may be compressed independently from the SOAP message.
2. Documents may be legally binding business documents. At a minimum, their original form needs to be conserved through the exchange and, more than likely, they may need to be archived and kept as evidence in case of disagreement. For these documents, the complete infoset of the original document should be preserved.

In essence, the service, which receives the request with the XML document, starts a business process to perform a series of steps to complete the request. The contents of the XML document are used throughout the business process. Handling this type of scenario effectively requires some considerations in addition to the general ones for all Web services.

1. Good design expects XML documents to be received as `javax.xml.transform.Source` objects.
2. Validation and transformation should be done before applying any processing logic to the document content.
3. When a service expects an XML document as input and starts a lengthy business process based on the document contents, then clients typically do not want to wait for the response. Good

design when processing time may be extensive is to delegate a request to a JMS queue or topic and return a correlation identifier for the client's future reference.

The J2EE platform provides **three ways to exchange XML documents**:

1. The first option is to use the Java-MIME mappings provided by the J2EE platform. With this option, the Web service endpoint receives documents as `javax.xml.transform.Source` objects. Along with the document, the service endpoint can also expect to receive other JAX-RPC arguments containing metadata, processing requirements, security information, and so forth. When an XML document is passed as a `Source` object, the container automatically handles the document as an attachment—effectively, the container implementation handles the document-passing details for you. This frees you from the intricacies of sending and retrieving documents as part of the endpoint's request/response handling. However, **sending documents as `Source` objects may not be interoperable with non-Java clients**.
2. The second option is to design your service endpoint such that it receives documents as `String` types. **If you are developing your service using the Java-to-WSDL approach, and the service must exchange XML documents and be interoperable with clients on any platform, then passing documents as `String` objects may be your only option.** The XML document may be large and the SOAP message payload size bloats resulting in performance overhead. Also the original format of the XML document is lost as it is sent as a String object (in canonical format).
3. The third option is to **exchange the XML document as a SOAP document fragment**. With this option, you map the XML document to `xsd:anyType` in the service's WSDL file. It is recommended that Web services exchange XML documents as SOAP document fragments because passing XML documents in this manner is both portable across J2EE implementations and interoperable with all platforms. **To pass SOAP document fragments, you must implement your service using the WSDL-to-Java approach**. A WSDL mapping of the XML document type to `xsd:anyType` requires the platform to map the document parameter as a `javax.xml.soap.SOAPElement` object. The service can parse the received SOAP document fragment using the `javax.xml.soap.SOAPElement` API. Or, the service can use JAXB to map the document fragment to a Java `Object` or transform it to another schema. A client of this Web service builds the purchase order document using the client platform-specific API for building SOAP document fragments—on the Java platform, this is the `javax.xml.soap.SOAPElement` API—and sends the document as one of the Web service's call parameters.
   When using the WSDL-to-Java approach, you can directly map the document to be exchanged to its appropriate schema in the WSDL. The corresponding generated Java interface represents the document as its equivalent Java `Object`. As a result, the service endpoint never sees the document that is exchanged in its original document form. It also means that the endpoint is tightly coupled to the document's schema: Any change in the document's schema requires a corresponding change to the endpoint. If you do not want such tight coupling, consider using `xsd:anyType` to map the document.
4. Rather than pass the entire document to different components handling various stages of the business process, it's best if the processing logic breaks the document into fragments and passes only the required fragments to other components or services that implement portions of the business process logic.

The pros and cons of passing XML documents through business processing stages take on greater significance when the business logic implementation spans multiple containers.

SAAJ is better for developers who want more control over the SOAP messages being exchanged and for developers using handlers.

Deploying and Packing a Service Endpoint

1. To begin, the service implementation declares its deployment details in the appropriate module-specific deployment descriptors (WEB-INF/web.xml for JSEs and META-INF/ejb-jar.xml for EJB Endpoints).

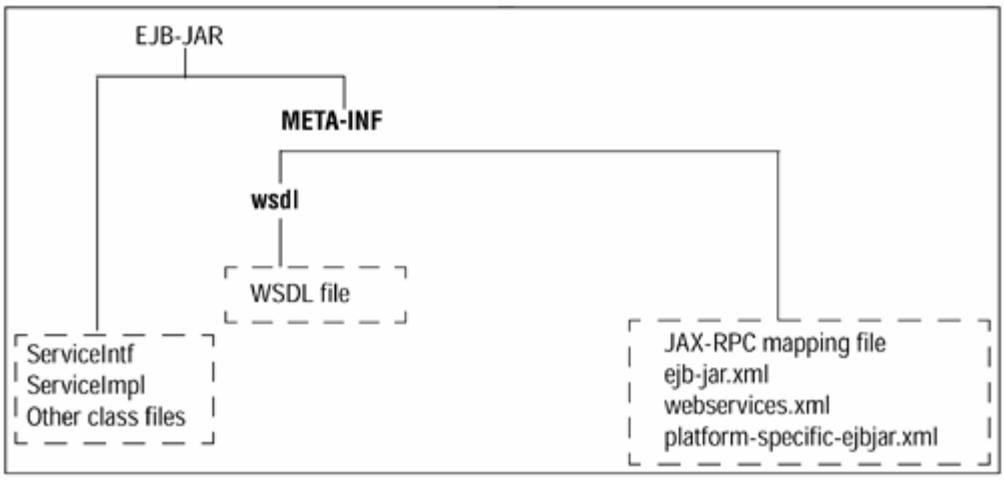
2. Next, the details of the port are specified. The Web service deployment descriptor, called `webservices.xml`, defines and declares the structural details for the port of a Web service. This file contains the following information:
    a. A logical name for the port that is also unique among all port components (`port-component-name` element)
    b. The service endpoint interface for the port (`service-endpoint-interface` element)
    c. The name of the class that implements the service interface (`service-impl-bean` element)
    d. The WSDL file for the service (`wsdl-file` element)
    e. A `QName` for the port (`wsdl-port` element)
    f. A correlation between WSDL definitions and actual Java interfaces and definitions using the mapping file (`jaxrpc-mapping-file` element)
    g. Optional details on any handlers

    The reference to the service implementation bean, specified using the `service-impl-bean` element in `webservices.xml`, is either a `servlet-link` or an `ejb-link` depending on whether the endpoint is a JAX-RPC or EJB service endpoint.

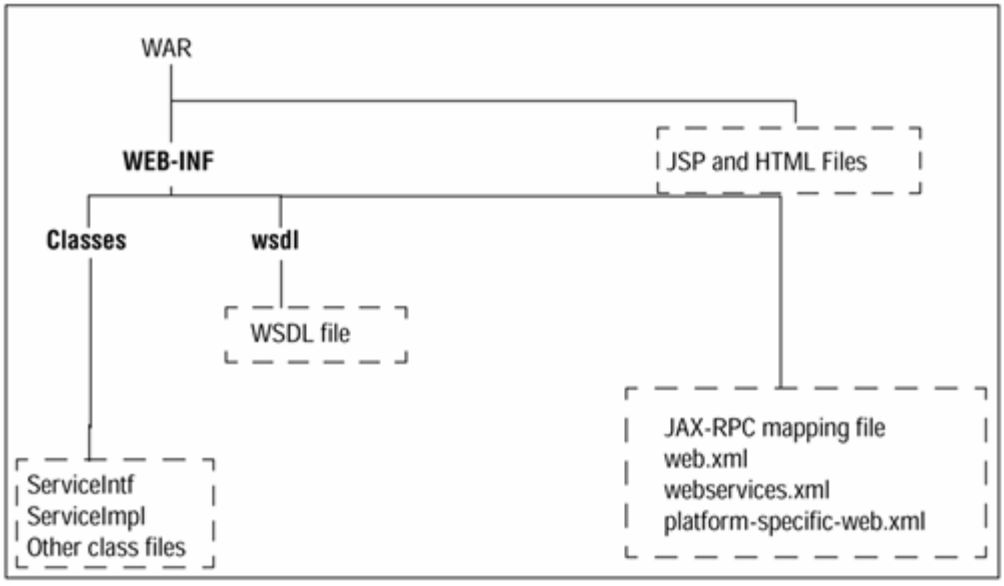
    The JAX-RPC mapping file, which is specified using the `jaxrpc-mapping-file` element in `webservices.xml`, keeps details on the relationships and mappings between WSDL definitions and corresponding Java interfaces and definitions. The information contained in this file, along with information in the WSDL, is used to create stubs and ties for deployed services.

    Thus, the Web services deployment descriptor, `webservices.xml`, links the WSDL port information to a unique port component and from there to the actual implementation classes and Java-to-WSDL mappings.

Package Structure for EJB Endpoint



Package Structure for JAX-RPC Service Endpoint



XML Processing

While XSD plays a major role in Web services, Web services may still have to deal with DTD-based schemas because of legacy reasons.

XML Information Set— This specification, often referred to as Infoset, provides the definitions for information in XML documents that are considered well formed according to the Namespaces criteria.

Canonical XML— This specification addresses how to resolve syntactic variations between the XML 1.0 and the Namespaces specifications to create the physical, canonical representation of an XML document.

However, because of the limited capabilities of some schema languages, a valid XML document may still be invalid in the application's domain. This might happen, for example, when a document is validated using

DTD, because this schema language lacks capabilities to express strongly-typed data, complex unicity, and cross-reference constraints. Other modern schema languages, such as XSD, more rigorously—while still lacking some business constraint expressiveness—narrow the set of valid document instances to those that the business logic can effectively process. Regardless of the schema language, even when performing XML validation, the application is responsible for enforcing any uncovered domain-specific constraints that the document may nevertheless violate. That is, the application may have to perform its own business logic-specific validation in addition to the XML validation. Only documents originating from untrusted sources/clients need to be validated. Components internal to an application can exchange the documents without any further validations.

Some applications may have to receive documents that conform to different schemas or different versions of a schema. In these cases, the application cannot do the validation up front against a specific schema unless the application is given a directive within the request itself about which schema to use. If no directive is included in the request, then the application has to rely on a hint provided by the document itself. Note that to deal with successive versioning of the same schema—where the versions actually modify the overall application's interface—it sometimes may be more convenient for an application to expose a separate endpoint for each version of the schema.

To illustrate, an application must check that the document is validated against the expected schema, which is not necessarily the one to which the document declares it conforms. With DTD schemas, this checking can be done only after validation. When using DOM, the application may retrieve the system or public identifier (`SystemID` or `PublicID`) of the DTD to ensure it is the identifier of the schema expected, while when using SAX, it can be done on the fly by handling the proper event.

```
public static boolean checkDocumentType(Document document,  String dtdPublicId) {

   DocumentType documentType = document.getDoctype();

   if (documentType != null) {

       String publicId = documentType.getPublicId();

       return publicId != null && publicId.equals(dtdPublicId);
   }
   return false;
}
```

With JAXP 1.2 and XSD (or other non-DTD schema languages), the application can specify up-front the schema to validate against; the application can even ignore the schema referred to by the document itself.

```
public static final String W3C_XML_SCHEMA  = "http://www.w3.org/2001/XMLSchema";

public static final String JAXP_SCHEMA_LANGUAGE
   = "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

public static final String JAXP_SCHEMA_SOURCE
   = "http://java.sun.com/xml/jaxp/properties/schemaSource";

public static SAXParser createParser(boolean validating,

       boolean xsdSupport, CustomEntityResolver entityResolver,

       String schemaURI) throws ... {

   // Obtain a SAX parser from a SAX parser factory

   SAXParserFactory parserFactory = SAXParserFactory.newInstance();

   // Enable validation

   parserFactory.setValidating(validating);

   parserFactory.setNamespaceAware(true);

   SAXParser parser = parserFactory.newSAXParser();

   if (xsdSupport) { // XML Schema Support
```



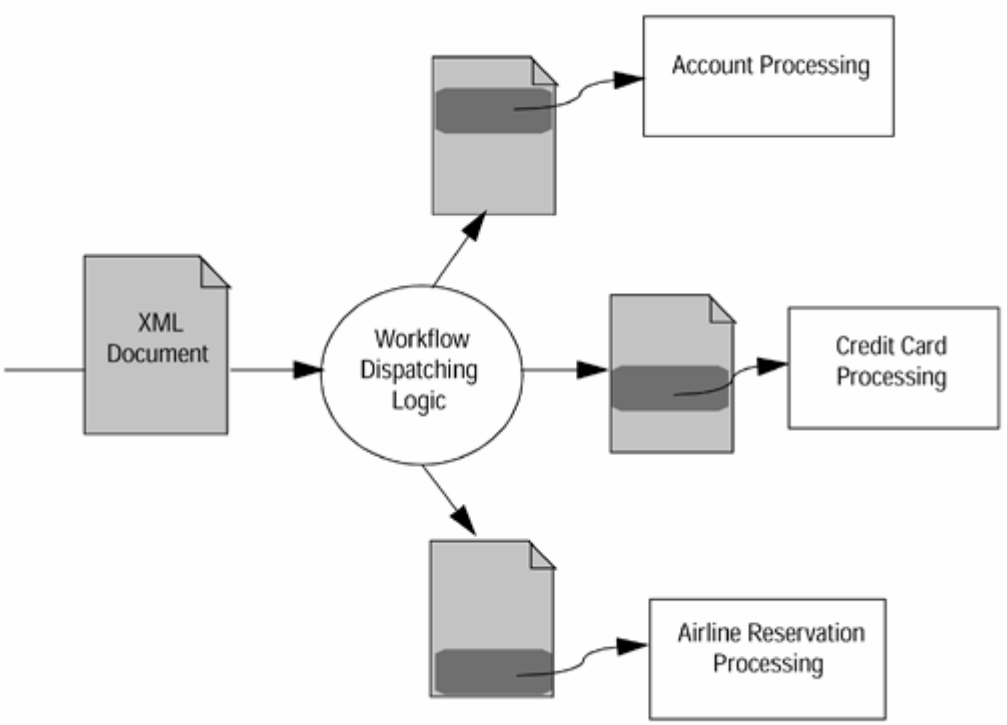
```
      try {
         // Enable XML Schema validation
         parser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
         // Set the validating schema to the resolved schema URI
         parser.setProperty(JAXP_SCHEMA_SOURCE, entityResolver.mapEntityURI(schemaURI));
      } catch(SAXNotRecognizedException exception) { ... }
   }
   return parser;
}
```

The above program sets the SAXParser for validation.

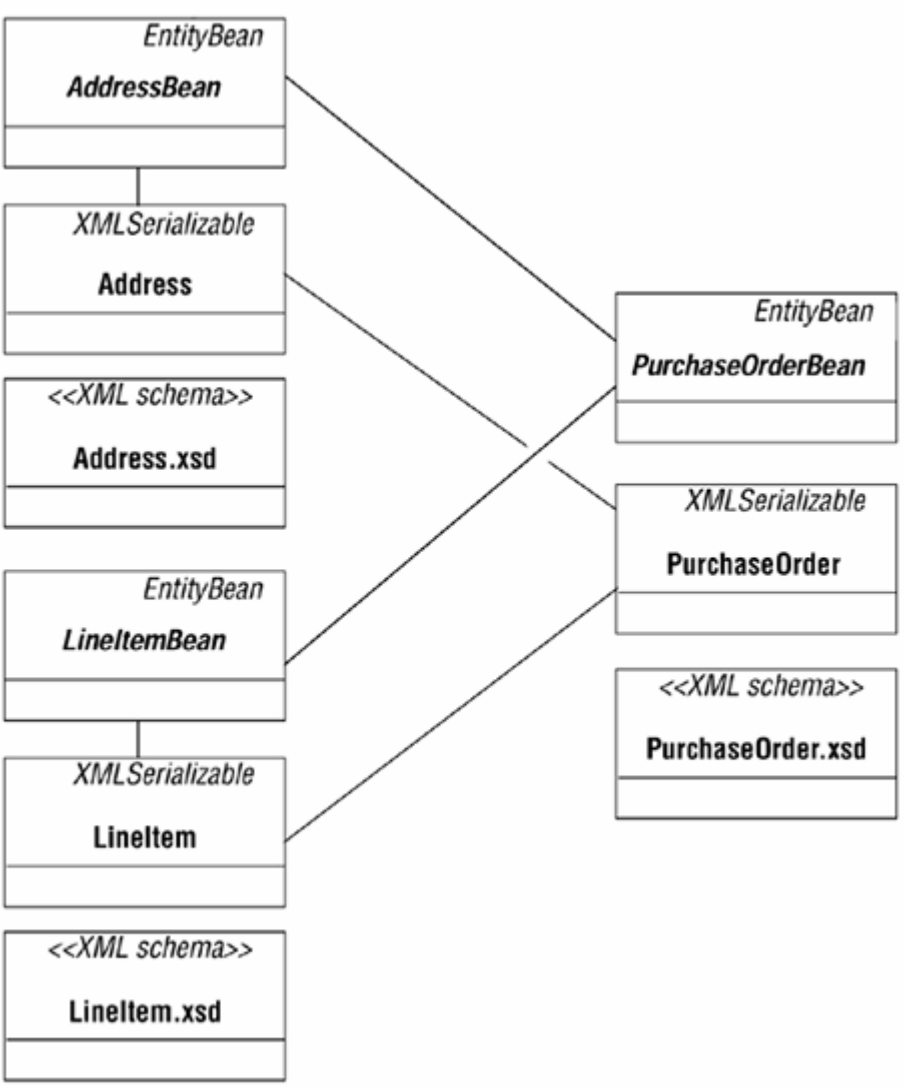
When relying on the schemas to which documents internally declare they are conforming (through a DTD declaration or an XSD hint), for security and to avoid external malicious modification, you should keep your own copy of the schemas and validate against these copies. This can be done using an entity resolver, which is an interface from the SAX API (`org.xml.sax.EntityResolver`), that forcefully maps references to well-known external schemas to secured copies.
Summary of recommendations for validation of XML document:

1. Validate incoming documents at the system boundary, especially when documents come from untrusted sources.
2. When possible, enforce validation up-front against the supported schemas.
3. When relying on internal schema declarations (DTD declaration, XSD hint, and so forth):
    a. Reroute external schema references to secured copies
    b. Check that the validating schemas are supported schemas.

A component may not need to know the complete XML document. A component may be coded against just a fragment of the overall document schema. This technique is particularly useful for implementing a document-oriented workflow where components exchange or have access to entire documents but only manipulate portions of the documents. For example, Each stage may process specific information within the document. A credit card processing stage may only retrieve the `CreditCard` element from the `PurchaseOrder` document. Upon completion, a stage may "stamp" the document by inserting information back into the document. In the case of a credit card processing stage, the credit card authorization date and status may be inserted back into the `PurchaseOrder` document.



For a document-centric processing model, especially when processing documents in the EJB tier, you may want to create generic, reusable components whose state is serializable to and from XML. For example, suppose your application works with an `Address` entity bean whose instances are initialized with information retrieved from various XML documents, such as purchase order and invoice documents. Although the XML documents conform to different schemas, you want to use the same component—the same `Address` bean—without modification regardless of the underlying supported schema. A good way to address this issue is to design a generic XML schema into which your component state can be serialized. From this generic schema, you can generate XML-serializable domain-specific or content objects that handle the serialization of your component state. You can generate the content objects manually or automatically by using XML data-binding technologies such as JAXB. Furthermore, you can combine these XML-serializable components into composite entities with corresponding composite schemas. When combined with the "meet-in-the-middle" approach discussed previously, you can apply XSLT transformations to convert XML documents conforming to external vertical schemas into your application's supported internal generic schemas. Transformations can also be applied in the opposite direction to convert documents from internal generic schemas to external vertical schemas. shows a `PurchaseOrderBean` composite entity and its two components, `AddressBean` and `LineItemBean`. The schemas for the components are composed in the same way and form a generic composite schema. Transformations can be applied to convert the internal generic composite `PurchaseOrder` schema to and from several external vertical schemas. Supporting an additional external schema is then just a matter of creating a new stylesheet.



An application's business logic may directly handle documents it consumes or produces, which is called a **document-centric processing model**, if the logic Relies on both document content and structure and Is required to punctually modify incoming documents while preserving most of their original form, including comments, external entity references, and so forth.

Moreover, the document-centric processing model does not promote a clean separation between business and XML programming skills, especially when an application developer who is more focused on the implementation of the business logic must additionally master one or several of the XML processing APIs.

There are cases that require a document-centric processing model, such as:

1. The schema of the processed documents is only partially known and therefore cannot be completely bound to domain-specific objects; the application edits only the known part of the documents before forwarding them for further processing.
2. Because the schemas of the processed documents may vary or change greatly, it is not possible to hard-code or generate the binding of the documents to domain-specific objects; a more flexible solution is required, such as one using DOM with Xpath.

A typical document-centric example is an application that implements a data-driven workflow: Each stage of the workflow processes only specific information from the incoming document contents, and there is no central representation of the content of the incoming documents. A stage of the workflow may receive a document from an earlier stage, extract information from the document, apply some business logic on the

extracted information, and potentially modify the document before sending it to the next stage. Generally, it is best to have the application's business logic directly handle documents only in exceptional situations, and to do so with great care. **You should instead consider applying the application's business logic on domain-specific objects that completely or partially encapsulate the content of consumed or produced documents. This helps to isolate the business logic from the details of XML processing.**

Because of these strong dependencies, and because they may still retain some document-centric characteristics (especially constraints), **applications may still be considered document centric when they apply business logic directly on classes generated by XML data-binding technologies from the schemas of consumed and produced documents. To change to a pure object-centric model, the developer may move the dependencies on the schemas down by mapping schema-derived objects to domain-specific objects**. The domain-specific object classes expose a constant, consistent interface to the business logic but internally delegate the XML processing details to the schema-derived classes. Overall, such a technique reduces the coupling between the business logic and the schema of the processed documents.

A pure object-centric processing model requires XML-related issues to be kept at the periphery of an application—that is, in the Web service interaction layer closest to the service endpoint, or, for more classical applications, in the Web tier. In this case, XML serves only as an additional presentation media for the application's inputs and outputs.

When implementing a document-oriented workflow in the processing layer of a Web service, or when implementing the asynchronous Web service interaction layer, an object-centric processing model may still be enforced by keeping the XML-related issues within the message-driven beans that exchange documents.

