SCDJWS Study Guide

Table of Contents

Chapter 1. XML Web Service Standards
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.

BP 1.0 - Service Description (WSDL) - Document Structure.

A DESCRIPTION MUST only use the WSDL "import" statement to import another WSDL description.

To import XML Schema Definitions, a DESCRIPTION MUST use the XML Schema "`import`" statement.

A DESCRIPTION MUST use the XML Schema "`import`" statement only within the `xsd:schema` element of the `types` section.

A DESCRIPTION MUST NOT use the XML Schema "`import`" statement to import a Schema from any document whose root element is not "`schema`" from the namespace "http://www.w3.org/2001/XMLSchema".

An XML Schema directly or indirectly imported by a DESCRIPTION MUST use either UTF-8 or UTF-16 encoding.

An XML Schema directly or indirectly imported by a DESCRIPTION MUST use version 1.0 of the eXtensible Markup Language W3C Recommendation.

CORRECT:

```
<definitions name="StockQuote"
      targetNamespace="http://example.com/stockquote/definitions">
      <import namespace="http://example.com/stockquote/definitions"
           location="http://example.com/stockquote/stockquote.wsdl"/>
      <message name="GetLastTradePriceInput">
         <part name="body" element="..."/>
      </message>
                  ...
</definitions>
```

CORRECT:

```
<definitions name="StockQuote"
   targetNamespace="http://example.com/stockquote/"
   xmlns:xsd1="http://example.com/stockquote/schemas"
                ...
   xmlns="http://schemas.xmlsoap.org/wsdl/">
   <import namespace="http://example.com/stockquote/definitions"
        location="http://example.com/stockquote/stockquote.wsdl"/>
   <message name="GetLastTradePriceInput">
      <part name="body" element="xsd1:TradePriceRequest"/>
   </message>
               ...
</definitions>
```

INCORRECT (imports not "wsdl" document):

```
<definitions name="StockQuote"
   targetNamespace="http://example.com/stockquote/definitions"
   xmlns:xsd1="http://example.com/stockquote/schemas""
                ...
   xmlns="http://schemas.xmlsoap.org/wsdl/">

   <import namespace="http://example.com/stockquote/schemas"
         location="http://example.com/stockquote/stockquote.xsd"/>

   <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
   </message>
               ...
</definitions>
```

A DESCRIPTION MUST specify a non-empty location `attribute` on the `wsdl:import` element.

When they appear in a DESCRIPTION, `wsdl:import` elements MUST precede all other elements from the WSDL namespace except `wsdl:documentation`.

When they appear in a DESCRIPTION, `wsdl:types` elements MUST precede all other elements from the WSDL namespace except `wsdl:documentation` and `wsdl:import`.

CORRECT:

```
<definitions name="StockQuote"
      targetNamespace="http://example.com/stockquote/definitions">
     <import namespace="http://example.com/stockquote/base"
       location="http://example.com/stockquote/stockquote.wsdl"/>
      <message name="GetLastTradePriceInput">
         <part name="body" element="..."/>
      </message>
                  ...
</definitions>
```

CORRECT (`wsdl:types` before all other elements):

```
<definitions name="StockQuote"                  ...
   xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
     <schema targetNamespace="http://example.com/stockquote/schemas"
          xmlns="http://www.w3.org/2001/XMLSchema">
           .......
     </schema>
   </types>
   <message name="GetLastTradePriceInput">
        <part name="body" element="tns:TradePriceRequest"/>
   </message>
               ...
   <service name="StockQuoteService">
      <port name="StockQuotePort" binding="tns:StockQuoteSoap">
           ....
      </port>
   </service>
</definitions>
```

INCORRECT (wrong position of `wsdl:types` element):

```

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

   <import namespace="http://example.com/stockquote/definitions"
         location="http://example.com/stockquote/stockquote.wsdl"/>

   <message name="GetLastTradePriceInput">
       <part name="body" type="tns:TradePriceRequest"/>
   </message>
               ...
   <service name="StockQuoteService">
      <port name="StockQuotePort" binding="tns:StockQuoteSoap">
           ....
      </port>
   </service>

   <types>
      <schema targetNamespace="http://example.com/stockquote/schemas"
               xmlns="http://www.w3.org/2001/XMLSchema">
           .......
      </schema>
   </types>
</definitions>
```

A DESCRIPTION MUST use version 1.0 of the eXtensible Markup Language W3C Recommendation.
DESCRIPTION MUST use either UTF-8 or UTF-16 encoding.

The `targetNamespace` attribute on the `wsdl:definitions` element of a description that is being imported MUST have same the value as the namespace attribute on the `wsdl:import` element in the importing DESCRIPTION.
A document-literal binding in a DESCRIPTION MUST, in each of its `soapbind:body` element(s), have at most one `part` listed in the `parts` attribute, if the `parts` attribute is specified.
If a document-literal binding in a DESCRIPTION does not specify the `parts` attribute on a `soapbind:body` element, the corresponding abstract `wsdl:message` MUST define zero or one `wsdl:parts`.
A `wsdl:binding` in a DESCRIPTION MAY contain `soapbind:body` element(s) that specify that zero `parts` form the `soap:Body`.
An rpc-literal binding in a DESCRIPTION MUST refer, in its `soapbind:body` element(s), only to `wsdl:part` element(s) that have been defined using the `type` attribute.
A MESSAGE described with an rpc-literal binding MUST NOT have the `xsi:nil` attribute with a value of "1" or "`true`" on the part accessors.
A `wsdl:message` in a DESCRIPTION MAY contain `wsdl:parts` that use the elements attribute provided those `wsdl:parts` are not referred to by a `soapbind:body` in an rpc-literal binding.
A document-literal binding in a DESCRIPTION MUST refer, in each of its `soapbind:body` element(s), only to `wsdl:part` element(s) that have been defined using the `element` attribute.
A binding in a DESCRIPTION MAY contain `soapbind:header` element(s) that refer to `wsdl:parts` in the same `wsdl:message` that are referred to by its `soapbind:body` element(s).
A `wsdl:binding` in a DESCRIPTION MUST refer, in each of its `soapbind:header`, `soapbind:headerfault` and `soapbind:fault` elements, only to `wsdl:part` element(s) that have been defined using the `element` attribute.
A `wsdl:binding` in a DESCRIPTION SHOULD bind every `wsdl:part` of a `wsdl:message` in the `wsdl:portType` to which it refers to one of `soapbind:body`, `soapbind:header`, `soapbind:fault` or `soapbind:headerfault`.
A `wsdl:message` in a DESCRIPTION containing a `wsdl:part` that uses the `element` attribute MUST refer, in that attribute, to a global element declaration.
CORRECT:

```
<message name="GetTradePriceInput">
      <part name="body" element="tns:SubscribeToQuotes"/>
</message>
```

INCORRECT (`xsd:string` is not a global element):

```
<message name="GetTradePriceInput">
      <part name="tickerSymbol" element="xsd:string"/>
      <part name="time" element="xsd:timeInstant"/>
</message>
```

INCORRECT (`xsd:string` is not a global element):

```
<message name="GetTradePriceInput">
      <part name="tickerSymbol" element="xsd:string"/>
</message>
```

BP 1.0 - Service Description (WSDL) - Port Types.
The order of the elements in the `soap:body` of a MESSAGE MUST be the same as that of the `wsdl:parts` in the `wsdl:message` that describes it.
A DESCRIPTION MAY use the `parameterOrder` attribute of an `wsdl:operation` element to indicate the return value and method signatures as a hint to code generators.
A DESCRIPTION MUST NOT use `Solicit-Response` and `Notification` type operations in a `wsdl:portType` definition.
A `wsdl:portType` in a DESCRIPTION MUST have operations with distinct values for their `name` attributes.

A wsdl:portType in a DESCRIPTION MUST be constructed so that the parameterOrder attribute, if present, omits at most 1 wsdl:part from the output message.
A wsdl:message in a DESCRIPTION MUST NOT specify both type and element attributes on the same wsdl:part.
BP 1.0 - Service Description (WSDL) - SOAP Binding.
The wsdl:binding element in a DESCRIPTION MUST be constructed so that its soapbind:binding child element specifies the transport attribute.
A wsdl:binding element in a DESCRIPTION MUST specify the HTTP transport protocol with SOAP binding. Specifically, the transport attribute of its soapbind:binding child MUST have the value "http://schemas.xmlsoap.org/soap/http".
A wsdl:binding in a DESCRIPTION MUST use either be a rpc-literal binding or a document-literal binding.
A wsdl:binding in a DESCRIPTION MUST use the value of "literal" for the use attribute in all soapbind:body, soapbind:fault, soapbind:header and soapbind:headerfault elements.
A wsdl:binding in a DESCRIPTION that contains one or more soapbind:body, soapbind:fault, soapbind:header or soapbind:headerfault elements that do not specify the use attribute MUST be interpreted as though the value "literal" had been specified in each case.
A wsdl:portType in a DESCRIPTION MAY have zero or more wsdl:bindings that refer to it, defined in the same or other WSDL documents.
The operations in a wsdl:binding in a DESCRIPTION MUST result in wire signatures that are different from one another.
A DESCRIPTION SHOULD NOT have more than one wsdl:port with the same value for the location attribute of the soapbind:address element.
A document-literal binding MUST be represented on the wire as a MESSAGE with a soap:Body whose child element is an instance of the global element declaration referenced by the corresponding wsdl:message part.
For one-way operations, an INSTANCE MUST NOT return a HTTP response that contains a SOAP envelope. Specifically, the HTTP response entity-body must be empty.
A CONSUMER MUST ignore a SOAP response carried in a response from a one-way operation.
For one-way operations, a CONSUMER MUST NOT interpret a successful HTTP response status code (i.e., 2xx) to mean the message is valid or that the receiver would process it.
A document-literal binding in a DESCRIPTION MUST NOT have the namespace attribute specified on contained soapbind:body, soapbind:header, soapbind:headerfault and soapbind:fault elements.
An rpc-literal binding in a DESCRIPTION MUST have the namespace attribute specified, the value of which MUST be an absolute URI, on contained soapbind:body elements.
An rpc-literal binding in a DESCRIPTION MUST NOT have the namespace attribute specified on contained soapbind:header, soapbind:headerfault and soapbind:fault elements.
A wsdl:binding in a DESCRIPTION MUST have the same set of wsdl:operations as the wsdl:portType to which it refers.
A wsdl:binding in a DESCRIPTION MAY contain no soapbind:headerfault elements if there are no known header faults.
A wsdl:binding in a DESCRIPTION SHOULD contain a soapbind:fault describing each known fault.
A wsdl:binding in a DESCRIPTION SHOULD contain a soapbind:headerfault describing each known header fault.
A MESSAGE MAY contain a fault detail entry in a SOAP fault that is not described by a wsdl:fault element in the corresponding WSDL description.
A MESSAGE MAY contain the details of a header processing related fault in a SOAP header block that is not described by a wsdl:headerfault element in the corresponding WSDL description.
A wsdl:binding in a DESCRIPTION MUST use the attribute named part with a schema type of "NMTOKEN" on all contained soapbind:header and soapbind:headerfault elements.
A wsdl:binding in a DESCRIPTION MUST NOT use the attribute named parts on contained soapbind:header and soapbind:headerfault elements.

CORRECT:

```
<binding name="StockQuoteSoap" type="tns:StockQuotePortType">
  <soapbind:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="SubscribeToQuotes">
      <input message="tns:SubscribeToQuotes">
        <soapbind:body parts="body" use="literal"/>
        <soapbind:header message="tns:SubscribeToQuotes" part="subscribeheader"
use="literal"/>
     </input>
   </operation>
</binding>
```

A `wsdl:binding` in a DESCRIPTION MUST have the name attribute specified on all contained `soapbind:fault` elements.

In a DESCRIPTION, the value of the name attribute on a `soapbind:fault` element MUST match the value of the name attribute on its parent `wsdl:fault` element.

...

...

Describe the use of XML Schema in J2EE Web services.

The W3C XML Schema Definition Language is a way of describing and constraining the content of XML documents. The XML Schema specification consists of three parts. One part defines a set of simple datatypes, which can be associated with XML element types and attributes; this allows XML software to do a better job of managing dates, numbers, and other special forms of information. The second part of the specification proposes methods for describing the structure and constraining the contents of XML documents, and defines the rules governing schema validation of documents. The third part is a primer that explains what schemas are, how they differ from DTDs, and how one builds a schema.

XML Schema introduces new levels of flexibility that may accelerate the adoption of XML for significant industrial use. For example, a schema author can build a schema that borrows from a previous schema, but overrides it where new unique features are needed. XML Schema allows the author to determine which parts of a document may be validated, or identify parts of a document where a schema may apply. XML Schema also provides a way for users of e-commerce systems to choose which XML Schema they use to validate elements in a given namespace, thus providing better assurance in e-commerce transactions and greater security against unauthorized changes to validation rules. Further, as XML Schema are XML documents themselves, they may be managed by XML authoring tools, or through XSLT.

Let's start from "Hello World !" example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<greeting>
  Hello World!
</greeting>

<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="greeting" type="xsd:string"/>
</xsd:schema>
```

Here is a simple data structure for a purchase order. The order example contains following elements: company name, product identifier, and price.

```
<?xml version="1.0" encoding="UTF-8" ?>
<order_request>
  <company_name>IBA USA, Inc.</company_name>
  <product_id>C-0YST</product_id>
  <product_price>500.00</product_price>
</order_request>
```

This example represents pricing data as a floating-point number and company name and product identifier as strings. Agreement among different programs about how to handle data is essential. XML solves data typing issues through the use of XML Schemas. The following example shows how each element in the order request can be designated a specific data type:

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

<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
  <xsd:element name="order_request">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="company_name" type="xsd:string"/>
        <xsd:element name="product_id" type="xsd:string"/>
        <xsd:element name="product_price" type="xsd:double"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Another example of XML document for more complex purchase order:

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

<purchase_order order_date="2004-04-07">
  <shipping_address country="US">
    <name>Mikalai Zaikin</name>
    <street>28th Street</street>
    <city>Boulder</city>
    <state>CO</state>
    <zip>80301</zip>
  </shipping_address>
  <items>
    <item part_number="008291">
      <product_name>PRESARIO S5140</product_name>
      <quantity>1</quantity>
      <price>898.00</price>
    </item>
    <item part_number="005376">
      <product_name>COMPAQ FP7317</product_name>
      <quantity>1</quantity>
      <price>398.00</price>
    </item>
  </items>
</purchase_order>
```

The XML Schema (one of many possible) for this document will be:

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

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="product_name" 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="price" type="xsd:decimal" />
      </xsd:sequence>
      <xsd:attribute name="part_number" type="xsd:string" use="required" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="items">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1" ref="item" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="purchase_order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="shipping_address" type="us_address" />
        <xsd:element ref="items" />
      </xsd:sequence>
      <xsd:attribute name="order_date" type="xsd:date" use="required" />
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="us_address">
    <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:string" />
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US" />
  </xsd:complexType>

</xsd:schema>
```

NOTE: `ref` attribute allows to refer in Schema to outside element definition using `name` attribute of element. You should differ it from reffering to outside element in SOAP 1.1 Message, which uses `href` attribute with value `#some_unique_value` and target element must have `id` attribute with value `some_unique_value`:

```
...
<e:Books>
  <e:Book>
    <title>SCDJWS Study Guide</title>
    <author href="#mz" />
  </e:Book>
  <e:Book>
    <title>SCWCD 1.4 Study Guide</title>
    <author href="#mz" />
  </e:Book>
  <e:Book>
    <title>SCBCD 1.3 Study Guide</title>
    <author href="#mz" />
  </e:Book>
</e:Books>

<author id="mz">
  <name>Mikalai Zaikin</name>
</author>
...
```

In cases when a field in a data structure is referred to in several places in that data structure (for example, in a doubly linked list), then the field is serialized as an independent element, an immediate child element of `Body`, and must have an `id` attribute of type `xsd:ID`. Such elements are called multireference accessors. They provide access to the data in the field from multiple locations in the message. Each reference to the field in the data structure is serialized as an empty element with an `href` attribute of type `xsd:anyURI`, where the value of the

attribute contains the identifier specified in the `id` attribute on the multireference accessor preceded by a fragment identifier, # (pound sign).
The following code:

```
public class PersonName {
    public String firstName;
    public String lastName;
}
public class Person {
    public PersonName name;
    public float age;
    public short height;

    public static boolean compare(Person person1, Person person2);
}
```

The SOAP 1.1 message for `Person.compare(...)` call may look like this:

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
       soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body xmlns:ns1='http://example'>
        <ns1:compare>
            <person1 href='#pid1' />
            <person2 href='#pid1' />
        </ns1:compare>
        <ns1:Person id='pid1' >
            <name>
                <fistName>Mikalai</firstName>
                <lastName>Zaikin</lastName>
            </name>
            <age>29</age>
            <height>1.78</height>
        </ns1:Person>
    </soap:Body>
</soap:Envelope>
```

Default value of `minOccurs` is 1.
Default value of `maxOccurs` is 1.
XML Schema defines four main elements:

1. `xsd:element` declares an element and assigns it a type.
2. `xsd:attribute` declares an attribute and assigns it a type.
3. `xsd:complexType` defines a new complex type.
4. `xsd:simpleType` defines a new simple type.

Attribute values are always simple types. Attributes are unordered.
It does not matter whether complex type is defined before or after the element declaration as long as it is present in the schema document.
You can derive new simple types from existing types. An `xsd:simpleType` element defines the subtype. The `name` attribute of `xsd:simpleType` assigns a name to the new type, by which it can be referred to in `xsd:element type` attributes. An `xsd:restriction` child element derives by restricting the legal values of the base type. An `xsd:list` child element derives a type as a white space separated list of base type instances. An `xsd:union` child element derives by combining legal values from multiple base types.
You can derive new simple types types from existing types by restricting the type to a subset of its normal values. An `xsd:simpleType` element defines the restricted type. The `name` attribute of `xsd:simpleType` assigns a name to the new type. An `xsd:restriction` child element specifies what type is being restricted via its base attribute. Facet children of `xsd:restriction` specify the constraints on the type. For example, this `xsd:simpleType` element defines a `myYear` as any year from 1974 on:

```
<xsd:simpleType name="myYear">
  <xsd:restriction base="xsd:gYear">
    <xsd:minInclusive value="1974"/>
  </xsd:restriction>
</xsd:simpleType>
```

Then you declare the `year` element like this:

```
<xsd:element name="year" type="myYear" />
```

Facets include: `length`, `minLength`, `maxLength`, `pattern`, `enumeration`, `whiteSpace`, `maxInclusive`, `maxExclusive`, `minInclusive`, `minExclusive`, `totalDigits`, `fractionDigits`. Not all facets apply to all types.
For example, new string type must contain between 1 and 255 characters:

```
<?xml version="1.0"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="Str255">
    <xsd:restriction base="xsd:string">
       <xsd:minLength value="1"/>
       <xsd:maxLength value="255"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

For example, the new year type must be between 1974 and 2100:

```
<?xml version="1.0"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="myYear">
    <xsd:restriction base="xsd:gYear">
      <xsd:minInclusive value="1974"/>
      <xsd:maxInclusive value="2100"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

The `enumeration` facet lists all allowed values. Applies to all simple types except `boolean`. For example, the computer brand name must be one of the following : IBM, COMPAQ, DELL, HP.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="computerBrandName">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="IBM"/>
      <xsd:enumeration value="COMPAQ"/>
      <xsd:enumeration value="DELL"/>
      <xsd:enumeration value="HP"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Each element in the `xsd:all` group must occur zero or once; that is `minOccurs` and `maxOccurs` must each be 0 or 1. The `xsd:all` group must be the top level element of its type. The `xsd:all` group may contain only individual element declarations; no choices or sequences. Example:

```
<?xml version="1.0"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="personName">
    <xsd:all>
      <xsd:element name="firstName"  type="xsd:string"/>
      <xsd:element name="lastName" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
</xsd:schema>
```

`xsd:choice` requires exactly one of a group of specified elements to appear. The `choice` can have `minOccurs` and `maxOccurs` attributes that adjust this from zero to any given number. Example:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="computer">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
        <xsd:choice minOccurs="1" maxOccurs="1">
          <xsd:element name="desktop" type="xsd:string"/>
          <xsd:element name="notebook" type="xsd:string"/>
          <xsd:element name="handheld"   type="xsd:string"/>
        </xsd:choice>
      </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

`xsd:sequence` requires each child element it specifies to appear in the specified order. The sequence can have `minOccurs` and `maxOccurs` attributes that repeat each sequence zero to any given number of times.

Describe the use of namespaces in an XML document.

The XML namespaces recommendation defines a way to distinguish between duplicate element type and attribute names. Such duplication might occur, for example, in an XSLT stylesheet or in a document that contains element types and attributes from two different DTDs.
An XML namespace is a collection of element type and attribute names. The namespace is identified by a unique name, which is a URI. Thus, any element type or attribute name in an XML namespace can be uniquely identified by a two-part name: the name of its XML namespace and its local name. This two-part naming system is the only thing defined by the XML namespaces recommendation.
XML namespaces are declared with an `xmlns` attribute, which can associate a prefix with the namespace. You can declare an XML namespace on any element in an XML document. The declaration is in scope for the element containing the attribute and all its descendants (unless it is overridden or undeclared on one of those descendants). It is a common practice to declare all namespaces within the root element. For example:

```
<!-- Declares two XML namespaces. Their scope is the 'aaa' and 'bbb' elements. -->

<aaa xmlns:foo="http://www.foo.org/" xmlns="http://www.bar.org/">
  <bbb>abcd</bbb>
</aaa>
```

If an XML namespace declaration contains a prefix, you refer to element type and attribute names in that namespace with the prefix. For example:

```
<!-- 'aaa' and 'bbb' are in the 'http://www.foo.org/' namespace,
     which is associated with the 'foo' prefix. -->

<foo:aaa xmlns:foo="http://www.foo.org/">
  <foo:bbb>abcd</foo:bbb>
</foo:aaa>
```

If an XML namespace declaration does not contain a prefix, the namespace is the default XML namespace and you refer to element type names in that namespace without a prefix. For example:

```
<!-- This is equivalent to the previous example but uses a
     DEFAULT namespace instead of the 'foo' prefix. -->

<aaa xmlns="http://www.foo.org/">
  <bbb>abcd<bbb>
</aaa>
```

The value (unique URI) of the `xmlns` attribute identifies the namespace, not the prefix. In this example, all elements belong to the same namespace although different prefixes are used.

```
<!-- 'aaa' and 'bbb' belong to the same 'http://www.foo.org/' namespace. -->

<foo:aaa xmlns:foo="http://www.foo.org/" xmlns:bar="http://www.foo.org/">
  <bar:bbb>abcd</bar:bbb>
</foo:aaa>
```

In this example, all elements belong to different namespaces although they have the same prefixes.

```
<!-- 'aaa' and 'bbb' belong to different namespaces. -->
<foo:aaa xmlns:foo="http://www.foo.org/">
  <foo:aaa />
</foo:aaa>
<foo:bbb xmlns:foo="http://www.bar.org/">
  <foo:bbb />
</foo:bbb>
```

You may override the prefix used in an XML namespace declaration, simply declare another XML namespace with the same prefix. For example, in the following, the `foo` prefix is associated with the `http://www.foo.org/` namespace on the aaa element and the `http://www.bar.org/` namespace on the bbb element. That is, the name aaa is in the `http://www.foo.org/` namespace and the name bbb is in the `http://www.bar.org/` namespace.

```
<!-- 'bbb' belongs to overriden 'http://www.bar.org/' namespace. -->
<foo:aaa xmlns:foo="http://www.foo.org/">
  <foo:bbb xmlns:foo="http://www.bar.org/">abcd</foo:bbb>
</foo:aaa>
```

This practice leads to documents that are confusing to read and should be avoided.

When an XML namespace declaration goes out of scope, it simply no longer applies. For example, in the following, the declaration of the `http://www.foo.org/` namespace does not apply to the bbb element because this is outside its scope.

```
<!-- 'bbb' does NOT belong to any  namespace. -->
<aaa xmlns="http://www.foo.org/">abcd</aaa>
<bbb>abcd</bbb>
```

You may undeclare the default XML namespace - declare a default XML namespace with an empty (zero-length) name (URI). Within the scope of this declaration, unprefixed element type names do not belong to any XML namespace. For example, in the following, the default XML namespace is the `http://www.foo.org/` for the aaa and there is no default XML namespace for the bbb elements. The name aaa is in the `http://www.foo.org/` namespace and the name bbb is not in any XML namespace.

```
<!-- 'bbb' does NOT belong to any  namespace. -->
<aaa xmlns="http://www.foo.org/">
  <aaa>
    <bbb xmlns="">
      <bbb>abcd</bbb>
    </bbb>
  </aaa>
</aaa>
```

You may NOT undeclare XML namespace prefix. It remains in scope until the end of the element on which it was declared unless it is overridden. Furthermore, trying to undeclare a prefix by redeclaring it with an empty (zero-length) name (URI) results in a namespace error. For example:

```
<!-- You may NOT undeclare XML namespace prefix. -->
<foo:aaa xmlns:foo="http://www.foo.org/">
  <foo:aaa>
    <foo:bbb xmlns:foo=""> <!-- ERROR -->
      <foo:bbb>abcd</foo:bbb>
     </foo:bbb>
  </foo:aaa>
</foo:aaa>
```

Attributes can be also explicitly assigned to the given namespace. See the example:

```
<!--
     You may assign namespaces to attributes.
     'bbb' element belongs to 'http://www.bar.org/' namespace.
     'attr' attribute belongs to 'http://www.foo.org/' namespace.
-->
<foo:aaa xmlns:foo="http://www.foo.org/" xmlns:bar="http://www.bar.org/">
  <bar:bbb foo:attr="attribute">abcd</bar:bbb>
</foo:aaa>
```

Attributes without a prefix never belongs to any namespace. The attributes do not belong to any namespace even if a default namespace is defined for the relevant element.

Chapter 2. SOAP 1.1 Web Service Standards

List and describe the encoding types used in a SOAP message.

The SOAP encoding style is based on a simple type system that is a generalization of the common features found in type systems in programming languages, databases and semi-structured data. A type either is a simple (scalar) type or is a compound type constructed as a composite of several parts, each with a type. This is described in more detail below. This section defines rules for serialization of a graph of typed objects. It operates on two levels. First, given a schema in any notation consistent with the type system described, a schema for an XML grammar may be constructed. Second, given a type-system schema and a particular graph of values conforming to that schema, an XML instance may be constructed. In reverse, given an XML instance produced in accordance with these rules, and given also the original schema, a copy of the original value graph may be constructed.
There are following SOAP encoding types:

- Simple Types:
  - Strings
  - Enumerations
  - Array of Bytes
- Compound Types:
  - Arrays
  - Structures

Simple Types

For simple types, SOAP adopts all the types found in the section "Built-in datatypes" of the "XML Schema Part 2: Datatypes" Specification, both the value and lexical spaces. Examples include: `boolean` (`true`, `false`, `0` or `1`), `byte`, `short`, `int`, `long`, `float`, `double`, `string` (`java.lang.String`), `decimal` (`java.math.BigDecimal`), `date` (`java.util.GregorianCalendar`), `dateTime` (`java.util.Date`), `SOAP-ENC:base64` (`byte []`).
The following examples are a SOAP representation of these primitive data types:

```
<element name="age" type="int"/>
<element name="height" type="float"/>
<element name="displacement" type="negativeInteger"/>
<element name="color">
  <simpleType base="xsd:string">
    <enumeration value="Green"/>
    <enumeration value="Blue"/>
  </simpleType>
</element>

<age>45</age>
<height>5.9</height>
<displacement>-450</displacement>
<color>Blue</color>
```

Strings

The datatype "`string`" is defined in "XML Schema Part 2: Datatypes" Specification. Note that this is not identical to the type called "`string`" in many database or programming languages, and in particular may forbid some characters those languages would permit. (Those values must be represented by using some datatype other than `xsd:string`.) A string MAY be encoded as a single-reference or a multi-reference value. The containing element of the string value MAY have an "`id`" attribute. Additional accessor elements MAY then have matching "`href`" attributes. For example, two accessors to the same string could appear, as follows:

```
<greeting id="String-0">Hello</greeting>
<salutation href="#String-0"/>
```

However, if the fact that both accessors reference the same instance of the string (or subtype of string) is immaterial, they may be encoded as two single-reference values as follows:

```
<greeting>Hello</greeting>
<salutation>Hello</salutation>
```

Schema fragments for these examples could appear similar to the following:

```
<element name="greeting" type="SOAP-ENC:string"/>
<element name="salutation" type="SOAP-ENC:string"/>
```

(In this example, the type SOAP-ENC:string is used as the element's type as a convenient way to declare an element whose datatype is "xsd:string" and which also allows an "id" and "href" attribute. See the SOAP Encoding schema for the exact definition. Schemas MAY use these declarations from the SOAP Encoding schema but are not required to.)

Enumerations

Enumeration as a concept indicates a set of distinct names. A specific enumeration is a specific list of distinct values appropriate to the base type. For example the set of color names ("Green", "Blue", "Brown") could be defined as an enumeration based on the string built-in type. The values ("1", "3", "5") are a possible enumeration based on integer, and so on. "XML Schema Part 2: Datatypes" supports enumerations for all of the simple types except for boolean. The language of "XML Schema Part 1: Structures" Specification can be used to define enumeration types. If a schema is generated from another notation in which no specific base type is applicable, use "string". In the following schema example "EyeColor" is defined as a string with the possible values of "Green", "Blue", or "Brown" enumerated, and instance data is shown accordingly:

```
<element name="EyeColor" type="tns:EyeColor"/>
<simpleType name="EyeColor" base="xsd:string">
   <enumeration value="Green"/>
   <enumeration value="Blue"/>
   <enumeration value="Brown"/>
</simpleType>
<Person>
   <Name>Mikalai Zaikin</Name>
   <Age>29</Age>
   <EyeColor>Brown</EyeColor>
</Person>
```

Array of Bytes

An array of bytes MAY be encoded as a single-reference or a multi-reference value. The rules for an array of bytes are similar to those for a string. In particular, the containing element of the array of bytes value MAY have an "id" attribute. Additional accessor elements MAY then have matching "href" attributes. The recommended representation of an opaque array of bytes is the 'base64' encoding defined in XML Schemas, which uses the base64 encoding algorithm. However, the line length restrictions that normally apply to base64 data in MIME do not apply in SOAP. A "SOAP-ENC:base64" subtype is supplied for use with SOAP:

```
<picture xsi:type="SOAP-ENC:base64">
  aG93IG5vDyBicm73biBjb3cNCg==
</picture>
```

Polymorphic Accessor

Many languages allow accessors that can polymorphically access values of several types, each type being available at run time. A polymorphic accessor instance MUST contain an "xsi:type" attribute that describes the type of the actual value. For example, a polymorphic accessor named "cost" with a value of type "xsd:float" would be encoded as follows:

```
<cost xsi:type="xsd:float">29.95</cost>
```

as contrasted with a cost accessor whose value's type is invariant, as follows:

```
<cost>29.95</cost>
```

Compound types

A "struct" is a compound value in which accessor name is the only distinction among member values, and no accessor has the same name as any other.

An "array" is a compound value in which ordinal position serves as the only distinction among member values.