Rather than pass the entire document to different components handling various stages of the business process, it's best if the processing logic breaks the document into fragments and passes only the required fragments to other components or services that implement portions of the business process logic. Passing the entire XML document to all stages of the business process results in unnecessary information flows and extra processing. It is more efficient to extract the logical fragments—account fragment, credit card fragment, and so forth—from the incoming XML document and then pass these individual fragments to the appropriate business process stages in an appropriate format (DOM tree, Java object, serialized XML, and so forth) expected by the receiver.

Fragmenting a document has the following benefits:

1. It avoids extra processing and exchange of superfluous information throughout the workflow.
2. It maximizes privacy because it limits sending sensitive information through the workflow.
3. It centralizes some of the XML processing tasks of the workflow and therefore simplifies the overall implementation of the workflow.
4. It provides greater flexibility to workflow error handling since each stage handles only business logic-related errors while the workflow dispatching logic handles document parsing and validation errors.

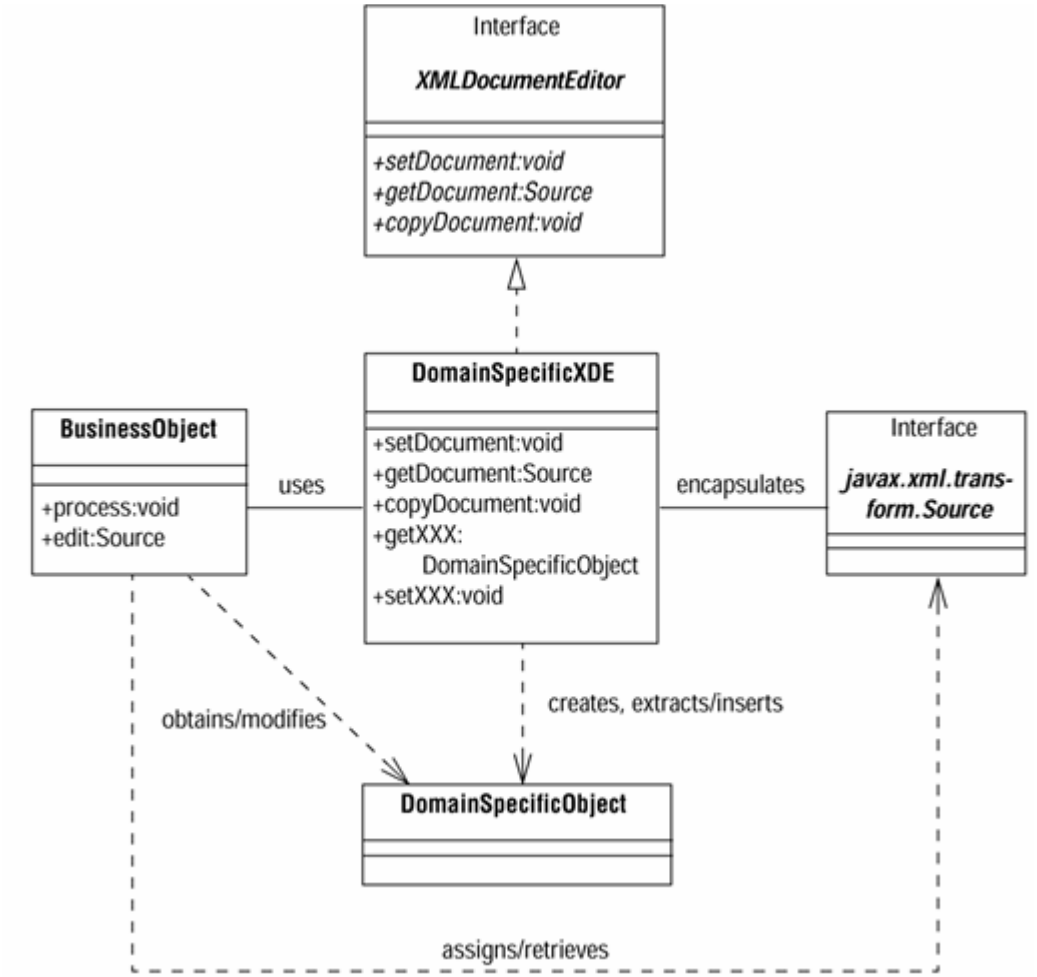
However, it loses some flexibility because the workflow dispatching logic is required to specifically know about (and therefore depend on) the document fragments and formats expected by the different stages.

With the object-centric processing model—when XML document content is mapped to domain-specific objects—the application applies its business logic on the domain-specific objects rather than the documents. In this case, only the interaction logic may handle documents. However, in the document-centric model, the application business logic itself may directly have to handle the documents. In other words, some aspects of the business model may be expressed in terms of the documents to be handled.

For example, consider a system processing a purchase order that sends the order to a supplier warehouse. The supplier, to process the order, may need to translate the incoming purchase order from the external, agreed-upon schema (such as an XSD-based schema) to a different, internal purchase order schema (such as a DTD-based schema) supported by its components. Additionally, the supplier may want to map the purchase order document to a purchase order business object. The business logic handling the incoming

purchase order must use an XML-processing API to extract the information from the document and map it to the purchase order entity. In such a case, the business logic may be mixed with the document-handling logic. If the external purchase order schema evolves or if an additional purchase order schema needs to be supported, the business logic will be impacted. Similarly, if for performance reasons you are required to revisit your choice of the XML-processing API, the business logic will also be impacted.

XML Document Editor design



The term "Editor" used here refers to the capability to programmatically create, access, and modify—that is, edit—XML documents. The XML document editor design is similar to the data access object design strategy, which abstracts database access code from a bean's business logic. The XML document editor implements the XML document processing using the most relevant API, but exposes only methods relevant to the application logic. Additionally, the XML document editor should provide methods to set or get documents to be processed, but should not expose the underlying XML processing API. These methods should use the abstract `Source` class (and `Result` class) from the JAXP API, in a similar fashion as JAX-RPC, to ensure that the underlying XML-processing API remains hidden. Also, a business object (such as an enterprise bean) that processes XML documents through an XML document editor should itself only expose accessor methods that use the JAXP abstract `Source` or `Result` class. Moreover, a business object or a service endpoint can use different XML document editor design strategies, combined with other strategies for creating factory methods or abstract factories (strategies for creating new objects where the instantiation of those objects is deferred to a subclass), to uniformly manipulate documents that conform to different schemas. The business object can invoke a factory class to create instances of different XML

document editor implementations depending on the schema of the processed document. This is an alternate approach to applying transformations for supporting several external schemas.\

```
public class SupplierOrderXDE extends XMLDocumentEditor.DefaultXDE {

   public static final String DEFAULT_ENCODING = "UTF-8";
   private Source source = null;
   private String orderId = null;

   public SupplierOrderXDE(boolean validating, ...) {
      // Initialize XML processing logic
   }

   // Sets the document to be processed
   public void setDocument(Source source) throws ... {
      this.source = source;
   }

   // Invokes XML processing logic to validate the source document,
   // extract its orderId, transform it into a different format,
   // and copy the resulting document into the Result object
   public void copyDocument(Result result) throws ... {
      orderId = null;
      // XML processing...
   }

   // Returns the processed document as a Source object
   public Source getDocument() throws ... {
      return new StreamSource(new StringReader(
          getDocumentAsString()));
   }

   // Returns the processed document as a String object
   public String getDocumentAsString() throws ... {
      ByteArrayOutputStream stream = new ByteArrayOutputStream();
      copyDocument(new StreamResult(stream));
      return stream.toString(DEFAULT_ENCODING);
   }

   // Returns the orderId value extracted from the source document
   public String getOrderId() {
      return orderId;
   }
}
```

To summarize, it is recommended that you use a design similar to the XML Document Editor presented above to abstract and encapsulate all XML document processing. In turn, the business object or service endpoint using such a document editor only invokes the simple API provided by the document editor. This hides all the complexities and details of interacting with the XML document from the business object clients. This design is not limited to the document-centric processing model where the application applies its business logic on the document itself. In an object-centric processing model, document editors can be used by the Web service interaction layer closest to the service endpoint, to validate, transform, and map documents to or from domain-specific objects. In this case, using the document editor isolates the interaction logic from the XML processing logic.

Summary:

1. When designing application-specific schemas, promote reuse, modularization, and extensibility, and leverage existing vertical and horizontal schemas.
2. When implementing a pure object-centric processing model, keep XML on the boundary of your system as much as possible—that is, in the Web service interaction layer closest to the service endpoint, or, for more classical applications, in the presentation layer. Map document content to domain-specific objects as soon as possible.
3. When implementing a document-centric processing model, consider using the flexible mapping technique. This technique allows the different components of your application to handle XML in a way that is most appropriate for each of them.

4. Strongly consider validation at system entry points of your processing model—specifically, validation of input documents where the source is not trusted.
5. When consuming or producing documents, as much as possible express your documents in terms of abstract `Source` and `Result` objects that are independent from the actual XML-processing API (SAX, DOM, XSLT, JAXB technology, and so forth) you are using.
6. Consider a "meet-in-the-middle" mapping design strategy when you want to decouple the application data model from the external schema that you want to support.
7. Abstract XML processing from the business logic processing using the XML document editor design strategy. This promotes separation of skills and independence from the actual API used.

There are four main XML programming models:
1. SAX – event-based programming model
2. DOM – in-memory tree-traversal programming model
3. JAXB – in-memory Java content class-bound programming model
4. XSLT – template-based programming model

The developer has two choices when using **SAX processing**:
1. The application can "on the fly" invoke the business logic on the extracted information. That is, the logic is invoked as soon as the information is extracted or after only a minimal consolidation. With this approach, referred to as **stream processing**, the document can be processed in one step. Thus, an application that wants to implement a stream processing model may have to perform the document parsing and the application's business logic within the context of a transaction. Keeping these operations within a transaction leverages the container's transaction capabilities: The container's transaction mode accounts for unexpected parsing errors and rolls back any invalidated business logic processing.
2. The application invokes the business logic after it completes parsing the document and has completely consolidated the extracted information. This approach takes two steps to process a document. Before invoking the application's business logic, the application first ensures that the document and the information extracted from the document are valid. Once the document data is validated, the application invokes the business logic, which may be executed within a transaction if need be.

The SAX programming model provides no facility to produce XML documents. SAX generally is very convenient for extracting information from an XML document. It is also very convenient for data mapping when the document structure maps well to the application domain-specific objects—this is especially true when only part of the document is to be mapped. Using SAX has the additional benefit of avoiding the creation of an intermediate resource-consuming representation. Finally, SAX is good for implementing stream processing where the business logic is invoked in the midst of document processing. However, SAX can be tedious to use for more complex documents that necessitate managing sophisticated context, and in these cases, developers may find it better to use DOM or JAXB.

To use SAX remember that, the structure of a document and the order of its information map well to the domain-specific objects or corresponds to the order in which discrete methods of the application's logic must be invoked. Otherwise, you may have to maintain rather complicated contexts.

The **DOM API** prior to version level 3 does not support serialization of DOM trees back to XML. Although some implementations do provide serialization features, these features are not standard. Thus, developers should instead rely on XSLT identity transformations, which provide a standard way to achieve serialization back to XML. Although not yet standard for the Java platform (not until JAXP 1.3), the Xpath API enables using it in conjunction with the DOM programming model. (The Xpath API can be found along with some DOM implementations such as Xerces or dom4j.) Developers use Xpath to locate and extract information from a source document's DOM tree.
DOM is beneficial when,
1. You want to implement data binding but you cannot use JAXB technology because the document either has no schema or it conforms to a DTD schema definition rather than to an XSD schema definition. The document may also be too complex to use SAX to implement data binding.

2. memory usage is not a big issue – ie the document is not very large.
3. you want to create/modify the document.
4. you want random access to any part of the document.

JAXB programming model has all benefits of DOM API (random access, able to modify/create document) and gives some additional benefits like:

1. Memory usage may be less of an issue. A JAXB implementation, such as the standard implementation, creates a Java representation of the content of a document that is much more compact than the equivalent DOM tree. The standard implementation is layered on top of SAX 2.0 and does not maintain an additional underlying representation of the source document. Additionally, where DOM represents all XML schema numeric types as strings, JAXB's standard implementation maps these values directly to much more compact Java numeric data types. Not only does this use less memory to represent the same content, the JAXB approach saves time, because converting between the two representations is not necessary.
2. You deal directly with POJO bound to the XML document.

The catch is, that the JAXB compiler can generate classes only for documents which validate against XML schema types. Using a binding compiler, the XML data-binding programming model, as implemented by JAXB, binds components of a source XSD schema to schema-derived Java content classes. JAXB binds an XML namespace to a Java package. XSD schema instance documents can be unmarshalled into a tree of Java objects (called a content tree), which are instances of the Java content classes generated by compiling the schema. Applications can access the content of the source documents using JavaBeans-style get and set accessor methods. In addition, you can create or edit an in-memory content tree, then marshal it to an XML document instance of the source schema. Whether marshalling or unmarshalling, the developer can apply validation to ensure that either the source document or the document about to be generated satisfy the constraints expressed in the source schema. If the document validates against a DTD or it does not conform to either DTD or XML schema then JAXB is not an option for such documents and you will have to use DOM/SAX to work with such documents.

XSLT does not compare with other processing models and should be regarded as complementary, to be used along with these other models. For the most part, XSLT requires developers to code rules, or templates, that are applied when specified patterns are encountered in the source document. The application of the rules adds new fragments or copies fragments from the source tree to a result tree. The patterns are expressed in the Xpath language, which is used to locate and extract information from the source document.
The XSLT API available with JAXP provides an abstraction for the source and result of transformations, allowing the developer not only the ability to chain transformations but also to interface with other processing models, such as SAX, DOM, and JAXB technology. To interface with SAX and DOM, use the classes `SAXSource`, `SAXResult`, `DOMSource`, and `DOMResult` provided by JAXP. To interface with JAXB, use the classes `JAXBSource` and `JAXBResult`.
Note that although the DOM version level 2 API does not support serialization—that is, transformation of a DOM tree to an XML document—the JAXP implementation of XSLT addresses the serialization of a DOM tree using an identity transformer. An identity transformer copies a source tree to a result tree and applies the specified output method, thus solving the serialization problem in an easy, implementation-independent manner.

In general, when you must deal with non-interactive presentation or you must integrate various XML data sources or perform XML data exchanges.

Summary for XML implementation programming model usage :
If you need to deal with the content and structure of the document, consider using DOM and SAX because they provide more information about the document itself than JAXB usually does. On the other hand, if your focus is more on the actual, domain-oriented objects that the document represents, consider using JAXB, since JAXB hides the details of unmarshalling, marshalling, and validating the document. Developers should use JAXB—XML data-binding—if the document content has a representation in Java that is directly usable by the application (that is, close to domain-specific objects).

DOM, when used in conjunction with XPath, can be a very flexible and powerful tool when the focus is on the content and structure of the document. DOM may be more flexible than JAXB when dealing with documents whose schemas are not well-defined.

Finally, use XSLT to complement the three other processing models, particularly in a pre- or post-processing stage.

DOM	SAX	XML Data-Binding
Tree traversal model	Event-based model	Java-bound content tree model
Random access (in-memory data structure) using generic (application independent) API	Serial access (flow of events) using parameters passed to events	Random access (in-memory data structure) using Java-Beans style accessors
High memory usage (The document is often completely loaded in memory, though some techniques such as deferred or lazy DOM node creation may lower the memory usage.)	Low memory usage (only events are generated)	Intermediate memory usage (The document is often completely loaded in memory, but the Java representation of the document is more effective than a DOM representation. Nevertheless, some implementations may implement techniques to lower the memory usage.)
To edit the document (processing the in-memory data structure)	To process parts of the document (handling relevant events)	To edit the document (processing the in-memory data structure)
To process multiple times (document loaded in memory)	To process the document only once (transient flow of events)	To process multiple times (document loaded in memory)
Processing once the parsing is finished	Stream processing (start processing before the parsing is finished, and even before the document is completely read)	Processing once the parsing is finished

The JAXP API provides support for chaining XML processings: The JAXP `javax.xml.Source` and `javax.xml.Result` interfaces constitute a standard mechanism for chaining XML processings. JAXP also provides support for chaining transformations with the use of `javax.xml.transform.sax.SAXTransformerFactory`. Code [example 4.10 in book] illustrates an XML processing pipeline that combines SAX and XSLT to validate an incoming purchase order document, extract on-the-fly the purchase order identifier, and transform the incoming document from its external, XSD-based schema to the internal, DTD-based schema supported by the business logic. The code uses a SAX filter chain as the `Source` of a transformation. Alternatively, the code could have used a `SAXTransformerFactory` to create an `org.xml.sax.XMLFilter` to handle the transformation and then chain it to the custom `XMLFilter`, which extracts the purchase order identifier.

A custom entity resolution allows you to implement the desired mapping of external entity references to actual trusted physical locations. In summary, you may want to consider implementing a custom entity resolution—or, even better, resort to a more elaborate XML catalog solution—in the following circumstances:

1. To protect the integrity of your application against malicious modification of external schemas by redirecting references to secured copies (either local copies or those on trusted repositories)

2. During design and, even more so, during production, to protect your application against unexpected, legitimate evolution of the schemas upon which it is based. Instead, you want to defer accounting for this evolution until after you have properly assessed their impact on your application.
3. To improve performance by maintaining local copies of otherwise remotely accessible schemas.

XML processing is potentially CPU, memory, and network intensive, for these reasons:

1. It may be CPU intensive. Incoming XML documents need not only to be parsed but also validated, and they may have to be processed using APIs which may themselves be CPU intensive. It is important to limit the cost of validation as much as possible without jeopardizing the application processing and to use the most appropriate API to process the document.
2. It may be memory intensive. XML processing may require creating large numbers of objects, especially when dealing with document object models (or JAXB).
3. It may be network intensive. A document may be the aggregation of different external entities that during parsing may need to be retrieved across the network. It is important to reduce as much as possible the cost of referencing external entities.

To improve performance while processing XML, use the following guidelines:

1. In general, it is best to parse incoming XML documents only when the request has been properly formulated. In the case of a Web service application, if a document is retrieved as a `Source` parameter from a request to an endpoint method, it is best first to enforce security and validate the meta information that may have been passed as additional parameters with the request.
2. In general, without considering memory consumption, processing using the DOM API tends to be slower than processing using the SAX API. This is because DOM may have to load the entire document into memory so that the document can be edited or data retrieved, whereas SAX allows the document to be processed as it is parsed. However, despite its initial slowness, it is better to use the DOM model when the source document must be edited or processed multiple times.
3. In any case, JAXB uses less memory resources as a JAXB content tree is by nature smaller than an equivalent DOM tree.
4. When building complex XML transformation pipelines, use the JAXP class `SAXTransformerFactory` to process the results of one style sheet transformation with another style sheet. You can optimize performance—by avoiding the creation of in-memory data structures such as DOM trees—by working with SAX events until at the last stage in the pipeline.
5. JDOM and dom4j are particularly appropriate for applications that implement a document-centric processing model and that must manipulate a DOM representation of the documents. Developers find JDOM easy to use because it relies on the Java `Collection` API. JDOM documents can be built directly from, and converted to, SAX events and DOM trees, allowing JDOM to be seamlessly integrated in XML processing pipelines and in particular as the source or result of XSLT transformations
6. When receiving documents through a service endpoint (either a JAX-RPC or EJB service endpoint) documents are parsed as abstract `Source` objects. As already noted, do not assume a specific implementation—`StreamSource`, `SAXSource`, or `DOMSource`—for an incoming document. Instead, you should ensure that the optimal API is used to bridge between the specific `Source` implementation passed to the endpoint and the intended processing model. Keep in mind that the JAXP XSLT API does not guarantee that identity transformations are applied in the most effective way.
7. A developer may also want to implement stream processing for the application so that it can receive the processing requirements as part of the SOAP request and start processing the document before it is completely received. Document processing in this manner improves overall performance and is useful when passing very large documents. Extreme caution should be taken if doing this, since there is no guarantee that the underlying JAX-RPC implementation will not wait to receive the complete document before passing the `Source` object to the endpoint and that it will effectively pass a `Source` object that allows for stream processing, such as `StreamSource` or `SAXSource`. The same holds true when implementing stream processing for outgoing documents. While you can pass a `Source` object that allows for stream processing, there is no guarantee on how the underlying JAX-RPC implementation will actually handle it.

8. Xerces defines a deferred expansion feature called `http://apache.org/xml/features/dom/defer-node-expansion`, which enables or disables a lazy DOM mode. In lazy mode (enabled by default), the DOM tree nodes are lazily evaluated, their creation is deferred: They are created only when they are accessed. As a result, DOM tree construction from an XML document returns faster since only accessed nodes are expanded. This feature is particularly useful when processing only parts of the DOM tree. Grammar caching, another feature available in Xerces, improves performance by avoiding repeated parsing of the same XML schemas. This is especially useful when an application processes a limited number of schemas, which is typically the case with Web services.
9. When the underlying implementation encounters a feature or a property that it does not support or recognize, the `SAXParserFactory`, the `XMLReader`, or the `DocumentBuilderFactory` may throw these exceptions: a `SAXNotRecognizedException`, a `SAXNotSupportedException`, or an `IllegalArgumentException`. You may design your application in such a way that features and properties specific to the underlying implementations may also be defined externally to the application, such as in a configuration file.
10. Parsers, document builders, and transformers, as well as style sheets, can be pooled using a custom pooling mechanism. Or, if the processing occurs in the EJB tier, you may leverage the EJB container's instance pooling mechanism by implementing stateless session beans or message-driven beans dedicated to these tasks. Since these beans are pooled by the EJB container, the parsers, document builders, transformers, and style sheets to which they hold a reference are pooled as well.
11. Style sheets can be compiled into `javax.xml.transform.Templates` objects to avoid repeated parsing of the same style sheets. `Templates` objects are thread safe and are therefore easily reusable.
12. Although you must validate external incoming XML documents, you can exchange freely—that is, without validation—internal XML documents or already validated external XML documents. When you are both the producer and consumer of XML documents, you may use validation as an assertion mechanism during development, then turn off validation when in production. Additionally, during production validation can be used as a diagnostic mechanism by setting up validation so that it is triggered by fault occurrences.
13. You can improve the efficiency of locating references to external entities that are on a remote repository by setting up a proxy that caches retrieved, external entities.
14. SAX parsers allow XML applications to handle external entities in a customized way. Such applications have to register their own implementation of the `org.xml.sax.EntityResolver` interface with the parser using the `setEntityResolver` method. The applications are then able to intercept external entities (including schemas) before they are parsed. Similarly, JAXP defines the `javax.xml.transform.URIResolver` interface. Implementing this interface enables you to retrieve the resources referred to in the style sheets by the `xsl:import` or `xsl:include` statements. You can use `EntityResolver` and `URIResolver` to implement:
    a. A caching mechanism in the application itself, or
    b. A custom URI lookup mechanism that may redirect system and public references to a local copy of a public repository.
15. Use a proxy cache for static entities whose lifetime is greater than the application's lifetime. This particularly works with public schemas, which include the version number in their public or system identifier, since they evolve through successive versions. A custom entity resolver may first map public identifiers (usually in the form of a URI) into system identifiers (usually in the form of an URL). Afterwards, it applies the same techniques as a regular cache proxy when dealing with system identifiers in the form of an URL, especially checking for updates and avoiding caching dynamic content. Using these caching approaches often results in a significant performance improvement, especially when external entities are located on the network. [For code of how to implement a Caching Entity Resolver using SAX API see book pg 190.]
16. Dynamically generated documents are typically assembled from values returned from calls to business logic. Generally, it is a good idea to cache dynamically generated XML documents to avoid having to refetch the document contents, which entails extra round trips to a business tier.

This is a good rule to follow when the data is predominantly read only, such as catalog data. Furthermore, if applicable, you can cache document content (DOM tree or JAXB content tree) in the user's session on the interaction or presentation layer to avoid repeatedly invoking the business logic. When you take this approach, keep in mind that it must not be done to the detriment of other users. That is, be sure that the application does not fail because of a memory shortage caused by holding the cached results. To help with memory management, use **soft references** (java.lang.ref.**SoftReference**<T> class), which allow more enhanced interaction with the garbage collector to **implement caches**.

17. When caching a DOM tree in the context of a distributed Web container, the reference to the tree stored in an HTTP session may have to be declared as **`transient`**. This is because `HttpSession` requires objects that it stores to be Java serializable, and not all DOM implementations are Java serializable. Also, Java serialization of a DOM tree may be very expensive, thus countering the benefits of caching.
18. Rely on XML protocols, such as those implemented by JAX-RPC and others, to interoperate with heterogeneous systems and to provide loosely coupled integration points – only. Avoid using XML for unexposed interfaces or interaction between components (which should be tightly coupled) and implemented using protocols like RMI.

Client Design

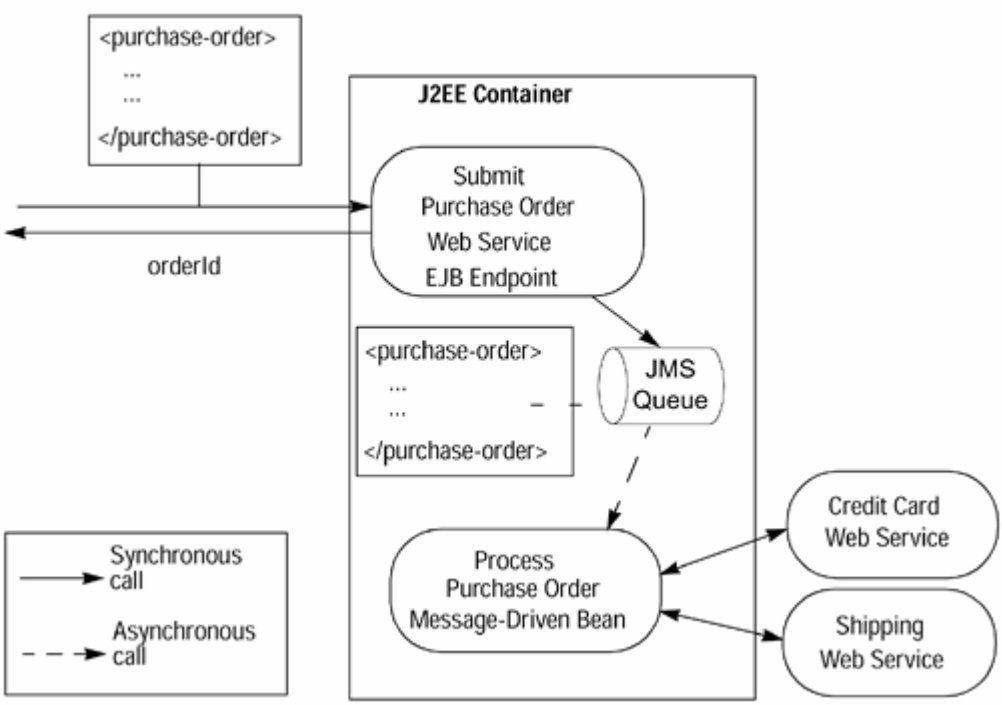
Types of webservice clients:

1. J2EE Component
2. J2SE client
3. J2ME Client
4. Non-Java Client

Which Webservice client to use?

1. J2EE Client - J2EE clients have good access to Web services. J2EE clients have other advantages provided by the J2EE platform, such as declarative security, transactions, and instance management. J2EE clients may also access Web services from within a workflow architecture, and they may aggregate Web services
2. J2SE Client - Generally, J2SE clients are best when you need to provide a rich interface or when you must manipulate large sets of data. J2SE clients may also work in a disconnected mode of operation.
3. J2ME clients - are best for applications that require remote and immediate access to a Web service. J2ME clients may be restricted to a limited set of interface components. Like J2SE clients, J2ME clients may also work in a disconnected mode of operation.

Often, EJB components are used in a workflow to provide Web services with the additional support provided by an EJB container—that is, declarative transactional support, declarative security, and life-cycle management.



In the fig. above, a Web service endpoint used in combination with a message-driven bean component to provide a workflow operation that runs asynchronously once the initial service starts. The Web service endpoint synchronously puts the purchase order in a JMS message queue and returns an `orderId` to the calling application. The message-driven bean listens for messages delivered from the JMS queue. When one arrives, the bean retrieves the message and initiates the purchase order processing workflow. The purchase order is processed asynchronously while the Web service receives other purchase orders. In this example, the workflow consists of three additional stages performed by separate Web services: a credit card charging service, a shipping service, and a service that sends an order confirmation. The **message-driven bean aggregates the three workflow stages**. Other systems within an organization may provide these services and they may be shared by many applications.

J2SE Clients are good for:

1. Long running applications
2. Rich GUI for complex data
3. Require only intermittent network access
4. Requiring complex computation on client – image manipulation

Best practices for J2SE clients are:

1. Whenever possible, use Java Web Start to provide a standardized means of deployment for J2SE clients.
2. When designing a Web service client, try to keep Web service access code separate from GUI code.
3. Since J2SE clients may operate in either a connected or disconnected mode, when developing these clients keep in mind issues related to maintenance of state and client and service synchronization.

Networks supporting J2ME devices may not always provide consistent connectivity. Applications using such networks must consider connection failure, or sporadic connectivity, and be designed so that recovery is possible. J2ME network providers may charge for network usage by the kilobyte. J2ME-targeted applications may be expensive for the user unless care is taken to limit the data transferred. Applications for J2ME devices may work in an offline, or disconnected, mode as well as an online, or connected, mode. When working in an offline mode, applications should collect data and batch it into requests to the Web service, as well as obtain data from the service in batches. Consideration should be given to the amount of data that is passed between the Web service and the client.

If you are using stubs or dynamic proxies, you must first locate and gain access to the full WSDL document representing the Web service, since you develop the client application from the stubs and supporting files generated from the WSDL. If you have access to only a partial WSDL file, then use DII, since DII lets you locate the service at runtime.

When using stubs and dynamic proxies, invocations are synchronous. Use DII if you choose to have a one-way call.

When using **stubs**, a JAX-RPC runtime tool generates during development static stub classes that enable the service and the client to communicate. The stub, which sits between the client and the client representation of the service endpoint interface, is responsible for converting a request from a client to a SOAP message and sending it to the service. The stub also converts responses from the service endpoint, which it receives as SOAP messages, to a format understandable by the client. In a sense, a stub is a local object that acts as a proxy for the service endpoint.

Dynamic proxies provides the same functionality as the stubs, but do so in a more dynamic fashion. Stubs and dynamic proxies both provide the developer access to the `javax.xml.rpc.Stub` interface, which represents a service endpoint. With both models, it is easy for a developer to program against the service endpoint interface, particularly because the JAX-RPC runtime does much of the communication work behind the scenes. The dynamic proxy model differs from the stub model principally because the dynamic proxy model does not require code generation during development.

DII is a call interface that supports a programmatic invocation of JAX-RPC requests. Using DII, a client can call a service or a remote procedure on a service without knowing at compile time the exact service name or the procedure's signature. A DII client can discover this information at runtime and can dynamically look up the service and its remote procedures.

The stubs rely on a tool that uses the WSDL file to create the service endpoint interface, plus generate stub and other necessary classes. These generated classes eliminate the need for the developer to use the JAX-RPC APIs directly. By contrast, the dynamic proxy and DII approaches both require the developer to use the JAX-RPC APIs.

Applications in J2ME environments can use only stubs to access Web services, and stubs are portable for J2ME devices. The JAX-RPC profile for J2ME environments does not support dynamic proxies and DII.

With Dynamic Proxies, unlike stubs, the client developer needs only the client-side interface that matches the service endpoint interface. That is, clients using dynamic proxies program to an interface that ensures the client application is portable across other JAX-RPC runtime implementations. Developers using dynamic proxies must create Java classes to serve as JAX-RPC value types—these classes have an empty constructor and set methods for each field, similar to JavaBeans classes. Client applications can access the Web service ports using the `javax.xml.rpc.Service` method `getPort`. Consider using the dynamic proxy approach if portability is important to your application, since this approach uses the service's endpoint interface to communicate with a service at runtime. Dynamic proxy communication is the most portable mode across JAX-RPC implementations. Because of how they access a service at runtime, dynamic proxies may have additional overhead when calls are made.

A client application may also dynamically access a Web service by locating the service at runtime from a registry. The client does not know about the service when it is compiled; instead, the client discovers the service's name from a JAXR registry at runtime. Along with the name, the client discovers the required parameters and return values for making a call to the service. Using a dynamic invocation interface, the client locates the service and calls it at runtime. A developer may choose to use the DII approach when a complete WSDL document is not available or provided, particularly when the WSDL document does not specify ports. The DII approach is more suitable when used within a framework, since from within a framework, client applications can generically and dynamically access services with no changes to core application code. The client uses the information from the registry to construct a `javax.xml.rpc.Call`,

which it uses to access the Web service. DII involves more work than stubs or dynamic proxies and should be used sparingly. You may consider using this mode if a service changes frequently.

Client Considerations	Stub	Dynamic Proxy	DII
Portable client code across JAX-RPC implementations	Yes, in the J2EE platform when an application uses a neutral means for accessing the stubs. (For an example, see Code Example 5.2.) Since stubs are bound to a specific JAX-RPC runtime, reliance on JAX-RPC-specific access to a stub may not behave the same on all platforms. Stub code needs to be generated for an application.	Yes	Yes
Requires generation of code using a tool	Yes	No. A tool may be used to generate JAX-RPC value types required by a service endpoint interface (but not serializers and other artifacts).	No
Ability to programmatically change the service endpoint URL	Yes, but the WSDL must match that used to generate the stub class and supporting classes.	Yes, but the client-side service endpoint interface must match the representation on the server side.	Yes
Supports service specific exceptions	Yes	Yes	No. All are `java.rmi`.Remote exceptions. Checked exceptions cannot be used when calls are made dynamically.
Supports one way communication mode	No	No	Yes
Supports the ability to dynamically specify JAX-RPC value types at runtime	No. Developer must program against a service endpoint interface.	No. Developer must program against a service endpoint interface.	Yes. However, returns Java objects which the developer needs to cast to application-specific objects as necessary.
Supported in J2ME platform	Yes	No	No
Supported in J2SE and J2EE platforms	Yes	Yes	Yes
Requires WSDL	No. A service endpoint interface may generate a stub class along with information concerning the protocol binding.	No. A partial WSDL (one with the service port element undefined) may be used.	No. Calls may be used when partial WSDL or no WSDL is specified. Use of methods other than the createCall method on the Call interface may result in unexpected behavior in such cases.

By using the JAX-RPC `javax.xml.rpc.Service` interface method **`getPort`**, you can access a Web service in the same manner regardless of whether you use stubs or dynamic proxies. The `getPort` method returns either an instance of a generated stub implementation class or a dynamic proxy, and the client can then uses this returned instance to invoke operations on the service endpoint. The `getPort` method

removes the dependencies on generated service-specific implementation classes. When this method is invoked, the JAX-RPC runtime selects a port and protocol binding for communicating with the port, then configures the returned stub that represents the service endpoint interface. Furthermore, since the J2EE platform allows the deployment descriptor to specify multiple ports for a service, the container, based on its configuration, can choose the best available protocol binding and port for the service call.

J2EE client using Dynamic Proxy:

```
Context ic = new InitialContext();

Service service = (Service)ic.lookup("java:comp/env/service/OpcPurchaseOrderService");

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

Using the `Service` interface in this manner reduces the dependency on generated stub classes. The client developer, by invoking the `getPort` method, uses the client-side representation of the service endpoint interface to look up the port. After obtaining the port, the client may make any calls desired by the application on the port.

1. When using stubs or dynamic proxies, the recommended strategy to reduce the dependency on generated classes is to use the `java.xml.rpc.Service` interface and the `getPort` method as a proxy for the service implementation class.
2. A client developer should not circumvent the J2EE platform's management of a service. A client should not create or destroy a Web service port.
3. A client developer should not assume that the same port instance for the service is used for all calls to the service. Port instances are stateless, and the J2EE platform is not required to return a previously used port instance to a client.

J2SE using generated Stub:

```
Stub stub = (Stub)(new OpcOrderTrackingService_Impl() getOrderTrackingIntfPort());

OrderTrackingIntf port = (OrderTrackingIntf)stub;
```

An application in a non-J2EE environment uses a stub to make a Web services call in a different manner. The client application accesses a stub for a service using the method `getOrderTrackingIntfPort` on the generated implementation class, `OpcOrderTrackingService_Impl`, which is specific to each JAX-RPC runtime. J2SE or J2ME clients use these generated `_Impl` files because they do not have access to the naming services available to clients in a J2EE environment through JNDI APIs. In addition, a J2SE or J2ME client can access a service by using the `javax.xml.rpc.ServiceFactory` class to instantiate a stub object.

```
ServiceFactory factory =  ServiceFactory.newInstance();

Service service = factory.createService(new QName(
   "urn:OpcOrderTrackingService", "OpcOrderTrackingService"));
```

J2SE using Dynamic Proxy service lookup:

```
ServiceFactory sf = ServiceFactory.newInstance();

String wsdlURI = "http://localhost:8001/webservice/OtEndpointEJB?WSDL";

URL wsdlURL = new URL(wsdlURI);

Service ots = sf.createService(wsdlURL,
       new QName("urn:OpcOrderTrackingService",
       "OpcOrderTrackingService"));

OrderTrackingIntf port = ( OrderTrackingIntf)ots.getPort(new QName(
       "urn:OpcOrderTrackingService", "OrderTrackingIntfPort"),
       OrderTrackingIntf.class);
```

A J2SE client uses a `ServiceFactory` to look up and obtain access to the service, represented as a `Service` object. The client uses the qualified name, or `QName`, of the service to obtain the service's port. The WSDL document defines the `QName` for the service. The client needs to pass as arguments the `QName` for the target service port and the client-side representation of the service endpoint interface.

It is simpler for J2EE clients to look up and access a service than it is for J2SE clients, since a JNDI lookup from the `IntialContext` of an existing service is much simpler than configuring the parameters for `ServiceFactory`. The J2EE client just invokes a `getPort` call on the client-side representation of the service endpoint interface.
DII communication supports two invocation modes: synchronous and one way, also called fire and forget. Both invocation modes are configured with the `javax.xml.rpc.Call` object. Note that DII is the only communication model that supports one-way invocation.

J2SE using DII:

```
Service service = //get service

QName port = new QName("urn:OpcOrderTrackingService", "OrderTrackingIntfPort");

Call call = service.createCall(port);

call.setTargetEndpointAddress("http://localhost:8000/webservice/OtEndpointEJB");

call.setProperty(Call.SOAPACTION_USE_PROPERTY, new Boolean(true));

call.setProperty(Call.SOAPACTION_URI_PROPERTY,"");

call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);

QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");

call.setReturnType(QNAME_TYPE_STRING);

call.setOperationName(new QName(BODY_NAMESPACE_VALUE "getOrderDetails"));

call.addParameter("String_1", QNAME_TYPE_STRING, ParameterMode.IN);

String[] params = {orderId};

OrderDetails = (OrderDetails)call.invoke(params);
```

It shows how the `Call` interface used by DII is configured with the property values required to access the order tracking Web service. The values set for these properties may have been obtained from a registry. Keep in mind that using DII is complex and often requires more work on the part of the client developer.

Stubs may be configured statically or dynamically. A stub's static configuration is set from the WSDL file description at the time the stub is generated. Instead of using this static configuration, a client may use methods defined by the `javax.xml.rpc.Stub` interface to dynamically configure stub properties at runtime. Two methods are of particular interest: **`_setProperty`** to configure stub properties and **`_getProperty`** to obtain stub property information. Clients can use these methods to obtain or configure such properties as the service's endpoint address, user name, and password. Generally, it is advisable to cast vendor-specific stub implementations into a `javax.xml.rpc.Stub` object for configuration. This ensures that configuration is done in a portable manner and that the application may be run on other JAX-RPC implementations with minimal changes, if any.

```
Service opcPurchaseOrderSvc = (Service) ic.lookup(AdventureKeys.PO_SERVICE);

PurchaseOrderIntf port = opcPurchaseOrderSvc.getPort(PurchaseOrderIntf.class);

((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,

   "http://localhost:8000/webservice/PoEndpointEJB");
```

In a J2EE environment, the J2EE container reserves the right to use two security-related properties for its own purposes. As a result, J2EE clients using any of the three communication modes (stub, dynamic proxy, or DII) should not configure these two security properties:
`javax.xml.rpc.security.auth.username`


```
javax.xml.rpc.security.auth.password
```

However, J2SE developers do need to set these two security properties if they invoke a Web service that requires basic authentication. When using DII, the client application may set the properties on the `Call` interface.

1. J2EE client developers should avoid setting properties other than the `javax.xml.rpc.endpoint.address` property.
2. Avoid setting nonstandard properties if it is important to achieve portability among JAX-RPC runtimes. Nonstandard properties are those whose property names are not preceded by `javax.xml.rpc`.
3. Avoid using `javax.xml.rpc.session.maintain` property. This property pertains to a service's ability to support sessions. Unless you have control over the development of both the client and the endpoint, such as when both are developed within the same organization, you cannot be sure that the Web service endpoints support sessions, and you may get into trouble if you set this property incorrectly.

WSDL-to-Java type mapping

Generally, the JAX-RPC runtime handles the mapping of parameters, exceptions, and return values to JAX-RPC types. When a client invokes a service, the JAX-RPC runtime maps parameter values to their corresponding SOAP representations and sends an HTTP request containing a SOAP message to the service. When the service responds to the request, the JAX-RPC runtime receives this SOAP response and maps the return values to Java objects or standard types. If an exception occurs, then the runtime maps the `WSDL:fault` to a Java exception, or to a `javax.rmi.RemoteException` if a `soap:fault` is encountered.

The JAX-RPC runtime supports the following standard value types: `String`, `BigInteger`, `Calender`, `Date`, `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and arrays of these types. Services can return mime types as images mapped to the `java.awt.Image` class and XML text as `javax.xml.transform.Source` objects. (The WSDL Basic Profile 1.0 does not support `javax.xml.transform.Source` objects as mime types. As a result, you should avoid this usage until it is supported by a future WSDL version.) A service may also return complex types, and these are mapped to Java `Object` representations.

Use WSDL-to-Java tools to generate support classes, even if using the dynamic proxy or DII approach.

1. Generally, whenever possible client developers should use JSP to generate responses by a service and to present the data as a view to clients (such as browsers that use Web tier technologies).
2. For clients in a non-Web tier environment where JSP technology is not available (like EJB endpoints), developers should use XSLT transformations.

Developers can use JSP technology to access content in XML documents and to build an HTML page from the XML document contents.

```
<%@ taglib prefix="x" uri="/WEB-INF/x-rt.tld" %>

<x:parse xml="${orderDetailsXml}" var="od" scope="application"/>

<html>

  Order:<x:out select="$od/orderdetails/id"/><br>

  Status:<x:out select="$od/orderdetails/status"/><br>

  Name:<x:out select="$od/orderdetails/shippinginfo/given-name"/>

      <x:out select="$od/orderdetails/shippinginfo/family-name"/>

</html>
```

The above code processes the following input data:

```
<orderdetails>
    <id>54321</id>
    <status>SHIPPED</status>
```

```
    <shippinginfo>
       <family-name>Smith</family-name>
       <given-name>Duke</given-name>
    </shippinginfo>
</orderdetails>
```

And the following is the o/p produced for the client in html format:

```
<html>

    Order: 54321<br>

    Status: SHIPPED<br>

    Name: Duke Smith<br>

</html>
```

In this example, the J2EE application first places the order details document received from the service in the request scope using the key `orderDetailsXML`. The next lines are JSP code that use the `x:out` JSTL tag to access the order details XML content. These lines of code select fields of interest (such as order identifier, status, and name fields) using XPath expressions, and convert the data to HTML for presentation to a browser client.

Often **system exceptions** happen because of network failures or server errors. They also may be the result of a SOAP fault. Since a `RemoteException` usually contains an explanation of the error, the application can use that message to provide its own error message to the user and can prompt the user for an appropriate action to take. If an EJB component client is doing its work on behalf of a Web tier client, the EJB client should throw an exception to the Web tier client. The Web tier client notifies the user, giving the user a chance to retry the action or choose an alternative action. When using a dynamic proxy, the `getPort` method may throw a `javax.xml.rpc.ServiceException` if the WSDL document has insufficient metadata to properly create the proxy.

Service exceptions occur when a Web service call results in the service returning a fault. A service throws such faults when the data presented to it does not meet the service criteria. For example, the data may be beyond boundary limits, it may duplicate other data, or it may be incomplete. These exceptions are defined in the service's WSDL file as `operation` elements, and they are referred to as `wsdl:fault` elements. These exceptions are checked exceptions in client applications. For example, a client accessing the order tracking service may pass to the service an order identifier that does not match orders kept by the service. The client may receive an `OrderNotFoundException`, since that is the error message defined in the WSDL document:

```
<fault name="OrderNotFoundException"

          message="tns:OrderNotFoundException"/>
```

This exception-mapping mechanism may not be used with DII, since this communication mode returns all exceptions as `java.rmi.RemoteException`.

Use the JAX-RPC tools to map faults to Java objects. Generated exceptions classes extend `java.lang.Exception`. A J2EE Web component client may handle the exception using the facilities provided by the J2EE environment. The client may wrap the application exception and throw an unchecked exception, such as a `javax.servlet.ServletException`, or it may map the exception directly to an error page in the Web deployment descriptor. Notice that using the servlet error mechanism in this way tightly binds the Web application client to the service. If the service changes the faults it throws, or any of the fault parameters, the client is directly affected.