Structures

The members of a Compound Value are encoded as accessor elements. When accessors are distinguished by their name (as for example in a struct), the accessor name is used as the element name. Accessors whose names are local to their containing types have unqualified element names; all others have qualified names. The following is an example of a struct of type "book":

```
<book>
  <author>Mikalai Zaikin</author>
  <title>SCDJWS Study Guide</title>
  <intro>This is a certification guide</intro>
</book>
```

And this is a schema fragment describing the above structure:

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="author" type="xsd:string" />
      <xsd:element name="title" type="xsd:string" />
      <xsd:element name="intro" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Arrays

SOAP arrays are defined as having a type of "SOAP-ENC:Array" or a type derived there from. Arrays are represented as element values, with no specific constraint on the name of the containing element (just as values generally do not constrain the name of their containing element). Arrays can contain elements which themselves can be of any type, including nested arrays. New types formed by restrictions of SOAP-ENC:Array can also be created to represent, for example, arrays limited to integers or arrays of some user-defined enumeration. The representation of the value of an array is an ordered sequence of elements constituting the items of the array. Within an array value, element names are not significant for distinguishing accessors. Elements may have any name. In practice, elements will frequently be named so that their declaration in a schema suggests or determines their type. As with compound types generally, if the value of an item in the array is a single-reference value, the item contains its value. Otherwise, the item references its value via an "href" attribute. The following example is a schema fragment and an array containing integer array members:

```
<element name="myFavoriteNumbers" type="SOAP-ENC:Array"/>
<myFavoriteNumbers SOAP-ENC:arrayType="xsd:int[3]">
   <number>1</number>
   <number>2</number>
   <number>3</number>
</myFavoriteNumbers>
```

In that example, the array "myFavoriteNumbers" contains several members each of which is a value of type xsd:int. This can be determined by inspection of the SOAP-ENC:arrayType attribute. Note that the SOAP-ENC:Array type allows unqualified element names without restriction. These convey no type information, so when used they must either have an xsi:type attribute or the containing element must have a SOAP-ENC:arrayType attribute. Naturally, types derived from SOAP-ENC:Array may declare local elements, with type information. As previously noted, the SOAP-ENC schema contains declarations of elements with names corresponding to each simple type in the "XML Schema Part 2: Datatypes" Specification. It also contains a declaration for "Array". Using these, we might write:

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[3]">
   <SOAP-ENC:int>1</SOAP-ENC:int>
   <SOAP-ENC:int>2</SOAP-ENC:int>
   <SOAP-ENC:int>3</SOAP-ENC:int>
</SOAP-ENC:Array>
```

Arrays can contain instances of any subtype of the specified arrayType. That is, the members may be of any type that is substitutable for the type specified in the arrayType attribute, according to whatever substitutability rules are expressed in the schema. So, for example, an array of integers can contain any type derived from integer (for example "int" or any user-defined derivation of integer). Similarly, an array of "address" might contain a restricted or extended type such as "internationalAddress". Because the supplied SOAP-ENC:Array type admits members of any type, arbitrary mixtures of types can be contained unless specifically limited by use of the arrayType attribute.

Array values may be structs or other compound values. For example an array of "my:order" structs :

```
<SOAP-ENC:Array SOAP-ENC:arrayType="my:order[2]">
   <order>
       <product>Melon</product>
       <price>0.99</price>
   </order>
   <order>
       <product>Apple</product>
       <price>1.49</price>
   </order>                   <
```

```
/SOAP-ENC:Array>
```

Arrays may have other arrays as member values. The following is an example of an array of two arrays, each of which is an array of strings:

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[][2]">
  <item href="#array-1"/>
  <item href="#array-2"/>
</SOAP-ENC:Array>

<SOAP-ENC:Array id="array-1" SOAP-ENC:arrayType="xsd:string[3]">
  <item>row1column1</item>
  <item>row1column2</item>
  <item>row1column3</item>
</SOAP-ENC:Array>

<SOAP-ENC:Array id="array-2" SOAP-ENC:arrayType="xsd:string[2]">
  <item>row2column1</item>
  <item>row2column2</item>
</SOAP-ENC:Array>
```

Arrays may be multi-dimensional. In this case, more than one size will appear within the `asize` part of the `arrayType` attribute:

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[2,3]">
  <item>row1column1</item>
  <item>row1column2</item>
  <item>row1column3</item>
  <item>row2column1</item>
  <item>row2ccolumn2</item>
  <item>row2column3</item>
</SOAP-ENC:Array>
```

NOTE: According to WS-I BP 1.0 you MUST NOT use `soapenc:Array` type for array declarations or `soapenc:arrayType` attribute in the type declarations.

Mapping between XML Schema types and SOAP Java types

Table 2.1. Mapping between XML Schema types and SOAP Java types

XML Schema type	SOAP Java type
`string`	`java.lang.String`
`integer`	`java.math.BigInteger`
`int`	`int`
`int`	`java.lang.Integer` (if `nillable="true"`)
`long`	`long`
`long`	`java.lang.Long` (if `nillable="true"`)
`short`	`short`
`short`	`java.lang.Short` (if `nillable="true"`)
`decimal`	`java.math.BigDecimal`
`float`	`float`
`float`	`java.lang.Float` (if `nillable="true"`)
`double`	`double`
`double`	`java.lang.Double` (if `nillable="true"`)

XML Schema type	SOAP Java type
boolean	boolean
boolean	java.lang.Boolean (if nillable="true")
byte	byte
byte	java.lang.Byte (if nillable="true")
dateTime	java.util.GregorianCalendar
base64Binary	byte[]
hexBinary	byte[]
time	java.util.GregorianCalendar
date	java.util.GregorianCalendar
anySimpleType	java.lang.String
any	javax.xml.soap.SOAPElement

Element that is *nillable*, meaning that the element CAN BE EMPTY without causing a validation error. 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="true"` attribute is specified.

Example of mapping XML Schema-Java class

XML Schema:

```
<schema>
  <complexType name="Address">
    <sequence>
      <element name="street" nillable="true" type="xsd:string"/>
      <element name="city" nillable="true" type="xsd:string"/>
      <element name="state" nillable="true" type="xsd:string"/>
      <element name="zip" type="xsd:int"/>
    </sequence>
  </complexType>
</schema>
```

Java class:

```
public class Address {
        public String street;
        public String city;
        public String state;
        public int zip;
}
```

NOTE: Since `zip` is not nillable, it can be primitive, otherwise it would be:

XML Schema:

```
<schema>
  <complexType name="Address">
    <sequence>
      <element name="street" nillable="true" type="xsd:string"/>
      <element name="city" nillable="true" type="xsd:string"/>
      <element name="state" nillable="true" type="xsd:string"/>
      <element name="zip" nillable="true" type="xsd:int"/>
    </sequence>
  </complexType>
</schema>
```

Java class:

```
public class Address {
        public String street;
        public String city;
        public String state;
        public Integer zip;
}
```

In XML Schema, we can use `nillable` attribute to indicate that whether the element's content could be `nil`, as in `<xsd:element name="birthDate" type="xsd:date" nillable="true"/>`. If the content of an element is `nil`, we can use `xsi:nil` attribute to signal the processor, as in `<birthDate xsi:nil="true" />` and this element must not contain any content.
More examples on `nillable` attribute. Consider following XML Schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="rootElement" nillable="true">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="myElement" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

VALID:

```
<rootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <myElement>I am valid</myElement>
</rootElement>
```

VALID:

```
<rootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:nil="true"  />
```

INVALID (when `xsi:nil` is `true`, the element MUST BE EMPTY):

```
<rootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:nil="true">
        <myElement>I am NOT valid</myElement>
</rootElement>
```

INVALID (element `rootElement` MUST have `myElement` child and `xsi:nil` has not been set to `true`):

```
<rootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" />
```

Mapping arbitrary XML content to Java
The `<xsd:any/>` element is an element that represents arbitrary XML content within an XML document. It is what its name indicates: any kind of XML. This lets you create complex type definitions in an XML Schema without describing what the exact structure of certain parts of the complex type is. Here is an example that shows the definition of a type called `Order`. It contains two regular elements and one `<xsd:any/>` element:

```
<schema elementFormDefault="qualified"
   xmlns="http://www.w3.org/2001/XMLSchema">

   <complexType name="Order">
      <sequence>
         <element name="date" nillable="true" type="xsd:dateTime"/>
         <element name="customer" nillable="true" type="xsd:string"/>
         <xsd:any maxOccurs="unbounded"/>
      </sequence>
   </complexType>
</schema>
```

An instance of this type can contain any number of additional XML elements without violating the schema definition. You can add additional information to an Order element without defining its format in the schema. The JAX-RPC 1.1 specification defines that `<xsd:any/>` element is mapped to the SAAJ's `javax.xml.soap.SOAPElement` interface. This means that the Service Endpoint Interface [SEI] will contain a parameter or return value of type javax.xml.soap.SOAPElement for each place in the schema where <xsd:any/> was used and, respectively, javax.xml.soap.SOAPElement[] if the maxOccurs attribute is bigger than 1. Therefore, a JAX-RPC tool will generate the following class from the sample schema above:

```
public class Order implements java.io.Serializable {
    private java.util.GregorianCalendar date;
    private java.lang.String customer;
    private javax.xml.soap.SOAPElement[] _any;
    ...
}
```

This approach can be usefule when your Web service uses some data that you don't want to be mapped into a Java class, but rather want to let the JAX-RPC engine hand it to the implementation in its XML form. The implementation could then parse it or simply pass it on as XML for further processing in the backend application. Similarly, you can create a client proxy that lets you pass in a SOAPElement rather than a mapped Java object.

Describe how SOAP message header blocks are used and processed.

SOAP provides a flexible mechanism for extending a message in a decentralized and modular way without prior knowledge between the communicating parties. Typical examples of extensions that can be implemented as header entries are authentication, transaction management, payment etc.

The Header element is encoded as the first immediate child element of the SOAP Envelope XML element. NOTE: The Header element is OPTIONAL. All immediate child elements of the Header element are called header entries (Header element may contain multiple child elements - *header entries*). Header entries can have following attributes: actor, encodingStyle, mustUnderstand.

The encoding rules for header entries are as follows:

1. A header entry is identified by its fully qualified element name, which consists of the namespace URI and the local name. All immediate child elements of the SOAP Header element MUST be namespace-qualified.
2. The SOAP encodingStyle attribute MAY be used to indicate the encoding style used for the header entries.
3. The SOAP mustUnderstand attribute and SOAP actor attribute MAY be used to indicate how to process the entry and by whom.

The SOAP Header attributes defined in this section determine how a recipient of a SOAP message should process the message. A SOAP application generating a SOAP message SHOULD only use the SOAP Header attributes on immediate child elements of the SOAP Header element. The recipient of a SOAP message MUST ignore all SOAP Header attributes that are not applied to an immediate child element of the SOAP Header element.

An example is a header with an element identifier of "Transaction", a "mustUnderstand" value of "1", and a value of 12345. This would be encoded as follows:

```
<SOAP-ENV:Header>
  <t:Transaction xmlns:t="some-URI" SOAP-ENV:mustUnderstand="1">
    12345
  </t:Transaction>
</SOAP-ENV:Header>
```

or

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

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    <t:Transaction xmlns:t="some-URI" soap:mustUnderstand="1">
      12345
    </t:Transaction>
  </soap:Header>
  ...
  ...
</soap:Envelope>
```

A SOAP message travels from the originator to the ultimate destination, potentially by passing through a set of SOAP intermediaries along the message path. A SOAP intermediary is an application that is capable of both receiving and forwarding SOAP messages. Both intermediaries as well as the ultimate destination are identified by a URI. Not all parts of a SOAP message may be intended for the ultimate destination of the SOAP message but, instead, may be intended for one or more of the intermediaries on the message path. The role of a recipient of a header element is similar to that of accepting a contract in that it cannot be extended beyond the recipient. That is, a recipient receiving a header element MUST NOT forward that header element to the next application in the SOAP message path. The recipient MAY insert a similar header element but in that case, the contract is between that application and the recipient of that header element. The SOAP `actor` global attribute can be used to indicate the recipient of a header element. The value of the SOAP `actor` attribute is a URI:

```
SOAP-ENV:actor="some-URI"
```

or

```
soap:actor="some-URI"
```

The special URI "`http://schemas.xmlsoap.org/soap/actor/next`" indicates that the header element is intended for the very first SOAP application that processes the message. This is similar to the hop-by-hop scope model represented by the `Connection` header field in HTTP. NOTE: Omitting the SOAP `actor` attribute indicates that the recipient is the ultimate destination of the SOAP message. This attribute MUST appear in the SOAP message instance in order to be effective.
An example:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    <t:Transaction xmlns:t="some-URI"
                   soap:actor="http://myserver.com/myactor">
      12345
    </t:Transaction>
  </soap:Header>
  ...
  ...
</soap:Envelope>
```

In this example header will be processed by first application:

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

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    <t:Transaction xmlns:t="some-URI"
                   soap:actor="http://schemas.xmlsoap.org/soap/actor/next">
      12345
    </t:Transaction>
  </soap:Header>
  ...
  ...
</soap:Envelope>
```

The SOAP `mustUnderstand` global attribute can be used to indicate whether a header entry is mandatory or optional for the recipient to process. The recipient of a header entry is defined by the SOAP `actor` attribute. The value of the `mustUnderstand` attribute is either "1" or "0".

```
SOAP-ENV:mustUnderstand="0|1"
```

or

```
soap:mustUnderstand="0|1"
```

The absence of the SOAP `mustUnderstand` attribute is semantically equivalent to its presence with the value "0". If a header element is tagged with a SOAP `mustUnderstand` attribute with a value of "1", the recipient of that header entry either MUST obey the semantics (as conveyed by the fully qualified name of the element) and process correctly to those semantics, or MUST fail processing the message. The SOAP `mustUnderstand` attribute allows for robust evolution. Elements tagged with the SOAP `mustUnderstand` attribute with a value of "1" MUST be presumed to somehow modify the semantics of their parent or peer elements. Tagging elements in this manner assures that this change in semantics will not be silently (and, presumably, erroneously) ignored by those who may not fully understand it. This attribute MUST appear in the instance in order to be effective.
An example:

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

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    <t:Transaction xmlns:t="some-URI"
                   soap:actor="http://myserver.com/myactor"
                   soap:mustUnderstand="1">
      12345
    </t:Transaction>
  </soap:Header>
  ...
  ...
```

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.

A SOAP message is an XML document that consists of a mandatory SOAP envelope, an optional SOAP header, and a mandatory SOAP body. The namespace identifier for the elements and attributes from SOAP message is "`http://schemas.xmlsoap.org/soap/envelope/`". A SOAP message contains the following:

- The `Envelope` is the top element of the XML document representing the message.
- The `Header` is a generic mechanism for adding features to a SOAP message in a decentralized manner without prior agreement between the communicating parties. SOAP defines a few attributes that can be used to indicate who should deal with a feature and whether it is optional or mandatory. NOTE: `Header` element is OPTIONAL.
- The `Body` is a container for mandatory information intended for the ultimate recipient of the message. SOAP defines one element for the body, which is the `Fault` element used for reporting errors. NOTE: `Body` element is MANDATORY and MUST be exactly 1 per message.

The grammar rules are as follows:

1. Envelope
   - The element name is "`Envelope`".
   - The element MUST be present in a SOAP message.
   - The element MAY contain namespace declarations as well as additional attributes. If present, such additional attributes MUST be namespace-qualified. Similarly, the element MAY contain additional sub elements. If present these elements MUST be namespace-qualified and MUST follow the SOAP Body element.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
           targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">

  <xs:complexType name="Envelope">
    <xs:sequence>
      <xs:element ref="tns:Header" minOccurs="0" />
      <xs:element ref="tns:Body" minOccurs="1" />
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
processContents="lax" />
   </xs:sequence>
   <xs:anyAttribute namespace="##other" processContents="lax" />
  </xs:complexType>
  ....
</xs:schema>

```

2. Header
   - The element name is "`Header`".
   - The element MAY be present in a SOAP message. If present, the element MUST be the first immediate child element of a SOAP `Envelope` element.

- The element MAY contain a set of header entries each being an immediate child element of the SOAP Header element. All immediate child elements of the SOAP Header element MUST be namespace-qualified.
  NOTE: WS-I BP 1.0 requires all immediate children of Header element be namespace qualified.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
           targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
  ....
  <xs:complexType name="Header">
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##other" processContents="lax" />
  </xs:complexType>
  ....
</xs:schema>
```

3. Body
   - The element name is "Body".
   - The element MUST be present in a SOAP message and MUST be an immediate child element of a SOAP Envelope element. It MUST directly follow the SOAP Header element if present. Otherwise it MUST be the first immediate child element of the SOAP Envelope element.
   - The element MAY contain a set of body entries each being an immediate child element of the SOAP Body element. Immediate child elements of the SOAP Body element MAY be namespace-qualified. SOAP defines the SOAP Fault element, which is used to indicate error messages.
     NOTE: WS-I BP 1.0 requires all immediate children of Body element be namespace qualified.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
           targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
  ....
  <xs:complexType name="Body">
    <xs:sequence>
      <xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded"
processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##any" processContents="lax">
      <xs:annotation>
        <xs:documentation>
          Prose in the spec does not specify that attributes are allowed on
          the Body element
        </xs:documentation>
      </xs:annotation>
    </xs:anyAttribute>
  </xs:complexType>
  ....
</xs:schema>
```

The SOAP Body element provides a simple mechanism for exchanging mandatory information intended for the ultimate recipient of the message. Typical uses of the Body element include marshalling RPC calls and error reporting. The Body element is encoded as an immediate child element of the SOAP Envelope XML element. If a Header element is present then the Body element MUST immediately follow the Header element, otherwise it MUST be the first immediate child element of the Envelope element. All immediate child elements of the Body element are called body entries and each body entry is encoded as an independent element within the SOAP Body element. The encoding rules for body entries are as follows:

1. A body entry is identified by its fully qualified element name, which consists of the namespace URI and the local name. Immediate child elements of the SOAP Body element MAY be namespace-qualified.
   NOTE: WS-I BP 1.0 requires all immediate children of Body element be namespace qualified.

2. The SOAP `encodingStyle` attribute MAY be used to indicate the encoding style used for the body entries (this attribute MAY appear on any element, and is scoped to that element's contents and all child elements not themselves containing such an attribute, much as an XML namespace declaration is scoped).

SOAP Fault

SOAP defines one body entry, which is the `Fault` entry used for reporting errors. Here is schema definition of `Fault` element:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
           targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
  ....
  <xs:complexType name="Fault" final="extension">
    <xs:annotation>
      <xs:documentation>Fault reporting structure</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="faultcode" type="xs:QName" />
      <xs:element name="faultstring" type="xs:string" />
      <xs:element name="faultactor" type="xs:anyURI" minOccurs="0" />
      <xs:element name="detail" type="tns:detail" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
  ....
</xs:schema>
```

The SOAP `Fault` element is used to carry error and/or status information within a SOAP message. If present, the SOAP `Fault` element MUST appear as a body entry and MUST NOT appear more than once within a `Body` element. The SOAP `Fault` element defines the following four subelements:

1. **faultcode** - The `faultcode` element is intended for use by software to provide an algorithmic mechanism for identifying the fault. The `faultcode` MUST be present in a SOAP `Fault` element and the `faultcode` value MUST be a qualified name. SOAP defines a small set of SOAP fault codes covering basic SOAP faults.
2. **faultstring** - The `faultstring` element is intended to provide a human readable explanation of the fault and is not intended for algorithmic processing. The `faultstring` element is similar to the '`Reason-Phrase`' defined by HTTP. It MUST be present in a SOAP `Fault` element and SHOULD provide at least some information explaining the nature of the fault.
3. **faultactor** - The `faultactor` element is intended to provide information about who caused the fault to happen within the message path. It is similar to the SOAP `actor` attribute but instead of indicating the destination of the header entry, it indicates the source of the fault. The value of the `faultactor` attribute is a URI identifying the source. Applications that do not act as the ultimate destination of the SOAP message MUST include the `faultactor` element in a SOAP `Fault` element. The ultimate destination of a message MAY use the `faultactor` element to indicate explicitly that it generated the fault.
4. **detail** - The `detail` element is intended for carrying application specific error information related to the Body element. It MUST be present if the contents of the Body element could not be successfully processed. It MUST NOT be used to carry information about error information belonging to header entries. Detailed error information belonging to header entries MUST be carried within header entries. The absence of the `detail` element in the `Fault` element indicates that the fault is not related to processing of the Body element. This can be used to distinguish whether the Body element was processed or not in case of a fault situation. All immediate child elements of the detail element are called *detail entries* and each detail entry is encoded as an independent element within the `detail` element.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
           targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
  ....
  <xs:complexType name="detail">
    <xs:sequence>
      <xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded"
processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##any" processContents="lax" />
  </xs:complexType>
  ....
</xs:schema>
```

The encoding rules for *detail entries* are as follows:

a. A *detail entry* is identified by its fully qualified element name, which consists of the namespace URI and the local name. Immediate child elements of the `detail` element MAY be namespace-qualified.
b. The SOAP `encodingStyle` attribute MAY be used to indicate the encoding style used for the detail entries

The `faultcode` values MUST be used in the `faultcode` element when describing faults. The namespace identifier for these `faultcode` values is "`http://schemas.xmlsoap.org/soap/envelope/`". The default SOAP `faultcode` values are defined in an extensible manner that allows for new SOAP faultcode values to be defined while maintaining backwards compatibility with existing `faultcode` values. The mechanism used is very similar to the `1xx`, `2xx`, `3xx` etc basic status classes classes defined in HTTP. However, instead of integers, they are defined as XML qualified names. The character "." (dot) is used as a separator of `faultcode` values indicating that what is to the left of the dot is a more generic fault code value than the value to the right. Example:

```
Client.Authentication
```

NOTE: WS-I BP 1.0 PROHIBITS the use of "dot" notation of `faultcode` element.
The set of predefined `faultcode` values is:

Table 2.2. SOAP Fault Codes

Error	Description
`VersionMismatch`	The processing party found an invalid namespace for the SOAP `Envelope` element.
`MustUnderstand`	An immediate child element of the SOAP `Header` element that was either not understood or not obeyed by the processing party contained a SOAP `mustUnderstand` attribute with a value of "`1`".
`Client`	The `Client` class of errors indicate that the message was incorrectly formed or did not contain the appropriate information in order to succeed. For example, the message could lack the proper authentication or payment information. It is generally an indication that the message should not be resent without change.
`Server`	The `Server` class of errors indicate that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to the processing of the message. For example, processing could include communicating with an upstream processor, which didn't respond. The message may succeed at a later point in time.

NOTE: WS-I BP 1.0 ALLOWS using of custom values of `faultcode` element. In this case they MUST be fully qualified:

```
<faultcode>ns1:ProcessingError</faultcode>
```

Using SOAP in HTTP

Binding SOAP to HTTP provides the advantage of being able to use the formalism and decentralized flexibility of SOAP with the rich feature set of HTTP. Carrying SOAP in HTTP does not mean that SOAP overrides existing semantics of HTTP but rather that the semantics of SOAP over HTTP maps naturally to HTTP semantics.
SOAP naturally follows the HTTP request/response message model providing SOAP request parameters in a HTTP request and SOAP response parameters in a HTTP response. Note, however, that SOAP intermediaries are NOT the same as HTTP intermediaries. That is, an HTTP intermediary addressed with the HTTP `Connection` header field cannot be expected to inspect or process the SOAP entity body carried in the HTTP request.
HTTP applications MUST use the media type "`text/xml`" when including SOAP entity bodies in HTTP messages.
The `SOAPAction` HTTP request header field can be used to indicate the intent of the SOAP HTTP request. The value is a URI identifying the intent. SOAP places no restrictions on the format or specificity of the URI or that it is resolvable. An HTTP client MUST use this header field when issuing a SOAP HTTP Request.
The presence and content of the `SOAPAction` header field can be used by servers such as firewalls to appropriately filter SOAP request messages in HTTP. The header field value of empty string ("") means that the intent of the SOAP message is provided by the HTTP Request-URI. No value means that there is no indication of the intent of the message. Examples:
CORRECT:

```
SOAPAction: "http://electrocommerce.org/abc#MyMessage"
```

CORRECT:

```
SOAPAction: "myapp.sdl"
```

CORRECT (empty quoted string):

```
SOAPAction: ""
```

INCORRECT (SOAPAction value MUST be a quoted string):

```
SOAPAction:
```

SOAP HTTP follows the semantics of the HTTP Status codes for communicating status information in HTTP. For example, for a two-way operations a 2xx status code indicates that the client's request including the SOAP component was successfully received, understood, and accepted etc. In case of a SOAP error while processing the request, the SOAP HTTP server MUST issue an HTTP 500 "Internal Server Error" response and include a SOAP message in the response containing a SOAP Fault element indicating the SOAP processing error (for two-way operations only).

Below is an example of HTTP request, successful response and fault response of simple Web Service for retrieving Quote information.

Quote SOAP HTTP request:

```
POST /StockQuoteProj/servlet/rpcrouter HTTP/1.0
Host: localhost:9080
Content-Type: text/xml; charset=utf-8
Content-Length: 469
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getQuote xmlns:ns1="http://tempuri.org/StockQuoteService"
                  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <symbol xsi:type="xsd:string">ibm</symbol>
    </ns1:getQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

NOTE: WS-I BP 1.0 MANDATES the use of HTTP POST method for SOAP messages sending.

Quote SOAP HTTP successful response:

```
HTTP/1.1 200 OK
Server: WebSphere Application Server/5.1
Content-Type: text/xml; charset=utf-8
Content-Length: 488
Content-Language: ru-RU
Connection: close

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getQuoteResponse xmlns:ns1="http://tempuri.org/StockQuoteService"
             SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:float">93.12</return>
    </ns1:getQuoteResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Quote SOAP HTTP fault response (exception is thrown by service endpoint object):

```
HTTP/1.1 500 Internal Server Error
Server: WebSphere Application Server/5.1
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Set-Cookie: JSESSIONID=0000XEtXnz75hI--bFdY49XCHGU:-1;Path=/
Cache-Control: no-cache="set-cookie,set-cookie2"

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>Exception from service object: null</faultstring>
      <faultactor>/StockQuoteProj/servlet/rpcrouter</faultactor>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example of SOAP HTTP request Using POST with a Mandatory Header:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
            SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Header>
    <t:Transaction xmlns:t="some-URI" SOAP-ENV:mustUnderstand="1">
      12345
    </t:Transaction>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DEF</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example of SOAP HTTP request Using POST with multiple request parameters:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
            SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Body>
    <m:GetLastTradePriceDetailed xmlns:m="Some-URI">
      <Symbol>IBM</Symbol>
      <Company>IBM Corp</Company>
    </m:GetLastTradePriceDetailed>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example of SOAP HTTP response with a Mandatory Header:

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

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
           SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Header>
    <t:Transaction xmlns:t="some-URI" xsi:type="xsd:int" mustUnderstand="1">
      12345
    </t:Transaction>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
      <Price>134.5</Price>
     </m:GetLastTradePriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example of SOAP HTTP response with a Struct:

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

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
           SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
      <PriceAndVolume>
        <LastTradePrice>
          134.5
        </LastTradePrice>
        <DayVolume>
          10000
        </DayVolume>
      </PriceAndVolume>
    </m:GetLastTradePriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example of SOAP HTTP response Failing to honor Mandatory Header:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
           SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:MustUnderstand</faultcode>
      <faultstring>SOAP Must Understand Error</faultstring>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example of SOAP HTTP response Failing to handle Body:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
           SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>Server Error</faultstring>
      <detail>
        <e:myfaultdetails xmlns:e="Some-URI">
          <message>
            My application didn't work
          </message>
          <errorcode>
            1001
          </errorcode>
        </e:myfaultdetails>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Create a SOAP message that contains an attachment.

The SOAP with Attachments API for Java (SAAJ) provides a standard way to send XML documents over the Internet from the Java platform. It is based on the SOAP 1.1 and SOAP with Attachments specifications, which define a basic framework for exchanging XML messages.

The process of creation and sending SOAP message includes following steps:

- Creating a SOAP connection
- Creating a SOAP message
- Populating the message
- Sending the message
- Retrieving the reply

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. Request-response messaging is synchronous, meaning that a request is sent and its response is received in the same operation. 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 represents an XML document that is a SOAP message. 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 represents an XML document that is a SOAP message. 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. There is no limitation on the content in the attachment part, so it can include images or any other kind of content, including XML fragments and documents. Common types of attachment include sound, picture, and movie data: .mp3, .jpg, and .mpg files.

The first thing a SAAJ client needs to do is get a connection in the form of a `SOAPConnection` object. A `SOAPConnection` object is a point-to-point connection that goes directly from the sender to the recipient. 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();
```

The client can use `factory` to create a `SOAPConnection` object.

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

Messages, like connections, are created by a factory. To obtain a MessageFactory object, you get an instance of the default implementation for the MessageFactory class. This instance can then be used to create a SOAPMessage object:

```
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, and the following sections will illustrate some of the typical ways to add content. Content can be added to the SOAPPart object, to one or more AttachmentPart objects, or to both parts of a message.

```
package javax.xml.soap;
public abstract class SOAPPart implements org.w3c.dom.Document {
    public abstract SOAPEnvelope getEnvelope() throws SOAPException;
    ...
}
```

As stated earlier, all messages have a SOAPPart object, which has a SOAPEnvelope object containing a SOAPHeader object and a SOAPBody object.

```
package javax.xml.soap;

public interface SOAPEnvelope extends SOAPElement {
    public abstract Name createName(String localName, String prefix, String uri)
        throws SOAPException;
    public abstract Name createName(String localName) throws SOAPException;
    public abstract SOAPHeader getHeader() throws SOAPException;
    public abstract SOAPHeader addHeader() throws SOAPException;
    public abstract SOAPBody getBody() throws SOAPException;
    public abstract SOAPBody addBody() throws SOAPException;
}
```

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. The first three lines of the following code fragment access the SOAPBody object body, which is used to create a new SOAPBodyElement object and add it to body. The argument passed to the createName method is a Name object identifying the SOAPBodyElement being added. The last line adds the XML string passed to the method 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.

The following code fragments illustrates adding content as a DOMSource object. The first step is to get the SOAPPart object from the SOAPMessage object. Next the code uses methods from the JAXP API to build the XML document to be added. It uses a DocumentBuilderFactory object to get a DocumentBuilder object. Then it parses the given file to produce the document that will be used to initialize a new DOMSource object. Finally, the code passes the DOMSource object domSource to the method SOAPPart.setContent:

```
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);
```

This code would work equally well with a `SAXSource` or a `StreamSource` object.

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);
```

This allows you to keep your application data in a document that is separate from the SOAP envelope unless and until it is time to send that data as a message.

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. There may be any number of attachment parts, and they may contain anything from plain text to image files. 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. The `Message` object message creates the `AttachmentPart` object `attachPart`, which is initialized with the data handler containing the URL for the image. Finally, the message adds `attachPart` to itself:

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

A `SOAPMessage` object can also give content to an `AttachmentPart` object by passing an `Object` and its content type to the method `createAttachmentPart`:

```
AttachmentPart attachPart = message.createAttachmentPart(
    "content-string", "text/plain");
message.addAttachmentPart(attachPart);
```

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 arguments to the method `call` are the message being sent and a URL object that contains the URL specifying the endpoint of the receiver.

```
SOAPMessage response = soapConnection.call(message, endpoint);
```

The following example shows a SOAP 1.1 message with an attached facsimile image of the signed claim form (`claim.tiff`), also it illustrates the use of the `cid` reference in the body of the SOAP 1.1 message:

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
        start="<claim.xml@claiming-it.com>"
Content-Description: This is the optional message description.
--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim.xml@claiming-it.com>
<?xml version='1.0' ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    ..
    <theSignedForm href="cid:claim.tiff@claiming-it.com"/>
    ..
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
--MIME_boundary
Content-Type: image/tiff
Content-Transfer-Encoding: binary
Content-ID: <claim.tiff@claiming-it.com>
d3d3Lm1hcmNoYWwuY29taesgfSEVFES45345sdvgfszd==
--MIME_boundary--
```

NOTE: In this example the "`Content-Type`" header line has been continued across two lines so the example prints easily. SOAP message senders should send headers on a single long line.
NOTE: Associate the attachment to `theSignedForm` element by adding an `href` attribute. The attachment is referred to through a `cid` (`Content-ID`) URL:

```
...
<myElement href="cid:xxxx" />
...
--MIME_boundary
Content-Type: image/tiff
Content-Transfer-Encoding: binary
Content-ID: <xxxx>
...
```

NOTE: Unlike SOAP 1.1 XML references, do not use '#' (pound sign) in `href` attribute value when you refer to attachment, also when you refer to attachment you must prepend unique identifier with prefix '`cid:`'.
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. Using SAAJ API:

```
// Using SAAJ
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--
```

Describe the restrictions placed on the use of SOAP by the WS-I Basic Profile 1.0a.

BP 1.0 - Messaging - XML Representation of SOAP Messages.

When a MESSAGE contains a `soap:Fault` element, that element MUST NOT have element children other than `faultcode`, `faultstring`, `faultactor` and `detail`.

CORRECT:

```
<soap:Fault xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' >
  <faultcode>soap:Client</faultcode>
  <faultstring>Invalid message format</faultstring>
  <faultactor>http://example.org/someactor</faultactor>
  <detail>
     <m:msg xmlns:m='http://example.org/faults/exceptions'>
         There were <b>lots</b> of elements in the message that I did not understand
     </m:msg>
     <m:Exception xmlns:m='http://example.org/faults/exceptions'>
       <m:ExceptionType>Severe</m:ExceptionType>
     </m:Exception>
   </detail>
</soap:Fault>
```

INCORRECT (not allowed child element inside `Fault`):

```
<soap:Fault xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' >
  <faultcode>soap:Client</faultcode>
  <faultstring>Invalid message format</faultstring>
  <faultactor>http://example.org/someactor</faultactor>
  <detail>There were <b>lots</b> of elements in the message
    that I did not understand
  </detail>
  <m:Exception xmlns:m='http://example.org/faults/exceptions' >
    <m:ExceptionType>Severe</m:ExceptionType>
  </m:Exception>
</soap:Fault>
```

When a MESSAGE contains a `soap:Fault` element its element children MUST be unqualified.
CORRECT:

```
<soap:Fault xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'   xmlns='' >
  <faultcode>soap:Client</faultcode>
  <faultstring>Invalid message format</faultstring>
  <faultactor>http://example.org/someactor</faultactor>
  <detail>
      <m:msg xmlns:m='http://example.org/faults/exceptions'>
          There were <b>lots</b> of elements in the message that
          I did not understand
      </m:msg>
  </detail>
</soap:Fault>
```

INCORRECT (child elements have namespace prefixes):

```
<soap:Fault xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' >
  <soap:faultcode>soap:Client</soap:faultcode>
  <soap:faultstring>Invalid message format</soap:faultstring>
  <soap:faultactor>http://example.org/someactor</soap:faultactor>
  <soap:detail>
      <m:msg xmlns:m='http://example.org/faults/exceptions'>
          There were <b>lots</b> of elements in the message that
          I did not understand
      </m:msg>
  </soap:detail>
</soap:Fault>
```

A RECEIVER MUST accept fault messages that have any number of ELEMENTS, including zero, appearing as children of the `detail` element. Such children can be qualified or unqualified.
A RECEIVER MUST accept fault messages that have any number of qualified or unqualified ATTRIBUTES, including zero, appearing on the `detail` element. The namespace of qualified attributes can be anything other than `"http://schemas.xmlsoap.org/soap/envelope/"`.
A RECEIVER MUST accept fault messages that carry an `xml:lang` attribute on the `faultstring` element.
When a MESSAGE contains a `faultcode` element the content of that element SHOULD be one of the fault codes defined in SOAP 1.1 or a namespace qualified fault code.
SOAP 1.1 defines following `faultcode` values:

Table 2.3. SOAP Fault Codes

Error	Description
`VersionMismatch`	The processing party found an invalid namespace for the SOAP `Envelope` element.
`MustUnderstand`	An immediate child element of the SOAP `Header` element that was either not understood or not obeyed by the processing party contained a SOAP `mustUnderstand` attribute with a value of "1".
`Client`	The Client class of errors indicate that the message was incorrectly formed or did not contain the appropriate information in order to succeed. For example, the message could lack the proper authentication or payment information. It is generally an indication that the message should not be resent without change.

Error	Description
`Server`	The Server class of errors indicate that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to the processing of the message. For example, processing could include communicating with an upstream processor, which didn't respond. The message may succeed at a later point in time.

When a MESSAGE contains a `faultcode` element the content of that element SHOULD NOT use of the SOAP 1.1 "dot" notation to refine the meaning of the Fault.

CORRECT (use of custom namespace qualified value):

```
<soap:Fault xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
            xmlns:c='http://example.org/faultcodes' >
  <faultcode>c:ProcessingError</faultcode>
  <faultstring>An error occured while processing the message
  </faultstring>
</soap:Fault>
```

CORRECT (use of predefined SOAP 1.1 value):

```
<soap:Fault xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' >
  <faultcode>soap:Server</faultcode>
  <faultstring>An error occured while processing the message
  </faultstring>
</soap:Fault>
```

INCORRECT ("dot" notation):

```
<soap:Fault xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
            xmlns:c='http://example.org/faultcodes' >
  <faultcode>soap:Server.ProcessingError</faultcode>
  <faultstring>An error occurred while processing the message
  </faultstring>
</soap:Fault>
```

A MESSAGE MUST NOT contain `soap:encodingStyle` attributes on any of the elements whose namespace name is "`http://schemas.xmlsoap.org/soap/envelope/`".

A MESSAGE MUST NOT contain `soap:encodingStyle` attributes on any element that is a child of `soap:Body`.

A MESSAGE described in an rpc-literal binding MUST NOT contain `soap:encodingStyle` attribute on any elements are grandchildren of `soap:Body`.

A MESSAGE MUST NOT contain a Document Type Declaration (DTD).

A MESSAGE MUST NOT contain Processing Instructions (PI).

A RECEIVER MUST accept messages that contain an XML Declaration.

A MESSAGE MUST NOT have any element children of `soap:Envelope` following the `soap:Body` element.

CORRECT:

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' >
  <soap:Body>
    <p:Process xmlns:p='http://example.org/Operations' >
         <m:Data xmlns:m='http://example.org/information' >
  Here is some data with the message
      </m:Data>
    </p:Process>
  </soap:Body>
</soap:Envelope>
```

INCORRECT (child elements after Body element):

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' >
  <soap:Body>
    <p:Process xmlns:p='http://example.org/Operations' />
  </soap:Body>
  <m:Data xmlns:m='http://example.org/information' >
    Here is some data with the message
  </m:Data>
</soap:Envelope>
```

A MESSAGE MUST be serialized as either UTF-8 or UTF-16.
The media type of a MESSAGE's envelope MUST indicate the correct character encoding, using the `charset` parameter.
A MESSAGE containing a `soap:mustUnderstand` attribute MUST only use the lexical forms "0" and "1".
The children of the `soap:Body` element in a MESSAGE MUST be namespace qualified.
A RECEIVER MUST generate a fault if they encounter a message whose document element has a local name of "`Envelope`" but a namespace name that is not "`http://schemas.xmlsoap.org/soap/envelope/`".
A RECEIVER MUST NOT mandate the use of the `xsi:type` attribute in messages except as required in order to indicate a derived type.

Describe the function of SOAP in a Web service interaction and the advantages and disadvantages of using SOAP messages.

HTTP is a transport-level protocols and SOAP is a messaging-layer (communication) protocol. SOAP can be used in combination with a variety of transport protocols - including SMTP, JMS, and other protocols in addition to HTTP - and does not depend on any particular network protocol. Although HTTP is a widely used protocol for SOAP, SOAP toolkit vendors have also started providing support for other protocols, like SMTP. SOAP messages may travel across several different transport-layer protocols before they reach their ultimate destination

SOAP advantages.

- Platform independent.
  SOAP decouples the encoding and communications protocol from the runtime environment. Web service can receive a SOAP payload from a remote service, and the platform details of the source are entirely irrelevant.
- Language independent.
  Anything can generate XML, from Perl scripts to C++ code to J2EE app servers. So, as of the 1.1 version of the SOAP specification, anyone and anything can participate in a SOAP conversation, with a relatively low barrier to entry.
- Uses XML to send and receive messages.
  SOAP is also a simple way to accomplish remote object/component/service communications. It formalizes the vocabulary definition in a form that's now familiar, popular, and accessible (XML). If you know XML, you can figure out the basics of SOAP encoding pretty quickly.
- Uses standard internet HTTP protocol.
  SOAP runs over HTTP, which eliminates firewall problems. When using HTTP as the protocol binding, an RPC call maps naturally to an HTTP request and an RPC response maps to an HTTP response.
- SOAP is very simple compared to RMI, CORBA, and DCOM because it does not deal with certain ancillary but important aspects of remote object systems.
- A protocol for exchanging information in a decentralized and distributed environment.
- SOAP is, transport protocol-independent and can therefore potentially be used in combination with a variety of protocols.
- Vendor neutral.

SOAP disadvantages

- The SOAP specification contains no mention of security facilities.
- SOAP 1.1 specification does not specify a default encoding for the message body. There is an encoding defined in the spec, but it is not required that you use this encoding to be compliant: Any custom encoding that you choose can be specified in the `encodingStyle` attribute of the message or of individual elements in the message.
- Because SOAP deals with objects serialized to plain text and not with stringified remote object references (interoperable object references, IORs, as defined in CORBA), distributed garbage collection has no meaning.
- SOAP clients do not hold any stateful references to remote objects.

Chapter 3. Describing and Publishing (WSDL and UDDI)

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.

WSDL is an XML-based language that allows formal XML desriptions of the interfaces of Web services:

- Interface information describing all publicly available functions.
- Data type information for all message requests and message responses.
- Binding information about the transport protocol to be used.
- Address information for locating the specified service.

WSDL benefits:

- It is an interface description is a contract between the server developers and the client developers (like Java interface represents a contract between client code and the actual Java object).
- It has formal descriptions which allows tool support, e.g. code template generators, integrate new services with little or no manual code.

WSDL language can be described as having two layers:

1. The service definition layer describes abstract properties:
    - data types
    - message types
    - operations
    - services
2. The binding layer describes concrete properties:
    - protocols
    - data formats

The `definitions` element MUST be the root element of all WSDL documents. It defines the name of the web service, declares multiple namespaces used throughout the remainder of the document. An actual WSDL document consists of a set of `definitions` of the following kinds:

- `types` - Contains XML Schema element and type definitions. The `types` element describes all the data types used between the client and server. WSDL is not tied exclusively to a specific typing system, but it uses the W3C XML Schema specification as its default choice. If the service uses only XML Schema built-in simple types, such as strings and integers, the `types` element is not required.
- `message` - Consistes of either a number of named `parts` typed by XML Schema elements, or a single `part` typed by a XML Schema type. The `message` element describes a one-way message, whether it is a single message request or a single message response. It defines the name of the message and contains zero or more message `part` elements, which can refer to message parameters or message return values.
- `portType` - describing a set of `operations`, each being either:
    - one-way: The endpoint receives an `input` message. (NOTE: The WS-I BP 1.0 restricts the valid `wsdl:operations` to one-way and request-response operations).