```
<error-page>
   <exception-type>com.sun.blueprints.adventure.OrderNotFoundException</exception-type>
   <location>/order_not_found_exception.jsp</location>
</error-page>
```

A J2SE client will handle the service exception in a try-catch block.

Client developers should isolate service-specific exceptions as much as possible by wrapping them in their own application-specific exceptions to keep the client application from being too closely tied to a service. This is especially important when the service is outside the control of the client developer or if the service changes frequently. A client may require refactoring when a service changes because the stubs and

supporting Java object representations of the exceptions were generated statically. Client developers may also generalize exception handling and instead handle all exceptions in a single point in the application. Keeping exception handling to one place in an application reduces the need to include code for handling exceptions throughout the application.

```
try {

   OrderDetails od = stub.getOrderDetails(orderId);

} catch (OrderNotFoundException onx) {

   RuntimeException re= new RuntimeException(onx);

}
```

The exception thrown as a result of the Web service call is set as the cause of the runtime exception. Do boundary checking and other validations on the client side so that Web service calls and round-trips to the server are kept to a minimum.

Managing conversational state

Storing conversation state at service endpoint: Often, such endpoints are designed to use a unique, nonreplicable token to identify communication from a specific client, much like browsers use the cookie mechanism. The service endpoint and the client application pass the token between them with each call during their conversation.

```
public interface OrderManagementSEI extends Remote {

   public void updatePurchaseOrder(PurchaseOrder po, String clientToken)
                              throws RemoteException;

   public PurchaseOrder getPurchaseOrder(String id, String clientToken)
                              throws RemoteException;

}
```

When an EJB component is the basis for a service endpoint, the EJB component can persist conversational state to a data store during the session. The service endpoint may still be designed to use tokens, and such **tokens may represent the primary key of an entity bean**. In this case, the endpoint designer must be sure to **clean up stale session data from the persistent store**. This can be done using a time stamp on the entity bean holding the conversational state and a timer bean to track elapsed time. An endpoint that does not properly clean up stale session data might eventually persist a large amount of data.

Packaging

J2EE clients require the following:

1. service-ref element in the DD
2. JAX-RPC mapping file
3. WSDL – in META-INF/WSDL for EJB client or WEB-INF/WSDL for web client.
4. Service endpoint interface
5. Generated classes

WSDL files, including partial WSDL files, are packaged within clients. Their location is dependent on the type of module. Since clients using DII do not require a WSDL file, they leave the `wsdl-file` element portion of the `service-ref` element undefined and they must not specify the `jaxrpc-mapping-file` element.

Figure 5.8. Web Application Module Packaging

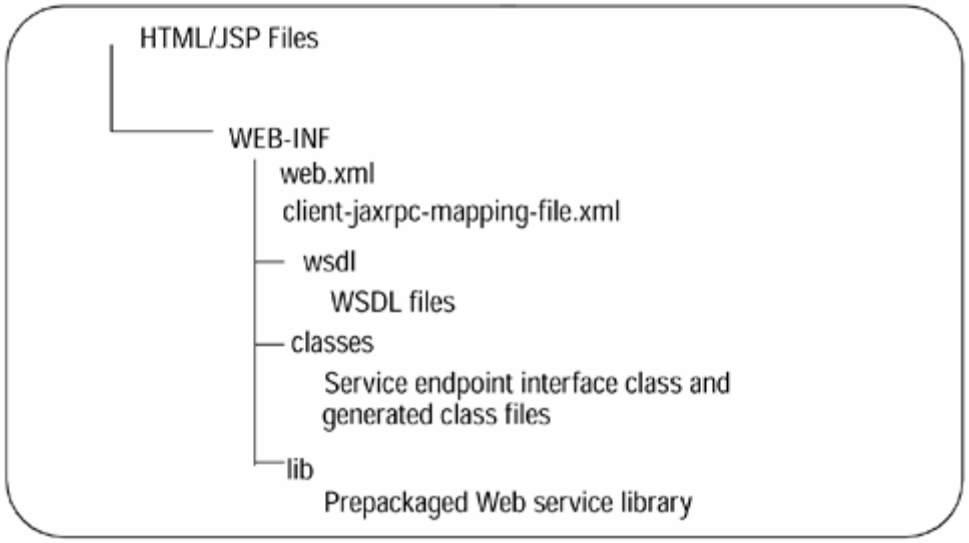
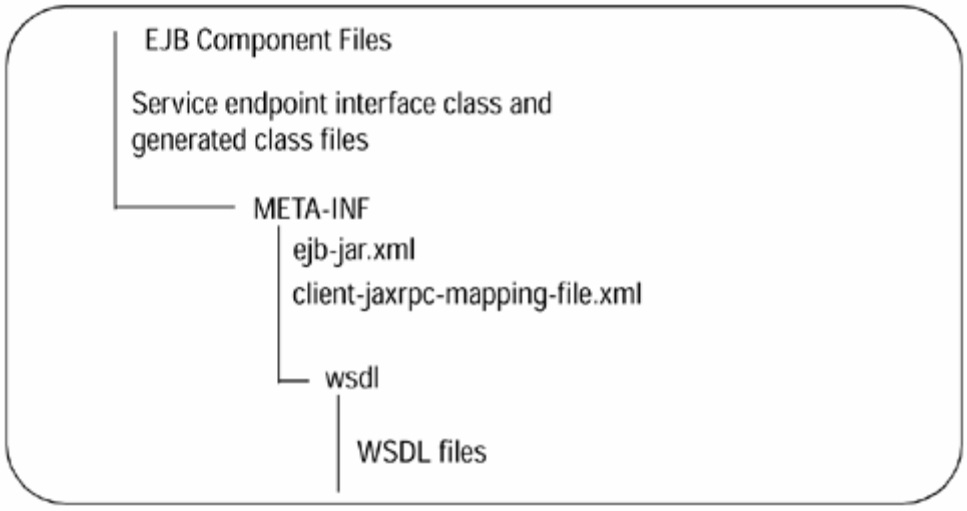


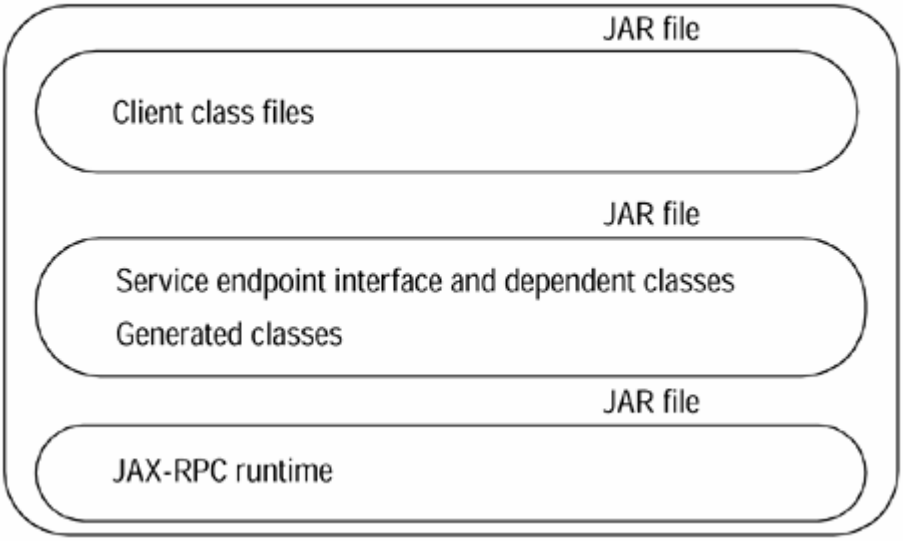
Figure 5.9. EJB Module Packaging



For Web tier clients, a `service-ref` element in the `web.xml` file contains the location of the JAX-RPC mapping file, `client-jaxrpc-mapping-file.xml`. The service endpoint interface (if provided) is either a class file in the `WEB-INF/classes` directory or it is packaged in a JAR file in the `WEB-INF/lib` directory. Generated classes are located in the same directory.

For EJB tier client components, the `service-ref` element is defined in the deployment descriptor of the `ejb-jar.xml` file and the `client-jaxrpc-mapping-file.xml` mapping file. The WSDL files are in a `META-INF/wsdl` directory. The service endpoint interface as well as generated class files are stored in the module's root directory.

Figure 5.10. Packaging a J2SE Client with Web Service Library



J2ME applications are packaged in a MIDlet format. A MIDlet is a Java Archive (JAR) file that contains class files, application resources, and a manifest file (`manifest.mf`), which contains application attributes. The `foo.jar` MIDlet file contains the client application classes and the respective artifacts generated by the J2ME Web service development tools, as well as a manifest file. A foo.jad file (A Java Archive Descriptor which is an external file similar in content to the manifest file in the jar where developer can override the attributes as defined in the manifest file) describes the `foo.jar` MIDlet. Similar to the J2EE platform, the J2ME platform with the optional Web service packages provides the resources required for Web service communication.

Enterprise Application Integration

Enterprise Information Systems = relational/legacy DB, ERP systems and mainframe transaction processing systems. The emergence of Web-based architectures and Web services adds impetus for enterprises to integrate their EISs and expose them to the Web.
A business process is a series of (often asynchronous) steps that together complete a business task or function. For example, the adventure builder enterprise has a business process for fulfilling purchase orders submitted to its order processing center. The order fulfillment business process includes such steps as validating a customer's credit card, communicating with various suppliers to fill different parts of an order, and notifying the customer of the order status at various stages of processing. Integration is often accomplished by exchanging documents, which more and more are XML documents, among business processes according to defined business rules. The different processes transform the documents by applying their individual business rules, and then they route the documents to other processes.

The J2EE platform provides a set of EIS integration technologies that address the EIS integration problem.

1. relational database integration technologies (such as JDBC, Enterprise JavaBeans technology container-managed persistence, and Java Data Objects),
2. messaging technologies (such as Java Message Service and message-driven beans),
3. EIS access technologies (particularly the J2EE Connector architecture), Web services, and XML technologies for manipulating documents.

It is not unusual to compare JDO to enterprise beans with container-managed persistence, since both provide object-relational mapping capabilities. The principal difference is that JDO maps database relationships to plain Java objects, while EJB CMP maps relationships to transactional components managed by a container. EJB CMP essentially provides a higher layer of service than JDO. Some J2EE

application servers, such as the J2EE 1.4 platform SDK, internally use JDO to implement enterprise bean container-managed persistence.
Message-driven beans are also useful when the delivery of a message should be the event initiating a workflow process or when a specific message must trigger a subsequent action.
The J2EE Connector architecture provides a standard architecture for integrating J2EE applications with existing EISs and applications, and particularly for data integration with non-relational databases. The Connector architecture enables adapters for external EISs to be plugged into the J2EE application server. Enterprise applications can use these adapters to support and manage secure, transactional, and scalable, bi-directional communication with EISs.

Although messaging systems provide many of the same EAI advantages as Web services, Web services go a step further. Principally, Web services support multiple vendors and the ability to go through firewalls using Internet standards. Web services also support a flexible XML format.
The adventure builder enterprise decides to use several layers for integration:

1. Web services as an integration layer for its supply chain.
2. Web services as an integration layer for communicating among different departments. For example, the adventure builder Web site uses a Web service to send an order to the order processing center.
3. EJB/JMS components as the integration layer with EISs. The order processing center integrates EISs within its department using JMS. Hence, the order processing center fulfills an order using JMS and EJB technologies for integrating its various EIS systems, customer relations management, billing systems, and so forth.

With Webservices approach to integration, an enterprise's EIS systems expose their functionality by implementing Web services. They make their Web service interfaces available to other applications by providing WSDL descriptions of them. In addition, the integration layer may also include XML schemas for the documents used as parameters and return values. Essentially, the **WSDL** description of the service interface and **document schemas** becomes the **integration layer**, or the point of stability.

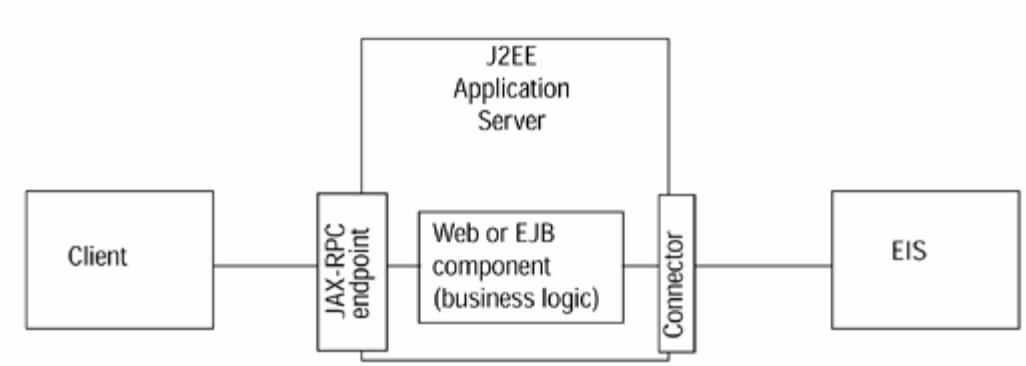
As noted, the adventure builder enterprise uses Web services for integrating its supply chain. The adventure builder architects decide, in consultation with the suppliers, on the schemas for the documents that they intend to exchange. Since their business depends on this exchange of documents—the adventure builder application sends purchase orders to various suppliers who fulfill adventure requests, and, in turn, the suppliers invoice adventure builder—the adventure builder enterprise and the suppliers need to agree on schemas that describe the content of these documents. For example, a lodging supplier might use an invoice document schema. A sample invoice document instance is shown below:

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

<Invoice xmlns="http://java.sun.com/blueprints/ns/invoice"

   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

   xsi:schemaLocation="http://java.sun.com/blueprints/ns/invoice  http://java.sun.com

  /blueprints/schemas/invoice-lodging.xsd">

   <ID>1234</ID>

   <OPCPoId>AB-j2ee-1069278832687</OPCPoId>

   <SupplierId>LODGING_INVOICE</SupplierId>

   <status>COMPLETED</status>

   <HotelId>LODG-6</HotelId>

   <HotelAddress>1234 Main Street, Sometown 12345, USA

   </HotelAddress>

<CancelPolicy>No Cancelations 24 hours prior</CancelPolicy>

</Invoice>
```


Similarly, the architects standardize on the WSDL to use for invoices when fulfilling an order. Suppliers, such as airlines, hotels, and activity providers, use these schemas and WSDL descriptions when interacting with the adventure builder enterprise. The WSDL shown below provides a single operation, `submitDocument`, which has a single **`xsd:anyType` parameter representing the invoice XML document** and which returns a single string value indicating a confirmation. The use of `xsd:anyType` enables an endpoint to receive multiple types of invoice documents with one method.

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

<definitions name="WebServiceBroker" targetNamespace= ...>

   ...

   <message name="BrokerServiceIntf_submitDocument">

      <part name="Invoice" type="xsd:anyType"/></message>

   <message name="BrokerServiceIntf_submitDocumentResponse">

      <part name="result" type="xsd:string"/></message>

   <message name="InvalidDocumentException"><part name=

          "InvalidDocumentException" element=

          "tns:InvalidDocumentException"/>

   </message>

   <portType name="BrokerServiceIntf">

      <operation name="submitDocument" parameterOrder="Invoice">

         <input message="tns:BrokerServiceIntf_submitDocument"/>

         <output message

            ="tns:BrokerServiceIntf_submitDocumentResponse"/>

         <fault name="InvalidDocumentException" message=

                        "tns:InvalidDocumentException"/>

      </operation>

   </portType>

   <binding name="BrokerServiceIntfBinding" type=

                              "tns:BrokerServiceIntf">

      ...

   </binding>

   <service name="WebServiceBroker">

      ...

   </service>

</definitions>
```

Using connectors, to plug the EIS system into the J2EE application server, allows the transactional and security contexts of a J2EE application server to carry forward to the EIS application. Plus, the EIS can use the thread pools of the application server. In conjunction with connectors, you use JAX-RPC from the J2EE application server to expose a Web service interface.



The adventure builder enterprise wants to use Web services to integrate its existing CRM system, which provides services to manage customer relations, to process orders. The department that owns the CRM module not only wants to maintain control of the software, it wants to use a generic interface that can handle user requests from multiple sources (Web site, telephone, OEM channels, and so forth). Web services are the best way to create such a generic interface, for these reasons:

1. Since Web services can provide an interface with clear, defined boundaries of responsibilities, the CRM department has the responsibility to only maintain the endpoint and publish a WSDL describing the endpoint.
2. Web services provide controlled access. Outside requests to the CRM must come in through the service interface, and the CRM department can apply its access control parameters to limit access, plus it can log each access.
3. Web services support multiple platforms. Because it does not control the hardware and software platforms other departments use, the CRM department can accommodate any platform by using Web services.
4. The current generation of Web services is best suited for applications with a limited need for transactions and security. The main purpose of the CRM system is to allow status queries on existing orders. As such, it has little need for transactions. It also has limited need for security, since all access to the module happens within the corporate firewall.

The adventure builder enterprise receives invoices using an EJB service endpoint, as well as JAX-RPC service endpoints for other functions. The adventure builder enterprise could use a small J2EE application that uses a J2EE connector to connect to the CRM system. Such an application needs to have a JAX-RPC endpoint to expose the required Web service functionality.

Consider using the built-in Web services support provided by the EIS vendors to avoid writing additional interfaces. If the webservices support is not present on the EIS then you can go the connector way (either write your own or consider the recommended connectors for the EIS) and provide a JAX-RPC service endpoint.

The adventure builder enterprise uses the strategy of using EJB and JMS for integrating applications within one department. For example, a single department owns the order processing module. Within that department, different groups handle various aspects of order processing, such as credit card payments, supply chain interactions, customer relations, and so on. In this workflow arrangement, interactions among these departmental groups are handled in a loosely coupled, asynchronous manner using JMS. When it needs to provide synchronous access, a group may use a remote enterprise bean interface. With this approach, still the application can use XML documents. The adventure builder enterprise uses the same invoice document listed above when sending a JMS message within its order processing center. The message-driven bean that receives the JMS message applies XML validation and translation logic just like any Web service endpoint.

Use DAO class when accessing an EIS via a connector: For new applications that are also J2EE applications, you use a connector to access the EIS. You can either buy an off-the-shelf connector or write your own. If you want to provide a simple isolation layer, you should consider writing a data access object, or DAO class, to hide the internal details of the EIS. A data access object encapsulates access to persistent data, such as that stored in a data management system. It decouples the user of the object from the programming mechanism for accessing the underlying data. It can also expose an easier-to-use API.

Typically, you develop a data access class for a specific business function or set of functions, then client applications use these classes. Another common use is to layer data access objects on top of connectors, or resource adapters, thus isolating applications from the details of the connector interactions. Often, each resource adapter interaction has a corresponding data access object, whose methods pass input records to the adapter as parameters and return output records to the client application as return values.

Note that when using the connector approach, the EAI application is tightly coupled with the EIS since the application directly uses the data model and functionality of the EIS. Since there is minimal layering, it also increases performance. However, it is an approach that works best when the integration problem is small in scope. Since it does not put into place a true integration architecture, this approach may limit the ability to scale the integration layer as the enterprise grows. Given these advantages and disadvantages, consider using this approach as a basic building block for other strategies.

In many cases, architects combine these various integration layers into a single integration architecture. **Web services, as they exist today, have some shortcomings:** They do not deliver the heavy-duty process integration, data transformation, and security capabilities required by many EAI scenarios. Similarly, services for transactional integrity and reliable messaging are not yet in place. Since security and transactional context propagation are critical business requirements, these are important factors to consider when using a Web services approach.

Web services are a good solution when data binding requirements are straightforward, such as mapping simple data types to Java data types. However, when it is necessary to manipulate complex relational or binary from an EIS, you may want to consider other solutions, such as using the J2EE Connector architecture, which provides a metadata facility to dynamically discover the data format in the EIS.

Operation	Connector Approach	EJB/JMS Approach	Web Services Approach
Coupling with EIS	Tight coupling. Uses EIS data model directly.	Can add a layer of abstraction in the EJB/JMS layer.	No hardware/software platform coupling. Can add multiple layers of abstractions and translations.
Transactional support	Available	Declarative, automatic context propagation	Global transaction propagation is not currently available. The endpoint implementation can use transactions for the business logic.
Supporting asynchronous operations	J2EE 1.4 platform adds asynchronous capabilities to connectors.	Message-driven beans provide an easy-to-use abstraction for receiving asynchronous events from EISs.	Currently no asynchronous support. WSDL provides a primitive mechanism for one-way calls, although the quality of service is low.
Performance	Highest	Overheads because of remote calls, and requirements of running a server.	Significant overheads because of remote calls, requirements of running a server, and of XML processing and validation.
Heterogeneous platform support	Requires that the client is programmed in Java.	Requires a J2EE application server (available on a broad range of hardware/software platforms).	Supported on a variety of hardware/software platforms.

Operation	Connector Approach	EJB/JMS Approach	Web Services Approach
Security features	Can directly integrate with the EISs security model.	Provides application server security mechanisms.	Limited. HTTPS is supported.

For relational data sources, you can either use ORM (hibernate or CMP entity beans) or create a data-holder layer using JDBC (esp. RowSet feature). The `RowSet` technology, through the `WebRowSet` feature, gives you an XML view of the data source. Its `CachedRowSet` capabilities lets you access data in a disconnected fashion, while the `FilteredRowSet` functions give you the ability to manage data in a disconnected manner, without requiring heavyweight DBMS support. By using a generic layer to hold data, you have a simpler design, since there is no real layering, and you avoid the conceptual weight of a formal data model. This approach may have better performance, particularly when access is tabular in nature (such as when selecting several attributes from all rows that match a particular condition).

To access data stored in non-relational data sources, it is best to use connectors.

A good strategy for data transformation is to use the canonical data model. Essentially, a canonical data model is a data model independent of any application. For example, a canonical model might use a standard format for dates, such as `MM/DD/YYYY`. Rather than transforming data from one application's format directly to another application's format, you transform the data from the various communicating applications to this common canonical model. You write new applications to use this common format and adapt legacy systems to the same format. By using XML to represent your canonical model, you can write various schemas that unambiguously define the data model. You can validate an XML document to ensure that it conforms to the schema of the canonical data model. Listing below shows an XML document representing invoice information. This might be the canonical model of an invoice used by the adventure builder enterprise. Since this document is published internally, all new applications requiring this data type can make use of it.