```
<wsdl:definitions .... >
  <wsdl:portType .... > *
    <wsdl:operation name="nmtoken">
      <wsdl:input name="nmtoken"? message="qname"/>
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

    - request-response: The endpoint receives an `input` message and then responds with an `output` message (like RPC - Remote Procedure Call). (NOTE: The WS-I BP 1.0 restricts the valid `wsdl:operations` to one-way and request-response operations).

```
<wsdl:definitions .... >
  <wsdl:portType .... > *
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:input name="nmtoken"? message="qname"/>
      <wsdl:output name="nmtoken"? message="qname"/>
      <wsdl:fault name="nmtoken" message="qname"/>*
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

    - solicit-response: The endpoint sends an `output` message and then receives an `input` message (NOTE: A DESCRIPTION MUST NOT use `Solicit-Response` and `Notification` type operations in a `wsdl:portType` definition - R2303 - BP 1.0).

```
<wsdl:definitions .... >
  <wsdl:portType .... > *
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:output name="nmtoken"? message="qname"/>
      <wsdl:input name="nmtoken"? message="qname"/>
      <wsdl:fault name="nmtoken" message="qname"/>*
    </wsdl:operation>
  </wsdl:portType >
```

```
</wsdl:definitions>
```

- notification: The endpoint sends an `output` message (NOTE: A DESCRIPTION MUST NOT use `Solicit-Response` and `Notification` type operations in a `wsdl:portType` definition - R2303 - BP 1.0).

```
<wsdl:definitions .... >
  <wsdl:portType .... > *
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:output name="nmtoken"? message="qname"/>
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

- The `portType` element combines multiple `message` elements to form a complete one-way or round-trip operation. For example, a `portType` can combine one request and one response message into a single request/response operation, most commonly used in SOAP services. Note that a `portType` can (and frequently does) define multiple `operations`.
- `binding` - Selects communication protocol and data formats for each `operation` and `message`. The `binding` element describes the concrete specifics of how the service will be implemented on the wire. WSDL includes built-in extensions for defining SOAP services, and SOAP-specific information therefore goes here. (NOTE: For interoperability the WS-I BP 1.0 requires that all messages must be sent using the SOAP 1.1 protocol over an HTTP transport as described in Section 3 of the WSDL 1.1 spec. The SOAP messages must be in either "document-literal" or "rpc-literal" form). The WS-I BP 1.0 requires that a `wsdl:binding` and its `wsdl:portType` have the same list of `wsdl:operations`. A perfect matching between the two lists is established through a 1-1 and onto relation from the `wsdl:binding` to the `wsdl:portType`. The `wsdl:binding` should completely bind all operations within a `wsdl:portType`.
- `service` - Describes a collection of named `ports`, each associated with a binding and a network address. The `service` element defines the address for invoking the specified service. Most commonly, this includes a URL for invoking the SOAP service.

The simplified structure of a WSDL document is:

```

<definitions> <!-- root WSDL element -->

  <types>
    <!-- defines data types to be transmitted -->
  </types>

  <message>
    <!-- defines messages to be transmitted -->
  </message>

  <portType>
    <!-- defines operations (functions) to be supported -->
  </portType>

  <binding>
    <!-- defines how will the messages be transmitted on the wire -->
  </binding>

  <service>
    <!-- defines location of web service -->
  </service>

</definitions>
```

WSDL document grammar:

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
    <import namespace="uri" location="uri"/>*
    <wsdl:documentation .... /> ?
    <wsdl:types> ?
        <wsdl:documentation .... />?
        <xsd:schema .... />*
        <-- extensibility element --> *
    </wsdl:types>
    <wsdl:message name="nmtoken"> *
        <wsdl:documentation .... />?
        <part name="nmtoken" element="qname"? type="qname"?/> *
    </wsdl:message>
    <wsdl:portType name="nmtoken">*
        <wsdl:documentation .... />?
        <wsdl:operation name="nmtoken">*
           <wsdl:documentation .... /> ?
           <wsdl:input name="nmtoken"? message="qname">?
               <wsdl:documentation .... /> ?
           </wsdl:input>
           <wsdl:output name="nmtoken"? message="qname">?
               <wsdl:documentation .... /> ?
           </wsdl:output>
           <wsdl:fault name="nmtoken" message="qname"> *
               <wsdl:documentation .... /> ?
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="nmtoken" type="qname">*
        <wsdl:documentation .... />?
        <-- extensibility element --> *
        <wsdl:operation name="nmtoken">*
           <wsdl:documentation .... /> ?
           <-- extensibility element --> *
           <wsdl:input> ?
               <wsdl:documentation .... /> ?
               <-- extensibility element -->
           </wsdl:input>
           <wsdl:output> ?
               <wsdl:documentation .... /> ?
               <-- extensibility element --> *
           </wsdl:output>
           <wsdl:fault name="nmtoken"> *
               <wsdl:documentation .... /> ?
               <-- extensibility element --> *
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="nmtoken"> *
        <wsdl:documentation .... />?
        <wsdl:port name="nmtoken" binding="qname"> *
           <wsdl:documentation .... /> ?
           <-- extensibility element -->
        </wsdl:port>
        <-- extensibility element -->
    </wsdl:service>
    <-- extensibility element --> *
</wsdl:definitions>
```

Example of simple WSDL (SOAP 1.1 Request-Response via HTTP):

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
              xmlns="http://www.w3.org/2000/10/XMLSchema">
           <element name="TradePriceRequest">
              <complexType>
                  <all>
                      <element name="tickerSymbol" type="string"/>
                  </all>
              </complexType>
           </element>
           <element name="TradePrice">
              <complexType>
                  <all>
                      <element name="price" type="float"/>
                  </all>
              </complexType>
           </element>
       </schema>
    </types>
    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>
    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>
    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
           <input message="tns:GetLastTradePriceInput"/>
           <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>
    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
           <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>
    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
           <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>
</definitions>
```

Describe how W3C XML Schema is used as a typing mechanism in WSDL 1.1.

The types element encloses data type definitions that are relevant for the exchanged messages. For maximum interoperability and platform neutrality, WSDL prefers the use of XSD as the canonical type system, and treats it as the intrinsic type system.

```
<definitions .... >
    <types>
        <xsd:schema .... />*
    </types>
</definitions>
```

The XSD type system can be used to define the types in a message regardless of whether or not the resulting wire format is actually XML, or whether the resulting XSD schema validates the particular wire format. This is especially interesting if there will be multiple bindings for the same message, or if there is only one binding but that binding type does not already have a type system in widespread use.

A DESCRIPTION MUST NOT use QName references to elements in namespaces that have been neither imported, nor defined in the referring WSDL document.

A QName reference to a Schema component in a DESCRIPTION MUST use the namespace defined in the `targetNamespace` attribute on the `xsd:schema` element, or to a namespace defined in the namespace attribute on an `xsd:import` element within the `xsd:schema` element.

All `xsd:schema` elements contained in a `wsdl:types` element of a DESCRIPTION MUST have a `targetNamespace` attribute with a valid and non-null value, UNLESS the `xsd:schema` element has `xsd:import` and/or `xsd:annotation` as its only child element(s).

In a DESCRIPTION, array declarations MUST NOT extend or restrict the `soapenc:Array` type.

In a DESCRIPTION, array declarations MUST NOT use `wsdl:arrayType` attribute in the type declaration.

In a DESCRIPTION, array declaration wrapper elements SHOULD NOT be named using the convention `ArrayOfXXX`.

A MESSAGE containing serialized arrays MUST NOT include the `soapenc:arrayType` attribute.

CORRECT:

Given the WSDL Description:

```
<xsd:element name="MyArray1" type="tns:MyArray1Type"/>
<xsd:complexType name="MyArray1Type">
  <xsd:sequence>
   <xsd:element name="x" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

The SOAP message would serialize as (omitting namespace declarations for clarity):

```
<MyArray1>
  <x>abcd</x>
  <x>efgh</x>
</MyArray1>
```

INCORRECT (uses `soapenc:arrayType` attribute and `soapenc:Array` type):

Given the WSDL Description:

```
<xsd:element name="MyArray2" type="tns:MyArray2Type"/>
<xsd:complexType name="MyArray2Type"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" >
  <xsd:complexContent>
     <xsd:restriction base="soapenc:Array">
       <xsd:sequence>
          <xsd:element name="x" type="xsd:string"
           minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute ref="soapenc:arrayType"
        wsdl:arrayType="tns:MyArray2Type[]"/>
   </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>
```

The SOAP message would serialize as (omitting namespace declarations for clarity):

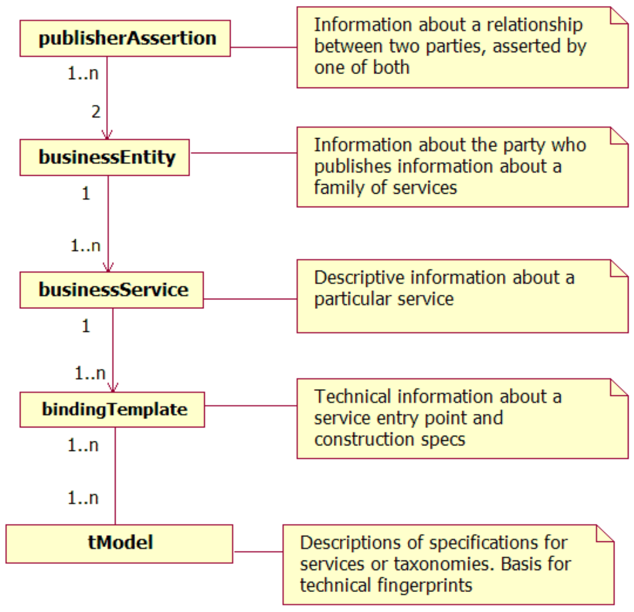
```
<MyArray2 soapenc:arrayType="tns:MyArray2Type[]" >
  <x>abcd</x>
  <x>efgh</x>
</MyArray2>
```

Describe the use of UDDI data structures. Consider the requirements imposed on UDDI by the WS-I Basic Profile 1.0a.

UDDI supports the following core data structures:

1. Business Entity
2. Business Service
3. Binding Template
4. tModel
5. Publisher Assertion

This division by information type provides simple partitions to assist in the rapid location and understanding of the different information that makes up a registration.


publisherAssertion
Information about a relationship between two parties, asserted by one of both
1..n
2
businessEntity
Information about the party who publishes information about a family of services
1
1..n
businessService
Descriptive information about a particular service
1
1..n
bindingTemplate
Technical information about a service entry point and construction specs
1..n
1..n
tModel
Descriptions of specifications for services or taxonomies. Basis for technical fingerprints

The `businessEntity` structure

The `businessEntity` structure represents all known information about a business or entity that publishes descriptive information about the entity as well as the services that it offers. From an XML standpoint, the `businessEntity` is the top-level data structure that accommodates holding descriptive information about a business or entity. Service descriptions and technical information are expressed within a `businessEntity` by a containment relationship.

Structure Specification:

```
<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 businessServices structure provides a way for describing information about families of services. This simple collection accessor contains zero or more businessService structures and has no other associated structures.

The identifierBag element allows businessEntity or tModel structures to include information about common forms of identification such as D-U-N-S numbers, tax identifiers, etc. This data can be used to signify the identity of the businessEntity, or can be used to signify the identity of the publishing party. Including data of this sort is optional, but when used greatly enhances the search behaviors exposed via the find_xx messages defined in the UDDI Version 2.0 API Specification.

The categoryBag element allows businessEntity, businessService and tModel structures to be categorized according to any of several available taxonomy based classification schemes. 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). Including data of this sort is optional, but when used greatly enhances the search behaviors exposed by the find_xx messages defined in the UDDI Version 2.0 API Specification.

The businessService structure

The businessService structures each represent a logical service classification. The name of the element includes the term "business" in an attempt to describe the purpose of this level in the service description hierarchy. Each businessService structure is the logical child of a single businessEntity structure. The identity of the containing (parent) businessEntity is determined by examining the embedded businessKey value. If no businessKey value is present, the businessKey must be obtainable by searching for a businessKey value in any parent structure containing the businessService. Each businessService element contains descriptive information in business terms outlining the type of technical services found within each businessService element.

In some cases, businesses would like to share or reuse services, e.g. when a large enterprise publishes separate businessEntity structures. This can be established by using the businessService structure as a projection to an already published businessService.

Any businessService projected in this way is not managed as a part of the referencing businessEntity, but centrally as a part of the referenced businessEntity. This means that changes of the businessService by the referenced businessEntity are automatically valid for the service projections done by referencing businessEntity structures.

In order to specify both referenced and referencing businessEntity structures correctly, service projections can only be published by a save_business message with the referencing businessKey present in the businessEntity structure and both the referenced businessKey and the referenced businessService present in the businessService structure.

Structure Specification:

```
<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>
```

The bindingTemplates structure is a container for zero or more bindingTemplate structures. This structure holds the technical service description information related to a given business service family.

The categoryBag is an optional element. This is an optional list of name-value pairs that are used to tag a businessService with specific taxonomy information (e.g. industry, product or geographic codes). These can be used during search via find_service.

The bindingTemplate structure

Technical descriptions of Web services are accommodated via individual contained instances of bindingTemplate structures. These structures provide support for determining a technical entry point or optionally support remotely hosted services, as well as a lightweight facility for describing unique technical characteristics of a given implementation. Support for technology and application specific parameters and settings files are also supported.

Since UDDI's main purpose is to enable description and discovery of Web Service information, it is the bindingTemplate that provides the most interesting technical data.
Each bindingTemplate structure has a single logical businessService parent, which in turn has a single logical businessEntity parent.
Structure Specification:

```
<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 is an attribute-qualified pointer to a service entry point. The notion of service at the metadata level seen here is fairly abstract and many types of entry points are accommodated. A single attribute is provided (named URLType). The purpose of the URLType attribute is to facilitate searching for entry points associated with a particular type of entry point. An example might be a purchase order service that provides three entry points, one for HTTP, one for SMTP, and one for FAX ordering. In this example, we'd find a businessService element that contains three bindingTemplate entries, each with identical data with the exception of the accessPoint value and URLType value.
The tModelInstanceDetails structure is a simple accessor container for one or more tModelInstanceInfo structures. When taken as a group, the data that is presented in a tModelInstanceDetails structure forms a technically descriptive fingerprint by virtue of the unordered list of tModelKey references contained within this structure. What this means in English is that when someone registers a bindingTemplate (within a businessEntity structure), it will contain one or more references to specific and identifiable specifications that are implied by the tModelKey values provided with the registration. During an inquiry for a service, an interested party could use this information to look for a specific bindingTemplate that contains a specific tModel reference, or even a set of tModel references. By registering a specific fingerprint in this manner, a software developer can readily signify that they are compatible with the specifications implied in the tModelKey elements exposed in this manner.

The tModel structure

Being able to describe a Web service and then make the description meaningful enough to be useful during searches is an important UDDI goal. Another goal is to provide a facility to make these descriptions useful enough to learn about how to interact with a service that you don't know much about. In order to do this, there needs to be a way to mark a description with information that designates how it behaves, what conventions it follows, or what specifications or standards the service is compliant with. Providing the ability to describe compliance with a specification, concept, or even a shared design is one of the roles that the tModel structure fills.
The tModel structure takes the form of keyed metadata (data about data). In a general sense, the purpose of a tModel within the UDDI registry is to provide a reference system based on abstraction. Thus, the kind of data that a tModel represents is pretty nebulous. In other words, a tModel registration can define just about anything, but in the current revision, two conventions have been applied for using tModels: as sources for determining compatibility and as keyed namespace references.
The information that makes up a tModel is quite simple. There's a key, a name, an optional description, and then a URL that points somewhere – presumably somewhere where the curious can go to find out more about the actual concept represented by the metadata in the tModel itself.
There are two places within a businessEntity registration that you'll find references to tModels. In this regard, tModels are special. Whereas the other data within the businessEntity (e.g. businessService and bindingTemplate data) exists uniquely with one uniquely keyed instance as a member of one unique parent businessEntity, tModels are used as references. This means that you'll find references to specific tModel instances in many businessEntity structures.

Defining the technical fingerprint.

The primary role that a tModel plays is to represent a technical specification. An example might be a specification that outlines wire protocols, interchange formats and interchange sequencing rules. Examples can be seen in the RosettaNet Partner Interface Processes specification, the Open Applications Group Integration Specification and various Electronic Document Interchange (EDI) efforts.

Software that communicates with other software across some communication medium invariably adheres to some pre-agreed specifications. In situations where this is true, the designers of the specifications can establish a unique technical identity within a UDDI registry by registering information about the specification in a tModel.

Once registered in this way, other parties can express the availability of Web services that are compliant with a specification by simply including a reference to the tModel identifier (called a tModelKey) in their technical service descriptions bindingTemplate data.

This approach facilitates searching for registered Web services that are compatible with a particular specification. Once you know the proper tModelKey value, you can find out whether a particular business or entity has registered a Web service that references that tModel key. In this way, the tModelKey becomes a technical fingerprint that is unique to a given specification.

Defining an abstract namespace reference.

The other place where tModel references are used is within the identifierBag, categoryBag, address and publisherAssertion structures that are used to define organizational identity and various classifications. Used in this context, the tModel reference represents a relationship between the keyed name-value pairs to the super-name, or namespace within which the name-value pairs are meaningful.

An example of this can be seen in the way a business or entity can express the fact that their US tax code identifier (which they are sure they are known by to their partners and customers) is a particular value. To do this, let's assume that we find a tModel that is named “US Tax Codes”, with a description “United States business tax code numbers as defined by the United States Internal Revenue Service”. In this regard, the tModel still represents a specific concept – but instead of being a technical specification, it represents a unique area within which tax code ID’s have a particular meaning.

Once this meaning is established, a business can use the tModelKey for the tax code tModel as a unique reference that qualifies the remainder of the data that makes up an entry in the identifierBag data.

To get things started, the UDDI Operator Sites have registered a number of useful tModels, including NAICS (an industry code taxonomy), UNSPC (a product and service category code taxonomy), and ISO 3166 (a geographical region code taxonomy).

Structure Specification:

```
<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 publisherAssertion structure

Many businesses, like large enterprises or marketplaces, are not effectively represented by a single businessEntity, since their description and discovery are likely to be diverse. As a consequence, several businessEntity structures can be published, representing individual subsidiaries of a large enterprise or individual participants of a marketplace. Nevertheless, they still represent a more or less coupled community and would like to make some of their relationships visible in their UDDI registrations. Therefore, two related businesses use the xx_publisherAssertion messages, publishing assertions of business relationships.

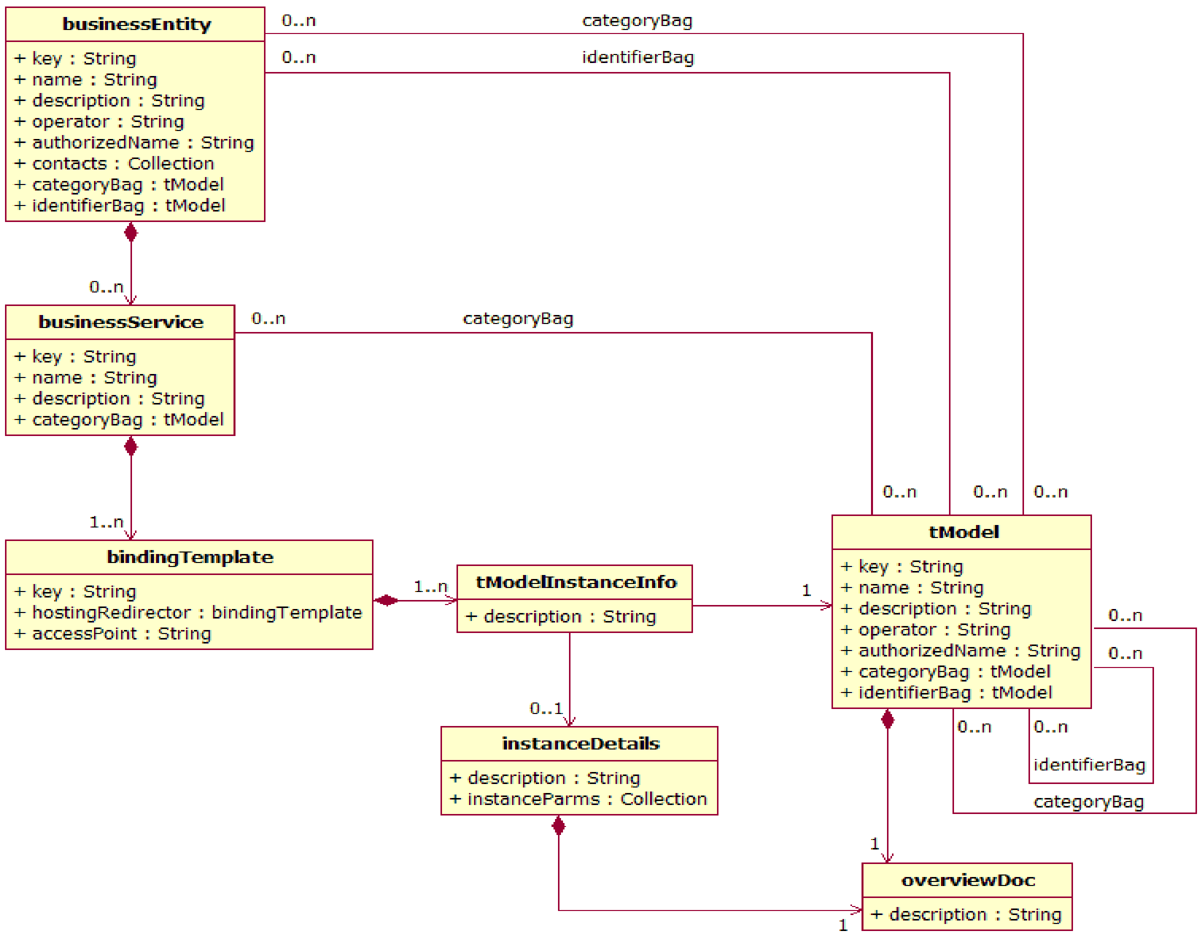
In order to eliminate the possibility that one publisher claims a relationship between both businesses that is in fact not reciprocally recognized, both publishers have to agree that the relationship is valid by publishing their own publisherAssertion. Therefore, both publishers have to publish exactly the same information. When this happens, the relationship becomes visible.

In the case that a publisher is responsible for both businesses, the relationship automatically becomes visible after publishing just one of both assertions that make up the relationship.

The publisherAssertion structure consists of the three elements fromKey (the first businessKey), toKey (the second businessKey) and keyedReference. The keyedReference designates the asserted relationship type in terms of a keyName keyValue pair within a tModel, uniquely referenced by a tModelKey. All three parts of the keyedReference – the tModelKey, the keyName, and the keyValue – are mandatory in this context. Empty (zero length) keyName and keyValue elements are permitted.

```
<element name="publisherAssertion" type="uddi:publisherAssertion" />
<complexType name="publisherAssertion">
  <sequence>
    <element ref="uddi:fromKey" />
    <element ref="uddi:toKey" />
    <element ref="uddi:keyedReference" />
  </sequence>
</complexType>
```

Simplified UML Model for UDDI Information Model:



BP 1.0 Requirements - Service Publication and Discovery

REGDATA of type uddi:bindingTemplate representing a conformant INSTANCE MUST contain the uddi:accessPoint element.

CORRECT:

```
<bindingTemplate bindingKey="...">
   <description xml:lang="EN">BarSOAPPort</description>
   <accessPoint>http://example.org/myBarSOAPPort</accessPoint>
   <tModelInstanceDetails>
      ...
   </tModelInstanceDetails>
</bindingTemplate>
```

INCORRECT:

```
<bindingTemplate bindingKey="...">
   <description xml:lang="EN">BarSOAPPort</description>
   <hostingRedirector bindingKey="..."/>
   <tModelInstanceDetails>
      ...
   </tModelInstanceDetails>
</bindingTemplate>
```

REGDATA of type `uddi:tModel` representing a conformant Web service type MUST use WSDL as the description language.
REGDATA of type `uddi:tModel` representing a conformant Web service type MUST be categorized using the `uddi:types` taxonomy and a categorization of "`wsdlSpec`".
The `wsdl:binding` that is referenced by REGDATA of type `uddi:tModel` MUST itself conform to the Profile.

Describe the basic functions provided by the UDDI Publish and Inquiry APIs to interact with a UDDI business registry.

Publish API functions

The messages in this section represent commands that require authenticated access to an UDDI Operator Site, and are used to publish and update information contained in a UDDI compatible registry. Each business should initially select one Operator Site to host their information. Once chosen, information can only be updated at the site originally selected. UDDI provides no automated means to reconcile multiple or duplicate registrations. The messages defined in this section all behave synchronously and are callable via HTTP-POST only. HTTPS is used exclusively for all of the calls defined in this publisher's API. The publishing API calls defined that UDDI operators support are:

- `add_publisherAssertions`: Used to add relationship assertions to the existing set of assertions.
- `delete_binding`: Used to remove an existing `bindingTemplate` from the `bindingTemplates` collection that is part of a specified `businessService` structure.
- `delete_business`: Used to delete registered `businessEntity` information from the registry.
- `delete_publisherAssertions`: Used to delete specific publisher assertions from the assertion collection controlled by a particular publisher account. Deleting assertions from the assertion collection will affect the visibility of business relationships. Deleting an assertion will cause any relationships based on that assertion to be invalidated.
- `delete_service`: Used to delete an existing `businessService` from the `businessServices` collection that is part of a specified `businessEntity`.
- `delete_tModel`: Used to hide registered information about a `tModel`. Any `tModel` hidden in this way is still usable for reference purposes and accessible via the `get_tModelDetail` message, but is simply hidden from `find_tModel` result sets. There is no way to actually cause a `tModel` to be deleted, except by administrative petition.
- `discard_authToken`: Used to inform an Operator Site that a previously provided authentication token is no longer valid and should be considered invalid if used after this message is received and until such time as an `authToken` value is recycled or reactivated at an operator's discretion. See `get_authToken`.
- `get_assertionStatusReport`: Used to get a status report containing publisher assertions and status information. This report is useful to help an administrator manage active and tentative publisher assertions. Publisher assertions are used in UDDI to manage publicly visible relationships between `businessEntity` structures. Relationships are a feature introduced in generic 2.0 that help manage complex business structures that require more than one `businessEntity` or more than one publisher account to manage parts of a `businessEntity`. Returns an `assertionStatusReport` that includes the status of all assertions made involving any `businessEntity` controlled by the requesting publisher account.
- `get_authToken`: Used to request an authentication token from an Operator Site. Authentication tokens are required when using all other API's defined in the publishers API. This function serves as the program's equivalent of a login request.
- `get_publisherAssertions`: Used to get a list of active publisher assertions that are controlled by an individual publisher account. Returns a `publisherAssertions` message containing all publisher assertions associated with a specific publisher account. Publisher assertions are used to control publicly visible business relationships.
- `get_registeredInfo`: Used to request an abbreviated synopsis of all information currently managed by a given individual.

- `save_binding`: Used to register new `bindingTemplate` information or update existing `bindingTemplate` information. Use this to control information about technical capabilities exposed by a registered business.
- `save_business`: Used to register new `businessEntity` information or update existing `businessEntity` information. Use this to control the overall information about the entire business. Of the `save_xx` API's this one has the broadest effect. In UDDI V2, a feature is introduced where `save_business` can be used to reference a `businessService` that is parented by another `businessEntity`.
- `save_service`: Used to register or update complete information about a `businessService` exposed by a specified `businessEntity`.
- `save_tModel`: Used to register or update complete information about a `tModel`.
- `set_publisherAssertions`: used to save the complete set of publisher assertions for an individual publisher account. Replaces any existing assertions, and causes any old assertions that are not reasserted to be removed from the registry. Publisher assertions are used to control publicly visible business relationships.

Inquiry API functions

The messages in this section represent inquiries that anyone can make of any UDDI Operator Site at any time. These messages all behave synchronously and are required to be exposed via HTTP-POST only. Other synchronous or asynchronous mechanisms may be provided at the discretion of the individual UDDI Operator Site or UDDI compatible registry. The publicly accessible queries are:

- `find_binding`: Used to locate specific bindings within a registered `businessService`. Returns a `bindingDetail` message.
- `find_business`: Used to locate information about one or more businesses. Returns a `businessList` message.
- `find_relatedBusinesses`: Used to locate information about `businessEntity` registrations that are related to a specific business entity whose key is passed in the inquiry. The Related Businesses feature is used to manage registration of business units and subsequently relate them based on organizational hierarchies or business partner relationships. Returns a `relatedBusinessesList` message.
- `find_service`: Used to locate specific services within a registered `businessEntity`. Returns a `serviceList` message.
- `find_tModel`: Used to locate one or more `tModel` information structures. Returns a `tModelList` structure.
- `get_bindingDetail`: Used to get full `bindingTemplate` information suitable for making one or more service requests. Returns a `bindingDetail` message.
- `get_businessDetail`: Used to get the full `businessEntity` information for one or more businesses or organizations. Returns a `businessDetail` message.
- `get_businessDetailExt`: Used to get extended `businessEntity` information. Returns a `businessDetailExt` message.
- `get_serviceDetail`: Used to get full details for a given set of registered `businessService` data. Returns a `serviceDetail` message.
- `get_tModelDetail`: Used to get full details for a given set of registered `tModel` data. Returns a `tModelDetail` message.

Chapter 4. JAX-RPC

Explain the service description model, client connection types, interaction modes, transport mechanisms/protocols, and endpoint types as they relate to JAX-RPC.

JAX-RPC is for Web services interoperability across heterogeneous platforms and languages. This makes JAX-RPC a key technology for Web services integration.

You can use the standard JAX-RPC programming model to develop Web service clients and endpoints based on SOAP. A Web service endpoint is described using a Web Services Description Language (WSDL) document. JAX-RPC enables JAX-RPC clients to invoke Web services developed across heterogeneous platforms. In a similar manner, JAX-RPC Web service endpoints can be invoked by heterogeneous clients. JAX-RPC requires SOAP and WSDL standards for this cross-platform interoperability.

JAX-RPC provides an easy to develop programming model for development of SOAP based Web services. You can use the RPC programming model to develop Web service clients and endpoints. For typical scenarios, you are not exposed to the complexity of the underlying runtime mechanisms (for example, SOAP protocol level mechanisms, marshalling and unmarshalling). A JAX-RPC runtime system (a library) abstracts these runtime mechanisms for the Web services programming model. This simplifies Web service development.

JAX-RPC provides support for WSDL-to-Java and Java-to-WSDL mapping as part of the development of Web service clients and endpoints. In a typical development environment, tools provide these mapping functionality. This further simplifies the application development.