```
<?xml version="1.0" encoding="UTF-8"?>
<bpi:Invoice
   xmlns:bpi="http://java.sun.com/blueprints/ns/invoice"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/blueprints/ns/
       invoice  http://java.sun.com/blueprints/schemas/
       invoice.xsd">
   <bpi: InvoiceId>1234</bpi:InvoiceId>
   <bpi:OPCPoId>AB-j2ee-1069278832687</bpi: OPCPoId>
   <bpi:SupplierId>LODGING_INVOICE</bpi:SupplierId>
   <bpi: status>COMPLETED</bpi:status>
</bpi:Invoice>
```

It is usually a good idea to provide the schema for the canonical data model. By having the schema available, you can validate the translated documents against it and newer applications can use the schema to define their own models. Once the XML is defined, you can also use tools such as JAXB to generate the Java classes.

That is, only those components with external interfaces—Web service applications, remote enterprise beans, and so forth—expose the canonical data model. Since the external world needs (and sees) only the canonical data model, the adventure builder enterprise must transform its internal data representations—which have their own data model devised by its various EISs—to this same canonical model. The data translation between the internal and external representations can be done before the data goes from the EIS into the application server—that is, the application server internally uses the canonical data model, which is generally recommended. Or, the data translation can take place just prior to sending the data out to the external world—that is, the application server uses the various EISs' native data representations. Sometimes the business logic necessitates this latter approach because the logic needs to know the precise native format.

1. Use XSL style sheets to transform these alternate data representations, either when data comes in or when data goes out. In this approach, the application server internally uses the EIS native formats and the translation happens just before the data either goes out to the external world or comes in from the outside. For example, In adventure builder's case, various suppliers submit invoices, and each supplier may have a different representation (that is, a different format) of the invoice information. Furthermore, adventure builder's various EISs may each have a different representation of the same invoice information. A typical supplier invoice is shown:

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

<Invoice xmlns="http://java.sun.com/blueprints/ns/invoice"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation=

       "http://java.sun.com/blueprints/ns/invoice

          http://java.sun.com/blueprints/schemas/

          invoice-lodging.xsd">

   <ID>1234</ID>

   <OPCPoId>AB-j2ee-1069278832687</OPCPoId>

   <SupplierId>LODGING_INVOICE</SupplierId>

   <status>COMPLETED</status>

   <HotelId>LODG-6</HotelId>

   <HotelAddress>1234 Main Street, Sometown 12345, USA

   </HotelAddress>

<CancelPolicy>No Cancellations 24 hours prior</CancelPolicy>

</Invoice>
```

The adventure builder can convert an invoice from this supplier to its canonical data model by applying in the interaction layer of the Web service the style sheet shown below:

```
<xsl:stylesheet version='1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
    xmlns:bpi='http://java.sun.com/blueprints/ns/invoice'>

     <xsl:template match="text()"/>

     <xsl:template match="@*"/>

     <xsl:template match="bpi:Invoice">
      <bpi:Invoice xmlns:xsi=
           "http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
          "http://java.sun.com/blueprints/ns/invoice
          http://java.sun.com/blueprints/schemas/invoice.xsd">

       <xsl:apply-templates/>
```



```
      </bpi:Invoice>
    </xsl:template>

    <xsl:template match="bpi:InvoiceRef">
          <bpi:InvoiceId><xsl:value-of  select="text()"/>
  </bpi:InvoiceId>
    </xsl:template>
    <xsl:template match="bpi:OPCPoId">
        <bpi:OPCPoId><xsl:value-of select="text()"/></bpi:OPCPoId>
    </xsl:template>
    <xsl:template match="bpi:SupplierId">
         <bpi:SupplierId><xsl:value-of  select="text()"/>
         </bpi:SupplierId>
    </xsl:template>

    <xsl:template match="bpi:Status">
          <bpi:status><xsl:value-of select="text()"/></bpi:status>
    </xsl:template>

</xsl:stylesheet>
```

The adventure builder application applies the style sheet when an invoice is sent to a supplier.

2. Use a façade approach and do programmatic object mapping in the DAO layer. That is, you set up a DAO to connect an EIS to the application server. Write the DAO so that it exposes only the canonical data model and have it map any incoming data to the appropriate internal data model. Since this approach converts incoming data to the canonical form when the data arrives at the application server, the business logic internally uses the canonical data representation.

Summary for data integration is:

1. When data is in an XML document, it is easier to write XSL style sheets that do the required transformations.
2. When you access data using EJB container-managed persistence, you can either directly modify the container-managed persistent classes or write façades to do the required transformations. To use a façade, you write a Java class within which you do manual data transformation and mappings.

With **screen scraping**, you write an adapter layer that acts as an end user entering data into the mainframe application, and this adapter layer serves as the programming interface. You then write a connector that uses this programming interface to accomplish its integration actions. When resorting to screen scraping, be sure to keep in mind the limitations of the legacy system.

The **Java Business Integration (JBI)** Systems Programming Interface, based on JSR 208, extends the J2EE platform with a pluggable integration infrastructure using WSDL-based message exchange. JBI is of most interest to integration-independent software vendors (ISVs) rather than enterprise developers. JBI enables ISVs to write integration modules that support business protocols, such as **Business Process Execution Language (BPEL)**, and plug them into a J2EE application server using JBI mechanisms. As a developer, you deal with the Web services for the various JBI-based integration services provided by these ISVs.

Security

- Identity, which enables a business or service to know who you are
- Authentication, which enables you to verify that a claimed identity is genuine
- Authorization, which lets you establish who has access to specific resources
- Data integrity, which lets you establish that data has not been tampered with
- Confidentiality, which restricts access to certain messages only to intended parties
- **Nonrepudiation**, which lets you prove a user performed a certain action such that the user cannot deny it
- Auditing, which helps you to keep a record of security events

Message-level security can be useful in XML document-centric applications, since different sections of the XML document may have different security requirements or be intended for different users.

Programmatic security, which allows an application to include code that explicitly uses a security mechanism, is useful when declarative security alone cannot sufficiently express the security model of an application.

When the proof occurs in two directions—the caller and service both prove their identity to the other party—it is referred to as **mutual authentication**.

The J2EE platfrom ensures that the client's authenticated identity can be propagated along the chain of calls. It is also possible to configure a component to establish a new identity when it acts as a client in a chain of calls. When so configured, a component can change the authenticated identity from the client's identity to its own identity. The J2EE platform makes it possible to group entities into special domains, called protection domains, so that they can communicate among themselves without having to authenticate themselves. A protection domain is a logical boundary around a set of entities that are assumed or known to trust each other. Entities in such a domain need not be authenticated to one another.

The deployment descriptor holds declarations of the references made by each J2EE component to other components and to external resources. These declarations, which appear in the descriptor as `ejb-ref` elements, `resource-ref` elements, and `service-ref` elements, indicate where authentication may be necessary. The declarations are made in the scope of the calling component, and they serve to expose the application's inter-component or resource call tree. Deployers use J2EE platform tools to read these declarations, and they can then use these references to properly secure interactions between the calling and called components. The container uses this information at runtime to determine whether authentication is required and to provide the mechanisms for handling identities and credentials.

Web tier Authentication

J2EE Web containers must support three different authentication mechanisms:

1. **HTTP basic authentication**— The Web server authenticates a principal using the username and password obtained from the Web client. The username and password are included in the HTTP headers and are handled at the transport layer.
2. **Form-based authentication**— A developer can customize a form for entering username and password information, and then use this form to pass the information to the J2EE Web container. This type of authentication, geared toward Web page presentation applications, is **not used for Web services**.
3. **HTTPS mutual authentication**— Both the client and the server use digital certificates to establish their identity, and authentication occurs over a channel protected by Secure Sockets Layer.

EJB tier Authentication

1. When a client directly interacts with a Web service endpoint implemented by an enterprise bean, the EJB container establishes the authentication with the client. Since the caller is making the SOAP request over HTTP, the Web service authentication model handles authentication using similar mechanisms—basic authentication and mutual SSL—to the Web tier component use case. However, rather than use a Web component in front of the EJB component, the EJB container directly handles the authentication.
2. Optionally, you can structure an application so that a Web container component may handle authentication for an EJB component. In these cases, the application developer places a Web component in front of the enterprise bean and lets the Web component handle the authentication. Thus, the Web container vouches for the identity of those clients who want to access enterprise beans, and these clients access the beans via protected Web components.
3. A third use case entails calls made directly to an enterprise bean using RMI-IIOP. This scenario is not common for Web services since they are not accessed with RMI-IIOP. However, some Web

service endpoints, while processing a request, may need to access a remote enterprise bean component using RMI-IIOP. The Common Secure Interoperability (CSIv2) specification, which is an Object Management Group (OMG) standard supported by the J2EE platform, defines a protocol for secure RMI-IIOP invocations. Using the CSIv2-defined Security Attribute Service, client authentication is enforced just above the transport layer. The Security Attribute Service also permits identity assertion, which is an impersonation mechanism, so that an intermediate component can use an identity other than its own.

EIS tier Authentication

When integrating with enterprise information systems, J2EE components may use different security mechanisms and operate in different protection domains than the resources they access. In these cases, you can configure the calling container to manage for the calling component the authentication to the resource, a form of authentication called **container-managed resource manager sign-on.** The J2EE architecture also recognizes that some components need to directly manage the specification of caller identity and the production of a suitable authenticator. For these applications, the J2EE architecture provides a means for an application component to engage in what is called **application-managed resource manager sign-on**. Use application-managed resource manager sign-on when the ability to manipulate the authentication details is fundamental to the component's functionality.

The `resource-ref` elements of a component's deployment descriptor declare the resources used by the component. The value of the `res-auth` subelement declares whether sign-on to the resource is managed by the container or the application. With application-managed resource manager sign-on, it is possible for components that programmatically manage resource sign-on to use the `EJBContext.getCallerPrincipal` or `HttpServletRequest.getUserPrincipal` methods to obtain the identity of their caller. A component can map the identity of its caller to a new identity or authentication secret as required by the target enterprise information system. With container-managed resource manager sign-on, the container performs principal mapping on behalf of the component.

The J2EE Connector architecture offers a standard API for application-managed resource manager sign-on. This API ensures portability of components that authenticate with enterprise information systems.

Authorization

In the J2EE architecture, a container serves as an authorization boundary between the components it hosts and their callers. The authorization boundary exists inside the container's authentication boundary so that authorization is considered in the context of successful authentication. For inbound calls, the container compares security attributes from the credential associated with a component invocation to the access control rules for the target component. If the rules are satisfied, the container allows the call; otherwise, it rejects the call.

The deployment descriptor defines logical privileges called *security roles* and associates them with components. Security roles are ultimately granted permission to access components. At deployment, the security roles are mapped to identities in the operational environment to establish the capabilities of users in the runtime environment. Callers authenticated by the container as one of these identities are assigned the privilege represented by the role. The EJB container grants permission to access a method only to callers that have at least one of the privileges associated with the method. The Web container enforces authorization requirements similar to those for an EJB container. Security constraints with associated roles also protect Web resource collections, that is, a URL pattern and an associated HTTP method, such as `GET or POST`. Both the EJB and Web tiers define access control policy at deployment, rather than during application development. The access control policy can be stated in the deployment descriptors, and the policy is often adjusted at deployment to suit the operational environment.

A J2EE container decides access control before dispatching method calls to a component. In addition to these container pre-dispatch access control decisions, a developer might need to include some additional application logic for access control decisions. This logic may be based on the state of the component, the parameters of the invocation, or some other information. A component can use two methods, `EJBContext.isCallerInRole` (for use by enterprise bean code) and

`HttpServletRequest.isUserInRole` (for use by Web components), to perform additional access control within the component code.
To use these functions, a component must specify in the deployment descriptor the complete set of distinct `roleName` values used in all calls. These declarations appear in the deployment descriptor as `security-role-ref` elements. Each `security-role-ref` element links a privilege name embedded in the application as a `roleName` to a security role. Ultimately, deployment establishes the link between the privilege names embedded in the application and the security roles defined in the deployment descriptor. The link between privilege names and security roles may differ for components in the same application. Additionally, a component might want to use the identity of the caller to make decisions about access control. As noted, a component can use the methods `EJBContext.getCallerPrincipal` and `HttpServletRequest.getUserPrincipal` to obtain the calling principle. Note that containers from different vendors may represent the returned principal differently. If portability is a priority, then care should be taken when code is embedded with a dependence on a principle.

Attaching a *message signature* to a message ensures that a particular person is responsible for the content: In addition, the modification of the message by anyone other than the creator of the content is detectable by the receiver. The J2EE platform requires that containers support transport layer integrity and confidentiality mechanisms based on SSL so that security properties applied to communications are established as a side effect of creating a connection.

Enabling SSL security

The key requirements for a secure Web service interaction are authentication and establishing a secure SSL channel for the interaction. The J2EE platform supports the following authentication mechanisms for Web services using HTTPS:

1. The server authenticates itself to clients with SSL and makes its certificate available.
2. The client uses basic authentication over an SSL channel.
3. Mutual authentication with SSL, using the server certificate as well as the client certificate, so that both parties can authenticate to each other.

In a machine-to-machine interaction, trust must be established proactively, since there can be no real-time interaction with a user about whether to trust a certificate. With Web services, the individuals involved in the deployment of the Web service interaction must distribute and exchange the server certificate, and possibly the client certificate if mutual authentication is required, prior to the interaction occurrence. This is unlike the case when with a web site where a user is prompted to accept or deny a server certificate if the browser does not have the server certificate already installed.

The endpoint type determines the mechanism for declaring that a Web service endpoint requires SSL. For a Web tier endpoint (a JAX-RPC service endpoint), you indicate you are using SSL by setting to `CONFIDENTIAL` the `transport-guarantee` subelement of a `security-constraint` element in the `web.xml` deployment descriptor.

```
<web-app>
   <security-constraint>
       ...

       <web-resource-collection>
          <web-resource-name>orderService</web-resource-name>
          <url-pattern>/mywebservice</url-pattern>
          <http-method>POST</http-method>
          <http-method>GET</http-method>
       </web-resource-collection>
       <user-data-constraint>
          <transport-guarantee>CONFIDENTIAL</transport-guarantee>
       </user-data-constraint>
   </security-constraint>
</web-app>
```

Generally, for EJB endpoints a developer uses a `description` subelement of the target EJB component to indicate that the component requires SSL when deployed. Although EJB endpoints are required to support SSL and mutual authentication, the specifications have not defined a standard, portable mechanism

for enabling this. As a result, you must follow application server-specific mechanisms to indicate that an EJB endpoint requires SSL. Often, these are **application server-specific deployment descriptor** elements for EJB endpoints that are similar to the `web.xml` elements for Web tier endpoints.

Specifying Mutual Authentication

For Web tier endpoints, to enable mutual authentication, you first specify a secure transport (see the above) and then, in the same deployment descriptor, set the `auth-method` element to `CLIENT-CERT`.

```
<login-config>
   <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

The combination of the two settings—`CONFIDENTIAL` for `transport-guarantee` and `CLIENT-CERT` for `auth-method`—enables mutual authentication. When set to these values, the containers for the client and the target service both provide digital certificates sufficient to authenticate each other. (These digital certificates contain client-specific identifying information.)

For EJB endpoints, similar setting has to be done in the application-server specific deployment descriptor.

Specifying Basic and Hybrid Authentication

With basic authentication, a Web service endpoint requires a client to authenticate itself with a username and password. For a Web tier (JAX-RPC) service endpoint, set the `auth-method` element to `BASIC` for the login configuration (`login-config`) element in the `web.xml` deployment descriptor.

```
<login-config>
       <auth-method>BASIC</auth-method>
       <realm-name>some_realm_name</realm-name>
</login-config>
```

For a Web service with an EJB endpoint, you use the application server-specific mechanisms to require basic authentication.

A Web service may also require hybrid authentication, which is when a client authenticates with basic authentication and SSL is the transport. The client authenticates with a username and password, the server authenticates with its digital certificate, and all of this occurs over a HTTPS connection. Hybrid authentication compensates for HTTP basic authentication's inability to protect passwords for confidentiality. You can enable hybrid authentication by setting the transport to use the confidentiality mechanism of HTTPS and setting the authentication of the client to use basic authentication.

The security functionality specified by the WS-I Basic Profile 1.0 only requires that Web services using HTTPS have **`https`** in the URI of the location attribute of the `address` element in its `wsdl:port` description.

```
<service name ="SomeService">

<port name="SomeServicePort" binding="tns:SomeServiceBinding">

<soap:address location="https://myhostname:7000/

   adventurebuilder/opc/getOrderDetails"/>

</port>

</service>
```

Since current WSDL documents have no standard mechanism to indicate whether an endpoint requires basic or mutual authentication, such information needs to be made available through service-level agreements between the client and endpoint. You can:

1. It is recommended that you list security assumptions and requirements in the description elements that are part of a service component's deployment descriptor.
2. In addition, have available for endpoint developers a separate document that describes the security policy for an endpoint. In this document, clearly describe the information needed by a client.

You can use the Java Authentication and Authorization Service (JAAS), along with tools such as `keytool`, to manage certificates and other security artifacts. As just noted, you can also include the JAX-

RPC runtime, then use its mechanisms to set up username and password properties in the appropriate stubs and make calls to the Web service.

Client Programming Model

The J2EE container provides support so that J2EE components, such as servlets and enterprise beans, can have secure interactions when they act as clients of Web service endpoints. The container provides this support regardless of whether or not the accessed Web service endpoint is based on Java.

1. The first step for a client is to discover the security policy of the target endpoint. Since the WSDL document may not describe all security, discovering the target endpoint's security policy is specific to each situation.
2. Once you know the client's security requirements for interacting with the service, you can set up the client component environment to make available the appropriate artifacts. For example, if the Web service endpoint requires basic authentication, the calling client container places the username and password identifying information in the HTTP headers.
3. For HTTP basic authentication, application server-specific mechanisms, such as additional deployment descriptor elements, are used to set the client username and password. These vendor-specific deployment descriptors may statically define at deployment the username and password needed for basic authentication. However, at runtime this username and identifier combination may have no relation to the principal associated with the calling component. When the JAX-RPC call is made, the container puts the username and password values into the HTTP header. Keep in mind that the J2EE specifications recommend against using programmatic JAX-RPC APIs to set the username and password properties on stubs for J2EE components. Thus, J2EE application servers are not required to support components programmatically setting these identifier values.
4. In other words, an enterprise bean or servlet component that interacts with a Web service requiring mutual authentication must, at deployment (in application server specific DD), make the appropriate digital certificates available to the component's host container. The client's container can then use these certificates when the component actually places the call to the service.
5. Once the environment is set, a J2EE component can make a secure call on a service endpoint in the same way that it ordinarily calls a Web service—it looks up the service using JNDI, sets any necessary parameters, and makes the call. The J2EE container not only manages the HTTPS transport, it handles the authentication for the call using the digital certificate or the values specified in the deployment descriptor.

Web service endpoints and other components can be clients of other Web services and J2EE components. Any given endpoint may be in a chain of calls between components and Web service endpoints. Also, non-Web service J2EE components can make calls to Web services. Each call between components and endpoints may have an identity associated with it, and this **identity may need to be propagated**. There are two cases of identity propagation, differentiated by the target of the call.

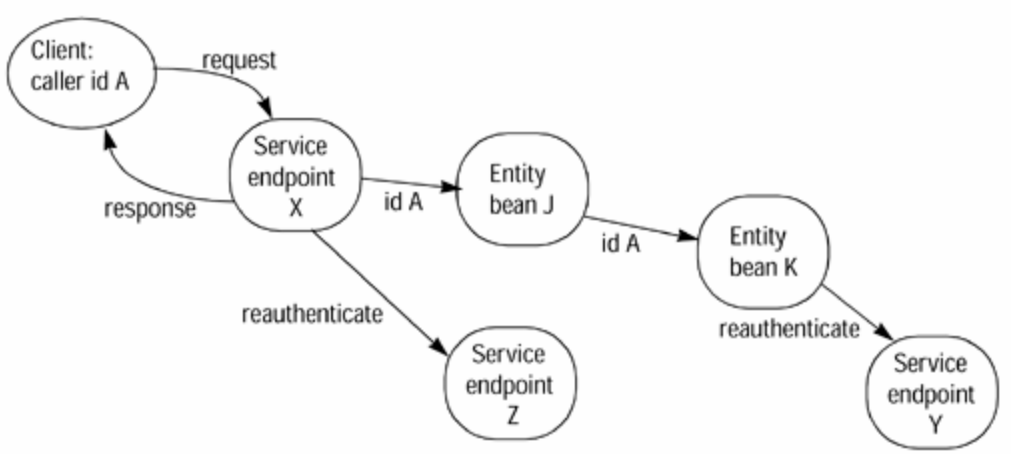
1. **Propagating identity to Non-Webservice components:** All J2EE components have an invocation identity, established by the container, that identifies them when they call other J2EE components. The container establishes this invocation identity using either the `run-as(role-name)` or `use-caller-identity` identity selection policy, both defined in the deployment descriptor. The container then uses either the calling component's identity (if the policy is to use the `use-caller-identity`) or, for `run-as(role-name)`, a static identity previously designated at deployment from the principal identities mapped to the named security role. If you want to hold callers accountable for their actions, you should associate a `use-caller-identity` policy with component callers. Using the `run-as(role-name)` identity selection policy does not maintain the chain of traceability and may be used to afford the caller with the privileges of the component. If `run-as` is not explicitly specified, the **`use-caller-identity`** policy is assumed.

   In ejb-jar.xml (for a EJB component):