JAX-RPC enables a Web service endpoint to be developed using either a Java Servlet or Enterprise JavaBeans (EJB) component model. A Web service endpoint is deployed on either the Web container or EJB container based on the corresponding component model. These endpoints are described using a WSDL document. This WSDL document can be published in public or private registry, though this is not required. A client uses this WSDL document and invokes the Web service endpoint. A JAX-RPC client can use stubs-based, dynamic proxy or dynamic invocation interface (DII) programming models to invoke a heterogeneous Web service endpoint.

JAX-RPC requires SOAP over HTTP for interoperability. JAX-RPC provides support for SOAP message processing model through the SOAP message handler functionality. This enables developers to build SOAP specific extensions to support security, logging and any other facility based on the SOAP messaging. JAX-RPC uses SAAJ API for SOAP message handlers. SAAJ provides a standard Java API for constructing and manipulating SOAP messages with attachments.

JAX-RPC provides support for document-based messaging. Using JAX-RPC, any MIME-encoded content can be carried as part of a SOAP message with attachments. This enables exchange of XML documents, images and other MIME types across Web services.

JAX-RPC supports HTTP level session management and SSL based security mechanisms. This enables you to develop secure Web services. More advanced SOAP message-level security will be addressed in the evolution of JAX-RPC technology.

Steps for Implementing a Service:

1. Define Web Service Endpoint Interface
2. Implement Web Service class and methods
3. Package and deploy

Endpoint Interface example:

```
package com.example;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloIF extends Remote {
       public String sayHello(String s) throws RemoteException;
}
```

Service endpoint interface [SEI] MUST extend `java.rmi.Remote` interface. NOTE: Service implementation class MAY not implement SEI (using `implements` Java reserved word). But it MUST provide all SEI methods implementations with same signatures.

Service endpoint interface's methods MUST throw `java.rmi.RemoteException` exception. NOTE: Service implementation class' methods MUST NOT throw `java.rmi.RemoteException`.

Service endpoint interface [SEI] MAY be generated from WSDL (WSDL-to-Java approach).

Web Service can be implemented in form:

1. Java class (for servlet-based endpoint)

```
package com.example;

public class HelloImpl implements HelloIF {
     public String message = "Hello ";
     public String sayHello(String s) {
            return message + s;
     }
}
```

   NOTE: Your implementation class does not throw `RemoteException`. This is the responsibility of the container as this implementation will be deployed in a managed J2EE container.

2. Stateless session bean (for EJB-based endpoint)
   `com.example.Greeting` is the bean's Web service endpoint interface. It provides the client's view of the Web service, hiding the stateless session bean from the client. A Web service endpoint interface must conform to the rules of a JAX-RPC service definition interface:
   - It extends the `java.rmi.Remote` interface.
   - It MUST NOT have constant declarations, such as `public final static`.
   - The methods MUST throw the `java.rmi.RemoteException` or one of its subclasses. The methods may also throw service-specific exceptions.

- Method parameters and return types must be supported JAX-RPC types.

Here is the source code for the `com.example.Greeting` endpoint interface:

```
package com.example; public interface Greeting extends Remote {
  public String sayHello(String name) throws RemoteException;
}
```

The `com.example.GreetingBean` class implements the `sayHello` method defined by the `com.example.Greeting` interface. The interface decouples the implementation class from the type of client access. For example, if you added remote and home interfaces to `com.example.GreetingBean`, the methods of the `com.example.GreetingBean` class could also be accessed by remote clients. No changes to the `com.example.GreetingBean` class would be necessary.

The source code for the `com.example.GreetingBean` Stateless Session Bean Class follows:

```
package com.example;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;
public class GreetingBean implements SessionBean {
  public String sayHello(String name) {
         return "Hello" + name;
  }
  public void ejbCreate() {
  }
  // Standard callback methods
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void ejbRemove() {
  }
  public void setSessionContext(SessionContext ctx) {
  }
}
```

Deployment Descriptor:

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

<ejb-jar version="2.1" 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/ejb-jar_2_1.xsd">

  <enterprise-beans>
         <session>
                 <ejb-name>GreetingBean</ejb-name>
                 <service-endpoint>com.example.Greeting</service-endpoint>
                 <ejb-class>com.example.GreetingBean</ejb-class>
                 <session-type>Stateless</session-type>
                 <transaction-type>Container</transaction-type>
                 <security-identity>
                         <use-caller-identity/>
                 </security-identity>
         </session>
  </enterprise-beans>

</ejb-jar>
```

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.

EJB based endpoint

JSR 109 standardizes webservice in J2EE 1.4 using JAX-RPC. EJB 2.1 exposes Stateless Session bean as a web service endpoint using JAX-RPC interface and any webservice client can access the EJB webservice using SOAP 1.1 over HTTP. The developer can choose a web service endpoint interface for a stateless session bean whenever he wants to expose the functionality of the bean as a web service endpoint through WSDL. The clients for EJB Web Service endpoint may be Java clients and/or clients written in a programming language other than Java. A Java client that accesses the EJB Web Service has to use JAX-RPC client APIs. This is an example shows how can you expose your existing EJB applications as a webservice endpoint and how a pure Java client accesses the ejb webservice.

We will use a simple Stateless Session bean `TimeBean` that displays the current time and locale information. For exposing the webservice endpoint you do not need to have home or remote interfaces for the EJBs, only the end-point interface that extends `java.rmi.Remote` and bean implementation class is required. Following the code for the service-endpoint for the EJB:

```
package time;
import java.rmi.RemoteException;
import java.rmi.Remote;
public interface TimeService extends Remote {
        public String getDateTime (String name) throws RemoteException;
}
```

Then we have to define the end-point interface in `ejb-jar.xml` as follows:

```
<session>
  <display-name>TimeServiceEJB</display-name>
  <ejb-name>TimeServiceEJB</ejb-name>
  <service-endpoint>time.TimeService</service-endpoint>
  <ejb-class>time.TimeServiceBean</ejb-class>
  <session-type>Stateless</session-type>
  ...
</session>
```

The WSDL file defines the web services e.g. the following `MyTimeService.wsdl` describes the Time ejb webservice:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MyTimeService" targetNamespace="urn:oracle-ws"
xmlns:tns="urn:oracle-ws"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types/>
  <message name="TimeService_getDateTime">
    <part name="String_1" type="xsd:string"/>
   </message>

  <message name="TimeService_getDateTimeResponse">
    <part name="result" type="xsd:string"/>
  </message>

  <portType name="TimeService">
    <operation name="getDateTime" parameterOrder="String_1">
      <input message="tns:TimeService_getDateTime"/>
      <output message="tns:TimeService_getDateTimeResponse"/>
    </operation>
  </portType>

  <binding name="TimeServiceBinding" type="tns:TimeService">
    <operation name="getDateTime">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

```
                    use="encoded" namespace="urn:oracle-ws"/>
      </input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded" namespace="urn:oracle-ws"/>
      </output>
      <soap:operation soapAction=""/>
    </operation>

    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
  </binding>
  <service name="MyTimeService">
    <port name="TimeServicePort" binding="tns:TimeServiceBinding">
    <soap:address location="REPLACE_WITH_ACTUAL_URL"/></port>
  </service>
</definitions>
```

The `mapping.xml` file specifies the Java to WSDL mapping i.e. it contains the mapping between package names and XML namespaces, WSDL root types and Java artifacts, and the set of mappings for services. For example we will have the following contents for our `mapping.xml`:

```
<package-mapping>
  <package-type>time</package-type>
  <namespaceURI>urn:oracle-ws</namespaceURI>
</package-mapping>
```

Deployment of webservices requires a deployment descriptor named `webservices.xml` in `META-INF` of the `ejb-jar` file. This descriptor specifies the set of web service descriptions that are to be deployed into the J2EE Application Server and the dependencies they have on container resources and services:

```
<webservice-description>
  <webservice-description-name>TimeServiceEJB</webservice-description-name>
  <wsdl-file>META-INF/MyTimeService.wsdl</wsdl-file>
  <jaxrpc-mapping-file>META-INF/mapping.xml</jaxrpc-mapping-file>
  <port-component>
    <description>port component description</description>
    <port-component-name>TimeServicePort</port-component-name>
    <wsdl-port>
      <namespaceURI>urn:oracle-ws</namespaceURI>
      <localpart>TimeServicePort</localpart>
    </wsdl-port>
    <service-endpoint-interface>time.TimeService</service-endpoint-interface>
    <service-impl-bean>
      <ejb-link>TimeServiceEJB</ejb-link>
    </service-impl-bean>
  </port-component>
</webservice-description>
```

Servlet based endpoint

During the deployment of a service endpoint component on a servlet container based JAX-RPC runtime system, a service endpoint class is associated with a servlet. The associated servlet class is provided by the JAX-RPC runtime system (not by service endpoint developer) during the deployment. This association is configured in a manner specific to a JAX-RPC runtime system and its deployment tool. For example, a JAX-RPC deployment tool may configure a 1-1 association between a servlet class and service endpoint class. The associated servlet class corresponds to the configured transport binding for the service endpoint. For example, the servlet class `javax.servlet.http.HttpServlet` is used for the HTTP transport.

The associated servlet typically takes the responsibility of handling transport specific processing of an RPC request and for initiating dispatch to the target service endpoint instance. Each `Servlet.service(...)` method maps to a single remote method invocation on the target service endpoint instance. The thread model (whether single threaded or concurrent) for the remote method invocation on the service endpoint instance depends on the runtime system specific servlet associated with the corresponding endpoint class. The Servlet specification provides facility for both concurrent and single threaded model (the latter through the `SingleThreadModel` interface) for the `service(...)` method on a servlet.

When processing an incoming SOAP request for a one-way operation, the associated servlet is required to send back an HTTP response code of 200 or 202 as soon as it has identified the incoming request as being one-way and before it dispatches it to the target service endpoint instance.
The term JAX-RPC Service Endpoint used within the JAX-RPC specification is somewhat confusing since both Service Implementation Beans require the use of a JAX-RPC run time. However, in this case it refers to the programming model defined within the JAX-RPC specification that is used to create Web services that run within the web container. The requirements are repeated here with clarification. Changes from the JAX-RPC defined programming model are required for running in a J2EE container-managed environment. A JAX-RPC Service Endpoint can be single or multi-threaded. The concurrency requirement is declared as part of the programming model. A JAX-RPC Service Endpoint must implement `javax.servlet.SingleThreadModel` if single threaded access is required by the component. A container must serialize method requests for a Service Implementation Bean that implements the `SingleThreadModel` interface. Note, the `SingleThreadModel` interface has been deprecated in the Servlet 2.4 specification. The Service Implementation Bean must follow the Service Developer requirements outlined in the JAX-RPC specification and are listed below except as noted:

- The Service Implementation Bean must have a default public constructor.
- The Service Implementation Bean may implement the Service Endpoint Interface as defined by the JAX-RPC Servlet model. The bean must implement all the method signatures of the SEI. In addition, a Service Implementation Bean may be implemented that does not implement the SEI. This additional requirement provides the same SEI implementation flexibility as provided by EJB service endpoints. The business methods of the bean must be `public` and must not be `static`. If the Service Implementation Bean does not implement the SEI, the business methods must not be `final`. The Service Implementation Bean may implement other methods in addition to those defined by the SEI, but only the SEI methods are exposed to the client.
- A Service Implementation must be a stateless object. A Service Implementation Bean must not save client specific state across method calls either within the bean instance's data members or external to the instance. A container may use any bean instance to service a request.
- The class must be `public`, must not be `final` and must not be `abstract`.
- The class must not define the `finalize()` method.

A Service Implementation Bean for the web container may implement the `java.xml.rpc.server.ServiceLifeCycle` interface:

```
package javax.xml.rpc.server;
public interface ServiceLifecycle {
        void init(Object context) throws ServiceException;
        void destroy();
}
```

The `ServiceLifecycle` interface allows the web container to notify a Service Implementation Bean instance of impending changes in its state. The bean may use the notification to prepare its internal state for the transition. If the bean implements the `ServiceLifecycle` interface, the container is required to call the `init(...)` and `destroy` methods as described below.
The container must call the `init(...)` method before it can start dispatching requests to the SEI methods of the bean. The `init(...)` method parameter value provided by the container is described by the JAX-RPC specification. The bean may use the container notification to ready its internal state for receiving requests.
The container must notify the bean of its intent to remove the bean instance from the container's working set by calling the `destroy()` method. A container may not call the `destroy()` method while a request is being processed by the bean instance. The container may not dispatch additional requests to the SEI methods of the bean after the `destroy()` method is called.

When should one implement Web services in J2EE 1.4 using stateless Session Bean as opposed to using Java class

Use a stateless Session Bean to expose Web services if you:

- Need to expose previously existing stateless Session Beans as Web services
- Need declarative transaction management
- Need the thread management provided by EJB Container
- Need role based security

Use Java classes to expose your Web services if you:

- Need to expose previously existing Java classes as Web services
- Want a light-weight system, and don't care much about transactional capabilities that an EJB container provides

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.

JAX-RPC Client Environment:

- Service endpoint can be implemented using any platform or language.
- May generate client code from WSDL:
    - Static stub (compile time)
    - Dynamic proxy (runtime)
- May call Web Service directly:
    - Dynamic invocation interface (DII)
- Can use either J2SE or J2EE programming model.

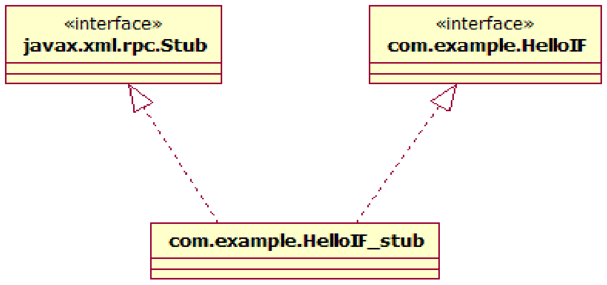
There are three Web Service Client programming models:

1. *Stub-based (least dynamic)*
   Both interface (WSDL) and implementation (stub) created at compile time.
2. *Dynamic proxy*
   Interface (WSDL) created at compile time. Implementation (dynamic proxy) created at runtime.
3. *Dynamic invocation interface (DII)*
   Both interface (WSDL) and implementation created at runtime.

Stub-based Invocation Model

- Stub class gets generated at compile time
- Instantiated using vendor-generated Service implementation class
- Best performance
- Stub class implements `javax.xml.rpc.Stub` interface and Web Service definition interface (`com.example.HelloIF`)

```
package javax.xml.rpc;
import java.util.Iterator;
public interface Stub {
    /**
     * Standard property: User name for authentication.
     */
    public static final String USERNAME_PROPERTY = Call.USERNAME_PROPERTY;
    /**
     * Standard property: Password for authentication.
     */
    public static final String PASSWORD_PROPERTY = Call.PASSWORD_PROPERTY;
    /**
     * Standard property: Target service endpoint address. The
     * URI scheme for the endpoint address specification must
     * correspond to the protocol/transport binding for this
     * stub class.
     */
    public static final String ENDPOINT_ADDRESS_PROPERTY =
"javax.xml.rpc.service.endpoint.address";
    /**
     * Standard property: This boolean property is used by a service
     * client to indicate whether or not it wants to participate in
     * a session with a service endpoint. If this property is set to
     * true, the service client indicates that it wants the session
     * to be maintained. If set to false, the session is not maintained.
     * The default value for this property is false.
     */
    public static final String SESSION_MAINTAIN_PROPERTY =
Call.SESSION_MAINTAIN_PROPERTY;
    public void _setProperty(String name, Object value);
    public Object _getProperty(String name);
    public Iterator _getPropertyNames();
}
```



Steps for Stub-Based Invocation Client:

1. Generate Stubs
2. Create Client code
3. Compile the Client code with remote interface and stubs in CLASSPATH
4. Run the Client with JAX-RPC generated code and runtime

Stand-alone Stub-based Client example:

```
package com.example;
import javax.xml.rpc.Stub;
public class HelloClient {
        public static void main(String[] args) {
                try {
                        Stub stub = (Stub) (new MyHelloService_Impl().getHelloIFPort());
        stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
                        HelloIF hello = (HelloIF) stub;
                        println(hello.sayHello("Duke!"));
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }
}
```

J2EE Stub-based Client example:

```
package com.example;

import javax.xml.rpc.Stub;

public class HelloClient {
        public void callSayHello {
                try {
                        Context ic = new InitialContext();
                        Service service = (Service)
ic.lookup("java:comp/env/service/HelloService");
                        HelloIF hello = (HelloIF) service.getHelloServiceProviderPort();
                        println(hello.sayHello("Duke!"));
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }
}
```

Dynamic Proxy-based Invocation Model

- At Runtime Application provides the WSDL
- Dynamic proxy is generated on the fly by JAX-RPC runtime system
- Slower than stub-based: proxy created and casted
- More portable than stub-based: does not depend on vendor generated service class before runtime

Dynamic Proxy Client example:

```
package com.example;
```

```
import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;

public class HelloClient {
	public static void main(String[] args) {
		try {
			String UrlString = "http://localhost:8080/ProxyHelloWorld.wsdl";
			String nameSpaceUri = "http://sample.proxy.org/wsdl";
		String serviceName = "HelloWorld"; // maps to service name in WSDL
			String portName = "HelloIFPort";   // maps to port name in WSDL
			URL helloWsdlUrl = new URL(UrlString);

			ServiceFactory serviceFactory = ServiceFactory.newInstance();

			// Create a Service object named helloService
			Service helloService =
serviceFactory.createService(helloWsdlUrl, new QName(nameSpaceUri, serviceName));

			// Destination for service endpoint retrieval
			QName qn = new QName(nameSpaceUri, portName);

			// Create a proxy with type of interface 'com.example.HelloIF'
			HelloIF myProxy = (HelloIF) helloService.getPort(qn,
com.example.HelloIF.class);

			System.out.println(myProxy.sayHello("Duke"));
		} catch (Exception ex) {
			ex.printStackTrace();
		}
	}
}
```

WSDL:

```
...
<service name="HelloWorld">
	<port name="HelloIFPort" binding="tns:HelloWorldBinding">
		<soap:address location="http://example.com/HelloWorld"/>
	</port>
</service>
...
```

The `javax.xml.rpc.Service` interface :

```
package javax.xml.rpc;

import javax.xml.namespace.QName;
import javax.xml.rpc.encoding.TypeMappingRegistry;
import javax.xml.rpc.handler.HandlerRegistry;

public interface Service {

    /**
     * The getPort method returns either an instance of a generated
     * stub implementation class or a dynamic proxy. A service client
     * uses this dynamic proxy to invoke operations on the target
     * service endpoint. The <code>serviceEndpointInterface</code>
     * specifies the service endpoint interface that is supported by
     * the created dynamic proxy or stub instance.
     */
    public java.rmi.Remote getPort(QName portName, Class serviceEndpointInterface)
            throws ServiceException;
    /**
     * The getPort method returns either an instance of a generated
     * stub implementation class or a dynamic proxy. The parameter
     * <code>serviceEndpointInterface</code> specifies the service
     * endpoint interface that is supported by the returned stub or
     * proxy. In the implementation of this method, the JAX-RPC
     * runtime system takes the responsibility of selecting a protocol
     * binding (and a port) and configuring the stub accordingly.
     * The returned <code>Stub</code> instance should not be
     * reconfigured by the client.
     */
    public java.rmi.Remote getPort(Class serviceEndpointInterface)
        throws ServiceException;
    public Call[] getCalls(QName portName) throws ServiceException;
    public Call createCall(QName portName) throws ServiceException;
    public Call createCall(QName portName, QName operationName)
        throws ServiceException;
    public Call createCall(QName portName, String operationName)
        throws ServiceException;
    public Call createCall() throws ServiceException;
    public QName getServiceName();
    public java.util.Iterator getPorts() throws ServiceException;
    public java.net.URL getWSDLDocumentLocation();
    public TypeMappingRegistry getTypeMappingRegistry();
    public HandlerRegistry getHandlerRegistry();
}
```

Dynamic Invocation Interface (DII) Model

- Gives complete control to client programmer
- Most dynamic but complex programming
- Create JAX-RPC `javax.xml.rpc.Call` object first, set operation and parameters during runtime
- Could combine with UDDI lookup and WSDL parsing for dynamic lookup and discovery
- Used when service definition interface is NOT known until runtime

Dynamic Invocation Interface (DII) Client example:

```
package com.example;

import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ParameterMode;

public class HelloClient {
        private static String endpoint = "http://localhost:8080/dynamic-
jaxrpc/dynamic";
        private static String qnameService = "Hello";
        private static String qnamePort = "HelloIF";
        private static String BODY_NAMESPACE_VALUE = "http://dynamic.org/wsdl";
        private static String ENCODING_STYLE_PROPERTY =
"javax.xml.rpc.encodingstyle.namespace.uri";
        private static String NS_XSD = "http://www.w3.org/2001/XMLSchema";
        private static String URI_ENCODING =
"http://schemas.xmlsoap.org/soap/encoding/";
        public static void main(String[] args) {
                try {
                        ServiceFactory factory = ServiceFactory.newInstance();
                Service service = factory.createService(new QName(qnameService));
                        QName port = new QName(qnamePort);

                //create JAX-RPC Call using JAX-RPC Service's createCall() method.
                        Call call = service.createCall(port);
                        // Configure your Call instance with its setter methods
                        call.setTargetEndpointAddress(endpoint);
                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 "sayHello"));
                call.addParameter("String_1", QNAME_TYPE_STRING, ParameterMode.IN);
                        String[] params = { "Duke!" };

                // Invoke the WS operation using the JAX-RPC Call's invoke method
                        String result = (String) call.invoke(params);

                        System.out.println(result);
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }
}
```

The `javax.xml.rpc.Call` interface:

```

package javax.xml.rpc;

import javax.xml.namespace.QName;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

/**
 * The <code>javax.xml.rpc.Call</code> interface provides support
 * for the dynamic invocation of a service endpoint. The
 * <code>javax.xml.rpc.Service</code> interface acts as a factory
 * for the creation of <code>Call</code> instances.
 */
public interface Call {

    /**
     * Standard property: User name for authentication
     */
    public static final String USERNAME_PROPERTY =
"javax.xml.rpc.security.auth.username";

    /**
     * Standard property: Password for authentication
     */
    public static final String PASSWORD_PROPERTY =
"javax.xml.rpc.security.auth.password";

    /**
```

```
     * Standard property for operation style. This property is
     * set to "rpc" if the operation style is rpc; "document"
     * if the operation style is document.
     */
    public static final String OPERATION_STYLE_PROPERTY =
"javax.xml.rpc.soap.operation.style";

    /**
     * Standard property for SOAPAction. This boolean property
     * indicates whether or not SOAPAction is to be used. The
     * default value of this property is false indicating that
     * the SOAPAction is not used.
     */
    public static final String SOAPACTION_USE_PROPERTY =
"javax.xml.rpc.soap.http.soapaction.use";

    /**
     * Standard property for SOAPAction. Indicates the SOAPAction
     * URI if the <code>javax.xml.rpc.soap.http.soapaction.use</code>
     * property is set to <code>true</code>.
     */
    public static final String SOAPACTION_URI_PROPERTY =
"javax.xml.rpc.soap.http.soapaction.uri";

    /**
     * Standard property for encoding Style:  Encoding style specified
     * as a namespace URI. The default value is the SOAP 1.1 encoding
     * <code>http://schemas.xmlsoap.org/soap/encoding/</code>
     */
    public static final String ENCODINGSTYLE_URI_PROPERTY =
"javax.xml.rpc.encodingstyle.namespace.uri";

    /**
     * Standard property: This boolean property is used by a service
     * client to indicate whether or not it wants to participate in
     * a session with a service endpoint. If this property is set to
     * true, the service client indicates that it wants the session
     * to be maintained. If set to false, the session is not maintained.
     * The default value for this property is <code>false</code>.
     */
    public static final String SESSION_MAINTAIN_PROPERTY =
"javax.xml.rpc.session.maintain";

    /**
     * Indicates whether <code>addParameter</code> and
     * <code>setReturnType</code> methods
     * are to be invoked to specify the parameter and return type
     * specification for a specific operation.
     */
    public boolean isParameterAndReturnSpecRequired(QName operationName);

    /**
     * Adds a parameter type and mode for a specific  operation.
     * Note that the client code may not call any
     * <code>addParameter</code> and <code>setReturnType</code>
     * methods before calling the <code>invoke</code> method. In
     * this case, the Call implementation class determines the
     * parameter types by using reflection on parameters, using
     * the WSDL description and configured type mapping registry.
     */
    public void addParameter(String paramName, QName xmlType, ParameterMode
parameterMode);

    /**
     * Adds a parameter type and mode for a specific  operation.
     * This method is used to specify the Java type for either
     * OUT or INOUT parameters.
     *
     * @param paramName Name of the parameter
     * @param xmlType XML datatype of the parameter
     * @param javaType The Java class of the parameter
     * @param parameterMode Mode of the parameter-whether
     *                ParameterMode.IN, OUT or INOUT
     */
    public void addParameter(String paramName, QName xmlType, Class javaType,
ParameterMode parameterMode);

    /**
     * Gets the XML type of a parameter by name.
     */
    public QName getParameterTypeByName(String paramName);

    /**
     * Sets the return type for a specific operation. Invoking
     * <code>setReturnType(null)</code> removes the return
     * type for this Call object.
     */
    public void setReturnType(QName xmlType);

    /**
     * Sets the return type for a specific operation.
     */
    public void setReturnType(QName xmlType, Class javaType);

    /**
     * Gets the return type for a specific operation.
     */
    public QName getReturnType();

    /**
     * Removes all specified parameters from this <code>Call</code> instance.
     * Note that this method removes only the parameters and not
     * the return type. The <code>setReturnType(null)</code> is
     * used to remove the return type.
     */
    public void removeAllParameters();

    /**
     * Gets the name of the operation to be invoked using this Call instance.
     */
    public QName getOperationName();

    /**
     * Sets the name of the operation to be invoked using this
     * <code>Call</code> instance.
     */
    public void setOperationName(QName operationName);

    /**
     * Gets the qualified name of the port type.
     */
    public QName getPortTypeName();

    /**
     * Sets the qualified name of the port type.
     */
    public void setPortTypeName(QName portType);

    /**
     * Sets the address of the target service endpoint.
     * This address must correspond to the transport specified
     * in the binding for this <code>Call</code> instance.
     */
    public void setTargetEndpointAddress(String address);

    /**
     * Gets the address of a target service endpoint.
     */
    public String getTargetEndpointAddress();

    /**
     * Sets the value for a named property. JAX-RPC specification
     * specifies a standard set of properties that may be passed
     * to the <code>Call.setProperty</code> method.
     */
    public void setProperty(String name, Object value);

    /**
     * Gets the value of a named property.
     */
    public Object getProperty(String name);

    /**
     * Removes a named property.
     */
    public void removeProperty(String name);

    /**
     * Gets the names of configurable properties supported by
     * this <code>Call</code> object.
     */
    public Iterator getPropertyNames();

    // Remote Method Invocation methods

    /**
     * Invokes a specific operation using a synchronous request-response
     * interaction mode.
     */
    public Object invoke(Object[] inputParams) throws java.rmi.RemoteException;

    /**
     * Invokes a specific operation using a synchronous request-response
     * interaction mode.
     */
    public Object invoke(QName operationName, Object[] inputParams)
        throws java.rmi.RemoteException;

    /**
     * Invokes a remote method using the one-way interaction mode. The
     * client thread does not block waiting for the completion of the
     * server processing for this remote method invocation. This method
```

```
      * must not throw any remote exceptions. This method may throw a
      * <code>JAXRPCException</code> during the processing of the one-way
      * remote call.
      */
     public void invokeOneWay(Object[] params);

     /**
      * Returns a <code>Map</code> of {name, value} for the output parameters of
      * the last invoked operation. The parameter names in the
      * returned Map are of type <code>java.lang.String</code>.
      */
     public Map getOutputParams();

     /**
      * Returns a <code>List</code> values for the output parameters
      * of the last invoked operation.
      */
     public List getOutputValues();
}
```

Table 4.1. Usage scenarios of the three Web Service service client styles.

Static stub	Dynamic proxy	Dynamic Invocation Interface (DII)
Web service not expected to change	Some changes to the Web Service expected, such as the location of the service	Considerable changes to the Web service expected, such as: • Location of the service • Request/response format • Data types
Most common scenario	Less common	Less common
You can generate a stub class either from WSDL (using `WSDL2Java`) or from a service endpoint interface. A generated stub class is required to implement both `javax.xml.rpc.Stub` and the service endpoint interface. This stub interface provides APIs to configure stubs by setting properties like endpoint address, session, user name, password, etc.	The client at runtime creates dynamic proxy stubs using the `javax.xml.rpc.Service` interface. The client has a priori knowledge of the WSDL and the service it is going to invoke. It uses the `javax.xml.rpc.ServiceFactory` classes to create the service and get the proxy.	This software pattern eliminates the need for clients to know in advance a service's exact name and parameters. A DII client can discover this information at runtime using a service broker that can look up the service's information. This flexibility in service discovery enables the run-time system to use service brokers, which can adopt varying service discovery mechanisms - ebXML registries, UDDI, etc.

Given a set of requirements, develop and configure a Web service client that accesses a stateful Web service.

The JAX-RPC specification requires that a service client be able to participate in a session with a service endpoint. In the JAX-RPC 1.1 version, the session management mechanisms require use of HTTP as the transport in the protocol binding. This version of the JAX-RPC specification does not specify (or require) session management using SOAP headers given that there is no standard SOAP header representation for the session information. SOAP based session management may be considered in the future versions of the JAX-RPC specification.
A JAX-RPC runtime system is required to use at least one of the following mechanisms to manage sessions:

- Cookie based mechanism: On the initial method invocation on a service endpoint, the server side JAX-RPC runtime system sends a cookie to the service client to initiate a new session. If service client wants to participate in this session, the client side JAX-RPC runtime system then sends the cookie for each subsequent method invocation on this service endpoint. The cookie associates subsequent method invocations from the service client with the same session.
- URL rewriting involves adding session related identifier to a URL. This rewritten URL is used by the server-side JAX-RPC runtime to associate RPC invocations to the service endpoint with a session. The URL that is rewritten depends on the protocol binding in use.
- SSL session may be used to associate multiple RPC invocations on a service endpoint as part of a single session.

A session (in JAX-RPC) is initiated by the server-side JAX-RPC runtime system. The server-side JAX-RPC runtime system may use `javax.servlet.http.HttpSession` (defined in the Servlet specification) to implement support for the HTTP session management.

A service client uses the `javax.xml.rpc.session.maintain` property (set using the `Stub` or `Call` interfaces) to indicate whether or not it wants to participate in a session with a service endpoint. By default, this property is `False`, so the client does not participate in a session by default. However, by setting `javax.xml.rpc.session.maintain` property to `True`, the client indicates that it wants to join the session initiated by the server. In the cookie case, the client runtime system accepts the cookie and returns the session tracking information to the server, thereby joining the session.

The client code by setting the `javax.xml.rpc.session.maintain` property assumes that it would participate in a session if one is initiated by the server. The actual session management happens transparent to the client code in the client-side runtime system.

Property `javax.xml.rpc.session.maintain` accepts objects of `java.lang.Boolean` class. This boolean property is used by a service client to indicate whether or not it wants to participate in a session with a service endpoint. If this property is set to `True`, the service client indicates that it wants the session to be maintained. If set to `False`, the session is not maintained. The default value for this property is `False`.