```
<enterprise-beans>
   <entity>
      <security-identity>
         <use-caller-identity/>
      </security-identity>
      ...
```

```
   </entity>

   <session>
      <security-identity>
         <run-as>
            <role-name> guest </role-name>
         </run-as>
      </security-identity>
      ...
   </session>

   ...
</enterprise-beans>

In web.xml (for a web component):
<web-app>
   <servlet>
      <run-as>
         <role-name> guest </role-name>
      </run-as>

      ...
   </servlet>
   ...
</web-app>
```

2. **Propagating identity to a Webservice:** Recall that a protection domain establishes an authentication boundary around a set of entities that are assumed to trust each other. Entities within this boundary can safely communicate with each other without authenticating themselves. Authentication is only required when the boundary is crossed. However, Web services are considered outside of any protection domain. Since Web service calls are likely to cross protection domains, identity propagation mechanisms (such as `run-as` and `use_caller_identity`) and security context are not useful and are not propagated to service endpoints. When a J2EE component acting as a Web service client specifies the `run-as` identity or the `use-caller-identity`, the container applies that identity only to the component's interactions with non-Web service components, such as enterprise beans.
   For the client making calls to a service that requires authentication, the client container provides the necessary artifacts, whether username and password for basic authentication or a digital certificate for mutual authentication. The container of the target Web service establishes the identity of calls to its service endpoint. The Web service bases this identity on the mapping principals designated by when the service was deployed, which may be based on either the client's username and password identity or the digital certificate attributes supplied by the client's container. However, since no standard mechanism exists for a target Web service to map an authenticated client to the identity of a component, each application server handles this mapping differently.
   As shown below, the initial client makes a request of Web service endpoint `X`. To fulfill the request, endpoint `X` makes a call on entity bean `J`, which in turn invokes a method on entity bean `K`. The client caller identifier `A` propagates from the endpoint through both entity beans. However, when entity bean `K` calls a method on service endpoint `Y`, since the Web service is not in the same protection domain, reauthentication must occur. Similarly, when endpoint `X` calls endpoint `Z`, the caller identifier cannot be propagated.



Applications can also use programmatic APIs to check client identity, and use that client identity to make identity decisions. For example, a Web tier endpoint, as well as other Web components, can use the `getUserPrincipal` method on the `HttpServletRequest` interface. An EJB endpoint, just like other enterprise bean components, can use the `EJBContext` method `getCallerPrincipal`. An application can use these methods to obtain information about the caller and then pass that information to business logic or use it to perform custom security checks.

Example code for enabling SSL on client:
You first need to import the server side certificate as trusted certificate into your applications keystore:
keytool –import –trustcacerts –alias verisign –file .\verisign.cer
The above command imports verisign primary class 2 ca certificate as a trusted root certificate in $JAVA_HOME/lib/security/cacerts keystore. The command prompts for password for the keystore. For cacerts it is “changeit” by default. Once the certificate is imported as trusted root into the client side JVM then one can use the following code to access the webservice over https. The code below uses Basic authentication with SSL for security.

```
import java.io.StringBufferInputStream;
import java.io.StringWriter;
import org.apache.axis.encoding.SerializationContext;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.message.SOAPBodyElement;
import org.apache.axis.client.ServiceClient;
import org.apache.axis.client.Transport;
import org.apache.axis.transport.http.HTTPTransport;
import org.apache.axis.Message;
import org.apache.axis.MessageContext;
import org.apache.axis.encoding.ServiceDescription;

final public class DoOrder {
  final String url;
  final Strint storetype;
  final Strint keystore;
  final String storepass;
  final String uid;
  final String password;

  public DoOrder(String url,
          Strint storetype,
          Strint keystore,
          String storepass,
          String uid,
          String password)
  {
    this.url = url;
    this.storetype = storetype;
    this.keystore = keystore;
    this.storepass = storepass;
```



```
    this.uid = uid;
    this.password = password;
  }

  public synchronized static String invoke(String xml) throws Exception {
        String[][] props = {
        { "javax.net.ssl.trustStore", keystore, },
        { "javax.net.ssl.keyStore", keystore, },
        { "javax.net.ssl.keyStorePassword", storepass, },
        { "javax.net.ssl.keyStoreType", storetype, },
        };
        for (int i = 0; i < props.length; i++)
        System.getProperties().setProperty(props[i][0], props[i][1]);

        ServiceClient client =  new ServiceClient(new HTTPTransport(url, "PO"));
        client.set(MessageContext.USERID, uid);
        client.set(MessageContext.PASSWORD, password);
        client.setRequestMessage(new Message(new StringBufferInputStream(xml), true));
        client.invoke();

        Message outMsg = client.getMessageContext().getResponseMessage();
        ServiceDescription svc = new ServiceDescription("doOrder", false);
        client.getMessageContext().setServiceDescription(svc);

        SOAPEnvelope envelope = outMsg.getAsSOAPEnvelope();
        SOAPBodyElement body = envelope.getFirstBody();
        StringWriter writer = new StringWriter();
        SerializationContext ctx = new SerializationContext(writer,
                                    client.getMessageContext());
        body.output(ctx);
        return writer.toString();
  }
}
```

To enable SSL on Tomcat server: you need to add the following to the $CATALINA_HOME/conf/server.xml file:

```
<Connector className="org.apache.tomcat.service.PoolTcpConnector">
  <Parameter name="handler"
   value="org.apache.tomcat.service.http.HttpConnectionHandler"/>
  <Parameter name="port" value="8443"/>
  <Parameter name="socketFactory"
   value="org.apache.tomcat.net.SSLSocketFactory"/>
  <Parameter name="keystore"
   value="c:\ws-book\SkatesTown.ks" />
  <Parameter name="keypass" value="wsbookexample"/>
  <Parameter name="clientAuth" value="false"/>
</Connector>
```

Keystore value can be cacerts also and the password changeit if you use the default cacerts keystore for your server side JVM.

When a Web service is called—and the calling client has been authenticated and its identity established—the container has the capability to check that the calling principal is authorized to access this service endpoint. A Web service is also free to leave its resources unprotected so that anyone can access its service. So, the authorization mechanisms for Web service endpoints are the same as for other components in the J2EE platform.

The tier on which your endpoint resides determines how you specify and configure access control. In general, to enable access control you specify a role and the resource you want protected. Components in both tiers specify a role in the same manner, using the `security-role` element as shown below. With Web tier endpoint components, access control entails specifying a URL pattern that determines the set of restricted resources. For EJB tier endpoints, you specify access control at the method level, and you can group together a set of method names that you want protected.

```
<security-role>

   <role-name>customer</role-name>

</security-role>
```

Your Web service access control policy may influence whether you implement the service as a Web tier or an EJB tier endpoint. For Web tier components, the granularity of security is specific to the resource and based on the URL for the Web resource. For EJB tier components, security granularity is at the method level, which is typically a finer-grained level of control.

To handle a service (on the web tier) with an interface containing multiple methods and different access policies, consider creating separate Web services where each service handles a different set of authorization requirements. OR You have more flexibility if you implement the same Web service that has an interface containing multiple methods with an EJB endpoint. By using an EJB endpoint, you can set different authorization requirements for each method.

```
public interface OrderingService extends java.rmi.Remote {
   public Details getCatalogInfo(ItemType someItem) throws java.rmi.RemoteException;
   public Details submitOrder(purchaseOrder po) throws java.rmi.RemoteException;
   public void updateCatalog(ItemType someItem) throws java.rmi.RemoteException;
}
```

For a service endpoint interface such as the above, you might want to permit the following: Any client can browse the catalog of items available for sale, only authorized customers—for example, those clients who have set up accounts—can place orders, and only administrators can alter the catalog data.

```
<web-app>
....
<security-constraint>
   <web-resource-collection>
      <web-resource-name>orderService</web-resource-name>
      <url-pattern>/mywebservice</url-pattern>
      <http-method>POST</http-method>
      <http-method>GET</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>customer</role-name>
   </auth-constraint>
</security-constraint>
...
<login-config>
   ...choose either basic or client(for mutual authentication)
</login-config>
<security-role>
   <role-name>customer</role-name>
</security-role>
</web-app>
```

The descriptor specifies that only clients acting in the role of `customer` can access the URL `/mywebservice`. Note that this URL maps to all the methods in the service endpoint interface. Hence, all methods have the same access control.
Omitting authentication rules allows unauthenticated users to access Web components.

The EJB deployment descriptors define security roles for an enterprise bean. These descriptors also specify, via the `method-permission` elements, the methods of a bean's home, component, and Web service endpoint interfaces that each security role is allowed to invoke.

```
<method-permission>
   <role-name>customer</role-name> <!—or use <unchecked/>→
   <method>
      <ejb-name>PurchaseOrder</ejb-name>
      <method-intf>ServiceEndpoint</method-intf>
      <method-name>submitOrder</method-name>
   </method>
</method-permission>
```

The example specifies that the method `submitOrder`, which occurs on an interface of an enterprise bean Web service endpoint, requires that a caller belonging to the `customer` role must have authenticated to be granted access to the method. It is possible to further qualify method specifications so as to identify methods with overloaded names by parameter signature or to refer to methods of a specific interface of the enterprise bean. For example, you can specify that all methods of all interfaces (that is, remote, home, local,

local home, and service) for a bean require authorization by using an asterisk (*) for the value in the method-name tag.
Use the unchecked element in the method-permission element to indicate that no authorization check is required.
In addition to defining authorization policy in the method-permission elements, you may also add method specifications to the exclude-list. Doing so denies access to these methods independent of caller identity and whether the methods are the subject of a method-permission element.

```
<exclude-list>
   <method>
      <ejb-name>SpecialOrder</ejb-name>
      <method-name>*</method-name>
   </method>
   <method>
   ...
</exclude-list>
```

Since a web.xml file can have only one type of authentication associated with its login configuration, you cannot put endpoints that require different authentication in a single .war file. Instead, group endpoints into .war files based on the type of client authentication they require. Because the J2EE platform permits multiple .war files in a single .ear file, you can put these .war files into the application .ear file.

The WSDL file is required to publish only a Web service's HTTPS URL. It has no standard annotation describing whether the service endpoint requires basic or mutual authentication. Use the description elements of the deployment descriptor to make known the security requirements of your endpoints.

Consider using a "guarding" component between the interaction and processing layers. Set up an application accessor component with security attributes and place it in front of a set of components that require protection. Then, allow access to that set of components only through the guarding or front component. A guarding component can make application security more manageable by centralizing security access to a set of components in a single component.

Message Level Webservice Security

Message-level security, which applies to XML documents sent as SOAP messages, makes security part of the message itself by embedding all required security information in a message's SOAP header. In addition, message-level security can apply security mechanisms, such as encryption and digital signature, to the data in the message itself.

Message-level security technology lets you embed into the message itself a range of security mechanisms, such as identity and security tokens and certificates, and message encryption and signature mechanisms. The technology associates this security information with the message and can process and apply the specified security mechanisms. Message-level security uses encryption and it uses a digital signature to bind the claims—the identity attributes—from a security token to message content.

The duration of protection using HTTPS is the lifetime of the message on the wire at the transport layer. Message-level security not only persists beyond the transport layer, it lasts for as long as the XML content is perceived as a SOAP message. The duration of protection for message-level security is the lifetime of the SOAP message, and this can span the transport boundary. Because security is part of the SOAP message, applications can support Web service interactions that require maintaining protection through out the entire system or into the application layer. Having security as part of the message also makes it possible to persist both the message data and its security information. For example, perhaps to prove that a message was sent, an application may need to persist the message data and the digital signature bound to the message. Or, to protect against internal threats, an application may need to keep data in a SOAP message confidential, even to the application layer. HTTPS, since it only protects a message during transport, cannot give the application layer this encryption protection for the data.

A Web service interaction that uses HTTPS supports peer entity authentication, because the interaction covers just the connection between two peers. Message-level security supports data origin authentication, since its security is tied to the SOAP message itself rather than the transport mechanism.

SSL, relying on peer entity authentication, does not support end-to-end multi-hop message exchange as it requires that each participant decrypt each received message, then encrypt the same message before transmitting it to the next participant in the workflow.

Web service scenarios that pass messages to multiple participants lend themselves to using message-level security. Since message-level security is based on data origin authentication, an application that passes messages to numerous intermediary participants on the way to the target recipient can verify, at each intermediary point and at the final target recipient, that the initial message creator identity associated with the message is as claimed. In other words, the initial message content creator's identity moves with the message through the chain of recipients.

If you use HTTPS, you essentially encrypt the entire SOAP message since HTTPS encrypts everything passed on the wire. If you use message-level security, you can encrypt just a portion of the XML document, then send a SOAP message that is partially encrypted. Since encryption is computationally intensive, encrypting an entire document (particularly a large one) can impact performance.
You could also apply different security mechanisms, such as different encryption algorithms, to various parts of a message, ensuring that only intended recipients can decrypt those parts of the message. Finer-grained control also supports intermediaries whose processing requires access to a small part of the message data, such as intermediaries that route messages to appropriate recipients.

Some message level security specs are:

XML Digital Signature	includes procedures for computing and verifying signatures
XML Encryption	for encrypting parts of an XML document
Web service Security Assertions	which is based on Security and Assertions Markup Language (SAML), is used for exchanging authentication and authorization information for XML documents
Web service Message Security Java APIs	which enable applications to construct secure SOAP message exchanges

Example: Putting XML Digital Signature on a purchase order document.
A client first embeds the digital signature into the XML message, signing the message before sending it. The signature, which is an X.509 certificate, is embedded into the purchase order XML document along with other information.

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
   xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext"
   xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">

   <S:Header>
      <wsse:Security>
         <!—Base 64 encoded X.509 certificate used to sign this message. -->
         <wsse:BinarySecurityToken
            ValueType="wsse:X509v3"
            EncodingType="wsse:Base64Binary" wsu:>
            MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
         </wsse:BinarySecurityToken>
          <!—signature algorithms being used. -->
         <ds:Signature>
            <ds:SignedInfo>
               <ds:CanonicalizationMethod Algorithm=
                           "http://www.w3.org/2001/10/xml-exc-c14n#"/>
               <ds:SignatureMethod Algorithm=
                           "http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
              <!—references the signed message body elements. -->
               <ds:Reference URI="#myBody">
```



```
                  <ds:Transforms>
                     <ds:Transform Algorithm=
                  "http://www.w3.org/2001/10/xml-exc-c14n#"/>
                  </ds:Transforms>
                  <ds:DigestMethod Algorithm=
                     "http://www.w3.org/2000/09/xmldsig#sha1"/>
                  <ds:DigestValue>
                     EULddytSo1...
                  </ds:DigestValue>
               </ds:Reference>
            </ds:SignedInfo>
             <!—Value of signature after signing the message body -->
            <ds:SignatureValue>
               BL8jdfToEb11/vXcMZNNjPOV...
            </ds:SignatureValue>
             <!— public Key(s) used to sign the message. -->
            <ds:KeyInfo>
               <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#X509Token"/>
               </wsse:SecurityTokenReference>
            </ds:KeyInfo>
         </ds:Signature>
      </wsse:Security>
   </S:Header>
   <S:Body wsu:>
      <myPO:PurchaseDetails xmlns:myPO=
         "http://www.someURL.com/purchaseOrder">
         some message content here ...
      </myPO:PurchaseDetails>
   </S:Body>

</S:Envelope>
```

For more on the use of XML Digital Signature related Java APIs refer an article at: http://java.sun.com/developer/technicalArticles/xml/dig_signatures/.

Handlers intercept all requests and responses that pass through a Web service endpoint, providing access to the actual SOAP message exchanged as part of the Web service request and response. Handlers let you apply different logic for service requests, responses, and faults. To do so, you add the appropriate code to the handler methods `handleRequest`, `handleResponse`, and `handleFault`. You can use handlers to apply message-level security to messages exchanged as part of your service. Since they are configurable on both the client and the endpoint, you can customize handlers to apply security services at both the client and service sides. You use the SAAJ API to inspect and manipulate raw SOAP messages. SAAJ also gives you a compound message view capability that lets you examine MIME-based attachments. With SAAJ, you can also embed the digital signature information into the XML document and add the necessary security information to the header and message.

If you choose to use any of the J2EE declarative or programmatic security mechanisms along with JAX-RPC handlers, keep in mind the order in which the security constraints are enforced:

1. Container applies the declarative security mechanisms first.
2. handlers run and apply their checks.
3. J2EE programmatic security mechanisms run after the handler checks.

Application Architecture

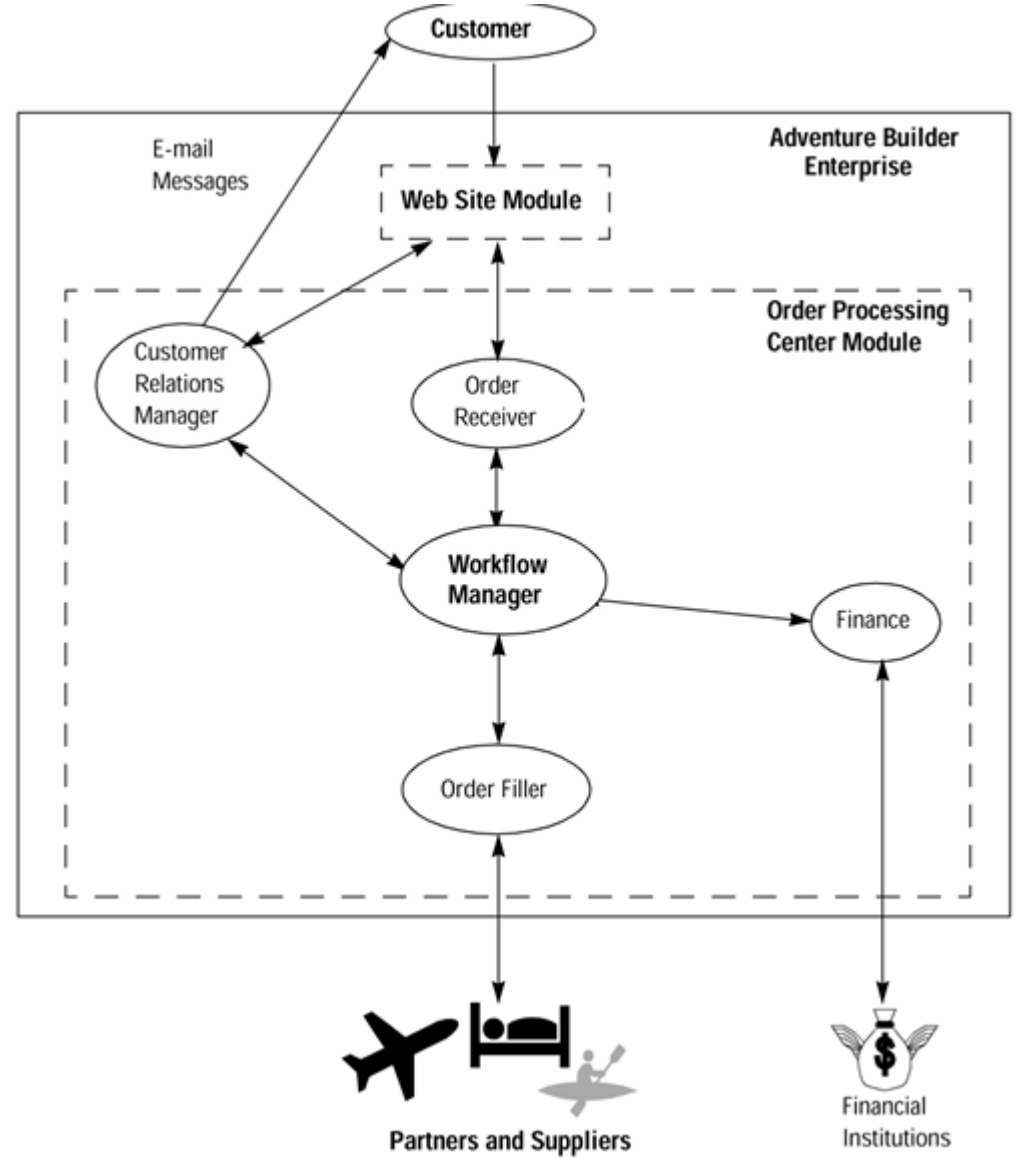
Functional Specification

Adventure builder enterprise provides **customers** with a catalog of adventure packages, accommodations, and transportation options. From a browser, customers select from these options to build a vacation, such as an Adventure on Mt Kilimanjaro. Building a vacation includes selecting accommodations, mode of transport, and adventure activities, such as mountaineering, mountain biking to a hidden waterfall, and hiking the mountain. After assembling the vacation package, the customer clicks the submit order option. The customer Web site builds a purchase order and sends it to the **order processing center (OPC)**. The order processing center, which is responsible for fulfilling the order, interacts with its internal departments

and the **external partners** and suppliers (such as airlines, hotels, and activity providers, supply the services or components of a vacation) to complete the order. In essence, the adventure builder enterprise consists of a front-end **customer Web site**, which provides a face to its customers, and a back-end order processing center, which handles the order fulfillment processing (Adventure Builder).

The order processing center module needs to perform the following functions:

- **Receive customer orders from the customer Web site** and process these orders
- Coordinate activities according to the **business workflow rules**
- **Track an order's progress** through the steps of the order fulfillment process
- **Manage customer relations**, including tracking customer preferences and updating customers on the status of an order. This includes sending formatted e-mails to customers about order status.
- **Manage financial information**, including verifying and obtaining approval for payment
- **Interact with business partners**—airlines, hotels, and adventure or activity providers—to fulfill a customer's adventure package.
- **Provide and maintain a catalog of adventures and allow customers to place orders.** The order processing center catalog manager needs to interact with external suppliers and also keep its customer offerings up to date.

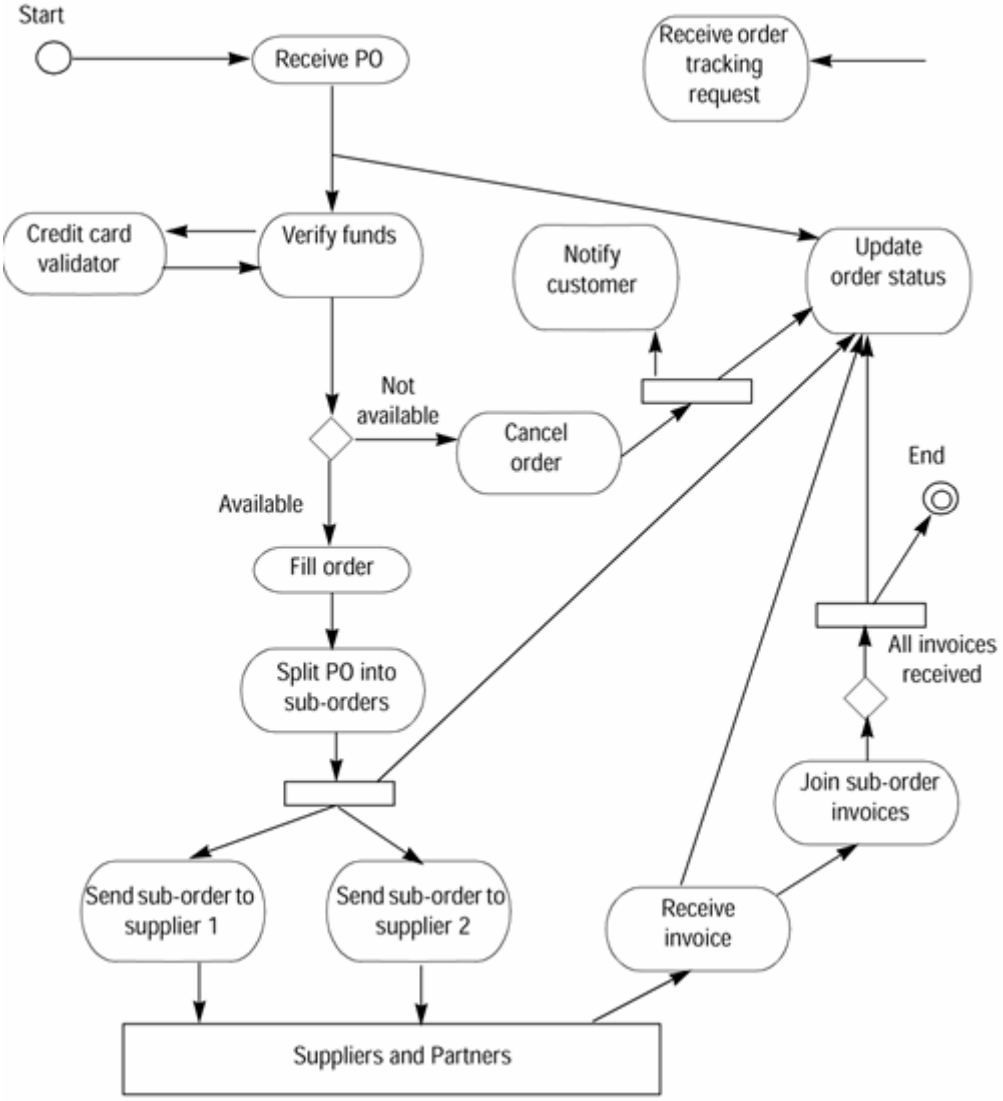


- Order Receiver— Accepts purchase orders from the customer Web site. This sub-module starts the order fulfillment processing in the back end of the enterprise. Each order is identified by a unique ID.
- Workflow Manager— Enforces the process flow rules within the adventure builder application and tracks the state of each order during its processing. The workflow manager interacts with the internal departments and coordinates all the participants in the business process.
- Finance— Interacts with external banks and credit card services that collect payments, and manages financial information.
- Customer Relations Manager (CRM)— Provides order tracking information to the customer Web site application. This sub-module also sends formatted e-mail notices about order status directly to clients.
- Order Filler— Exchanges messages with the various external suppliers to fulfill purchase orders. Messages include supplier purchase orders and invoices

The process flow is as:

- A purchase order flows from the customer Web site to the order processing center.
- The finance department and the credit card services exchange credit card payment requests and verifications.
- The order processing center order filler and the external suppliers exchange supplier purchase orders, invoices, and other documents.
- The workflow manager and other departments or sub-modules within the order processing center exchange internal messages.

When it receives an order for an adventure package from a customer, the order processing center persists a purchase order. Before proceeding, it verifies that the customer has the available funds or credit for the purchase. If not, the order processing center cancels the order, notifies the customer, and updates the order status. Otherwise, it proceeds to fulfill the order, which entails breaking the entire adventure package order into sub-orders (such as a sub-order for a hotel room, another sub-order for airline reservations, and so forth). It sends the sub-orders to the appropriate suppliers, who fulfill these portions of the order and return invoices. The order processing center then joins these invoices for sub-orders so that the entire order is invoiced. The order processing center also updates the status of the order and notifies the customer.



OPC Architecture and Design

Message exchanges that occur during the order fulfillment process:

- Clients or customers communicate with the customer Web site via a Web browser.
- Once a customer places an order, the customer Web site communicates a purchase order to the order processing center.
- The customer Web site also follows up with the order processing center to ascertain the current status of an order.
- The order processing center sends credit card processing details for verification to the credit card service.
- The order processing center sends purchase orders to suppliers and receives invoices from these suppliers.

Design Choices related to Communication:

1. For adventure builder, we chose to use Web services as the technology for communication between the order processing center and entities external to the order processing center, such as for the exchange of purchase orders and invoices with external partners and suppliers. We made this choice because we are primarily interested in achieving the greatest degree of interoperability among all modules and among all types of clients.
2. Within the order processing center module, the communication among sub-modules primarily uses JMS. Since the order processing center controls the entire environment within the module, communication can be more tightly coupled. Most of the communication is between the order processing center workflow manager and the various department sub-modules, and all sub-modules are within this environment. Given the control over the environment and that most communication is asynchronous, using JMS is appropriate.
3. Web services are a good choice for communication external to the enterprise, but they can also be applied to internal situations. The order processing center and customer Web sites are within the same enterprise, albeit different departments, and they use Web services rather than JMS. The order processing center module makes its services available as a Web service to the customer Web site—that is, the Web site uses the order processing center's Web service to fulfill an order. There are advantages to this implementation. For one, the Web site for the order processing center module may be hosted outside the firewall in a demilitarized zone (DMZ), even though the order processing center module itself is always inside the firewall. The Web site may conveniently use the HTTP port available on the firewall to communicate with the order processing center.

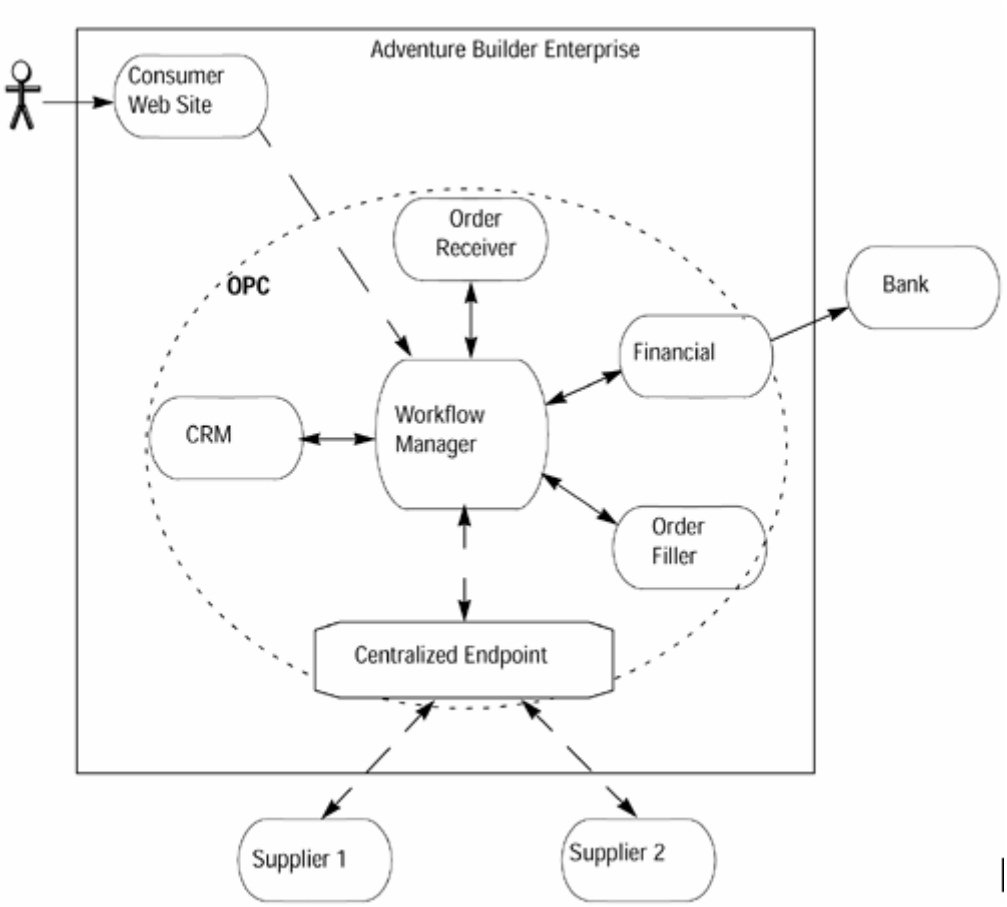
Since the system must handle multiple message types, we use XML as the message payload format. Primarily, we use XML documents for communication on the edges on the enterprise, especially for messages exchanged with trading partners.
But since we also use JMS internally within the order processing center, we had to choose to either pass XML documents between sub-modules or to convert documents to Java objects. Since we are modelling a document-oriented system, and since the sub-modules represent different departments, we decided to pass XML documents among them.

Communication Architecture

- The order processing center uses a workflow manager, which contains all process rules and flow logic, to coordinate processing of the orders. The workflow manager also keeps track of the current state of each order while it is processed.
- Each participant knows its business functionality and how to execute its own step in the workflow; it does not need to know other steps in the process.
- Generally, each order processing center participant (such as the order filler) has a JMS resource and receives XML documents. The JMS resource is the integration point between the manager and each work module.
- A centralized endpoint manages the interactions between the suppliers and the order processing center. Rather than having a separate Web service endpoint for each message, all interactions

between the order processing center and suppliers are grouped together and handled by a common endpoint and interaction layer.

- Other parts of the order processing center may act directly as clients to Web services without using the centralized endpoint. For example, the finance module uses the credit card service.



The order processing center internally uses a hub-and-spoke model where the workflow manager coordinates the participants in the order fulfillment process. Each participant receives an XML document from the workflow manager, processes the XML document by applying its portion of the business logic, and returns the result to the workflow manager. The workflow manager determines the next step to execute in the workflow and dispatches the appropriate request, by sending an XML document, to the proper participant. Each individual participant executes their part of the business functionality and has no need to know the overall process of the workflow.

Endpoint Design Issues

Three of adventure builder's Web service interfaces are:

1. The Web service interface between the order processing center and the customer Web site to receive customer orders - Because the same organization develops and controls the customer Web site and the order processing center, it is easy to propagate to the Web site module changes to the Web service interface that the order processing center provides to the Web site. Since we have control over all parties that have access to this Web service interface, the issue of stability of the interface is less important. Because it is easier, we opted to use the Java-to-WSDL approach to design and implement the Web service interfaces between the Web site and the order processing center. – **Order Receiver Webservice.**

2. The order tracking Web service, which handles client requests for order status – This is the **CRM module**.
3. The Web service interface between the order processing center and the external partners or suppliers - The Web service interaction between the order processing center and the suppliers is a business-to-business interaction. Any order processing center changes to the interface affect many different suppliers. A stable interface also enables the participants to evolve in different ways. It is of paramount importance with such interactions that the interface between the communicating entities be stable. Exchanged business documents should have an agreed-upon format and content, and for this it is best to use documents with existing standard schemas. For these reasons we used the WSDL-to-Java approach for the Web service interface between the order processing center and its suppliers. – This is the **Centralized Service Endpoint.**

The choice of endpoint type is primarily based on whether or not the business logic uses enterprise beans. If the business logic does use enterprise beans, it is often convenient to use an EJB service endpoint. If the application is primarily Web tier based, then it is best to use a JAX-RPC service endpoint. Since the order processing center is implemented with a set of enterprise beans, it makes sense to implement these Web services as EJB service endpoints. Using a JAX-RPC service endpoint would introduce a Web layer that acts merely as a proxy, directing requests from the Web tier to the EJB tier and adding no value. Advantages of using EJB service endpoints are:

1. You can declare an EJB method to be transactional, resulting in the method body executing within a transaction. We found this useful since the order processing center requires transactional access to the database.
2. An EJB service endpoint also allows method-based security controls, which is useful if you want certain methods to be publicly accessible but other methods to be restricted to authenticated users. You need to do additional work to get the equivalent capabilities in a JAX-RPC service endpoint (like either divide into multiple webservices based on the access required or within a single webservice implement security programmatically).

Because we want our application to perform well, we use coarse-grained interfaces for the Web services. When adding a Web service interface to existing applications, you want to identify the Web service interfaces that can be exposed. To do so, it is useful to look at the session bean-based application façades. Generally, it is not a good idea to convert each such application façade session bean into a Web service interface. You should design Web services to be more coarse grained than individual session beans. A good way to achieve a coarse-grained interface is to design the Web service around the documents it handles, since documents are naturally coarse grained. The adventure builder application applies these principles by exposing Web services that primarily exchange well-defined documents—purchase orders and invoices. These Web services may call multiple session bean methods to implement the services' business logic.

Parameters may be passed as either JAX-RPC value types or XML documents, which are represented as `SOAPElement` objects in the service interface. Passing XML documents raises different issues than passing Java objects. XML documents involve a certain amount of complexity, since the requestor must build the document containing the request and the receiver must validate and parse the document before processing the request. However, XML documents are better for addressing the business-to-business transaction considerations. We used object parameters for the service interfaces between the customer Web site and the order processing center applications. Moreover, the parameters map to specific document types that can be mapped easily to their equivalent JAX-RPC value types. Using these JAX-RPC value types eliminates the complexity of creating and manipulating XML documents.

```
public interface OrderTrackingIntf extends Remote {
    public OrderDetails getOrderDetails(String orderId)
                throws OrderNotFoundException, RemoteException;
}
```

A JAX-RPC interface generated from the WSDL description where the order processing center acts as a client of a supplier and an XML document passes between them. To fulfill part of the customer's order, the order processing center module sends a purchase order to the supplier's Web service. For example, to fulfill a customer's travel request, the order processing center sends an order contained in an XML document to the airline supplier's Web service.

```
public interface AirlineReservationIntf extends Remote {
   public String submitDocument(SOAPElement reservationRequest)
```

```
                                        throws RemoteException;
}
```

Keep in mind that when a Web service method passes different types of XML documents, the WSDL description should use the generic type `anyType` to indicate the type of the method's parameters which is mapped to SOAPElement in the generated JAX-RPC interface. Because the WSDL does not have the information to describe these documents, a separate schema holds the complete description of the documents. You need to publish the schemas for all documents that are exchanged. Publishing entails making the schemas available to clients at some known URL or registry.

On the Java platform, it is best to send an XML document as a `javax.xml.transform.Source` object. However, we chose to send the XML document as a `SOAPElement` object because the WS-I Basic Profile does not yet support `Source` objects.

The **order fulfillment** and **order tracking** Web service interfaces, since they use parameters that are objects, have no need for document validation or transformation. For both, there is minimal mapping from one data model to another, since the passed data mirrors for the most part the internal data model. Thus, both services do very minimal preprocessing of requests. As a result, we chose to merge the interaction and processing layers for these services.

The Web service interfaces between the **order processing center and the suppliers** must validate and transform incoming XML documents to the internal data model before processing the document contents. We chose to do the validation and transformation in the interaction layer of the service before delegating the request to the processing layer.

The order tracking Web service interface uses a synchronous interaction for handling requests. Adventure builder's Web services for processing the order workflow are asynchronous in nature. For these other Web service interactions—client submitting purchase order to the order processing center, order processing center placing orders with the suppliers, suppliers invoicing the order processing center—the service interfaces pre-process the requests and then delegate the request to the business logic. The interfaces use JMS messages to send the requests to a JMS queue associated with the business logic.

Adventure builder's Web service clients use stubs, which is the simplest of the three modes. The stub mode requires the WSDL document along with the JAX-RPC mapping file to generate a set of proxies at development. A client at runtime uses these generated proxies to access a service. The stub mode of proxy generation provides the client with a tightly coupled view of the Web service, and it is a good choice when the service endpoint interface does not change. With stubs, adventure builder uses the `javax.xml.rpc.Service` interface `getPort` method to access the Web service. Using this method removes port and protocol binding dependencies on the generated service implementation classes. As a result, it improves the portability of the client code since the client does not have to hard code a generated stub class (rather use the generic javax.xml.rpc.Service).
The customer Web site, when it catches exceptions thrown by the order tracking service, maps the exceptions to client-specific exceptions. The client application detects these exceptions and redirects the user to an appropriate error page.

We decided to make the adventure builder Web services available in a well-known location, which clients can obtain from the deployment environment. These Web service interfaces are meant for specific business interactions between specific entities rather than for consumption by the general public. For this reason, we decided against publishing the services in a registry. Similarly, we decided to place the business document schemas at a well-known location, from which intended clients use these schemas.

Communication Patterns

Often Web service interactions are part of larger business collaborations that may involve multiple synchronous and asynchronous Web service interactions. Although Web service interactions within a single business collaboration may be related, current Web service technologies treat these interactions as stateless. As a result, a developer needs a way to associate messages that are part of a single business collaboration.

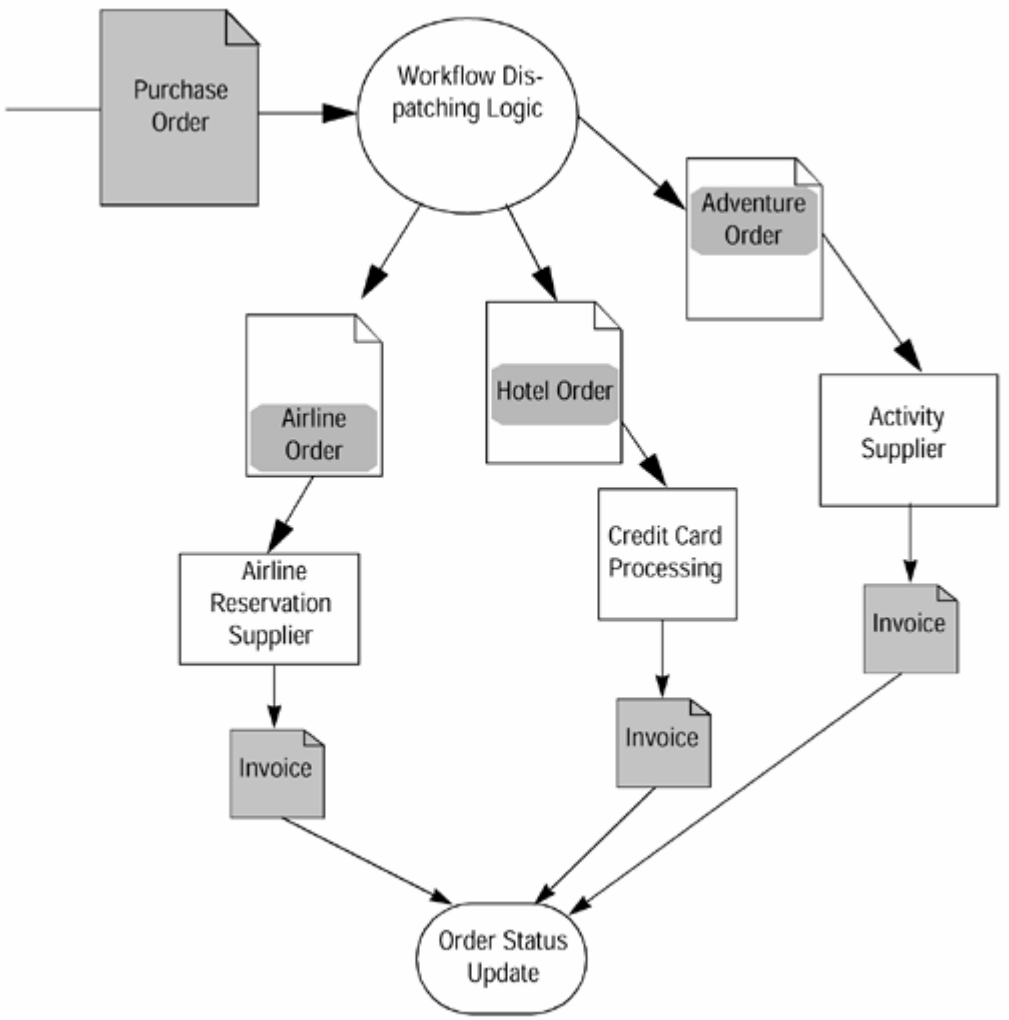
This issue is made more difficult because collaboration sometimes requires that messages be split into multiple sub-messages, where each sub-message is processed by a different entity or partner. Not only does the developer need to relate these sub-messages to the original message, but the developer may also need to join together replies to the sub-messages.
A particular request may take several Web service interactions to complete. This often happens when there are multiple steps in a workflow. Since different services process parts of a request, you need to correlate these interactions and identify individual requests that together make up a larger job.
Correlated Messages:
Sometimes a business collaboration requires multiple Web service calls. In these cases, the developer needs a way to indicate that the messages passed in these Web service calls are related. A developer should use a correlation identifier to keep together messages for related Web service calls. Scenarios where correlation identifier can be useful are:

- A client submits an order and later wants to track the order status.
- A request is broken up into multiple subtasks that are asynchronously processed. The request manager aggregates subtasks as they complete, finally marking the entire job complete. Often, many jobs run at the same time, requiring the manager to correlate each job's multiple subtasks. Adventure builder's order processing module illustrates this scenario.
- A Web service interaction may require a higher level of service in terms of handling errors and recovery. If an error occurs, the service may have to track the state of the request.
- A client needs to make several request and reply interactions as part of a conversation, and it needs to identify those calls belonging to the conversation.
- A client may make duplicate submissions of a message, and the service needs to eliminate these duplications.