```
package javax.xml.rpc;
public interface Stub {
        // ...
        void _setProperty(String name, Object value);
        Object _getProperty(String name);
        java.util.Iterator _getPropertyNames();
}
package javax.xml.rpc;
public interface Call {
        // ...
        void setProperty(String name, Object value);
        Object getProperty(String name);
        boolean removeProperty(String name);
        java.util.Iterator getPropertyNames();
}
```

The service endpoint class may implement the following `ServiceLifecycle` interface:

```
package javax.xml.rpc.server;
public interface ServiceLifecycle {
        void init(Object context) throws ServiceException;
        void destroy();
}
```

If the service endpoint class implements the `ServiceLifecycle` interface, the servlet container based JAX-RPC runtime system is required to manage the lifecycle of the corresponding service endpoint instances. The lifecycle of a service endpoint instance is realized through the implementation of the `init` and `destroy` methods of the `ServiceLifecycle` interface.

For service endpoint components deployed on a servlet container based JAX-RPC runtime system, the `context` parameter in the `ServiceLifecycle.init` method is required to be of the Java type `javax.xml.rpc.server.ServletEndpointContext`. The `ServletEndpointContext` provides an endpoint context maintained by the underlying servlet container based JAX-RPC runtime system. Note that the JAX-RPC specification specifies the standard programming model for a servlet based endpoint. The goal of JAX-RPC specification is not to define a more generic abstraction for the endpoint context or session that is independent of any specific component model, container and protocol binding. Such generic abstractions and endpoint model are outside the scope of the JAX-RPC specification. The following code snippet shows the `ServletEndpointContext` interface:

```
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();
        public javax.servlet.ServletContext getServletContext();
}
```

A servlet container based JAX-RPC runtime system is required to implement the `ServletEndpointContext` interface. The JAX-RPC runtime system is required to provide appropriate session, message context, servlet context and user principal information per method invocation on service endpoint instances.

The `getHttpSession` method returns the current HTTP session (as a `javax.servlet.http.HttpSession`). When invoked by the service endpoint instance within a remote method implementation, the `getHttpSession` returns the HTTP session associated currently with this method invocation. This method is required to return `null` if there is no HTTP session currently active and associated with this service endpoint instance. An endpoint class should not rely on an active HTTP session being always there; the underlying JAX-RPC runtime system is responsible for managing whether or not there is an active HTTP session.

The `getHttpSession` method throws `JAXRPCException` if it is invoked by a non HTTP bound endpoint. The JAX-RPC specification does not specify any transport level session abstraction for non-HTTP bound endpoints.

HTTP sesion timeout

To change the default session timeout globally, add the following element in your `web.xml` (service endpoint implementation deployment descriptor):

```
<web-app>
        ...
        <session-config>
                <!-- set global default timeout to 15 minutes -->
                <session-timeout>15</session-timeout>
        </session-config>
        ...
</web-app>
```

The `session-timeout` element defines the default session timeout interval for all sessions created in this web application. The specified timeout must be expressed in a whole number of MINUTES. If the timeout is 0 or less, the container ensures the default behaviour of sessions is never to timeout. If this element is not specified, the container must set its default timeout period.

The full syntax:

```
<!--
The session-config element defines the session parameters for this
web application.
-->
<!ELEMENT session-config (session-timeout?)>
<!--
The session-timeout element defines the default session timeout
interval for all sessions created in this web application.
The specified timeout must be expressed in a whole number of minutes.
-->
<!ELEMENT session-timeout (#PCDATA)>
```

Explain the advantages and disadvantages of a WSDL to Java vs. a Java to WSDL development approach.

WSDL to Java:

1. Convenient when WSDL document already exists
2. More powerful
3. Requires WSDL and XML Schema knowledge
4. Easy to stray outside of WS-I BP 1.0, harming interoperability

Java to WSDL:

1. Easier to use
2. No need to learn WSDL or XML Schema
3. Less control over published service contract

Describe the advantages and disadvantages of web service applications that use either synchronous/request response, one-way RPC, or non-blocking RPC invocation modes.

Synchrous request-response mode:

- Client's thread blocks until a return value or exception is returned

One-way RPC mode:

- Client's thread continues processing
- No return value or exception is expected

Example:

```
...
<portType name="HelloOneWayIF">
        <operation name="oneWayValueType">
                <input message="tns:HelloOneWayIF_oneWayValueType"/>
        </operation>
</portType>
<binding name="HelloOneWayIFBinding" type="tns:HelloOneWayIF">
        <operation name="oneWayValueType">
                <input>
                        <soap:body use="literal"/>
                </input>
                <soap:operation soapAction=""/>
        </operation>
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
</binding>
...
```

Non-blocking RPC invocation mode:

- 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

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 handlers allow you to intercept a SOAP message at various times during a service invocation. Handlers are similar to servlet filters. Handlers can exist on both the client and the server side. If you use JAX-RPC on the client side, you can have a handler process a SOAP request message right before it goes on the network, and you can process the response message before it is returned to the client. Similarly, you can intercept an incoming SOAP request message on the server before invoking the service implementation, as well as the outgoing response.

Several handlers can be combined into what is called a "handler chain". Each handler processes the SOAP message, which is then passed on to the next handler in the chain. The exact sequence in which this happens is configurable.

To develop a JAX-RPC handler, you simply create a class that implements the `javax.xml.rpc.handler.Handler` interface. It has three methods to handle SOAP *requests*, *responses* and *faults*, respectively.

Handlers are defined in the JAX-RPC specification. However, the "Enterprise Web Services" (JSR109) specification describes how they are used in a J2EE environment and adds some clarification to the way handlers are managed by the application server. We will assume that your Web service runs on a J2EE application server and hence we will follow the definitions of JSR109 as well as JAX-RPC.

Handlers are shared across multiple service invocations. In other words, they can store information that is only valid for a particular client or server instance. You can compare this to the way servlets are handled. When a new instance of a handler is created, its `init(...)` method is called. That allows you to set up things that you can use for multiple invocations. Before the handler is removed, the `destroy()` method is called, so that you can do cleanup in there. As a rule of thumb, however, you should avoid storing any state in a handler altogether.

Handlers can be configured either programmatically or, in case you are running a J2EE application server, they are configured in the Web service deployment descriptor. Handlers and handler chains are defined on a per service basis. They can be defined in both the server and client side deployment descriptors. Listing below shows the deployment descriptor for our sample service, showing how a handler class called `handler.PerformanceHandler` is registered for the `HelloWorldService` service (`webservices.xml`):

```
<webservices>
   <webservice-description>
      <webservice-description-name>HelloWorldService</webservice-description-name>
      <wsdl-file>WEB-INF/wsdl/HelloWorld.wsdl</wsdl-file>
      <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
      <port-component>
         <port-component-name>HelloWorld</port-component-name>
         <wsdl-port>
```

```
            <namespaceURI>http://pack</namespaceURI>
            <localpart>HelloWorld</localpart>
         </wsdl-port>
         <service-endpoint-interface>pack.HelloWorld</service-endpoint-interface>
         <service-impl-bean>
            <servlet-link>pack_HelloWorld</servlet-link>
         </service-impl-bean>
         <handler>
            <handler-name>handler.PerformanceHandler</handler-name>
            <handler-class>handler.PerformanceHandler</handler-class>
         </handler>
      </port-component>
   </webservice-description>
</webservices>
```

Multiple handlers would be defined here to form a chain as mentioned above.
If multiple handlers that are involved in one service invocation need to share information, they can do so by adding properties to the message context as it is passed from handler to handler. This message context will be available from the request to the response. In other words, we can use it to store information on an incoming request that we can reuse when the response comes back.
Now let's look at how you can create a handler that measures the response time of your service implementation. We will assume that you have created a `HelloWorld` web service, which simply returns a `String` message. Listing below shows the code for the service implementation bean:

```
public class HelloWorld {
        public String helloWorld(String message) {
                return "Hello " + message;
        }
}
```

The handler will be configured for the server that hosts the Web service. It will be invoked on both the request and the response message, so that we can measure the elapsed time.
Each handler must implement the `javax.xml.rpc.handler.Handler` interface. Or, you can make your life a bit easier by simply inheriting the `javax.xml.rpc.handler.GenericHandler` class, which provides default implementations for all the methods. For storing performance results, we use a class called `Logger`, which we set up in the `init()` method. Moreover, the application server passes a `javax.xml.rpc.handler.HandlerInfo` object into this method, which we need to cache as well:

```
public class PerformanceHandler extends GenericHandler {
        protected HandlerInfo info = null;
        protected Logger logger = null;
        public void init(HandlerInfo arg) {
                info = arg;
                logger = Logger.getLogger("c://temp//HelloWorldServiceLog");
        }
        public void destroy() {
                try {          logger.close();}
                catch (Exception x) {}
        }
        ...
}
```

Note that we close the `Logger` object when the handler instance is destroyed.
Each handler implements the `handleRequest` method, which is invoked when a request message arrives:

```
public boolean handleRequest(MessageContext context) {
        try {
                Date startTime = new Date();
                context.setProperty("startTime", startTime);
        } catch (Exception x) {
                x.printStackTrace();
        }
        return true;
}
```

Here you can see that we store the current time in the message context as a property called "`startTime`". The application server will guarantee that the same message context object is passed into the `handleResponse` method, so that we can measure the elapsed time there:

```
public boolean handleResponse(MessageContext context) {
        try {
                Date startTime = (Date)context.getProperty("startTime");
                Date endTime = new Date();
                long elapsedTime = endTime.getTime() - startTime.getTime();
                logger.write("Elapsed time is " + elapsedTime+"\n");
        } catch (Exception x) {
                x.printStackTrace();
        }
        return true;
}
```

JAX-RPC defines a mechanism with which you can manage service invocations by intercepting request and response messages without having to change the actual service consumer or provider. In J2EE, you can configure handlers in a deployment descriptor, without writing any code, providing you with a powerful way of controlling SOAP messages as they pass through your system.
Handlers let you access/modify SOAP request and response messages, but typically used to process service contexts in SOAP header blocks.
Possible example handlers: encryption, decryption, authentication, authorization, logging, auditing, caching.
Handlers are pluggable and chainable through standardized programming API, portable across implementations.
Handler has its own lifecycle: JAX-RPC runtime system calls `init(...)`, `destroy()` of a handler. Handler instances can be pooled (stateless). `MessageContext` is used to share properties among handlers in a handler chain.
On the service client side: a request handler is invoked before an RPC request is communicated to the target service endpoint, a response or fault handler is invoked before an RPC response is returned to the client.
On a service endpoint: a request handler is invoked before an RPC request is dispatched to the target service endpoint, a response or fault handler is invoked before communication back to the service client from the target service endpoint.
`javax.xml.rpc.handler.Handler` interface is required to be implemented by a SOAP message handler:

```
package javax.xml.rpc.handler;
import javax.xml.namespace.QName;
public interface Handler {

    public boolean handleRequest(MessageContext context); // 'false' will block the
HandlerChain
    public boolean handleResponse(MessageContext context); // 'false' will block the
HandlerChain
    public boolean handleFault(MessageContext context); // 'false' will block the
HandlerChain
    public abstract void init(HandlerInfo config);
    public abstract void destroy();
    public QName[] getHeaders();
}
```

or extend `GenericHandler` abstract class:

```
package javax.xml.rpc.handler;
import javax.xml.namespace.QName;
public abstract class GenericHandler implements Handler {
    protected GenericHandler() {}
    public boolean handleRequest(MessageContext context) {
        return true;
    }
    public boolean handleResponse(MessageContext context) {
        return true;
    }
    public boolean handleFault(MessageContext context) {
        return true;
    }
    public void init(HandlerInfo config) {}
    public void destroy() {}
    public abstract QName[] getHeaders();
}
```

Example of generic SOAP message handler:

```
package com.example;
public class SampleSOAPHeaderHandler implements javax.xml.rpc.handler.Handler {
        public SampleSOAPHeaderHandler() { ... }
        public boolean handleRequest(MessageContext context) {
                try {
                        SOAPMessageContext smc = (SOAPMessageContext)context;
                        SOAPMessage msg = smc.getMessage();
                        SOAPPart sp = msg.getSOAPPart();
                        SOAPEnvelope se = sp.getEnvelope();
                        SOAPHeader sh = se.getHeader();
                        // Process one or more header blocks
                        // ...
                        // Next step based on the processing model for this handler
                } catch(Exception ex) {
                        // throw exception
                }
        }
        ...
}
```

Sample handler which extends `GenericHandler`:

```
package com.example;
public class SampleSOAPHeaderHandler extends GenericHandler {
        public boolean handleRequest(MessageContext ctx) {
                try {
                        SOAPMessageContext mc = (SOAPMessageContext)ctx;
                        SOAPMessage msg = mc.getMessage();
                        SOAPPart sp = msg.getSOAPPart();
                        SOAPEnvelop se = sp.getEnvelope();
                        SOAPHeader header= se.getHeader();

                        // now we can process the header

                if (everything is fine) {
                        return true; // with 'true' handler chain continues processing
                        } else {
                        return false; // return 'false' results in chaining to stop
                        }
                } catch(Exception ex) {}
        }
}
```

Configurable on both client and endpoint side:

- Server side in `webservices.xml` file.
- Client side in `webservicesclient.xml` or `web.xml` or `ejb-jar.xml` file (inside the `service-ref` tag).

`webservices.xml` (server-side config):

```
<webservices>
    <webservice-description>
        ...
        <port-component>
            <port-component-name>GetQuote</port-component-name>
            <wsdl-port>
                <namespaceURI>http://example.org</namespaceURI>
                <localpart>GetQuote</localpart>
            </wsdl-port>
            ...
            <handler>
                <handler-name>server1</handler-name>
                <handler-class>service.ServerHandler1</handler-class>
                <init-param>
                    <param-name>name</param-name>
                    <param-value>My Handler Name</param-value>
                </init-param>
                <soap-header>
                    <namespaceURI>http://sample</namespaceURI>
                    <localpart>QuoteHeader</localpart>
                </soap-header>
                <soap-role>MyServerHandler</soap-role>
            </handler>
            <handler>
                <handler-name>server2</handler-name>
                <handler-class>service.ServerHandler2</handler-class>
            </handler>
            ...
        </port-component>
    </webservice-description>
</webservices>
```

Observe that there are 2 handlers defined to run in the order (for incoming message): `server1` -> `server2`. Also observe that the server-side Handler Chain is associated with a particular '`Port`' of a Web Service by `port-component` tag.

Server-side handler:

```
package service;
import javax.xml.namespace.QName;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;
import javax.xml.rpc.handler.*;
import javax.xml.soap.*;
/**
* This class implements a handler by extending the abstract
* class javax.xml.rpc.handler.GenericHandler.
*/
public class ServerHandler1 extends GenericHandler {
        private HandlerInfo handlerInfo;
        private String name;
        public void init(HandlerInfo info) {
                handlerInfo = info;
                // this parameter was configured in 'webservices.xml'
                name = (String) info.getHandlerConfig().get("name");
                System.out.println("ServerHandler1: name = " + name);
        }
        /*
        * This method is declared abstract in GenericHandler and must
        * be defined here.
        */
        public QName[] getHeaders() {
                return handlerInfo.getHeaders();
        }
        /*
        * This handler will check incoming messages for the header
        * specified in 'webservices.xml'. It doesn't do anything with the
        * information besides output it, but it could be used to determine
        * what type of processing should be performed on this message
        * before passing on to the ultimate recipient.
        */
        public boolean handleRequest(MessageContext context) {
                // ...
        }
}
```

webservicesclient.xml or web.xml or ejb-jar.xml (client-side config):

```
<webservicesclient>
    <service-ref>
        <description>WSDL Service StockQuoteService</description>
        <service-ref-name>service/StockQuoteService</service-ref-name>
        <service-interface>sample.StockQuoteService</service-interface>
        <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
        <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
        <service-qname>
            <namespaceURI>http://sample</namespaceURI>
            <localpart>StockQuoteService</localpart>
        </service-qname>
        <port-component-ref>
      <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
        </port-component-ref>
        <handler>
            <handler-name>client1</handler-name>
            <handler-class>client.ClientHandler1</handler-class>
            <init-param>
                <param-name>name</param-name>
                <param-value>My Client Handler</param-value>
            </init-param>
            <soap-header>
                <namespaceURI>http://sample</namespaceURI>
                <localpart>GetQuote</localpart>
            </soap-header>
            <soap-role>LoggingHandler</soap-role>
            <port-name>GetQuote</port-name>
        </handler>
    </service-ref>
</webservicesclient>
```

NOTE: Unlike the server-side handlers, client-side handlers are associated with service-ref (service references) instead of port-component (port component references). However, they have a configurable parameter, port-name, by which handlers can be associated with the port of the service which is invoked. With port names, you can restrict which handlers to run when a service endpoint (WSDL port) is invoked. In the above case, the handler 'client1' is only run if 'GetQuote' is invoked.

Client-side handler:

```
package client;
import javax.xml.namespace.QName;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;
import javax.xml.rpc.handler.*;
import javax.xml.soap.*;
/**
* This class implements a handler by extending the abstract
* class javax.xml.rpc.handler.GenericHandler.
*/
public class ClientHandler1 extends GenericHandler {
        HandlerInfo handlerInfo;
        String name;
        public void init(HandlerInfo info) {
                handlerInfo = info;
                // this parameter was configured in deployment descriptor
                name = (String) info.getHandlerConfig().get("name");
                System.out.println("ClientHandler1: name = " + name);
        }
        /*
        * This method is declared abstract in GenericHandler and must
        * be defined here. Another way to implement is to keep an array
        * of QNames and set them in the init() method to info.getHeaders().
        */
        public QName[] getHeaders() {
                return handlerInfo.getHeaders();
        }
        public boolean handleRequest(MessageContext context) {
                try {
                        // get the soap header
                        SOAPMessageContext smc = (SOAPMessageContext) context;
                        SOAPMessage message = smc.getMessage();
                        SOAPPart soapPart = message.getSOAPPart();
                        SOAPEnvelope envelope = soapPart.getEnvelope();
                        SOAPHeader header = message.getSOAPHeader();
                        if (header == null) {
                                header = envelope.addHeader();
                        }
                        // Add logger element with mustUnderstand="1".
                        // Will use the default actor "next" for this example, otherwise
                        // Use loggerElement.setActor(String actorURI) to define actor.
                        // The element will contain a ficticuous log level for
                        // this example.
                        System.out.println("ClientHandler1: adding loggerElement");
                        SOAPHeaderElement loggerElement = header.addHeaderElement(
                        envelope.createName("loginfo", "ns1", "http://example.com/"));
                        loggerElement.setMustUnderstand(true);
                        loggerElement.setValue("10");
                        // Add simple element describing the client making the request.
                        System.out.println("ClientHandler1: adding nameElement");
                        SOAPHeaderElement nameElement = header.addHeaderElement(
                envelope.createName("clientname", "ns1", "http://example.com/"));
                        nameElement.addTextNode("Duke");
                } catch (Exception e) {
                        throw new JAXRPCException(e);
                }
                // return true to continue message processing
                return true;
        }
}
```

this handler will add following headers to processed SOAP message:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
    <soap:Header>

        <ns1:loginfo xmlns:ns1="http://example.com/"
            soap:mustUnderstand="1"
            soap:actor="http://schemas.xmlsoap.org/soap/actor/next">
            10
        <n/s1:loginfo>

        <ns1:clientname xmlns:ns1="http://example.com/"
            soap:actor="http://schemas.xmlsoap.org/soap/actor/next">
            Duke
        </ns1:clientname>
    </soap:Header>
    <soap:Body>
        ...
    </soap:Body>
</soap:Envelope>
```

A handler chain represents an ordered list of handlers. This grouping helps you define policies that you want associated with the handler invocation model. Examples of such policies include order of invocation, style of invocation (for example, a one-way call invokes only `handleRequest()`; no `handleResponse()`), etc. Another possible policy you can set on the handler chain: A handler chain can invoke a handler based on the `qname` of the outermost element of a SOAP header. This association can be configured to the handler through the `Handler.init(...)` method passing a `HandlerInfo` object. The handler chain continues processing the handlers only if the current processing handler returns `true`. You can associate a handler chain with SOAP actors (or roles) by specifying the URIs of the actors. By default, a handler chain is always associated with the special SOAP actor "`http://schemas.xmlsoap.org/soap/actor/next`". A server-side handler chain is registered on a per service endpoint basis, as indicated by the qualified name of the WSDL port.

```
package javax.xml.rpc.handler;
import java.util.List;
import java.util.Map;
public interface HandlerChain extends List {
    public boolean handleRequest(MessageContext context);
    public boolean handleResponse(MessageContext context);
    public boolean handleFault(MessageContext context);
    public void init(Map config);
    public void destroy();
    public void setRoles(String[] soapActorNames);
    public java.lang.String[] getRoles();
}
```

In the following example, three `Handler` instances `Handler_1`, `Handler_2` and `Handler_3` are registered (in this order) in a single `HandlerChain` instance that is used for both request and response processing. The default invocation order for these handlers is as follows:

1. `Handler_1.handleRequest`
2. `Handler_2.handleRequest`
3. `Handler_3.handleRequest`
4. `Handler_3.handleResponse`
5. `Handler_2.handleResponse`
6. `Handler_1.handleResponse`

Handlers Concepts

A Handler can be likened to a Servlet Filter in that it is business logic that can examine and potentially modify a request before it is processed by a Web Service component. It can also examine and potentially modify the response after the component has processed the request. Handlers can also run on the client before the request is sent to the remote host and after the client receives a response.

JAX-RPC Handlers are specific to SOAP requests only and cannot be used for other non-SOAP Web services. Handlers may be transport independent. For instance, a Handler as defined by JAX-RPC may be usable for SOAP/JMS in addition to SOAP/HTTP if a JMS protocol binding was available. Handlers for non-SOAP encodings have not been defined yet.

Handlers are service specific and therefore associated with a particular Port component or port of a Service interface. This association is defined in the deployment descriptors. They are processed in an ordered fashion called a *Handler Chain*, which is defined by the deployment descriptors.

There are several scenarios for which Handlers may be considered. These include application specific SOAP header processing, logging, and caching. A limited form of encryption is also possible. For application specific SOAP header processing, it is important to note that the client and server must agree on the header processing semantics without the aid of a WSDL description that declares the semantic requirements. Encryption is limited to a literal binding in which the SOAP message part maps to a `SOAPElement`. In this case, a value within the `SOAPElement` may be encrypted as long as the encryption of that value does not change the structure of the `SOAPElement`.

Some Handler scenarios described within the JAX-RPC specification are not supported by this specification. For example, auditing cannot be fully supported because there is no means for a Handler to obtain the Principal. The secure stock quote example cannot be supported as stated because encrypting the body would prevent the container from determining which Port component the request should be directed to and therefore which Handler should decrypt the body.

A Handler always runs under the execution context of the application logic. On the client side, the Stub/proxy controls Handler execution. Client side Handlers run after the Stub/proxy has marshaled the message, but before container services and the transport binding occurs. Server side Handlers run after container services have run including method level authorization, but before demarshalling and dispatching the SOAP message to the endpoint. Handlers can access the `java:comp/env` context for accessing resources and environment entries defined by the Port component the Handler is associated with.

Handlers are constrained by the J2EE managed environment. Handlers are not able to re-target a request to a different component. Handlers cannot change the WSDL operation nor can Handlers change the message part types and number of parts. On the server, Handlers can only communicate with the business logic of the component using the `MessageContext`. On the client, Handlers have no means of communicating with the business logic of the client. There is no standard means for a Handler to access the security identity associated with a request, therefore Handlers cannot portably perform processing based on security identity.

Handlers are associated with the Port component on the server and therefore run in both the web and EJB containers.

J2EE applications that define one or more port components or service references include WSDL descriptions for each of them as well as application logic and (optionally) SOAP message handlers associated with them. In order for such applications to behave predictably, all three elements (description, handlers and application logic) must be well aligned. Developers should program handlers carefully in order not to create invalid SOAP envelope format that contradicts WS-I BP 1.0 requirements or violates the message schema declared in the WSDL. In particular, containers cannot provide any guarantees beyond those specified as part of the interoperability requirements on the behavior of an application that violates the assumptions embedded in a WSDL document either in its business logic or in SOAP message handlers.

Handlers Scenarios

Scenario 1: Handlers must be able to transform the SOAP header. One example is the addition of a SOAP header for application specific information, like `customerId`, by the handler.

Scenario 2: Handlers must be able to transform just parts of the body. This might include changing part values within the SOAP body. Encryption of some parameter values is an example of this scenario.

Scenario 3: Handlers must be able to just read a message where no additions, transformations, or modification to the message is made. Common scenarios are logging, metering, and accounting.

Handlers Programming Model

A Web Services for J2EE provider is required to provide all interfaces and classes of the `javax.xml.rpc.handler` package. The `HandlerInfo` `setHandlerConfig()` and `getHandlerConfig()` methods do not affect the container's Handler request processing. A Web Services for J2EE provider is not required to provide an implementation of `HandlerRegistry`. This functionality is specific to the container. A Web Services for J2EE provider is required to provide an implementation of `MessageContext`. A Web Services for J2EE provider is required to provide all the interfaces of the `javax.xml.rpc.handler.soap` package. The provider must also provide an implementation of the `SOAPMessageContext` interface. The programming model of a Port component can be single-threaded or multi-threaded. The concurrency of a JAX-RPC Handler must match the concurrency of the business logic it is associated with. Client handlers may need to support multi-threaded execution depending on the business logic which is accessing the Port. Handlers must be loaded using the same class loader the application code was loaded with. The class loading rules follow the rules defined for the container the Handler is running in.

The `init` and `destroy` methods of the `Handler` interface allows the container to notify a `Handler` instance of impending changes in its state. The `Handler` may use the notification to prepare its internal state for the transition. The container is required to call the `init` and `destroy` methods as described. The container must call the `init` method before it can start dispatching requests to the `handleRequest()`, `handleResponse()`, and `handleFault()` methods of the `Handler`. The `Handler` may use the container notification to ready its internal state for receiving requests. The container must notify the `Handler` of its intent to remove the instance from the container's working set by calling the `destroy` method. A container must not call the `destroy` method while a request is being processed by the `Handler` instance. The container must not dispatch additional requests to the `Handler` interface methods after the `destroy` method is called. As defined by JAX-RPC, a `RuntimeException` (other than `SOAPFaultException`) thrown from any method of the `Handler` results in the `destroy` method being invoked and transition to the "Does Not Exist" state. Pooling of `Handler` instances is allowed, but is not required. If `Handler` instances are pooled, they must be pooled by Port component. This is because `Handlers` may retain non-client specific state across method calls that are specific to the Port component. For instance, a `Handler` may initialize internal data members with Port component specific environment values. These values may not be consistent when a single `Handler` type is associated with multiple Port components. Any pooled instance of a Port component's `Handler` in a "Method Ready" state may be used to service the `handleRequest()`, `handleResponse()`, and `handleFault()` methods. It is not required that the same `Handler` instance service both the `handleRequest()` and `handleResponse()` or `handleFault()` method invocations of any given request.

A developer is not required to implement a Handler. Handlers are another means of writing business logic associated with processing a Web services request. A developer may implement zero or more Handlers that are associated with a Port component and/or a Service reference. If a developer implements a Handler, they must follow the requirements outlined in this section. A Handler is implemented as a stateless instance. A Handler does not maintain any message processing (client specific) related state in its instance variables across multiple invocations of the handle_XXX method. A Handler class must implement the javax.xml.rpc.handler.Handler interface:

```
package javax.xml.rpc.handler;
public interface Handler {
        boolean handleRequest(MessageContext context);
        boolean handleResponse(MessageContext context);
        boolean handleFault(MessageContext context);
        // ...
}
```

A Handler.handle<action>() method may access the component's environment entries by using JNDI lookup of the "java:comp/env" contenxt and accessing the env-entry-name defined in the deployment descriptor by performing a JNDI lookup. The container may throw a java.lang.IllegalStateException if the environment is accessed from any other Handler method and the environment is not available. In addition, the Handler may use java.util.Map HandlerInfo.getHandlerConfig() method to access the Handler's init-params declared in the deployment descriptor. The Handler.init() method must retain the information defined by HandlerInfo.getHeaders(). A Handler implementation must implement the getHeaders() method to return the results of the HandlerInfo.getHeaders() method. The headers that a Handler declares it will process (i.e. those returned by the Handler.getHeaders() method must be defined in the WSDL definition of the service. A Handler implementation should test the type of the MessageContext passed to the Handler in the handle<action>() methods. Although this specification only requires support for SOAP messages and the container will pass a SOAPMessageContext in this case, some providers may provide extensions that allow other message types and MessageContext types to be used. A Handler implementation should be ready to accept and ignore message types which it does not understand. A Handler implementation must use the MessageContext to pass information to other Handler implementations in the same Handler Chain and, in the case of the JAX-RPC service endpoint, to the Service Implementation Bean.

```
package javax.xml.rpc.handler;
public interface MessageContext {
        void setProperty(String name, Object value);
        Object getProperty(String name);
        void removeProperty(String name);
        boolean containsProperty(String name);
        java.util.Iterator getPropertyNames();
}
```

A container is not required to use the same thread for invoking each Handler or for invoking the Service Implementation Bean. A Handler may access the env-entry elements of the component it is associated with by using JNDI to lookup an appropriate subcontext of java:comp/env. Access to the java:comp/env contexts must be supported from the init() and handle<action>() methods. Access may not be supported within the destroy() method. A Handler may access transactional resources defined by a component's resource-refs. Resources are accessed under a transaction context. A Handler may access the complete SOAP message and can process both SOAP header blocks and body if the handle<action>() method is passed a SOAPMessageContext:

```
package javax.xml.rpc.handler.soap;
public interface SOAPMessageContext extends MessageContext {
        SOAPMessage getMessage();
        void setMessage(SOAPMessage message);
        // ...
}
```

A SOAPMessageContext Handler may add or remove headers from the SOAP message. A SOAPMessageContext Handler may modify the header of a SOAP message if it is not mapped to a parameter or if the modification does not change value type of the parameter if it is mapped to a parameter. A Handler may modify part values of a message if the modification does not change the value type. Handlers that define

application specific headers should declare the header schema in the WSDL document for the component they are associated with, but are not required to do so.
A container is required to provide an instance of a java.util.Map object in the HandlerInfo instance:

```
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 void setHandlerClass(Class handlerClass) { ... }
        public Class getHandlerClass() { ... }
        public void setHandlerConfig(java.util.Map config) { ... }
        public java.util.Map getHandlerConfig() { ... }
        public QName[] getHeaders() { ... }
        public void setHeaders(QName[] headers) { ... }
}
```

The HandlerInfo.getHeaders() method must return the set of soap-header elements defined in the deployment descriptor. The Map object must provide access to each of the Handler's init-param name/value pairs declared in the deployment descriptor as java.lang.String values. The container must provide a unique HandlerInfo instance and Map config instance for each Handler instance. A unique Handler instance must be provided for each Port component declared in the deployment descriptor. The container must call the init() method within the context of a Port component's environment. The container must ensure the Port component's env-entrys are setup for the init method to access. The container must provide a MessageContext type unique to the request type. For example, the container must provide a SOAPMessageContext to the handle<action>() methods of a Handler in a handler chain when processing a SOAP request. The SOAPMessageContext must contain the complete SOAP message:

```
package com.example;
public class MySOAPMessageHandler extends javax.xml.rpc.handler.GenericHandler {
        public MySOAPMessageHandler() { ... }
        public boolean handleRequest(MessageContext context) {
                try {
                        SOAPMessageContext smc = (SOAPMessageContext)context;
                        SOAPMessage msg = smc.getMessage();
                        SOAPPart sp = msg.getSOAPPart();
                        SOAPEnvelope se = sp.getEnvelope();
                        SOAPHeader sh = se.getHeader();
                        // Process one or more header blocks
                        // ...
                        // Next step based on the processing model for this
                        // handler
                } catch(Exception ex) {
                        // throw exception
                }
        }
        // Other methods: handleResponse, handleFault, init, destroy
}
```

The container must share the same MessageContext instance across all Handler instances and the target endpoint that are invoked during a single request and response or fault processing on a specific node. The container must setup the Port component's execution environment before invoking the handle<action>() methods of a handler chain. Handlers run under the same execution environment as the Port component's business methods. This is required so that handlers have access to the Port component's java:comp/env context.

Chapter 5. SOAP and XML Processing APIs (JAXP, JAXB, and SAAJ)

Describe the functions and capabilities of the APIs included within JAXP.

SAX

SAX allows you to process a document as it's being read, which avoids the need to wait for all of it to be stored before taking action.
Consider the following XML code snippet:

```
<?xml version="1.0"?>
<samples>
        <server>UNIX</server>
        <monitor>color</monitor>
</samples>
```

A SAX processor analyzing this code snippet would generate, in general, the following events:

```
Start document
Start element (samples)
Characters (white space)
Start element (server)
Characters (UNIX)
End element (server)
Characters (white space)
Start element (monitor)
Characters (color)
End element (monitor)
Characters (white space)
End element (samples)
End document
```

The SAX API allows a developer to capture these events and act on them.
SAX processing involves the following steps:

1. Create an event handler.
2. Create the SAX parser.
3. Assign the event handler to the parser.
4. Parse the document, sending each event to the handler.

Create the parser
This example uses a pair of classes, `SAXParserFactory` and `SAXParser`, to create the parser, so you don't have to know the name of the driver itself. First declare the `XMLReader`, `xmlReader`, and then use `SAXParserFactory` to create a `SAXParser`. It's the `SAXParser` that gives you the `XMLReader`:

```
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.XMLReader;

public class SurveyReader extends DefaultHandler {
        public static void main (String args[]) {
                XMLReader xmlReader = null;
                try {
                        SAXParserFactory spfactory = SAXParserFactory.newInstance();
                        SAXParser saxParser = spfactory.newSAXParser();
                        xmlReader = saxParser.getXMLReader();
                } catch (Exception e) {
                        System.err.println(e);
                }
        }
}
```

Set validation options
Turn off validation for any parser the `SAXParserFactory` creates by setting the validating property:

```
...
public static void main (String args[]) {
        XMLReader xmlReader = null;
        try {
                SAXParserFactory spfactory = SAXParserFactory.newInstance();
                spfactory.setValidating(false);
                SAXParser saxParser = spfactory.newSAXParser();
                xmlReader = saxParser.getXMLReader();
        } catch (Exception e) {
                System.err.println(e);
        }
}
...
```

Set the content handler

The parser has to send its events to a `ContentHandler`:

```
package org.xml.sax;
public interface ContentHandler {
    public void setDocumentLocator (Locator locator);
    /**
     * Receive notification of the beginning of a document.
     */
    public void startDocument () throws SAXException;
    /**
     * Receive notification of the end of a document.
     */
    public void endDocument() throws SAXException;
    /**
     * Begin the scope of a prefix-URI Namespace mapping.
     */
    public void startPrefixMapping (String prefix, String uri) throws SAXException;
    /**
     * End the scope of a prefix-URI mapping.
     */
    public void endPrefixMapping (String prefix) throws SAXException;
    /**
     * Receive notification of the beginning of an element.
     */



    public void startElement (String namespaceURI, String localName,
                         String qName, Attributes atts) throws SAXException;
    /**
     * Receive notification of the end of an element.
     */
    public void endElement (String namespaceURI, String localName,
                       String qName) throws SAXException;
    /**
     * Receive notification of character data.
     */
    public void characters (char ch[], int start, int length) throws SAXException;

    /**
     * Receive notification of ignorable whitespace in element content.
     */
    public void ignorableWhitespace (char ch[], int start, int length) throws
SAXException;
    /**
     * Receive notification of a processing instruction.
     */
    public void processingInstruction (String target, String data) throws
SAXException;    /**
     * Receive notification of a skipped entity.
     */
    public void skippedEntity (String name) throws SAXException;
}
```

To keep things simple, `SurveyReader` is both the main application and the content handler, so create a new instance of it and set it as the `ContentHandler` using the `XMLReader.setContentHandler()` method:

```
        ...
        xmlReader = saxParser.getXMLReader();
        xmlReader.setContentHandler(new SurveyReader());
} catch (Exception e) {
        ...
```

Parse the `InputSource`

To actually parse a file you need an `InputSource`:

```
package org.xml.sax;
import java.io.Reader;
import java.io.InputStream;
/**
 * A single input source for an XML entity.
 */
public class InputSource {
    public InputSource ()  { }
    /**
     * Create a new input source with a system identifier.
    public InputSource (String systemId)  {
                setSystemId(systemId);
    }
    /**
     * Create a new input source with a byte stream.
     */
    public InputSource (InputStream byteStream) {
                setByteStream(byteStream);
    }
    /**
     * Create a new input source with a character stream.
     */
    public InputSource (Reader characterStream) {
                setCharacterStream(characterStream);
    }
    ...
}
```

This SAX class wraps whatever data you're going to process. Now you're ready to actually parse the file. The `parse()` method takes the file, wrapped in the `InputSource`, and processes it, sending each event to the `ContentHander`:

```
...
import org.xml.sax.InputSource;
        ...
        xmlReader = saxParser.getXMLReader();
        xmlReader.setContentHandler(new SurveyReader());
        InputSource source = new InputSource("surveys.xml");
        xmlReader.parse(source);
} catch (Exception e) {
        ...
```

Set an `ErrorHandler`

You can create an error handler just as you created a content handler. Normally, you would create this as a separate instance of `ErrorHandler`, but to simplify the example, you'll include error handling right in `SurveyResults`. This dual-usage is possible because the class extends `DefaultHandler`, which includes implementations of both the `ContentHandler` methods and the `ErrorHandler` methods. Set a new `ErrorHandler` just as you set the `ContentHandler`:

```
...
xmlReader.setContentHandler(new SurveyReader());
xmlReader.setErrorHandler(new SurveyReader());
InputSource source = new InputSource("surveys.xml");
...
```

Event handlers and the SAX events

- `startDocument()`
  Start by noting the beginning of the document using the `startDocument()` event. This event, like the other SAX events, throws a `SAXException`:

```
...
import org.xml.sax.SAXException;

public class SurveyReader extends DefaultHandler {

  public void startDocument() throws SAXException {
         System.out.println("Tallying survey results...");
  }

  public static void main (String args[]) {
         XMLReader xmlReader = null;
         ...
```

- `startElement()`
  For each element, the example echoes back the name that is passed to the `startElement()` event. The parser actually passes several pieces of information for each element:
  - The qualified name, or `qName`. This is actually a combination of namespace information, if any, and the actual name of the element. The `qName` also includes the colon (:) if there is one - for example, `revised:question`.
  - The namespace URI. An actual namespace is a URI of some sort and not the alias that gets added to an element or attribute name. For example, `http://www.example.com` as opposed to simply `revised:`.
  - The local name. This is the actual name of the element, such as `question`. If the document doesn't provide namespace information, the parser may not be able to determine which part of the `qName` is the `localName`.
  - Any attributes. The attributes for an element are actually passed as a collection of objects.

  Start by listing the name of each element:

```
...
import org.xml.sax.Attributes;

public class SurveyReader extends DefaultHandler {
  ...
  public void startDocument() throws SAXException {
         System.out.println("Tallying survey results...");
  }

  public void startElement(String namespaceURI, String localName,
                        String qName, Attributes atts) throws SAXException {
         System.out.print("Start element: ");
         System.out.println(qName);
  }
  public static void main (String args[]) {            ...
```

The `startElement()` event also provides access to the attributes for an element. They are passed in within an `Attributes` data structure. You can retrieve an attribute value based on its position in the array, or based on the name of the attribute:

```
...
public void startElement(String namespaceURI, String localName,
         String qName, Attributes atts) throws SAXException {
  System.out.print("Start element: ");
  System.out.println(qName);
  for (int att = 0; att < atts.getLength(); att++) {
         String attName = atts.getQName(att);
         System.out.println(" " + attName + ": " + atts.getValue(attName));
  }
}
...
```

- `endElement()`
  You'll find plenty of good reasons to note the end of an element. For example, it might be a signal to process the contents of an element. Here you'll use it to pretty print the document to some extent, indenting each level of elements. The idea is to increase the value of the indent when a new element starts, decreasing it again when the element ends:

```
...
int indent = 0;

public void startDocument() throws SAXException {
  System.out.println("Tallying survey results...");
  indent = -4;
}

public void printIndent(int indentSize) {
  for (int s = 0; s < indentSize; s++) {
         System.out.print(" ");
  }
}
public void startElement(String namespaceURI, String localName,
                String qName, Attributes atts) throws SAXException {
  indent = indent + 4;
  printIndent(indent);
  System.out.print("Start element: ");
  System.out.println(qName);
  for (int att = 0; att < atts.getLength(); att++) {
         printIndent(indent + 4);
         String attName = atts.getLocalName(att);
         System.out.println(" " + attName + ": " + atts.getValue(attName));
  }
}
public void endElement(String namespaceURI, String localName, String qName)
         throws SAXException {
  printIndent(indent);
  System.out.println("End Element: "+localName);
  indent = indent - 4;
}
...
```

- `characters()`
  Now that you've got the elements, go ahead and retrieve the actual data using `characters()`. Take a look at the signature of this method for a moment:

```
public void characters(char[] ch, int start, int length) throws SAXException
```

Notice that nowhere in this method is there any information whatsoever as to what element these characters are part of. If you need this information, you're going to have to store it. This example adds

variables to store the current element and question information. (It also removes a lot of extraneous information that was displayed.)
Note two important things here:

- Range: The `characters()` event includes more than just a string of characters. It also includes start and length information. In actuality, the `ch` character array includes the entire document. The application must not attempt to read characters outside the range the event feeds to the `characters()` event.
- Frequency: Nothing in the SAX specification requires a processor to return characters in any particular way, so it's possible for a single chunk of text to be returned in several pieces. Always make sure that the `endElement()` event has occurred before assuming you have all the content of an element. Also, processors may use `ignorableWhitespace()` to return whitespace within an element. This is always the case for a validating parser.

```
...
public void printIndent(int indentSize) {
  for (int s = 0; s < indentSize; s++) {
          System.out.print(" ");
  }
}
String thisQuestion = "";
String thisElement = "";
public void startElement(String namespaceURI, String localName,
                 String qName, Attributes atts) throws SAXException {
  if (qName == "response") {
          System.out.println("User: " + atts.getValue("username"));
  } else if (qName == "question") {
          thisQuestion = atts.getValue("subject");
  }
  thisElement = qName;
}
public void endElement(String namespaceURI, String localName, String qName)
          throws SAXException {
  thisQuestion = "";
  thisElement = "";
}
public void characters(char[] ch, int start, int length)
          throws SAXException {
  if (thisElement == "question") {
          printIndent(4);
          System.out.print(thisQuestion + ": ");
          System.out.println(new String(ch, start, length));
  }
}
...
```

- `endDocument()`
  Once the document is completely parsed, you'll want to print out the final tally as shown below. This is also a good place to tie up any loose ends that may have come up during processing:

```
public void endDocument() {
  System.out.println("Appearance of the aliens:");
  System.out.println("A: " + getInstances(appearance, "A"));
  System.out.println("B: " + getInstances(appearance, "B"));
  ...
}
public static void main (String args[]) {
  ...
```

ErrorHandler events
Just as the `ContentHandler` has predefined events for handling content, the `ErrorHandler` has predefined events for handling errors. Because you specified `SurveyReader` as the error handler as well as the content handler, you need to override the default implementations of those methods. You need to be concerned with three events: `warning`, `error`, and `fatalError`:

```
...import org.xml.sax.SAXParseException;
public class SurveyReader extends DefaultHandler {
        public void error (SAXParseException e) {
                System.out.println("Error parsing the file: "+e.getMessage());
        }
        public void warning (SAXParseException e) {
                System.out.println("Problem parsing the file: "+e.getMessage());
        }
        public void fatalError (SAXParseException e) {
                System.out.println("Error parsing the file: "+e.getMessage());
                System.out.println("Cannot continue.");
                System.exit(1);
        }
        ...
```

Resolving Entities. `org.xml.sax.EntityResolver` interface

An XML document is made up of entities. Each entity is identified by public identifiers and/or system identifiers. The system IDs tend to be URLs. The public IDs generally require some sort of catalog system that can convert them into URLs. An XML parser reads each entity using an `InputSource` connected to the right URL. Most of the time you just give the parser a system ID or an `InputSource` pointing to the document entity, and let the parser figure out where to find any further entities referenced from the document entity. However, sometimes you want the parser to read from different URLs than the ones the document specifies. For example, the parser might ask for the XHTML DTD from the W3C web site. However, you might choose to replace that with a cached copy stored locally.

The `EntityResolver` API lets you convert a public ID (URN) into a system ID (URL). Your application may need to do that, for example, to convert something like href="`urn:/someName`" into "`http://someURL`".

The `org.xml.sax.EntityResolver` interface allows you to filter the parser's requests for external parsed entities so you can replace the files it requests with your own copies, either faithful or modified. You might even use this interface to provide some form of custom proxy server support, though chances are that would be better implemented at the socket level rather than in the parsing API.

`org.xml.sax.EntityResolver` is a callback interface much like `ContentHandler`. It is attached to an `org.xml.sax.XMLReader` interface with set and get methods:

```
package org.xml.sax;
public interface XMLReader {
        ...
        public void setEntityResolver(EntityResolver resolver);
        public EntityResolver getEntityResolver();
        ...
}
```

The `EntityResolver` interface contains just a single method, `resolveEntity(...)`. If you register an `EntityResolver` with an `XMLReader`, then every time that `XMLReader` needs to load an external parsed entity, it will pass the entity's public ID and system ID to `resolveEntity(...)` first. The external entities can be: external DTD subset, external parameter entities, etc.

The `EntityResolver` allows you to substitute your own URI lookup scheme for external entities. Especially useful for entities that use URL and URI schemes not supported by Java's protocol handlers; e.g. `jdbc:/` or `isbn:/`.

The `resolveEntity(...)` can either return an `InputSource` or `null`. If it returns an `InputSource`, then this `InputSource` provides the entity's replacement text. If it returns `null`, then the parser reads the entity in the same way it would have if there wasn't an `EntityResolver` - by using the system ID and the `java.net.URL` class.

You could replace the host in the system ID to load the DTDs from a mirror site. You could bundle the DTDs into your application's JAR file and load them from there. You could even hardwire the DTDs in the `EntityResolver` as string literals and load them with a `StringReader`.

```
package org.xml.sax;
public interface EntityResolver {
        public InputSource resolveEntity(String publicId,
                String systemId) throws SAXException, IOException;
}
```

The following resolver will redirect system identifier to local URI:

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import org.xml.sax.EntityResolver;
import org.xml.sax.InputSource;
public class UseLocal implements EntityResolver {
        public InputSource resolveEntity(String publicId, String systemId)
                throws FileNotFoundException {

                if (systemId.equals("http://server.com/DTD/quiz.dtd")) {
                        // use local version
                        return new InputSource(new
FileInputStream("local/DTD/quiz.dtd"));
                } else {
                        // use the default behaviour
                        return null;
                }
        }
}
```

DOM

A DOM Document is a collection of nodes, or pieces of information, organized in a hierarchy. This hierarchy allows a developer to navigate around the tree looking for specific information. Analyzing the structure normally requires the entire document to be loaded and the hierarchy to be built before any work is done. Because it is based on a hierarchy of information, the DOM is said to be tree-based, or object-based.

For exceptionally large documents, parsing and loading the entire document can be slow and resource-intensive. DOM provides an API that allows a developer to add, edit, move, or remove nodes at any point on the tree in order to create an application, while event-based models like SAX do not allow a developer to actually change the data in the original document.

The basic node types

1. Elements
   Elements are the basic building blocks of XML. Typically, elements have children that are other elements, text nodes, or a combination of both. Element nodes are also the only type of node that can have attributes.
2. Attributes
   Attribute nodes contain information about an element node, but are not actually considered to be children of the element, as in:
   ```
   <customerid limit="1000">12341</customerid>
   ```
3. Text
   A text node is exactly that - text. It can consist of more information or just white space.
4. Document
   The document node is the overall parent for all of the other nodes in the document.

Parsing a file into a document

To work with the information in an XML file, the file must be parsed to create a Document object.

The Document object is an interface, so it can't be instantiated directly; generally, the application uses a factory instead. In the example Java environment, parsing the file is a three-step process:

1. Create the DocumentBuilderFactory. This object creates the DocumentBuilder.
2. Create the DocumentBuilder. The DocumentBuilder does the actual parsing to create the Document object.
3. Parse the file to create the Document object.

Start by creating the basic application, a class called OrderProcessor:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.File;
import org.w3c.dom.Document;
public class OrderProcessor {
        public static void main (String args[]) {
                File docFile = new File("orders.xml");
                Document doc = null;
                try {
```

```
                        DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
                        DocumentBuilder db = dbf.newDocumentBuilder();
                        doc = db.parse(docFile);
                } catch (Exception e) {
                        System.out.print("Problem parsing the file: "+e.getMessage());
                }
        }
}
```

First, the Java code imports the necessary classes, and then it creates the OrderProcessor application. Within the try-catch block, the application creates the DocumentBuilderFactory, which it then uses to create the DocumentBuilder. Finally, the DocumentBuilder parses the file to create the Document.

One of the advantages of creating parsers with a DocumentBuilder lies in the control over various settings on the parsers created by the DocumentBuilderFactory. For example, the parser can be set to validate the document:

```
...
try {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setValidating(true);
        DocumentBuilder db = dbf.newDocumentBuilder();
        doc = db.parse(docFile);
} catch (Exception e) {
...
```

Get the root element

Once the document is parsed and a Document is created, an application can step through the structure to review, find, or display information. This navigation is the basis for many operations that will be performed on a Document. Stepping through the document begins with the root element. A well-formed document has only one root element, also known as the DocumentElement. First the application retrieves this element:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.File;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
public class OrderProcessor {
        ...
                //STEP 1: Get the root element
                Element root = doc.getDocumentElement();
                System.out.println("The root element is " + root.getNodeName());
        ...
}
```

Get the children of a node

Once the application determines the root element, it retrieves a list of the root element's children as a NodeList. The NodeList class is a series of items through which the application can iterate. In this example, for brevity, the application gets the child nodes and verifies the retrieval by showing only how many elements appear in the resulting NodeList:

```
...
import org.w3c.dom.NodeList;
        ...
                //STEP 1: Get the root element
                Element root = doc.getDocumentElement();
                System.out.println("The root element is "+root.getNodeName());

                //STEP 2: Get the children
                NodeList children = root.getChildNodes();
                System.out.println("There are "+children.getLength()+" nodes in this
document.");
        }          }
```

Using getFirstChild() and getNextSibling()

The parent-child and sibling relationships offer an alternative means for iterating through all of the children of a node that may be more appropriate in some situations, such as when these relationships and the order in which children appear is crucial to understanding the data. In Step 3, a for-loop starts with the first child of the root. The application iterates through each of the siblings of the first child until they have all been evaluated. Each time the application executes the loop, it retrieves a Node object, outputting its name and value. Notice also that the elements carry a value of null, rather than the expected text. It is the text nodes that are children of the elements that carry the actual content as their values:

```
...
import org.w3c.dom.Node;
        ...
                //STEP 3: Step through the children
                for (Node child = root.getFirstChild(); child != null; child =
child.getNextSibling()) {
                        System.out.println(child.getNodeName() + " = " +
child.getNodeValue());
                }
        }
}
```

A Node object carries member constants that represent each type of node, such as ELEMENT_NODE or ATTRIBUTE_NODE. If the nodeType matches ELEMENT_NODE, it is an element. For every element it finds, the application creates a NamedNodeMap that contains all of the attributes for the element. The application can iterate through a NamedNodeMap, printing each attribute's name and value, just as it iterated through the NodeList:

```
...
import org.w3c.dom.NamedNodeMap;
        ...
        private static void stepThroughAll (Node start) {
                System.out.println(start.getNodeName()+" = "+start.getNodeValue());
                if (start.getNodeType() == start.ELEMENT_NODE) {
                        NamedNodeMap startAttr = start.getAttributes();
                        for (int i = 0; i < startAttr.getLength(); i++) {
                                Node attr = startAttr.item(i);
System.out.println(" Attribute: "+ attr.getNodeName() +" = "+attr.getNodeValue());
                        }
                }
for (Node child = start.getFirstChild();child != null;child = child.getNextSibling())
{
                        stepThroughAll(child);
                }
        }
```

TrAX (Transformation API for XML)

TrAX is an API for transforming XML documents using XSLT style sheets. TrAX is a Java API and has been defined to provide common access to different XSLT Processors. TrAX is part of the JAXP API, which combines a number of Java APIs.

The TrAX API extends the original JAXP mission to include XML transformations: provide a vendor and implementation agnostic standard Java API for specifying and executing XML transformations. This is important to note, because TrAX is more than just a standard interface for XSLT engines - it is designed to be used as a general-purpose transformation interface for XML documents.

TrAX isn't a competitor to the existing DOM and SAX APIs used to represent and process XML, but a common Java API to bridge the various XML transformation methods (a la JDBC, JNDI, etc.) including SAX Events and XSLT Templates. In fact, TrAX relies upon a SAX2- and DOM-level-2-compliant XML parser/XSLT engine. JAXP 1.0 allows the developer to change XML parsers by setting a property, and TrAX provides the same functionality for XSLT engines.

Here is a sample of how to apply an XSLT stylesheet to an XML document and write the results out to a file. In this example, both the stylesheet and the XML document exist as files, but they could just as easily have come from any Java InputStream or Reader class. The same follows for the results of the transformation; I could've just as easily written the results out to any Java OutputStream or Writer class.

```
// 1. create the XML content input source:
//    can be a DOM node, SAX stream, or any Java input stream/reader
String xmlInputFile = "myXMLinput.xml";
Source xmlSource = new StreamSource(new FileInputStream(xmlInputFile));
// 2. create the XSLT Stylesheet input source
//    can be a DOM node, SAX stream, or a java input stream/reader
String xsltInputFile = "myXsltStylesheet.xsl";
Source xsltSource = new StreamSource(new FileInputStream(xsltInputFile));
// 3. create the result target of the transformation
//    can be a DOM node, SAX stream, or a java out stream/reader
String xmlOutputFile = "result.html";
Result transResult = new StreamResult(new FileOutputStream(xmlOutputFile));
// 4. create the transformerfactory and transformer instance
TransformerFactory tf = TransformerFactory.newInstance();
Transformer t = tf.newTransformer(xsltSource);
// 5. execute transformation and fill result target object
t.transform(xmlSource, transResult);
```

The first three stanzas simply establish our inputs and result targets, and aren't that interesting, with one exception. Notice that the XSLT stylesheet isn't handled via a different class in TrAX. It's treated just like any other XML source document, because that's exactly what it is. We use the stream implementations of the `Source` and `Result` interfaces from the `javax.xml.transform.stream` package to handle reading the data from our file streams.

In the fourth stanza, we use the `TransformerFactory` to get an instance of a `Transformer`, and then use the `Source` instance for the XSLT stylesheet we created in the second stanza to define the transformation that this transformer will perform. A `Transformer` actually executes the transformation and assembles the result. A single `Transformer` instance can be reused, but it is not thread-safe.

In this example, the XSLT stylesheet is reprocessed for each successive transformation. A very common case is that the same transformation is applied multiple times to different `Sources`, perhaps in different threads. A more efficient approach in this case is to process the transformation stylesheet once, and save this object for successive transformations. This is achieved through the use of the TrAX `Templates` interface:

```
// we've already set up our content Source instance,
// XSLT Source instance, TransformerFactory, and
// Result target from the previous example

// process the XSLT stylesheet into a Templates instance
// with our TransformerFactory instance
Templates t = tf.newTemplates(xsltSource);

// whenever you need to execute this transformation, create
// a new Transformer instance from the Templates instace
Transformer trans = t.newTransformer();

// execute transformation and fill result target object
trans.transform(xmlSource, transResult);
```

While the `Transformer` performs the transformation, a `Templates` instance is the actual run-time representation of the processed transformation instructions. `Templates` instances may be reused to increase performance, and they are thread-safe. It might seem odd that an interface has a plural name, but it stems from the fact that an XSLT stylesheet consists of a collection of one or more `xsl:template` elements. Each `template` element defines a transformation in that stylesheet, so it follows that the simplest name for a representation of a collection of template elements is `Templates`.

One of the main reasons the TrAX API is so clean and simple is the Interface-driven approach to design. The highest-level interfaces define the essential entities that are being modeled, and the interactions are left to the implementations. The interfaces themselves aren't very interesting. They are essentially marker interfaces.

Source

The `Source` interface is a generic container for existing XML documents that will be used in a transformation as either the input document or the stylesheet. Serve as a single vendor-neutral object for multiple types of input. Implementations of the `Source` interface provide access to the XML document to be processed. TrAX defines `Source` implementations for DOM trees (`DOMSource`); SAX 2.0 InputSources (`SAXSource`); and Java `InputStreams`, `Readers` and any of their derived classes (`StreamSource`).

```
package javax.xml.transform;

public interface Source {
    public void setSystemId(String systemId);
    public String getSystemId();
}
```

Result

The Result interface is a generic container for an XML document that will be produced by a transformation. Serve as a single object for multiple types of output, so there can be simple process method signatures. Implementations of the Result interface provide access to the transformed XML document. TrAX defines Result implementations for DOM trees (DOMResult); SAX 2.0 ContentHandlers (SAXResult); and Java OutputStreams, Writers and any of their derived classes (StreamResult).

```
package javax.xml.transform;

public interface Result  {

  public static final String PI_DISABLE_OUTPUT_ESCAPING;
  public static final String PI_ENABLE_OUTPUT_ESCAPING;

  public void   setSystemId(String url);
  public String getSystemId();
}
```

Templates

Templates is a thread-safe interface that represents a compiled stylesheet. It can quickly create new Transformer objects without having to reread and reparse the original stylesheet. It's particularly useful when you want to use the same stylesheet in multiple threads. The runtime representation of the transformation instructions. A template implementation is the optimized, in-memory representation of an XML transformation that is processed and ready to be executed. Templates objects are safe to use in concurrent threads. To reuse a single Template instance in multiple concurrent threads, multiple Transformer instances would have to be created via the Templates.newTransformer() factory method. Each Transformer instance may be used completely independently in concurrent threads, and both the Templates and the Transformer instances can be reused for subsequent transformations.

```
package javax.xml.transform;
public interface Templates  {

  public Transformer newTransformer() throws TransformerConfigurationException;
  public Properties  getOutputProperties();
}
```

Transformer

Transformer is the abstract class that represents a compiled stylesheet. It transforms Source objects into Result objects. A single Transformer can transform multiple input documents in sequence but not in parallel. Act as a per-thread execution context for transformations, act as an interface for performing the transformation. A Transformer is the object that actually applies the transformation to the source document and creates the result document. However, it is not responsible for outputting, or serializing, the result of the transformation. This is the responsibility of the transformation engine's serializer and this behavior can be modified via the setOutputProperty(java.lang.String name, java.lang.String value) method. The configurable OutputProperties are defined in the OutputKeys class, and are described in the XSLT 1.0 Specification. Transformers are immutable, they cannot change which Templates instance gets applied to the Source.

```
package javax.xml.transform;
public abstract class Transformer  {
  protected Transformer();
  public void transform(Source input, Result output) throws TransformerException;
  public void setParameter(String name, Object value);
  public Object getParameter(String name);
  public void clearParameters();
  public void setURIResolver(URIResolver resolver);
  public URIResolver getURIResolver();
  public void setOutputProperties(Properties serialization) throws
IllegalArgumentException;
  public Properties getOutputProperties();
  public void setOutputProperty(String name, String value) throws
IllegalArgumentException;
  public String getOutputProperty(String name) throws IllegalArgumentException;
  public void setErrorListener(ErrorListener listener) throws
IllegalArgumentException;
  public ErrorListener getErrorListener();
}
```

TransformerFactory

`TransformerFactory` is an abstract factory that creates new `Transformer` and `Templates` objects. The concrete subclass that `newInstance()` instantiates is specified by the `javax.xml.transform.TransformerFactory` Java system property. If this class is not set, a platform dependent default class is chosen. Serve as a vendor-neutral Processor interface for XSLT and similar processors. The `TransformerFactory` is primarily responsible for creating new `Transformers` and `Templates` objects. New instances of `Transformer` are created via the `static newTransformer()` method. Processing `Source` instances into `Templates` objects is handled by the `newTemplates(Source source)` method.

```
package javax.xml.transform;
public abstract class TransformerFactory  {
  protected TransformerFactory();
  public static TransformerFactory newInstance() throws
TransformerFactoryConfigurationError;
  public Transformer newTransformer(Source source) throws
TransformerConfigurationException;
  public Transformer newTransformer() throws TransformerConfigurationException;
  public Templates   newTemplates(Source source) throws
TransformerConfigurationException;

  public Source getAssociatedStylesheet(Source source, String media, String title,
String charset)
     throws TransformerConfigurationException;
  public void setURIResolver(URIResolver resolver);
  public URIResolver getURIResolver();
  public boolean getFeature(String name);
  public void setAttribute(String name, Object value) throws IllegalArgumentException;
  public Object getAttribute(String name) throws IllegalArgumentException;
  public void setErrorListener(ErrorListener listener) throws
IllegalArgumentException;
  public ErrorListener getErrorListener();
}
```

Given a scenario, select the proper mechanism for parsing and processing the information in an XML document.

SAX:

- SAX is an event-driven XML parser that is appropriate for high-speed processing of XML because it does not produce a representation of the data in memory.
- SAX automatically perform structure validation.
- Event-based parsers (SAX) provide a data-centric view of XML. When an element is encountered, the idea is to process it and then forget about it. The event-based parser returns the element, its list of attributes, and the content. This is more efficient for many types of applications, especially searches. It requires less code and less memory since there is no need to build a large tree in memory as you are scanning for a particular element, attribute, and/or content sequence in an XML document.

DOM:

- DOM produces an in-memory data representation, which allows an application to manipulate the contents in memory
- DOM automatically perform structure validation.
- Tree-based parsers (DOM) provide a document-centric view of XML. In tree-based parsing, an in-memory tree is created for the entire document, which is extremely memory-intensive for large documents. All

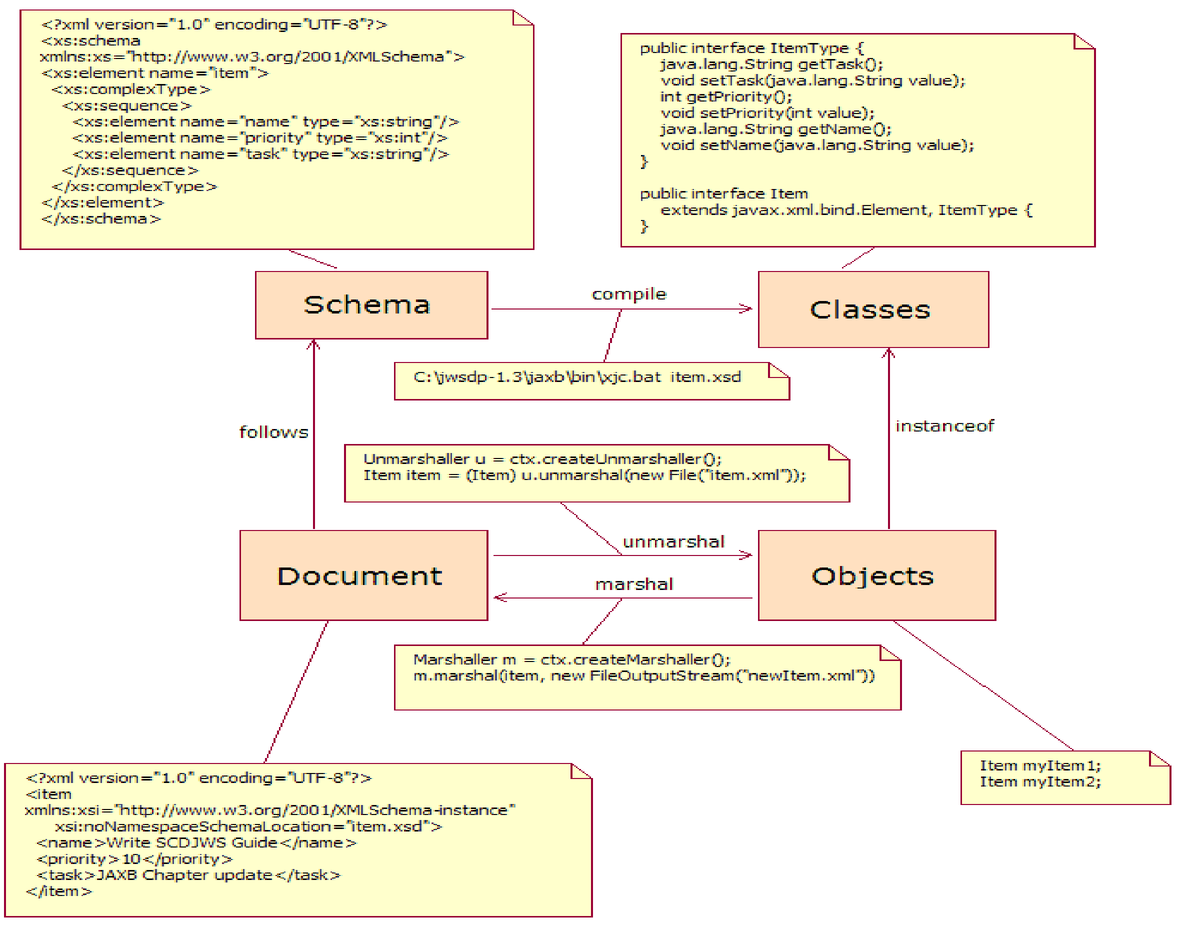
elements and attributes are available at once, but not until the entire document has been parsed. This technique is useful if you need to navigate around the document and perhaps change various document chunks, which is precisely why it is useful for the Document Object Model (DOM), the aim of which is to manipulate documents via scripting languages or Java.

JAXB:

- JAXB application can perform structure and content validation with Java classes that it generates from a schema. A JAXB application builds an in-memory data structure, like a DOM, by marshalling an XML document to build a content tree, which contains objects that are instances of the derived classes. However, unlike a DOM tree, a content tree is specific to one source schema, does not contain extra tree-manipulation functionality, allows access to its data with the derived classes' accessor methods, and is not built dynamically. For these reasons, a JAXB application uses memory more efficiently than a DOM application does. If the content of a document is more dynamic and not well-constrained, DOM and SAX are more appropriate than JAXB for processing XML content that does not have a well-known schema prior to processing the content.
- Build object trees representing XML data that is valid against the XML Schema by either unmarshalling the data from a document or instantiating the classes you created.
- Access and modify the data. Optionally validate the modifications to the data relative to the constraints expressed in the XML Schema.
- Marshal the data to new XML documents.

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.

JAXB defines an architecture for binding XML schemata to Java objects. These bindings allow you to unmarshal XML documents into a hierarchy of Java objects and marshal the Java objects into XML with minimal effort. If you work a lot with XML, you know how tedious it can be to write Simple API for XML (SAX) or Document Object Model (DOM) code to convert XML into Java objects that mean something to your program. JAXB generates code automatically so you can go about the business of processing data instead of parsing it.



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="item">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="priority" type="xs:int"/>
      <xs:element name="task" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

```
public interface ItemType {
    java.lang.String getTask();
    void setTask(java.lang.String value);
    int getPriority();
    void setPriority(int value);
    java.lang.String getName();
    void setName(java.lang.String value);
}

public interface Item
    extends javax.xml.bind.Element, ItemType {
}
```

```
C:\jwsdp-1.3\jaxb\bin\xjc.bat  item.xsd
```

```
Unmarshaller u = ctx.createUnmarshaller();
Item item = (Item) u.unmarshal(new File("item.xml"));
```

```
Marshaller m = ctx.createMarshaller();
m.marshal(item, new FileOutputStream("newItem.xml"))
```

```
Item myItem1;
Item myItem2;
```

```
<?xml version="1.0" encoding="UTF-8"?>
<item
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="item.xsd">
  <name>Write SCDJWS Guide</name>
  <priority>10</priority>
  <task>JAXB Chapter update</task>
</item>
```

Latest version of JAXB supports XML schema definitions and allows additional binding declarations to be defined inside of the schema using XML schema annotations.

The JAXB API, defined in the `javax.xml.bind` package, is a set of interfaces through which client applications communicate with code generated from a schema. The center of the JAXB API is `JAXBContext`, the client' s entry point. It provides an abstraction for managing the XML-Java binding information necessary to implement the JAXB binding framework operations: unmarshal, marshal and validate.
These three aspects of JAXB are covered by three separate interfaces. Instances of those interfaces can be created from a `JAXBContext` object:

- `javax.xml.bind.Unmarshaller` - governs the process of deserializing XML data into Java content trees, optionally validating the XML data as it is unmarshalled.

```
package javax.xml.bind;
import java.io.File;
import java.io.InputStream;
import java.net.URL;
import javax.xml.bind.JAXBException;
import javax.xml.bind.PropertyException;
import javax.xml.bind.UnmarshallerHandler;
import javax.xml.bind.ValidationEventHandler;
import javax.xml.transform.Source;

import org.w3c.dom.Node;
import org.xml.sax.InputSource;

public interface Unmarshaller {
     public Object unmarshal(File f) throws JAXBException;
     public Object unmarshal(InputStream is) throws JAXBException;
     public Object unmarshal(URL url) throws JAXBException;
     public Object unmarshal(InputSource source) throws JAXBException;
     public Object unmarshal(Node node) throws JAXBException;
     public Object unmarshal(Source source) throws JAXBException;

     public UnmarshallerHandler getUnmarshallerHandler();
     public void setValidating(boolean validating) throws JAXBException;
     public boolean isValidating() throws JAXBException;
     public void setEventHandler(ValidationEventHandler handler) throws
JAXBException;
     public ValidationEventHandler getEventHandler() throws JAXBException;
     public void setProperty(String name, Object value) throws PropertyException;
     public Object getProperty(String name) throws PropertyException;
}
```

- `javax.xml.bind.Marshaller` - governs the process of serializing Java content trees back into XML data.

```
package javax.xml.bind;
import java.io.OutputStream;
import java.io.Writer;
import javax.xml.bind.JAXBException;
import javax.xml.bind.PropertyException;
import javax.xml.bind.ValidationEventHandler;
import javax.xml.transform.Result;
import org.w3c.dom.Node;
import org.xml.sax.ContentHandler;
public interface Marshaller {
     public static final String JAXB_ENCODING = "jaxb.encoding";
     public static final String JAXB_FORMATTED_OUTPUT = "jaxb.formatted.output";
     public static final String JAXB_SCHEMA_LOCATION = "jaxb.schemaLocation";
     public static final String JAXB_NO_NAMESPACE_SCHEMA_LOCATION =
"jaxb.noNamespaceSchemaLocation";
     public void marshal(Object obj, Result result) throws JAXBException;
     public void marshal(Object obj, OutputStream os) throws JAXBException;
     public void marshal(Object obj, Writer writer) throws JAXBException;
     public void marshal(Object obj, ContentHandler handler) throws JAXBException;
     public void marshal(Object obj, Node node) throws JAXBException;
     public org.w3c.dom.Node getNode(Object contentTree) throws JAXBException;
     public void setProperty(String name, Object value) throws PropertyException;
     public Object getProperty(String name) throws PropertyException;
     public void setEventHandler(ValidationEventHandler handler) throws 
JAXBException;
     public ValidationEventHandler getEventHandler() throws JAXBException;
}
```

- `javax.xml.bind.Validator` - performs the validation on an in-memory object graph.

```
package javax.xml.bind;
import javax.xml.bind.JAXBException;
import javax.xml.bind.PropertyException;
import javax.xml.bind.ValidationEventHandler;
public interface Validator {
     public void setEventHandler(ValidationEventHandler handler) throws 
JAXBException;
     public ValidationEventHandler getEventHandler() throws JAXBException;
     public boolean validate(Object subrootObj) throws JAXBException;
     public boolean validateRoot(Object rootObj) throws JAXBException;
     public void setProperty(String name, Object value) throws PropertyException;
     public Object getProperty(String name) throws PropertyException;
}
```

`JAXBContext` is an `abstract` class defined in the API, so its actual implementation is vendor-dependent. To create a new instance of `JAXBContext`, you use the `static` `newInstance(contextPath)` method.