The correlation identifier must be unique in the participating systems. Server-side identifier generation also allows better control by providers, since they can employ their own policies. But the correlation identifier can even be generated at client. The customer Web site collects information from a Web user and puts that information together into an order. The Web site then places the order into adventure builder's system via a Web service call to the order receiver module, passing the order information as a message. The order information includes the client-generated correlation identifier. The order receiver module receives the order message and uses the correlation identifier to **eliminate duplicate order submissions.**

Split + Join Operations: The order processing center splits a purchase order document into multiple XML documents by extracting the relevant information for each supplier, then sending that relevant information as a sub-order to the suppliers. It includes a correlation identifier with the sub-order to correlate all submessages generated from a single message. For the join operation, the order processing center waits for all suppliers to fulfill the sub-orders, then it does some further processing to complete the order. Notification from suppliers may arrive in an arbitrary, nonsequential, order. The order processing center implements the join operation using message-driven beans. When it processes a sub-order, a supplier sends an invoice message to the order processing center through a Web service call. On receiving the invoice message, the order processing center workflow manager checks whether the join condition is met and updates its state accordingly. When all invoices are received, the workflow manager changes the order status to "COMPLETED" and directs the CRM system to notify the user.

The customer orders an adventure package, plus hotel and airline reservations. All this information is contained with the purchase order. The order processing center logic splits the purchase order into sub-orders, and sends each sub-order to the appropriate supplier for processing. For example, the order processing center sends the activity sub-order to an activity supplier and the airline sub-order to the airline reservation processing. Each supplier, when it completes a sub-order, submits an invoice for its work. The order processing center receives the separate invoices at varying times. It checks to see if the join condition is met and updates the order status accordingly.

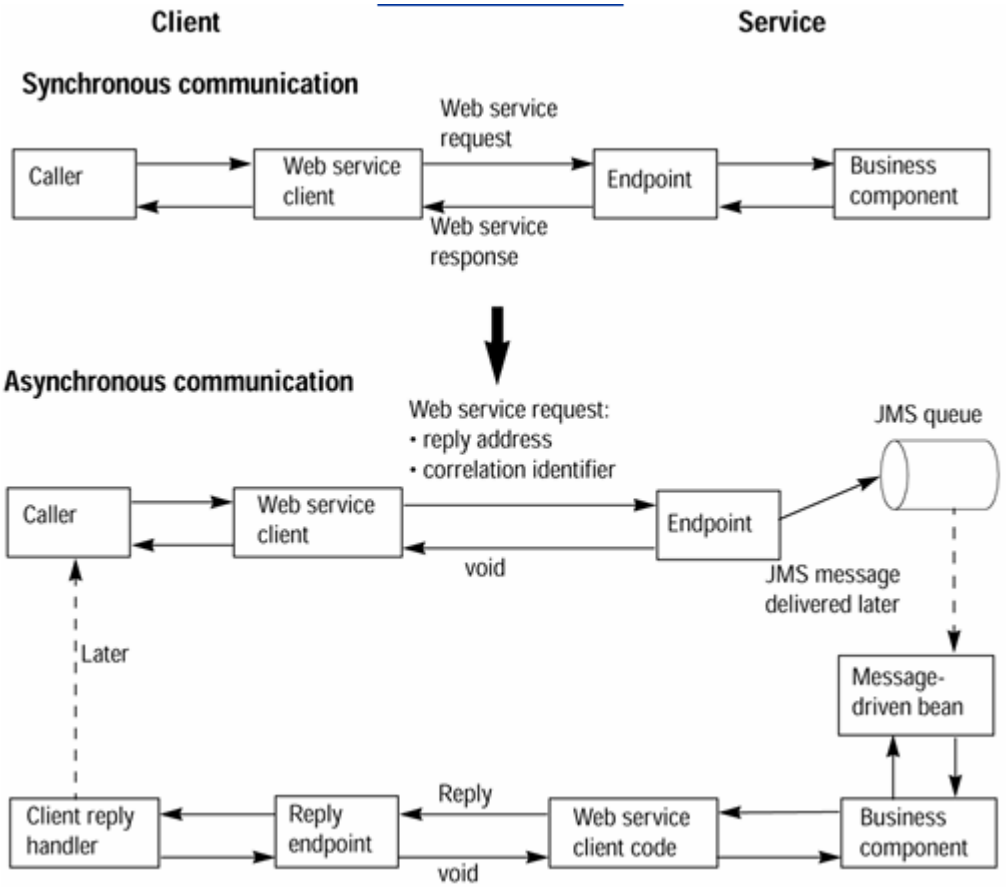


Converting Synchronous Endpoint to Asynchronous Endpoint

Web services that use asynchronous interactions tend to be more responsive and scale better. However, Web services provide only a synchronous mode of operation, especially when using the HTTP protocol. In your application you can convert this synchronous interaction to an asynchronous request and reply.

- Change a synchronous request/reply interaction into a request that returns `void` as the immediate reply.
- Change the client code to send the message request and immediately return.
- Have the client establish a reply address for receiving the response and include this reply address in the request. By including a reply address, the service knows where to send the response message.
- A correlation identifier needs to be established between the request message and the service response message. Both the client and the service use this identifier to associate the request and the reply.
- Refactor the server endpoint to accept the request, send it to a JMS queue for processing later, and immediately return an acknowledgement to the client. The service retains the correlation identifier.
- Have the service processing layer use a message-driven bean to process the received request. When a response is ready, the service sends the response to the reply address. In addition to the results, the response message contains the same correlation identifier so that the service requestor can identify the request to which this response relates.

- Have the client accept the response at a Web service that it establishes at the reply address.



Optionally, to further increase responsiveness , a stand-alone client that uses a Swing GUI might spawn a thread to make the Web service call, allowing the client to continue its user interactions while the Web service handles the call. Or, a J2EE client component might use a JMS queue to make Web service calls—that is, rather than making the JAX-RPC call directly, the client places the request into a JMS queue and a message-driven bean makes the Web service call to deliver the request.

Passing Context Information on Webservice Calls

Purchase order consists of the message contents—items to purchase, customer and financial information—and you also may need to pass some extra information about the message, such as its processing priority. The Web service may also need additional context information indicating how to process the message, such as processing instructions or hints. This context information is often passed along with the message. The following are some use cases where such additional data may be needed. Some use cases where Context information may be needed:

- Security information needs to be embedded within the document. For example, you may want to include some message-level security or add a digital signature to an accounts payment document as part of a credit card transaction.
- A correlation identifier needs to be included with the message to indicate that the message is associated with a logical group of messages exchanged in a workflow.
- Transactional or reliability information may be included along with the message contents to indicate additional quality of service activities that should be applied before processing the message contents.
- A message needs to be processed on a priority basis.

Strategies to include context information with a message exchange are:

1. Context information can be **passed as an extra parameter in the Web service method**. You can define the Web service interface to accept not only the XML document message, but to also include a second parameter that encapsulates metadata. Including context information as part of the service interface by adding an extra field for the input parameters or return values, effectively makes the context information part of the WSDL. Besides complicating the service interface, you must remember to update the interface—and regenerate the WSDL—if you add to or change the context information. For these reasons, the strategy of including context information as part of the service interface results in code that is harder to maintain.

```
public interface MyWebService extends Remote {
// priorityProcessing is the data indicating that the purchase
// order needs to be processed on a priority basis

public String submitPurchaseOrder(PurchaseOrder poObject,
          boolean priorityProcessing) throws RemoteException;
}
```

2. Context information also can be **passed as part of the document itself**. You can embed the extra metadata within the XML document. When you parse the document, you extract the needed information. There is also no logical separation between the schema for the document request and the schema for the context information and hence it also results in a code which is harder to maintain.

```
<PurchaseOrder>
<Id>123456789 </Id>
...
<POContextInfo> <!-- This is where the context data begins -->
   <PriorityProcessing>True</PriorityProcessing>
   <!-- Other context data elements -->
</POContextInfo>
...
</PurchaseOrder>
```

3. Context information also can be **passed in the SOAP message header**. You can embed the metadata in the SOAP message header and write a handler to extract it and pass it to the endpoint. For this strategy, clients need to embed this information in a SOAP header and the service needs to extract it. The service can use a JAX-RPC handler to intercept the SOAP message and then extract the header while still keeping it associated with the document. The handler uses this context information to take appropriate actions before the business logic is invoked in the endpoint. This strategy is more **elegant** because it does not require you to modify the XML schema for the data model exported by the service. Instead, the handler unobtrusively determines the context information from the envelope's header, leaving the message body untouched and unaffected.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="SoapEnvelopeURI"
       SOAP-ENV:encodingStyle="SoapEncodingURI">
   <SOAP-ENV:Header>
      <ci:PriorityProcessing>True</ci:PriorityProcessing>
   </SOAP-ENV:Header>
<SOAP-ENV:Body>
        <opc:submitPurchaseOrder xmlns:opc="ServiceURI">
          <opc:PurchaseOrder>
            <!-- contents of PurchaseOrder document -->
             </opc:PurchaseOrder>
        </opc:submitPurchaseOrder>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Sometimes, the handler needs to pass the context information (or the results of processing the context information) to the endpoint implementation. The way to do this is to use the JAX-RPC `MessageContext` interface. If the handler sets the context information in the `MessageContext` interface, the service endpoint can access it.

```
public class MyMessageHandler extends
          javax.xml.rpc.handler.GenericHandler {

   public boolean handleRequest(MessageContext mc) {
       SOAPMessage msg = ((SOAPMessageContext)mc).getMessage() ;
       SOAPPart sp = msg.getSOAPPart();
       SOAPEnvelope se = sp.getEnvelope();
       SOAPHeader header = se.getHeader();
       SOAPBody body = se.getBody();

       if (header == null) {
           // raise error
       }

       for (Iterator iter = header.getChildElements();
                    iter.hasNext();) {
           SOAPElement element = (SOAPElement) iter.next();
           if (element.getElementName().getLocalName()
                  .equals("PriorityProcessing")) {
              mc.setProperty("PriorityProcessing",
                 element.getValue());
           }
       }

   ...
   return true;

}
```

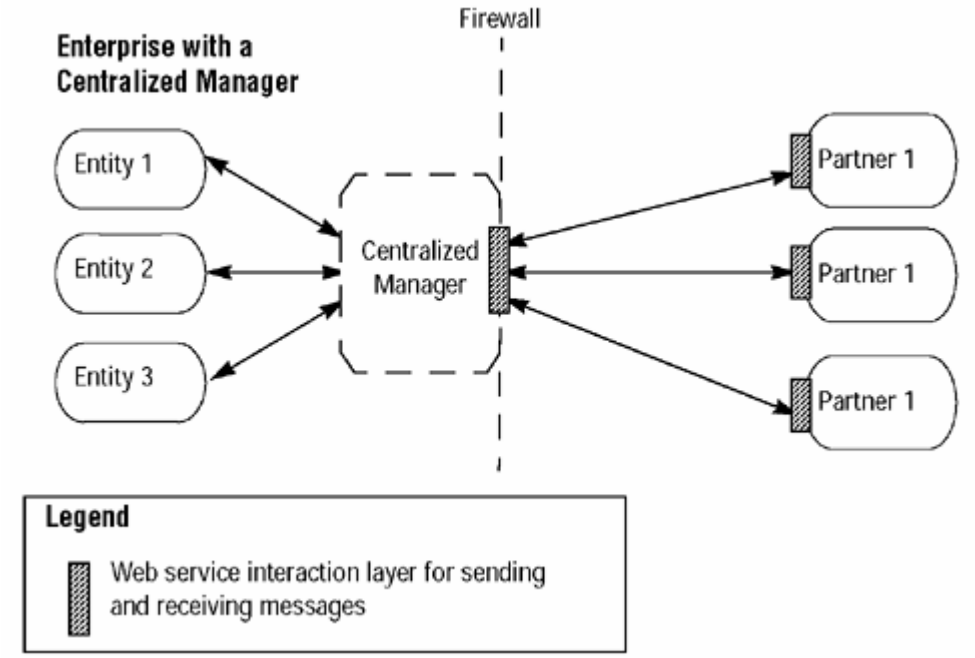
Then, you can get access to `MessageContext` in the endpoint that receives the request. For an EJB service endpoint, `MessageContext` is available with the bean's `SessionContext`.

```
public class EndpointBean implements SessionBean {
   private SessionContext sc;
   public void businessMethod() {
      MessageContext msgc= sc.getMessageContext();
      String s = (String)msgc.getProperty("PriorityProcessing");
      Boolean priority = new Boolean(s);
      ...
   }

   public void setSessionContext(SessionContext sc) {
      this.sc = sc;
   }

   ...
}
```

A better strategy for **handling multiple document types** is to design the service interface with this possibility in mind. You can do so by writing a generic method to represent a document-centric interaction. For example, instead of having multiple methods for each document type such as `submitSupplierInvoice`, `submitSupplier2Invoice`, and `submitBillingInfo`, the endpoint interface has a single method called `submitDocument` that receives all document types. To accept any type of document, you need to use `xsd:anyType` as the parameter type for the `submitDocument` method. The service interface does not have to change to accommodate new document types, just the service implementation changes.

The `submitDocument` method needs to map these documents to the processing logic that handles them. You can apply a **Command pattern** to this strategy to achieve flexibility in handling these documents. With a Command pattern, you place the commands to manage all schemas of the various document types in one place, identifying each command by schema type. To add a new schema, you simply add a new command that handles the new schema. The command mapping happens in the interaction layer, separate from the processing layer and the business logic.

When the number of Web service interactions grows beyond a certain point, you may want to factor out common code for reuse. For service interactions that rely on document passing, consider using a centralized manager that can broker the Web service interactions and consolidate the interaction layer of your Web services. This centralized manager can be responsible for both incoming and outgoing Web service interactions, and it can also consolidate all client access to a service. A centralized manager strategy applies only to interactions involving document passing, because such interactions can use an interface with a single method to handle all possible documents. Interfaces that pass objects as parameters do not have the flexibility to be this generic—a method passing or returning an object must declare the object type, which can only map to a single document type. For exmaple, the order processing center is both a client of and a service to multiple suppliers. The order processing center and the suppliers engage in XML document exchange interactions, and often the interactions are asynchronous since they may span a fair amount of time.



To illustrate, we implemented this strategy between the order processing center **Order Filler submodule** and the supplier Web services. The centralized manager handles XML documents sent in by suppliers. The centralized manager extracts information from these incoming requests, and this information lets it know how to handle the request. It may reformat the message content or transform the content to the internal canonical form used by the enterprise. The centralized manager routes the reformatted message to the correct target system. It can also validate incoming documents and centralize the management of document schemas. For outgoing Web service calls, the centralized manager acts as the client on behalf of the internal modules. Use of this strategy also decouples components that access a service from those that focus on business logic. Your enterprise ends up with a clean Web service layer that is decoupled from the processing layer.

For handling incoming requests, you can establish a single interface and endpoint that all clients use. You can add context information to each request to enable an easy identification of the type of the request, such as purchase order or invoice. When this single endpoint acts as the interaction layer for several Web services, it can perform a number of functions, depending on what is available to it. If it has access to

schemas, the endpoint can perform validation and error checking. If it has access to style sheets, the endpoint can transform incoming requests to match the canonical data model. A centralized manager may also use a **message router** to route requests to the appropriate business component recipients in the processing layer. A message router is responsible for mapping requests to recipients.
The adventure builder enterprise receives messages from suppliers through the centralized manager, which provides a natural place to hold a canonical data model. The canonical data model we use is represented in XML. The order processing center provides the canonical data model used for its interaction with the Web site.

Building Robust (Fault tolerant) Webservice

Creating **idempotent endpoints** is one strategy a service can use to handle duplicate messages. Idempotent refers to a situation where repeated executions of the same event have the same effect as a single execution of the event. Making your endpoints idempotent avoids the problem of a service processing duplicate instances of the same message or request. Even if a client mistakenly—or even maliciously—submits a request such as a purchase order (via a JAX-RPC call) more than once, the effect is the same as if the request was submitted just once.
Service endpoints that only perform read operations are naturally idempotent, since these operations do not have any side effects. For example, the adventure builder application's CRM order tracking Web service is naturally idempotent since it only invokes read operations. For Web services that perform updates or otherwise change state, you need to explicitly build idempotency. One way to do this is to leverage the semantics of the endpoint logic. If you know the effect of a single execution of an operation, you can change the program logic to ensure that multiple executions of the operation have the same result as a single execution. To do this:

1. First, to detect duplicate requests, you need to assign a correlation identifier to client interactions with a service.
2. This correlation identifier can be passed as context information (in the SOAPHeader) and intercepted and processed by a JAX-RPC handler.
3. When the request is received, the endpoint should check to see if it is a duplicate request. For example, the order processing center's `submitPurchaseOrder` method can be made idempotent since we know its operation depends on an order identifier key value. Before executing the business logic, the `submitPurchaseOrder` method stores the order in the database with the order identifier as its primary key. If the same purchase order is sent again, the second attempt to store the same order identifier in the database results in a duplicate key exception, preventing the order from being processed a second time.

Because multiple service requests have the same effect as a single request, clients of a idempotent service can retry message requests until they are successful, without fear of causing duplicate actions. However, using idempotent endpoints for a fault-tolerant Web service adds to the application's complexity and may adversely affect performance. It is also not interoperable: Specifications for this area are still being designed, and it is doubtful that the standards that ultimately result will work with current custom-built schemes.
Typically the client retries only a fixed number of times and then fails. You also design the endpoint to be idempotent. You may also need an acknowledgment message, which can be part of the JAX-RPC reply message or a separate message (if using asynchronous processing). Often before executing a retry, a client waits briefly so that transient conditions in the network or on the server may clear.
Remember that a failure can occur at any point during a JAX-RPC interaction—when the client makes a call or the service receives it, while the service processes the call, when the service sends a reply or the client receives it, or when the client processes the reply. To recover from such failures, you may want the application code to log interaction state on the client as well as the endpoint. If a failure occurs, you can use the log to recover and finish the interaction, deleting the log when complete or otherwise marking it as finished.
You need to formulate a contract—a set of understood interaction rules—between the client and the service endpoint. These rules might specify that retries are allowed and how many, the overall protocol, and so forth. Generally, the client makes the call and can repeat the call if it is not notified of success within a certain time limit. The service is responsible for detecting duplicate calls.

Handling Exceptions in Webservice Calls

For its Web service interactions, the adventure builder application gives the interaction requestor a correlation identifier for the submitted request. The service endpoint implementation catches any exceptions that occur when the request is preprocessed. Exceptions might be an invalid XML document or XML parsing and translation errors. For asynchronous interactions, the service endpoint implementation uses JMS to send requests to the processing layer. In those cases, it may also encounter JMS exceptions. For these errors, the endpoint passes a service-specific exception to the requestor.

In adventure builder, exceptions may occur as follows:

1. Case 1: During any Web service interaction between the Web site and the order processing center. These Web service interaction exceptions are synchronous in nature and thus easier to handle than exceptions that arise in cases 2 and 3.

```
public interface PurchaseOrderIntf extends Remote {
   public String submitPurchaseOrder(PurchaseOrder poObject)
          throws InvalidPOException, ProcessingException,
          RemoteException;

}
```

   Above is the code for the order processing center Web service interface that receives purchase orders from the Web module. The interface throws two kinds of service-specific exceptions: `InvalidPOException`, which it throws when the received purchase order is not in the expected format, and `ProcessingException`, which is throws when there is an error submitting the request to JMS. Note that the platform throws `RemoteException` when irrecoverable errors, such as network problems, occur.

2. Case 2: Within the order processing center, while processing an order during any stage of the workflow management operation.
3. Case 3: During the order processing center interaction with partners, such as the suppliers and the credit card service.

After the endpoint receives the purchase order from the client and successfully puts it in the workflow manager's queue, it returns a correlation identifier to the client. The client then goes about its own business. At this point, the workflow manager takes the order through the various workflow stages. Different exceptions can occur in each stage. These exceptions can be broadly categorized as those that require immediate client intervention and those that require an application retry.
Examples of exceptions that require human intervention might be an incoming purchase order that needs to be persisted to the data store but the database table does not exist. Or, the credit card agency might not grant credit authorization for an order because of an invalid credit card number. When these exception conditions occur, it is impossible to continue purchase order processing without human intervention. The adventure builder application informs the customer via e-mail when such conditions happen.
As an order progresses through the workflow stages, various exception conditions of a temporary nature may occur. There may be an error placing the order in a JMS queue, the database connection may be busy, a service may not be available, and so forth. For example, the workflow manager might try to invoke the supplier or bank Web service while that latter service is down. Or, the order processing center database may be temporarily unavailable due to a back-up operation. Since temporary error conditions often resolve themselves in a short while, asking the customer to intervene in these situations does not make for the best customer experience. A strategy for handling temporary error conditions involves keeping a status indicator to signal retrying a failed step at a later time. The adventure builder application workflow manager, as it moves a purchase order through various stages, tracks the status of the order at each stage. If an error occurs, the workflow manager flags the order's status to indicate an error. By using the timer bean mechanism available in the J2EE 1.4 platform, the workflow manager can periodically examine orders with an error status and attempt to move these orders to their next logical stage in the flow. The timer bean

activates after a specified period of time, initiating a check of orders flagged with errors and prompting the workflow manager to retry the order's previously failed operation. The order status, in addition to keeping an error flag, tracks the number of retry attempts. The workflow manager seeks human intervention when the number of retry attempts exceeds the fixed number of allowable retries.

Basic Profile 1.0 Spec Notes

[http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html]
Read from above link.