```
JAXBContext context = JAXBContext.newInstance("org.acme.foo:org.acme.bar");
```

The `contextPath` contains a list of Java package names that contain schema derived interfaces. The value of this parameter initializes the `JAXBContext` object so that it is capable of managing the schema derived interfaces.
The client application must supply a context path which is a list of colon (':') separated java package names that contain schema derived classes. In this way, the unmarshaller will look at a document and figure out which package to use. This makes it easy to read in different types of documents without knowing their type in advance.

Unmarshalling (XML-to-Java)

An unmarshaller is used to read XML and build an object tree from classes generated by the compiler. To read an XML file, you would simply do:

```
...
JAXBContext jaxbContext = JAXBContext.newInstance(packageName);
Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
unmarshaller.setValidating(true);
Item item = (Item) unmarshaller.unmarshal(new File("item.xml"));
...
```

There are other overloaded versions that take different types of input, such as `InputStream` or `InputSource`. You can even unmarshal a `javax.xml.transform.Source` object. All in all, it's similar to the way DOM trees are parsed.
Here are some more examples of unmarshalling:
Unmarshalling from a `File`:

```
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );
Unmarshaller u = jc.createUnmarshaller();
Object o = u.unmarshal( new File( "xmlFile.xml" ) );
```

Unmarshalling from a `java.io.InputStream`:

```
InputStream is = new FileInputStream( "xmlFile.xml" );
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );
Unmarshaller u = jc.createUnmarshaller();
Object o = u.unmarshal( is );
```

Unmarshalling from a `java.net.URL`:

```
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );
Unmarshaller u = jc.createUnmarshaller();
```

```
URL url = new URL( "http://server.com/xmlFile.xml" );
Object o = u.unmarshal( url );
Unmarshalling from a StringBuffer using a javax.xml.transform.
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );
Unmarshaller u = jc.createUnmarshaller();
StringBuffer xmlStr = new StringBuffer( "<?xml version='1.0'?> ..." );
Object o = u.unmarshal( new StreamSource( new StringReader( xmlStr.toString() ) ) );
```

Unmarshalling from a `org.w3c.dom.Node`:

```
JAXBContext jc = JAXBContext.newInstance( "com.acme.foo" );
Unmarshaller u = jc.createUnmarshaller();
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(new File( "xmlFile.xml"));
Object o = u.unmarshal( doc );
```

By default, `Unmarshaller` is very forgiving. Even if a document is invalid, it tries to recover from errors. If the document is so broken that it cannot be read, an `javax.xml.bind.UnmarshalException` (child of `javax.xml.bind.JAXBException`) will be thrown. It's often desirable to get more information about errors or reject documents with errors. The first step to do this is to set `ValidationEventHandler` to the `Unmarshaller`. A `ValidationEventHandler` can explicitly tell a JAXB implementation whether it should reject a document or try to recover from errors. It also gives you more information, such as line numbers, about errors. An `Unmarshaller` can validate a document with the schema while unmarshalling. With this option turned on, it rejects anything short of a valid document. However, W3C XML Schema validation can be very costly. Another possibility is to set up a SAX pipeline in such a way that your XML parser does the validation; alternately, you could install a stand-alone validator in the pipeline. In this way, for example, you can change your schema to change what you get from the compiler, while maintaining the scrutiny of the original schema.

Marshalling (Java-to-XML)

A `Marshaller` is used to write an object graph into XML. To write an object o to a file, you would do:

```
...
JAXBContext jaxbContext = JAXBContext.newInstance(packageName);
Marshaller marshaller = jaxbContext.createMarshaller();
ObjectFactory itemMaker = new ObjectFactory();
Item item = itemMaker.createItem();
marshaller.marshal(item, new FileOutputStream("newIem.xml"));
...
```

There are other overloaded versions which allow you to produce XML as a a DOM tree or as SAX events. For example, by using `StringWriter`, you can marshal an object into a string. You can also marshal an object graph to a `javax.xml.transform.Result` object.

Here are some more examples of marshalling:

Marshalling to a file (`FileOutputStream`):

```
OutputStream os = new FileOutputStream( "xmlFile.xml" );
m.marshal( obj, os );
```

Marshalling to a SAX `ContentHandler`:

```
// assume MyContentHandler instanceof ContentHandler
m.marshal( obj, new MyContentHandler() );
```

Marshalling to a DOM `Node`:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.newDocument();
m.marshal( obj, doc );
```

Marshalling to a `java.io.OutputStream`:

```
m.marshal( obj, System.out );
```

Marshalling to a `java.io.Writer`:

```
m.marshal( obj, new PrintWriter( System.out ) );
```

Marshalling to a `javax.xml.transform.SAXResult`:

```
// assume MyContentHandler instanceof ContentHandler
SAXResult result = new SAXResult( new MyContentHandler() );
m.marshal( obj, result );
```

Marshalling to a `javax.xml.transform.DOMResult`:

```
DOMResult result = new DOMResult();
m.marshal( obj, result );
```

Although each of the marshal methods accepts a `java.lang.Object` as its first parameter, JAXB Providers are not required to be able to marshal any arbitrary `java.lang.Object`. If the `JAXBContext` object that was used to create this `Marshaller` does not have enough information to know how to marshal the object parameter (or any objects reachable from it), then the `marshal` operation will throw a `MarshalException`.

By default, the `Marshaller` will use UTF-8 encoding when generating XML data to a `java.io.OutputStream`, or a `java.io.Writer`. Use the `setProperty` API to change the ouput encoding used during these `marshal` operations.

You can control the behavior of marshalling by setting `Marshaller` properties. For example, you can toggle indentation of the XML:

```
...
Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, new Boolean(true));
...
```

JAXB can produce XML as SAX events. That is, you can pass `ContentHandler` and have it receive SAX events from a JAXB object. This gives client apps plenty of chances to modify XML. For example, you can add and remove elements or attributes, use one of the freely available serializers for better output, or write your own XML serializer that prints XML in your preferred way.

Finally, you can ask a `Marshaller` to marshal an invalid object graph by setting a `ValidationEventHandler`. If a provider supports error recovery, you can tell it to write XML even if it's incomplete.

If the `boolean ValidationEventHandler.handleEvent(ValidationEvent event)` method throws an unchecked runtime exception, the JAXB Provider must treat that as if the method returned `false`, effectively terminating whatever operation was in progress at the time (unmarshal, validate, or marshal). The method returns: `true` if the JAXB Provider should attempt to continue the current unmarshal, validate, or marshal operation after handling this warning/error, `false` if the provider should terminate the current operation with the appropriate `UnmarshalException`, `ValidationException`, or `MarshalException`.

Validation

There are three forms of Validation in JAXB:

1. Unmarshal-Time Validation
   This form of validation enables a client application to receive information about validation errors and warnings detected while unmarshalling XML data into a Java content tree and is completely orthogonal to the other types of validation. To enable or disable it use method `Unmarhaller.setValidating(...)`. All JAXB Providers are REQUIRED to support this operation.
2. On-Demand Validation
   This form of 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). All JAXB Providers are REQUIRED to support this operation.
3. Fail-Fast Validation
   This form of validation enables a client application to receive immediate feedback about modifications to the Java content tree that violate type constraints on Java Properties as defined in the specification. JAXB Providers are NOT REQUIRED support this type of validation. Of the JAXB Providers that do support this type of validation, some may require you to decide at schema compile time whether or not a client application will be allowed to request fail-fast validation at runtime.

The `Validator` class is responsible for managing On-Demand Validation. The `Unmarshaller` class is responsible for managing Unmarshal-Time Validation during the unmarshal operations. Although there is no formal method of enabling validation during the marshal operations, the `Marshaller` may detect errors, which will be reported to the `ValidationEventHandler` registered on it.

JAXB has the capability to validate an object graph in memory without actually writing it to XML. This allows client apps to check if a graph is okay and ready to process; if not, validation will identify objects that contain errors so that, for example, client apps can ask users to fix those.
The following code validates the object "`item`":

```
...
JAXBContext jaxbContext = JAXBContext.newInstance(packageName);
ObjectFactory itemMaker = new ObjectFactory();
Item item = itemMaker.createItem();
Validator validator = jaxbContext.createValidator();
if(! validator.validate(item)) {
        System.err.println("Not valid !!!");
}
...
```

To receive detailed information about errors, you need to register `ValidationEventHandler` with the `Validator`, just like you did in `Unmarshaller` and `Marshaller`. This is analogous to registering an `ErrorHandler` for a SAX parser.
You can also first marshal an object graph and then validate XML (for example by Java API for validators). But doing so makes it much harder to associate errors with their sources, which makes debugging harder for humans. Validation after marshalling will give you errors like "missing `<foo>` element," but you can hardly know what is actually wrong in the object graph.
Validity is not enforced while you are modifying an object graph; you ALWAYS have to explicitly validate it. To edit a valid object graph into another valid object graph, you may need to go through invalid intermediate states. If validity is enforced on every step of mutation, this becomes impossible.

Customization Through the Schema

The binding language is an XML based language which defines constructs referred to as binding declarations. A binding declaration can be used to customize the default binding between an XML schema component and its Java representation.
The schema for binding declarations is defined in the namespace `http://java.sun.com/xml/ns/jaxb`. This specification uses the namespace prefix "`jaxb`" to refer to the namespace of binding declarations. For example:

```
<jaxb: binding declaration >
```

A binding compiler interprets the binding declaration relative to the source schema and a set of default bindings for that schema. Therefore a source schema need not contain a binding declarations for every schema component. This makes the job of a JAXB application developer easier.
There are two ways to associate a binding declaration with a schema element:

- as part of the source schema (inline annotated schema).
- external to the source schema in an external binding declaration.

The syntax and semantics of the binding declaration is the same regardless of which of the above two methods is used for customization.

Inline Annotated Schema

This method of customization utilizes on the `appinfo` element specified by the XML Schema. A binding declaration is embedded within the `appinfo` element as illustrated below:

```
<xs:annotation>
  <xs:appinfo>
    <binding declaration>
  </xs:appinfo>
</xs:annotation>
```

The inline annotation where the binding declaration is used identifies the schema component.
Here are the changes you must make to the schema to make JAXB generate `java.util.Vector` rather than `java.util.ArrayList`, its default collection (the `collecionType` value must be either "indexed" or any fully qualified class name that implements `java.util.List`). Note that the top-level schema tag needs to be changed too:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
    jxb:version="1.0">
  ...
  <xs:annotation>
    <xs:appinfo>
      <jxb:globalBindings collectionType="java.util.Vector" />
    </xs:appinfo>
  </xs:annotation>
  ...
```

The `annotation` tag introduces a part of the schema that is usually intended for schema processing software. The `appinfo` tag introduces instructions for a particular processing application (in this case, JAXB's `xjc` code-generation tool). Usually, each application uses its own namespace, as JAXB has done here.

Before customization:

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

  <xs:element name="todolist">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="item" type="entry"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  ...
</xs:schema>
public class TodolistTypeImpl implements ... {

    protected com.sun.xml.bind.util.ListImpl _Item = new
com.sun.xml.bind.util.ListImpl(new java.util.ArrayList());
    ...
}
```

After customization:

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
    jxb:version="1.0">

  <xs:annotation>
    <xs:appinfo>
      <jxb:globalBindings collectionType="java.util.Vector" />
    </xs:appinfo>
  </xs:annotation>

  <xs:element name="todolist">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="item" type="entry"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  ...
</xs:schema>

public class TodolistTypeImpl implements ... {

    protected com.sun.xml.bind.util.ListImpl _Item = new
com.sun.xml.bind.util.ListImpl(new java.util.Vector());
    ...
}
```

External Binding Declaration

The external binding declaration format enables customized binding without requiring modification of the source schema. Unlike inline annotation, the remote schema component to which the binding declaration applies must be identified explicitly. The `jaxb:bindings` element enables the specification of a remote schema context to associate its binding declaration(s) with. Minimally, an external binding declaration follows the following format:

```
<jaxb:bindings schemaLocation = "xs:anyURI">
  <jaxb:bindings node = "xs:string">*
    <binding declaration>
  <jaxb:bindings>
</jaxb:bindings>
```

The attributes `schemaLocation` and `node` are used to construct a reference to a node in a remote schema. The binding declaration is applied to this node by the binding compiler as if the binding declaration was embedded in the node's `xs:appinfo` element. The attribute values are interpreted as follows:

- `schemaLocation` - It is a URI reference to a remote schema.
- `node` - It is an XPath 1.0 expression that identifies the schema node within `schemaLocation` to associate binding declarations with.

Use the SAAJ APIs to create and manipulate a SOAP message.

SAAJ - SOAP with Attachments API for Java - contains APIs for creating and populating SOAP messages which might or might not contain attachments. It also contains APIs for sending point to point, non-provider-based, request and response SOAP messages.

SOAP message is made of SOAP Envelope and zero or more attachments. The SOAP Envelope is then made of SOAP Header and SOAP Body. SOAP attachment allows the SOAP message to contain not only the XML data but also non-XML data such as JPG file. And it uses the MIME multipart as container for these non-XML data.

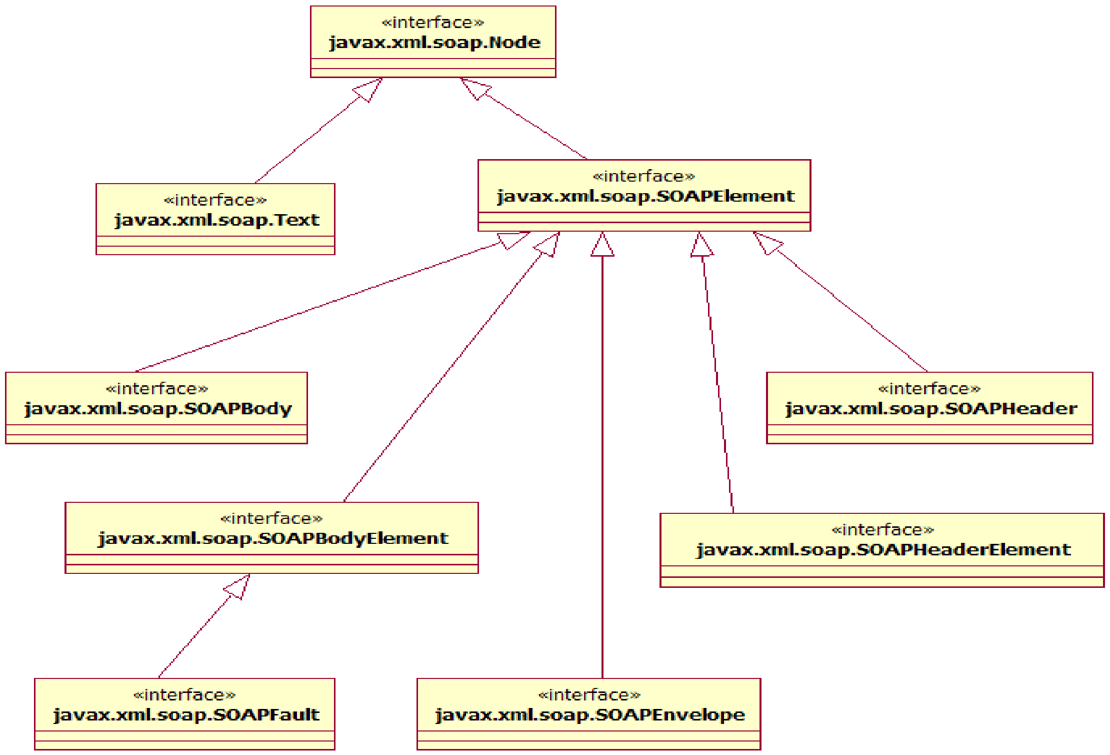
The SAAJ API provides the `SOAPMessage` class to represent a SOAP message, the `SOAPPart` class to represent the SOAP part, the `SOAPEnvelope` interface to represent the SOAP envelope, and so on.

When you create a new `SOAPMessage` object, it will automatically have the parts that are required to be in a SOAP message. In other words, a new `SOAPMessage` object has a `SOAPPart` object that contains a `SOAPEnvelope` object. The `SOAPEnvelope` object in turn automatically contains an empty `SOAPHeader` object followed by an empty `SOAPBody` object. If you do not need the `SOAPHeader` object, which is optional, you can delete it. The rationale for having it automatically included is that more often than not you will need it, so it is more convenient to have it provided. The `SOAPHeader` object may contain one or more headers with information about the sending and receiving parties. The `SOAPBody` object, which always follows the `SOAPHeader` object if there is one, provides a simple way to send information intended for the ultimate recipient. For example, if there is a `SOAPFault` object, it must be in the `SOAPBody` object.

A SOAP message may include one or more attachment parts in addition to the SOAP part. The SOAP part may contain only XML content; as a result, if any of the content of a message is not in XML format, it must occur in an attachment part. So, if for example, you want your message to contain a binary file, your message must have an attachment part for it. Note that an attachment part can contain any kind of content, so it can contain data in XML format as well.

The SAAJ API provides the `AttachmentPart` class to represent the attachment part of a SOAP message. A `SOAPMessage` object automatically has a `SOAPPart` object and its required subelements, but because `AttachmentPart` objects are optional, you have to create and add them yourself.

SAAJ API belongs to `javax.xml.soap.*` package. `SOAPConnection` provides request/response SOAP message exchange. `SOAPMessage` creates and populates SOAP message (consists of `SOAPPart` and `AttachmentPart`).



`javax.xml.soap.MessageFactory` is a factory for creating SOAP 1.1-based messages.

```
public abstract class MessageFactory {
    public static MessageFactory newInstance() throws SOAPException { ... }
    public abstract SOAPMessage createMessage() throws SOAPException;
    ...
}
```

`javax.xml.soap.SOAPMessage` is a Java technology abstraction for a SOAP 1.1 message. Contains EXACTLY ONE `SOAPPart` and ZERO OR MORE `AttachmentParts`.

```
public abstract class SOAPMessage {
    public abstract SOAPPart getSOAPPart();
    public abstract Iterator getAttachments();
    public abstract Iterator getAttachments(MimeHeaders headers);
    ...
}
```

`javax.xml.soap.SOAPPart` is the first part of a multi-part message when there are attachments.

```
public abstract class SOAPPart implements org.w3c.dom.Document {
    public abstract SOAPEnvelope getEnvelope() throws SOAPException;
    ...
}
```

`javax.xml.soap.AttachmentPart` can contain any data (for example: JPEG images, XML business documents, etc.) If a SOAPMessage object has one or more attachments, each `AttachmentPart` object must have a MIME header to indicate the type of data it contains. It may also have additional MIME headers to identify it or to give its location, which are optional but can be useful when there are multiple attachments. When a SOAPMessage object has one or more `AttachmentPart` objects, its SOAP`Part` object may or may not contain message content.

Steps of SAAJ Programming

1. Creating a message (SOAPMessage)
   Use `MessageFactory` as a factory of messages. SOAPMessage object has the following:
   - SOAP`Part` object
     - o SOAPEnve`lope` object
       - empty SOAP`Header` object
       - empty SOAPBody object

Example:

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

Result:

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

2. Accessing elements of a message
   Approach 1: from SOAPEnvelope object:

```
SOAPPart soapPart = message.getSOAPPart();
SOAPEnvelope envelope = soapPart.getEnvelope();
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();
```

   Approach 2: from SOAPMessage object:

```
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
```

3. Adding contents to the body

   Example:

```
SOAPBody body = message.getSOAPBody();
SOAPFactory soapFactory = SOAPFactory.newInstance();
Name bodyName = soapFactory.createName("GetLastTradePrice",
         "m", "http://wombat.ztrade.com");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

   Will produce following XML:

```
<m:GetLastTradePrice xmlns:m="http://wombat.ztrade.com">
  ...
</m:GetLastTradePrice>
```

   Example:

```
Name name = soapFactory.createName("symbol");
SOAPElement symbol = bodyElement.addChildElement(name);
symbol.addTextNode("SUNW");
```

   Generates XML fragment like following:

```
<symbol>SUNW</symbol>
```

4. Getting a SOAPConnection object

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

   All SOAP messages are sent and received over a connection. With the SAAJ API, the connection is represented by a SOAPConnection object, which goes from the sender directly to its destination. This kind of connection is called a point-to-point connection because it goes from one endpoint to another endpoint. Messages sent using the SAAJ API are called request-response messages. They are sent over a SOAPConnection object with the method call, which sends a message (a request) and then blocks until it receives the reply (a response).

5. Sending a message

```
// Create an endpint point which is either URL or String type
java.net.URL endpoint = new URL("http://wombat.ztrade.com/quotes");

// Send a SOAPMessage (request) and then wait for SOAPMessage (response)
SOAPMessage response = connection.call(message, endpoint);
```

A SAAJ client calls the SOAPConnection method call on a SOAPConnection object to send a message. The call method takes two arguments, the message being sent and the destination to which the message should go. This message is going to the stock quote service indicated by the URL object endpoint.

6. Getting the content of a message

```
SOAPBody soapBody = response.getSOAPBody();
java.util.Iterator iterator = soapBody.getChildElements(bodyName);
SOAPBodyElement bodyElement = (SOAPBodyElement)iterator.next();
String lastPrice = bodyElement.getValue();

System.out.print("The last price for SUNW is ");
System.out.println(lastPrice);
```

In order to access bodyElement, you need to call the method getChildElements on soapBody. Passing bodyName to getChildElements returns a java.util.Iterator object that contains all of the child elements identified by the Name object bodyName. You already know that there is only one, so just calling the method next on it will return the SOAPBodyElement you want. Note that the method Iterator.next() returns a Java Object, so it is necessary to cast the Object it returns to a SOAPBodyElement object before assigning it to the variable bodyElement.

Adding content to SOAPHeader

To add content to the header, you need to create a SOAPHeaderElement object. As with all new elements, it must have an associated Name object, which you can create using the message's SOAPEnvelope object or a SOAPFactory object. For example, suppose you want to add a conformance claim header to the message to state that your message conforms to the WS-I Basic Profile. The following code fragment retrieves the SOAPHeader object from message and adds a new SOAPHeaderElement object to it. This SOAPHeaderElement object contains the correct qualified name and attribute for a WS-I conformance claim header:

```
SOAPHeader header = message.getSOAPHeader();
Name headerName = soapFactory.createName("Claim", "wsi", "http://ws-
i.org/schemas/conformanceClaim/");
SOAPHeaderElement headerElement = header.addHeaderElement(headerName);
headerElement.addAttribute(soapFactory.createName("conformsTo"), "http://ws-
i.org/profiles/basic1.0/");
```

At this point, header contains the SOAPHeaderElement object headerElement identified by the Name object headerName. Note that the addHeaderElement method both creates headerElement and adds it to header.
XML fragment generated:

```
<SOAP-ENV:Header>
    <wsi:Claim conformsTo="http://ws-i.org/profiles/basic1.0/"
            xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"/>
</SOAP-ENV:Header>
```

A conformance claim header has no content.
For a different kind of header, you might want to add content to headerElement. The following line of code uses the method addTextNode to do this:

```
headerElement.addTextNode("order");
```

Now you have the SOAPHeader object header that contains a SOAPHeaderElement object whose content is "order".

Adding content to SOAPBody

The process for adding content to the SOAPBody object is the same as the process for adding content to the SOAPHeader object. You access the SOAPBody object, add a SOAPBodyElement object to it, and add text to the SOAPBodyElement object. It is possible to add additional SOAPBodyElement objects, and it is possible to add subelements to the SOAPBodyElement objects with the method addChildElement. For each element or child element, you add content with the method addTextNode. The following example shows adding multiple SOAPElement objects and adding text to each of them:

```
SOAPBody body = soapFactory.getSOAPBody();
Name bodyName = soapFactory.createName("PurchaseLineItems", "PO",
"http://sonata.fruitsgalore.com");
SOAPBodyElement purchaseLineItems = body.addBodyElement(bodyName);
Name childName = soapFactory.createName("Order");
SOAPElement order = purchaseLineItems.addChildElement(childName);
```

```
childName = soapFactory.createName("Product");
SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");
childName = soapFactory.createName("Price");
SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");
```

The code first creates the `SOAPBodyElement` object `purchaseLineItems`, which has a fully qualified name associated with it. That is, the `Name` object for it has a local name, a namespace prefix, and a namespace URI. As you saw earlier, a `SOAPBodyElement` object is required to have a fully qualified name, but child elements added to it, such as `SOAPElement` objects, may have `Name` objects with only the local name.
The SAAJ code in the preceding example produces the following XML in the SOAP body:

```
<PO:PurchaseLineItems xmlns:PO="http://www.sonata.fruitsgalore/order">
        <Order>
                <Product>Apple</Product>
                <Price>1.56</Price>
        </Order>
</PO:PurchaseLineItems>
```

Adding and accessing attachments
Create from `AttachmentPart` object:

```
AttachmentPart attachment = message.createAttachmentPart();
```

`AttachmentPart` is made of two parts: application-specific content and associated MIME headers (Content-Type):

```
attachment.setMimeHeader("Content-Type", "application/xml");
```

Adding contents to attachment (Option 1: Setting 'Content' and 'ContentId'):

```
String stringContent = "Update address for Sunny Skies "
stringContent += "Inc., to 10 Upbeat Street, Pleasant Grove, CA 95439";
attachment.setContent(stringContent, "text/plain");
attachment.setContentId("update_address");
message.addAttachmentPart(attachment);
```

The code fragment above shows one of the ways to use the method `setContent`. This method takes two parameters, the first being a Java Object containing the content and the second being a `String` giving the content type. The Java Object may be a `String`, a stream, a `javax.xml.transform.Source` object, or a `javax.activation.DataHandler` object. The Java Object being added in the following code fragment is a `String`, which is plain text, so the second argument must be "`text/plain`". The code also sets a content identifier, which can be used to identify this `AttachmentPart` object. After you have added content to attachment, you need to add it to the `SOAPMessage` object, which is done in the last line.
As with `AttachmentPart.setContent(...)`, the Object may be a `String`, a stream, a `javax.xml.transform.Source` object, or a `javax.activation.DataHandler` object.
Adding Contents to Attachment (Option 2: Using `DataHandler`):

```
// Create DataHandler object
URL url = new URL ("http://greatproducts.com/gizmos/img.jpg");
DataHandler dataHandler = new DataHandler(url);
AttachmentPart attachment = message.createAttachmentPart(dataHandler);
// Note that there is no need to set Content Type
attachment.setContentId("attached_image");
message.addAttachmentPart(attachment);
```

The other method for creating an `AttachmentPart` object with content takes a `DataHandler` object, which is part of the JavaBeans Activation Framework (JAF). Using a `DataHandler` object is fairly straightforward. First you create a `java.net.`URL object for the file you want to add as content. Then you create a `DataHandler` object initialized with the URL object: You might note two things about the previous code fragment. First, it sets a header for `Content-ID` with the method `setContentId(...)`. This method takes a `String` that can be whatever you like to identify the attachment. Second, unlike the other methods for setting content, this one does not take a `String` for `Content-Type`. This method takes care of setting the `Content-Type` header for you, which is possible because one of the things a `DataHandler` object does is determine the data type of the file it contains.

Accessing attachments:

```
java.util.Iterator iterator = message.getAttachments();
while (iterator.hasNext()) {
        AttachmentPart attachment = (AttachmentPart)iterator.next();
        String id = attachment.getContentId();
        String type = attachment.getContentType();
        System.out.print("Attachment " + id + " has content type " + type);
        if (type == "text/plain") {
                Object content = attachment.getContent();
                System.out.println("Attachment " + "contains:\n" + content);
        }
}
```

Adding attributes to SOAPHeaderElement

SOAP Header attributes: `actor` and `mustUnderstand`

The attribute `actor` is optional, but if it is used, it must appear in a `SOAPHeaderElement` object. Its purpose is to indicate the recipient of a header element. The default actor is the message's ultimate recipient; that is, if no `actor` attribute is supplied, the message goes directly to the ultimate recipient.

An actor is an application that can both receive SOAP messages and forward them to the next actor. The ability to specify one or more actors as intermediate recipients makes it possible to route a message to multiple recipients and to supply header information that applies specifically to each of the recipients.

For example, suppose that a message is an incoming purchase order. Its `SOAPHeader` object might have `SOAPHeaderElement` objects with `actor` attributes that route the message to applications that function as the order desk, the shipping desk, the confirmation desk, and the billing department. Each of these applications will take the appropriate action, remove the `SOAPHeaderElement` objects relevant to it, and send the message on to the next actor.

An actor is identified by its URI. For example, the following line of code, in which `orderHeader` is a `SOAPHeaderElement` object, sets the actor to the given URI:

```
orderHeader.setActor("http://gizmos.com/orders");
```

Additional actors may be set in their own `SOAPHeaderElement` objects. The following code fragment first uses the `SOAPMessage` object message to get its `SOAPHeader` object header. Then header creates two `SOAPHeaderElement` objects, each of which sets its `actor` attribute and `mustUnderstand` attribute:

```
SOAPHeader header = message.getSOAPHeader();

SOAPFactory soapFactory = SOAPFactory.newInstance();

String nameSpace = "ns";
String nameSpaceURI = "http://gizmos.com/NSURI";

Name order = soapFactory.createName("orderDesk", nameSpace, nameSpaceURI);
SOAPHeaderElement orderHeader = header.addHeaderElement(order);
orderHeader.setActor("http://gizmos.com/orders");
orderHeader.setMustUnderstand(true);

Name shipping = soapFactory.createName("shippingDesk", nameSpace, nameSpaceURI);
SOAPHeaderElement shippingHeader = header.addHeaderElement(shipping);
shippingHeader.setActor("http://gizmos.com/shipping");
shippingHeader.setMustUnderstand(true);
```

Retrieving all SOAPHeaderElements with a particular Actor

The `SOAPHeader` interface provides two methods that return a `java.util.Iterator` object over all of the `SOAPHeaderElement` objects with an actor that matches the specified `actor`. The first method, `examineHeaderElements`, returns an iterator over all of the elements with the specified Actor:

```
// Note that an Actor is identified by an URL
Iterator headerElements = header.examineHeaderElements("http://gizmos.com/orders");
```

The second method, `extractHeaderElements`, not only returns an iterator over all of the `SOAPHeaderElement` objects with the specified `actor` attribute but also detaches them from the `SOAPHeader` object. So, for example, after the order desk application has done its work, it would call `extractHeaderElements` to remove all of the `SOAPHeaderElement` objects that applied to it:

```
// All headers with defined Actor are detached from the SOAPHeader object
Iterator headerElements = header.extractHeaderElements("http://gizmos.com/orders");
```

Creating SOAPHeaderElement with mustUnderstand attribute

The Java code:

```
SOAPHeader header = message.getSOAPHeader();
Name name = soapFactory.createName("Transaction", "t", "http://gizmos.com/orders");
SOAPHeaderElement transaction = header.addHeaderElement(name);
transaction.setMustUnderstand(true);
transaction.addTextNode("5");
```

The XML fragment:

```
<SOAP-ENV:Header>
        <t:Transaction xmlns:t="http://gizmos.com/orders"
                        SOAP-ENV:mustUnderstand="1">
                5
        </t:Transaction>
</SOAP-ENV:Header>
```

SOAPFault object

Represents SOAP Fault element in SOAP body. Parties that can generate SOAPFault (for example):

- Recipient of a message in point-to-point messaging (Indicates missing address information in purchase order SOAP message)
- Messaging provider in messaging provider-based messaging (Messaging provider cannot deliver the message due to server failure)

SOAPFault object contains:

1. Fault Code (mandatory)
   Required in SOAPFault object (VersionMismatch, MustUnderstand, Client, Server).
2. Fault String (mandatory)
   Human readable explanation of the fault
3. Fault Actor (conditional)
   Required if SOAPHeader object has one or more actor attributes and fault was caused by header processing.
4. Detail object (conditional)
   Required if the fault is error related to the SOAPBody object. If not present in Client fault, SOAPBody is assumed to be OK.

SOAPFault with no Detail object:

```
// Create SOAPBody object
SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
SOAPBody body = envelope.getBody();

// Create and fill up SOAPFault object.
// Note that Detail object is not being set here since the fault has
// nothing to do with SOAPBody
SOAPFault fault = body.addFault();
fault.setFaultCode("Server");
fault.setFaultActor("http://gizmos.com/orders");
fault.setFaultString("Server not responding");
```

SOAPFault with Detail object:

```
// Create SOAPFault object
SOAPFault fault = body.addFault();

// Set fault code and fault string
fault.setFaultCode("Client");
fault.setFaultString("Message does not have necessary info");

// Detail object contains two DetailEntry's
Detail detail = fault.addDetail();
Name entryName = envelope.createName("order", "PO", "http://gizmos.com/orders/");
DetailEntry entry = detail.addDetailEntry(entryName);
```

```
entry.addTextNode("quantity element does not have a value");
Name entryName2 = envelope.createName("confirmation", "PO",
"http://gizmos.com/confirm");
DetailEntry entry2 = detail.addDetailEntry(entryName2);
entry2.addTextNode("Incomplete address: no zip code");
```

Retrieving `SOAPFault`:

```
// Get SOAPBody object
SOAPBody body = newmsg.getSOAPPart().getEnvelope().getBody();

// Check if SOAPFault is present in the message
if ( body.hasFault() ) {
        SOAPFault newFault = body.getFault();
        String code = newFault.getFaultCode();
        String string = newFault.getFaultString();
        String actor = newFault.getFaultActor();
        if ( actor != null ) { System.out.println(" fault actor = " + actor); }
}
```

Retrieving Detail Object:

```
// Get Detail object
Detail newDetail = newFault.getDetail();

// Get the list of DetailEntry's
if ( newDetail != null) {
        Iterator it = newDetail.getDetailEntries();
        while ( it.hasNext() ) {
                DetailEntry entry = (DetailEntry)it.next();
                String value = entry.getValue();
                System.out.println(" Detail entry = " + value);
        }
}
```

Chapter 6. JAXR

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.

JAXR, the Java API for XML Registries, provides a standard API for publication and discovery of Web services through underlying registries.

JAXR does not define a new registry standard. Instead, this standard Java API performs registry operations over a diverse set of registries and defines a unified information model for describing registry contents. Regardless of the registry provider accessed, your programs use common APIs and a common information model. The JAXR specification defines a general-purpose API, allowing any JAXR client to access and interoperate with any business registry accessible via a JAXR provider.

Capability profiles

Because some diversity exists among registry provider capabilities, the JAXR expert group decided to provide multilayer API abstractions through capability profiles. Each method of a JAXR interface is assigned a capability level, and those JAXR methods with the same capability level define the JAXR provider capability profile.

Currently, JAXR defines only two capability profiles: level 0 profile for basic features and level 1 profile for advanced features. Level 0's basic features support so-called business-focused APIs, while level 1's advanced features support generic APIs. At the minimum, all JAXR providers must implement a level 0 profile. A JAXR client application using only those methods of the level 0 profile can access any JAXR provider in a portable manner. JAXR providers for UDDI must be level 0 compliant.

JAXR providers can optionally support the level 1 profile. The methods assigned to this profile provide more advanced registry capabilities needed by more demanding JAXR clients. Support for the level 1 profile also implies full support for the level 0 profile. JAXR providers for ebXML must be level 1 compliant. A JAXR client can discover the capability level of a JAXR provider by invoking methods on the `CapabilityProfile` interface. If the client

attempts to invoke capability level methods unsupported by the JAXR provider, the provider will throw an UnsupportedCapabilityException.
NOTE, because WS-I BP sanctions the use of UDDI, not ebXML, we MUST use **JAXR level 0 profile**.

RegistryService interfaces

The JAXR provider supports capability profiles that group the methods on JAXR interfaces by capability level. RegistryService exposes the JAXR provider's key interfaces, that is, Web services discovery and registration. The JAXR client can obtain an instance of the RegistryService interface by invoking the getRegistryService() method on the connection established between the JAXR client and JAXR provider. Once the JAXR client has the RegistryService, it can obtain the primary registry interfaces and perform life-cycle management and query management through the JAXR provider.
The JAXR specification defines two life-cycle management interfaces:

1. BusinessLifeCycleManager - for level 0 (we MUST use this interface according to BP 1.0)
2. LifeCycleManager - for level 1 (we MUST NOT use this interface according to BP 1.0)

BusinessLifeCycleManager defines a simple business-level API for life-cycle management. This interface resembles the publisher's API in UDDI, which should prove familiar to the UDDI developer. For its part, LifeCycleManager interface provides complete support for all life-cycle management needs using a generic API. Life-cycle management includes creating, saving, updating, deprecating, and deleting registry objects. In addition, the LifeCycleManager provides several factory methods to create JAXR information model objects. In general, life-cycle management operations are privileged, while a user can use query management operations for browsing the registry.
JAXR's top-level interface for query management, QueryManager, has two extensions:

1. BusinessQueryManager - for level 0 (we MUST use this interface according to BP 1.0)
2. DeclarativeQueryManager - for level 1 (we MUST NOT use this interface according to BP 1.0)

Query management deals with querying the registry for registry data. A simple business-level API, the BusinessQueryManager interface provides the ability to query for the most important high-level interfaces in the information model, such as Organization, Service, ServiceBinding, ClassificationScheme, and Concept. Alternatively, the DeclarativeQueryManager interface provides a more flexible, generic API, enabling the JAXR client to perform ad hoc queries using a declarative query language syntax. Currently, the only declarative syntaxes JAXR supports are SQL-92 and OASIS/ebXML Registry Filter Queries. As noted in the JAXR specification, ebXML registry providers optionally support SQL queries. If a registry provider does support SQL queries, the JAXR ebXML provider will throw an UnsupportedCapabilityException on DeclarativeQueryManager methods.

JAXR information model

Invoking life-cycle and query management methods on the JAXR provider requires the JAXR client to create and use the JAXR information model objects. The JAXR information model resembles the one defined in the ebXML Registry Information Model 2.0, but also accommodates the data types defined in the UDDI Data Structure Specification. Although developers familiar with the UDDI information model might face a slight learning curve, once understood, the JAXR information model will provide a more intuitive and natural interface to most developers.
Most JAXR information-model interfaces are derived from the abstract RegistryObject interface, which defines the common state information, called attributes, that all registry objects share. Example attributes include: key, name, and description. The InternationalString interface defines attributes that must be internationalization compatible, such as name and description. The InternationalString interface contains a collection of LocalizedStrings, where each LocalizedString defines locale, character set, and string content.
The RegistryObject interface also defines collections of Classification, ExternalIdentifier, ExternalLink, and Association objects. The BusinessQueryManager often uses those collections as parameters in its find methods.
Also specializations of the RegistryObject interface, the concrete interfaces Organization, Service, ServiceBinding, Concept, and ClassificationScheme provide additional state information. For example, the Organization interface defines a collection of Service objects, and Service defines a collection of ServiceBinding objects. A ServiceBinding might contain a collection of SpecificationLink objects. UDDI developers should be familiar with these concrete interfaces; they map quite well to major UDDI data types shown in the table below:

Table 6.1. UDDI-to-JAXR information model mappings

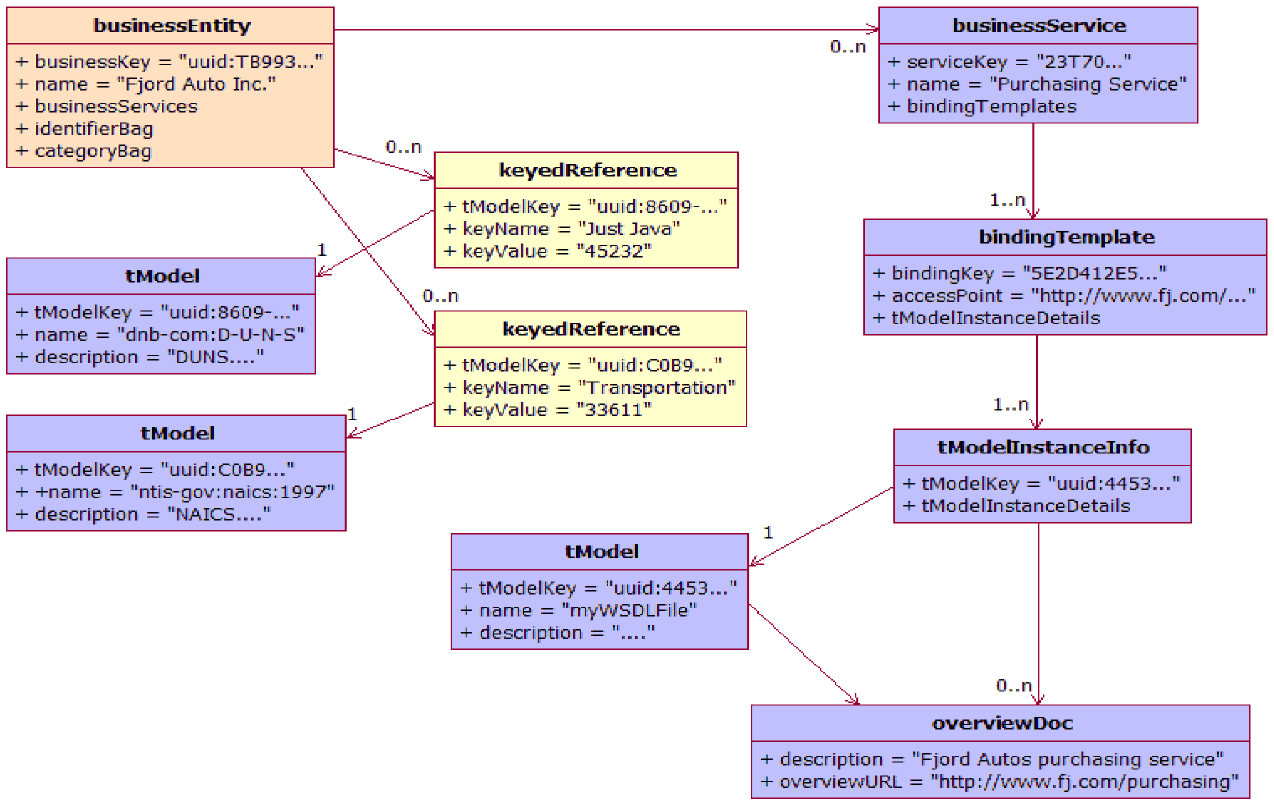
JAXR Information Model	UDDI Data Model
Organization A business, such as a corporation whose data is contained in the registry, is represented by an Organization	businessEntity The businessEntity structure contains information about a particular business organization

JAXR Information Model	UDDI Data Model
instance. An Organization can have multiple Organizations under it or can refer to another organization as a parent. An Organization can offer one or more services, represented by Service objects which can be accessed via Collection Organization.getServices() method.	and holds references to the services it offers. businessEntity is the highest in the hierarchy and contains descriptive information. Each businessEntity is identified by its businessKey. If its businessKey is not specified at publication time, the registry will automatically generate a key.
Service Each Service can have multiple ServiceBinding objects that specify the details about the protocol binding information for that service.	businessService businessService entry indicates a logical service and holds descriptive information about a Web service in business terms. A businessService is a child of a businessEntity that provides the service. Information about how a businessService can be instantiated is contained within a bindingTemplate. Each businessService is has a unique identifier - serviceKey. This value is assigned by each UDDI operator and cannot be edited by the publisher. It also contains the key to its parent businessEntity - businessKey.
ServiceBinding ServiceBinding has attributes that describe the location where the service can be accessed. A ServiceBinding is associated with one or more SpecificationLink objects that point to the technical specifications defining the service (e.g., a WSDL file for the Web service). ServiceBinding instances are RegistryObject objects that represent technical information on a specific way to access a specific interface offered by a Service instance. A ServiceBinding may have a set of SpecificationLink instances. Maps to a bindingTemplate in UDDI.	bindingTemplate A bindingTemplate contains information necessary for a client to invoke a service. It specifies the information for a particular Web service and provides the URL of the service where it can be invoked. The bindingTemplate also contains references to tModels as well as service-specific settings. A bindingTemplate is a child of a businessService.
ClassificationScheme A ClassificationScheme instance represents a taxonomy that may be used to classify or categorize RegistryObject instances. The classification scheme, or taxonomy, can be internal or external to the registry, meaning that the structure of the taxonomy is defined internal to the registry or is represented somewhere outside the registry and is represented by a ClassificationScheme interface. A good example of an external taxonomy for e-business is that of North American Industry Classification System (NAICS) devised by the U.S. Census Bureau. NAICS is a classification scheme used to classify businesses and services by the industry to which they belong and the business processes they follow.	tModel
Concept The Concept interface is used to represent taxonomy elements and their structural relationship with each other in order to describe an internal taxonomy. Concept instances are used to define tree structures where the root of the tree is a ClassificationScheme instance and each node in the tree is a Concept instance. As any RegistryObject, a Concept may be classified and also associated with a set of external identifiers and links.	

JAXR Information Model	UDDI Data Model
Association A RegistryObject instance may be associated with zero or more RegistryObject instances. The information model defines an Association interface, an instance of which may be used to associate any two RegistryObject instances. An Association instance represents an association between a source RegistryObject and a target RegistryObject. These are referred to as sourceObject and targetObject for the Association instance. It is important which object is the sourceObject and which is the targetObject as it determines the directional semantics of an Association. Each Association must have an associationType attribute that identifies the type of that association. The associationType attribute is a reference to an enumeration Concept as defined by the predefined associationType ClassificationScheme in the JAXR specification. An association may need to be confirmed by the parties whose objects are involved in that Association.	publisherAssertion
ExternalLink ExternalLink instances model a named URI to content that may reside outside the registry. RegistryObject may be associated with any number of ExternalLink instances to annotate a RegistryObject with external links to external content. Consider the case where a Submitting Organization submits a repository item (e.g. a DTD) and wants to associate some external content to that object (e.g. the Submitting Organization's home page). The ExternalLink enables this capability.	overviewDoc
Classification The Classification interface is used to classify RegistryObject instances. A RegistryObject may be classified along multiple dimensions by adding zero or more Classification instances to the RegistryObject. For example, an Organization may be classified by its industry, by the products it sells and by its geographical location. In this example the RegistryObject would have at least three Classification instances added to it. The RegistryObject interface provides several addClassification methods to allow a client to add Classification instances to a RegistryObject.	keyedReference (in categoryBag)
ExternalIdentifier ExternalIdentifier instances provide the additional identifier information to RegistryObject such as DUNS number, Social Security Number, or an alias name of the organization. The attribute name inherited from RegistryObject is used to contain the identification scheme (Social Security Number, etc), and the attribute value contains the actual information. Each RegistryObjects may have 0 or more association(s) with ExternalIdentifier.	keyedReference (in identifierBag)
User	contact

JAXR Information Model	UDDI Data Model
Collection of `ExternalIdentifier` instances	`identifierBag`
Collection of `Classification` instances	`categoryBag`
`PostalAddress`	`address`
`SpecificationLink` A `SpecificationLink` provides the linkage between a `ServiceBinding` and one of its technical specifications that describes how to use the service using the `ServiceBinding`. For example, a `ServiceBinding` may have a `SpecificationLink` instance that describes how to access the service using a technical specification in form of a WSDL document. It serves the same purpose as the union of `tModelInstanceInfo` and `instanceDetails` in UDDI.	The union of `tModelInstanceInfo` and `instanceDetails`

UDDI Example Mapped to JAXR:


businessEntity
+ businessKey = "uuid:TB993..."
+ name = "Fjord Auto Inc."
+ businessServices
+ identifierBag
+ categoryBag
0..n
businessService
+ serviceKey = "23T70..."
+ name = "Purchasing Service"
+ bindingTemplates
0..n
keyedReference
+ tModelKey = "uuid:8609-..."
+ keyName = "Just Java"
+ keyValue = "45232"
1..n
bindingTemplate
+ bindingKey = "5E2D412E5..."
+ accessPoint = "http://www.fj.com/..."
+ tModelInstanceDetails
1
tModel
+ tModelKey = "uuid:8609-..."
+ name = "dnb-com:D-U-N-S"
+ description = "DUNS...."
0..n
keyedReference
+ tModelKey = "uuid:C0B9..."
+ keyName = "Transportation"
+ keyValue = "33611"
1
tModel
+ tModelKey = "uuid:C0B9..."
+ +name = "ntis-gov:naics:1997"
+ description = "NAICS...."
1..n
tModelInstanceInfo
+ tModelKey = "uuid:4453..."
+ tModelInstanceDetails
1
tModel
+ tModelKey = "uuid:4453..."
+ name = "myWSDLFile"
+ description = "...."
0..n
overviewDoc
+ description = "Fjord Autos purchasing service"
+ overviewURL = "http://www.fj.com/purchasing"

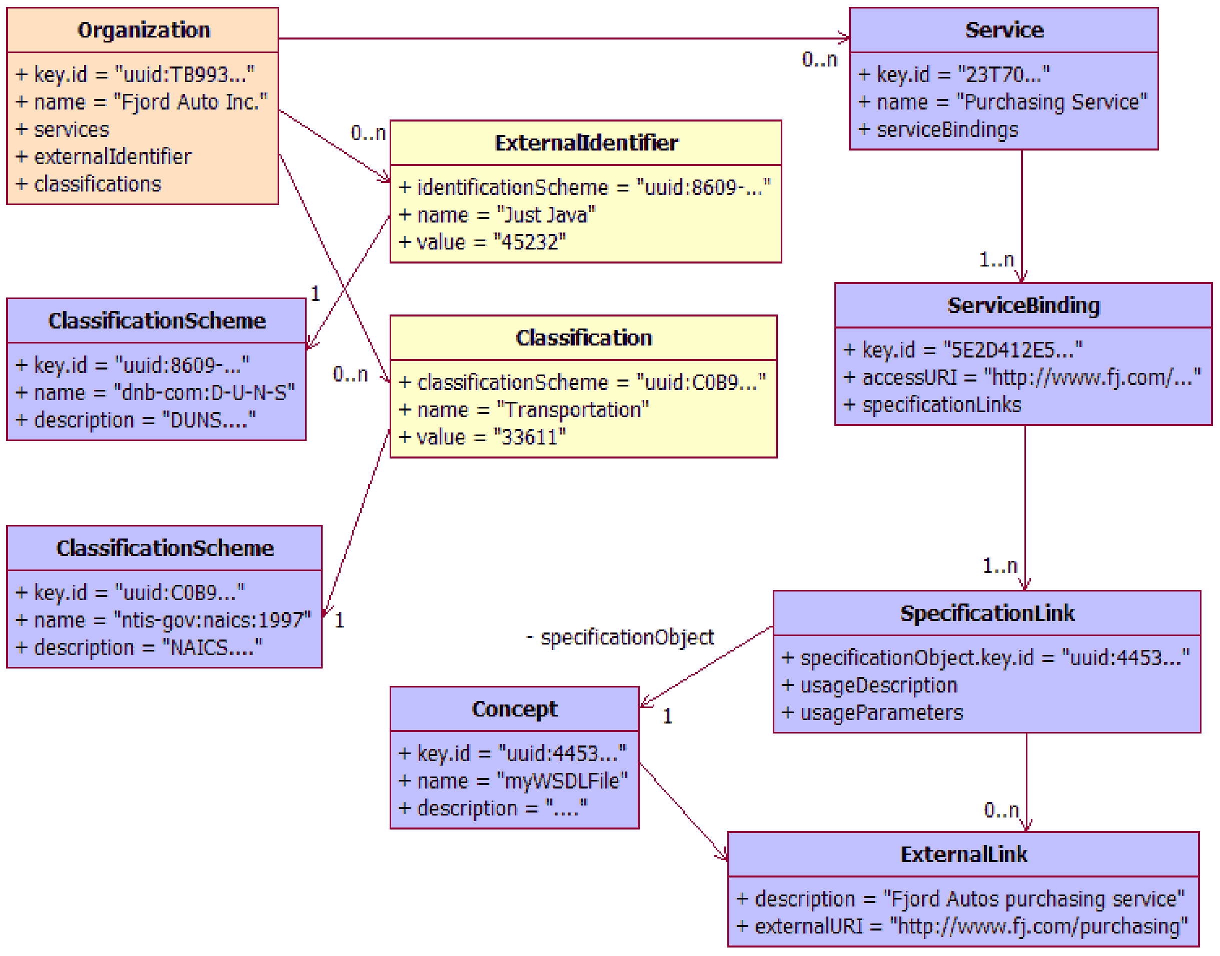

Organization
+ key.id = "uuid:TB993..."
+ name = "Fjord Auto Inc."
+ services
+ externalIdentifier
+ classifications
Service
+ key.id = "23T70..."
+ name = "Purchasing Service"
+ serviceBindings
0..n
ExternalIdentifier
+ identificationScheme = "uuid:8609-..."
+ name = "Just Java"
+ value = "45232"
1..n
ServiceBinding
+ key.id = "5E2D412E5..."
+ accessURI = "http://www.fj.com/..."
+ specificationLinks
1
ClassificationScheme
+ key.id = "uuid:8609-..."
+ name = "dnb-com:D-U-N-S"
+ description = "DUNS...."
0..n
Classification
+ classificationScheme = "uuid:C0B9..."
+ name = "Transportation"
+ value = "33611"
ClassificationScheme
+ key.id = "uuid:C0B9..."
+ name = "ntis-gov:naics:1997"
+ description = "NAICS...."
1
- specificationObject
SpecificationLink
+ specificationObject.key.id = "uuid:4453..."
+ usageDescription
+ usageParameters
Concept
+ key.id = "uuid:4453..."
+ name = "myWSDLFile"
+ description = "...."
1
0..n
ExternalLink
+ description = "Fjord Autos purchasing service"
+ externalURI = "http://www.fj.com/purchasing"

Table 6.2. Mapping of UDDI Inquiry API Calls To JAXR (BusinessQueryManager)

UDDI Method	`BusinessQueryManager` Method
`find_binding`	`findServiceBindings`
`find_business`	`findOrganizations`
`find_related_business`	`findAssociatedObjects(RegistryObject, asociationType)`
`find_service`	`findServices`
`find_tModel`	`findConcepts`, `findClassificationSchemes`
`get_bindingDetail`	Not needed. Handled transparently by JAXR provider
`get_businessDetail`	Not needed. Handled transparently by JAXR provider
`get_businessDetailExt`	Unsupported. Use `RegistryService.makeRegistrySpecificRequest`
`get _serviceDetail`	Not needed. Handled transparently by JAXR provider
`get_tModelDetail`	Not needed. Handled transparently by JAXR provider

Table 6.3. Mapping of UDDI Publisher API Calls to JAXR (BusinessLifeCycleManager)

UDDI Method	`BusinessLifeCycleManager` Method
`add_publisherAssertions`	`saveAssociations(associations, replace)`, `Association.confirm`

UDDI Method	**BusinessLifeCycleManager Method**
delete_binding	deleteServiceBindings
delete_business	deleteOrganizations
delete_publisherAssertions	deleteAssociations
delete_service	deleteServices
delete_tModel	deleteClassificationsSchemes, deleteConcepts. NOTE, 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 deleteClassificationSchemes call.
discard_authToken	Not needed. Handled transparently by JAXR provider
get_assertionStatusReport	findAssociations(findQulifiers, associationTypes, sourceObjectConfirmed, targetObjectConfirmed)
get_authToken	Not needed. Handled transparently by JAXR provider
get_publisherAssertions	QueryManager.getRegistryObjects(objectType)
get_registeredInfo	QueryManager.getRegistryObjects
save_binding	saveServiceBindings
save_business	saveOrganizations
save_service	saveServices
save_tModel	saveClassificationsSchemes, saveConcepts
set_publisherAssertions	saveAssociations(assiations,replace)

The JAXR specification defines the following information model interfaces:

- Organization instance is a RegistryObject that provides information on an organization that has been published to the underlying registry.
- Service instance is a RegistryObject that provides information on services (e.g., Web Service) offered by an Organization. An Organization can have a set of Service objects.
- ServiceBinding instance is a RegistryObject that represents technical information on how to access a specific interface offered by a Service instance. A Service can have a collection of ServiceBinding objects.
- SpecificationLink links a ServiceBinding with its technical specifications that describe how to use the service. For example, a ServiceBinding might have SpecificationLink instances that describe how to access the service using a technical specification in the form of a WSDL (Web Services Description Language) document.
- ClassificationScheme instance represents a taxonomy that you can use to classify or categorize RegistryObject instances.
- Classification instances classify a RegistryObject instance using a classification scheme. Because classification facilitates rapid discovery of RegistryObject instances within the registry, the ability to classify RegistryObject instances in a registry is one of the registry's most significant features.
- Concept instance represents an arbitrary notion, or concept. It can be virtually anything.
- Association instances define many-to-many associations between objects in the information model.
- RegistryPackage instances group together logically related RegistryObject instances. A RegistryPackage might contain any number of RegistryObject instances, and a RegistryObject can be a member of any number of RegistryPackage objects.

- `ExternalIdentifier` instances provide identification information to a `RegistryObject`. You can base such identification on well-known identification schemes such as DUNS (D&B's Data Universal Numbering System) number and social security number.
- `ExternalLink` instances provide a link to content managed outside the registry using a URI (uniform resource identifier).
- `Slot` instances dynamically add arbitrary attributes to `RegistryObject` instances at runtime, an ability that enables extensibility within the information model.
- Most interfaces in the JAXR information model extend the `ExtensibleObject` interface. It provides methods that allow the addition, deletion, and lookup of `Slot` instances.
- `AuditableEvent` instances are `RegistryObject` instances that provide an audit trail for `RegistryObject` instances.
- Affiliated with `Organization`, `User` objects are `RegistryObject` instances that provide information about registered users within the registry. You use `User` objects in a `RegistryObject`'s audit trail.
- `PostalAddress` defines a postal address's attributes. Currently, it provides address information for a `User` and an `Organization`.

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.

Establish a connection between a JAXR client and JAXR provider

The JAXR client must first connect to the JAXR provider. This connection contains the client state and preference information used when the JAXR provider invokes methods on the registry provider.

Before executing any registry operation, a JAXR client connects with a JAXR provider as shown in the following code:

```
import javax.xml.registry.*;
...
// Get an instance of ConnectionFactory object
ConnectionFactory connFactory = ConnectionFactory.newInstance();

// Define connection configuration properties
// Set URLs of the query service and publishing service
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
"http://uddi.ibm.com/testregistry/inquiryapi");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
"https://uddi.ibm.com/testregistry/protect/publishapi");
// If JAXR client goes outside firewall for query, set up HTTP proxy
props.setProperty("com.sun.xml.registry.http.proxyHost", "myhost.mydomain");
props.setProperty("com.sun.xml.registry.http.proxyPort", "8080");

// If JAXR client goes outside firewall for update, set up HTTPS proxy
props.setProperty("com.sun.xml.registry.https.proxyHost", "myhost.mydomain");
props.setProperty("com.sun.xml.registry.https.proxyPort", "8080");
// Set up properties and create Connection object
connFactory.setProperties(props);
Connection connection = connFactory.createConnection();
```

If the JAXR client wishes to submit data to the registry, the client must set authentication information on the connection. Note that the establishment of this authentication information with the registry provider is registry-provider specific and negotiated between the registry provider and the publishing user, often in the form of a Web-based interface. The user does not use the JAXR API to negotiate and establish an account with the registry provider. After establishing a connection, the JAXR client can obtain `RegistryService` interfaces for Web Services discovery and publishing:

```
// Get RegistryService object
RegistryService rs = connection.getRegistryService();
// Get QueryManager and LifeCycleManager objects for
// JAXR Business API (Capability Level 0 - UDDI oriented)
BusinessQueryManager bqm = rs.getBusinessQueryManager();
BusinessLifeCycleManager blcm = rs.getBusinessLifeCycleManager();
```

This example also demonstrates how to obtain the business-focused interfaces, the `BusinessLifeCycleManager` and the `BusinessQueryManager`, to perform registry operations shown in later examples:

```
/*
 *  Establish a connection to a JAXR provider.
 *  Set authentication information, communication preference.
 *  Get business-focused discovery and publish interfaces.
 */
public void makeConnection() {
        // URLs for RegistryServer
        String queryURL = "http://localhost/RegistryServerServlet";
        String publishURL = "http://localhost/RegistryServerServlet";
        /*
         * Define connection configuration properties.
         * For simple queries, you need the query URL.
         * To use a life-cycle manager, you need the publish URL.
         */
        Properties props = new Properties();
        props.setProperty("javax.xml.registry.queryManagerURL", queryUrl);
        props.setProperty("javax.xml.registry.lifeCycleManagerURL", publishUrl);
        try {
                // Create the connection, passing it the configuration properties
                ConnectionFactory factory = ConnectionFactory.newInstance();
                factory.setProperties(props);
                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 = registryService.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");

        // Set client authorization information for privileged registry operations
                PasswordAuthentication passwdAuth = new
PasswordAuthentication(username, password.toCharArray());
                Set creds = new HashSet();
                creds.add(passwdAuth);
                // Set credentials on the JAXR provider
                connection.setCredentials(creds);
                System.out.println("Established security credentials");
                // Set communication preference
                connection.setSynchronous(true);
        } catch (Exception e) {
                e.printStackTrace();
                if (connection != null) {
                        try {
                                connection.close();
                        } catch (JAXRException je) {
                        }
                }
        }
}
```

Registry operation: execute queries to locate services

Now that the JAXR client has established a connection with the JAXR provider and obtained the BusinessQueryManager and BusinessLifeCycleManager from the RegistryService interface, the client can now invoke methods on these interfaces. Often, a JAXR client will wish to discover the services an organization offers. To do so, the JAXR client would invoke methods on the BusinessQueryManager.
JAXR defines top-level interface for query management called QueryManager. There are two extensions of QueryManager interface and they are called BusinessQueryManager and DeclarativeQueryManager.
Interface BusinessQueryManager provides a simple business-level API that provides the ability to query for the most important high-level interfaces in the information model.
Interface DeclarativeQueryManager provides a more flexible generic API that provides the ability to perform ad hoc queries using a declarative query language syntax. Currently the only declarative syntax supported is SQL-92. In version 1 of JAXR, it is optional for a JAXR provider to provide support for SQL query syntax.
This example shows how to find all the organizations in the registry whose names begin with a specified string, qString, and to sort them in alphabetical order:

```
// Define find qualifiers. Sort by name.
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);

// Define name patterns. Organization name begins with qString.
Collection namePatterns = new ArrayList();
namePatterns.add(qString);

// Find organizations using the name
BulkResponse response = bqm.findOrganizations(findQualifiers, namePatterns, null,
null, null, null);
Collection orgs = response.getCollection();
```

Another example shows how to find all the organizations in the registry using case sensitive name pattern:

```
// Define find qualifiers. Do case sensitive search.
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);

// Define name patterns. Organization name contains qString.
Collection namePatterns = new ArrayList();
namePatterns.add("%" + qString + "%");

// Find organizations using the name
BulkResponse response = bqm.findOrganizations(findQualifiers, namePatterns, null,
null, null, null);
Collection orgs = response.getCollection();
```

Steps for finding Organization by Classification:

1. Decide on classification scheme (taxonomy)
2. Build a classification within a particular classification scheme
3. Specify the classification as an argument to the findOrganizations method

To find organizations by classification, you need to establish the classification within a particular classification scheme and then specify the classification as an argument to the findOrganizations method. The following code fragment finds all organizations that correspond to a particular classification within the North American Industry Classification System (NAICS) taxonomy:

```
// Get classification scheme. Taxonomy is NAICS.
ClassificationScheme cScheme = bqm.findClassificationSchemeByName(null, "ntis-
gov:naics");

// Build classifications
Classification classification = blcm.createClassification(cScheme,
        "Snack and Nonalcoholic Beverage Bars", "722213");
Collection classifications = new ArrayList();
classifications.add(classification);

// Find organizations via Classification
```

```
BulkResponse response = bqm.findOrganizations(null, null, classifications, null, null,
null);
Collection orgs = response.getCollection();
```

To find WSDL Specification Instances: use classification scheme of "`uddi-org:types`". In JAXR, a concept is used to hold information about a WSDL specification. JAXR client must find the specification concepts first, then the organizations that use those concepts. Once you get organizations, you can then get services and `bindingTemplates`:

```
// Get classification scheme. Taxonomy is uddi-org:types.
String schemeName = "uddi-org:types";
ClassificationScheme uddiOrgTypes = bqm.findClassificationSchemeByName(null,
schemeName);
// Create a classification, specifying the scheme
// and the taxonomy name and value defined for WSDL
// documents by the UDDI specification.
Classification wsdlSpecClassification = blcm.createClassification(uddiOrgTypes,
"wsdlSpec", "wsdlSpec");
Collection classifications = new ArrayList();
classifications.add(wsdlSpecClassification);
// Find concepts
BulkResponse br = bqm.findConcepts(null, null, classifications, null, null);
// Display information about the concepts found
Collection specConcepts = br.getCollection();
Iterator iter = specConcepts.iterator();
if (!iter.hasNext()) {
        System.out.println("No WSDL specification concepts found");
} else {
        while (iter.hasNext()) {
                Concept concept = (Concept) iter.next();
                String name = getName(concept);
                Collection links = concept.getExternalLinks();
                System.out.println("Specification Concept: Name: " + name +
                        "Key: " + concept.getKey().getId() + "Description: " +
getDescription(concept));
                if (links.size() > 0) {
                        ExternalLink link = (ExternalLink) links.iterator().next();
                        System.out.println("URL of WSDL document: '" +
link.getExternalURI() + "'");
                }
                // Find organizations that use this concept
                Collection specConcepts1 = new ArrayList();
                specConcepts1.add(concept);
        br = bqm.findOrganizations(null, null, null, specConcepts1, null, null);
                // Display information about organizations
                ...
        }
}
```

Finding Services and Service Bindings from Organization:

```
Iterator orgIter = orgs.iterator();
while (orgIter.hasNext()) {
        Organization org = (Organization) orgIter.next();
        Collection services = org.getServices();
        Iterator svcIter = services.iterator();
        while (svcIter.hasNext()) {
                Service svc = (Service) svcIter.next();
                Collection serviceBindings = svc.getServiceBindings();
                Iterator sbIter = serviceBindings.iterator();
                while (sbIter.hasNext()) {
                        ServiceBinding sb = (ServiceBinding) sbIter.next();   }}}
```

Another example use case might be the following:
A user browsing the UDDI registry wishes to find an organization that provides services of the NAICS (North American Industry Classification System) type Computer Systems Design and Related Services in the United States. To perform this query with JAXR, the user would invoke a `findOrganization()` method with classification listed under the well-known taxonomies NAICS and ISO 3166 Geographic Code System (ISO 3166). As JAXR provides a taxonomy service for these classifications, the client can easily access the classification information needed to be passed as `findOrganization()` parameters. A sample of this query to the taxonomy service and registry follows below:

```
public void findOrganizations() throws JAXRException {
        // Search criteria -- Organizations found will return in this sort order
        Collection findQualifiers = new ArrayList();
        findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
        // Query the JAXR taxonomy service
        ClassificationScheme naics =
businessQueryManager.findClassificationSchemeByName(
                findQualifiers, "ntis-gov:naics:1997");

// Create the classification that will be a parameter to findOrganization() method
        Classification computerSystemsDesign =
businessLifeCycleManager.createClassification(
                naics, "Computer Systems Design and Related Services", "5415");
        // Query the taxonomy service
        ClassificationScheme geography =
businessQueryManager.findClassificationSchemeByName(
                findQualifiers, "iso-ch:3166:1999");
// Create the classification passed as a parameter to findOrganization() method.
        Classification us = businessLifeCycleManager.createClassification(
                geography, "United States", "US");
        // Add classifications to the classifications collection parameter
        Collection classifications = new ArrayList();
        classifications.add(computerSystemsDesign);
        classifications.add(us);
        // Invoke the findOrganizations() method on BusinessQueryManager
        BulkResponse bulkResponse = businessQueryManager.findOrganizations(
                findQualifiers, null, classifications, null, null, null);
        if (bulkResponse.getStatus() == JAXRResponse.STATUS_SUCCESS) {
                System.out.println("Found Organization located in the United States  ");
        System.out.println("categorized Computer Systems Design and Related Service ");
        }
}
```

Most calls invoked on the registry provider via the JAXR provider return a `BulkResponse` that contains any registry exceptions encountered and a collection of concrete `RegistryObject` instances or `RegistryObject` keys. To ensure that a registry invocation always returns a response, any exceptions generated by the registry provider are wrapped in a `RegistryException` and stored in the `BulkResponse`'s exceptions collection. In the case of `findXXX(...)` methods, any `RegistryObjects` found are contained in the `BulkResponse` collection. For the above `findOrganization()` method, the `BulkResponse` contains a collection of `Organization` objects found in the registry provider that match the classifications passed as parameters to the method. However, these `Organization` objects provide limited information about the `Organization` and its `Service` objects such as `key`, `name`, and `description`. Another value-added feature of JAXR is the incremental loading of `RegistryObject` details. For instance, in the case of a JAXR UDDI provider, a JAXR `findOrganization()` method transforms to a UDDI `find_Business` request. After invocation, the `find_Business` request returns minimal business and service information such as ID, name, and description. Using UDDI APIs, a UDDI client would need to make an additional call such as `get_BusinessDetail()` to retrieve the organization details. With JAXR, the JAXR UDDI provider performs this invocation to the registry provider on an as-needed basis. The JAXR client can access `Organization` and other `RegistryObject` details by using the `getXXX()` methods on the JAXR information model interfaces. The `getOrganizationDetail()` method demonstrates how a JAXR client would obtain full `Organization` details:

```
public void getOrganizationDetail(BulkResponse bulkResponse) throws JAXRException {
  // Get a collection of Organizations from BulkResponse
  Collection organizations = bulkResponse.getCollection();
  // Iterate through the collection to get an individual Organization
  Iterator orgIter = organizations.iterator();
  while (orgIter.hasNext()) {
    Organization organization = (Organization) orgIter.next();
    // Get a collection of Services from an Organization
    Collection services = organization.getServices();
    // Iterate through the collection to get an individual Service
    Iterator serviceIter = services.iterator();
    while (serviceIter.hasNext()) {
      Service service = (Service) serviceIter.next();
      // Get a collection of ServiceBindings from a Service
      Collection serviceBindings = service.getServiceBindings();
      // Iterate through the collection to get an individual ServiceBinding
      Iterator sbIter = serviceBindings.iterator();
      while (sbIter.hasNext()) {
        ServiceBinding serviceBinding = (ServiceBinding) sbIter.next();
        // Get URI of the service.  You can access the service through this URI.
        String accessURI = serviceBinding.getAccessURI();
        System.out.println("Access the service " + service.getName().getValue()
          + " at this URI " + accessURI);
        // Get a collection of SpecificationLinks from a ServiceBinding.
 // SpecificationLinks provide further technical details needed to access the service.
        Collection specificationLinks = serviceBinding.getSpecificationLinks();
        // Iterate through the collection to get an individual SpecificationLink
        Iterator linkIter = specificationLinks.iterator();
        while (linkIter.hasNext()) {
          SpecificationLink specificationLink = (SpecificationLink) linkIter.next();
          // Get a collection of ExternalLinks from SpecificationLink
         // An ExternalLink points to technical detail necessary to invoke the service
          Collection externalLinks = specificationLink.getExternalLinks();
          // Iterate through the collection to get an ExternalLink
          Iterator elinkIter = externalLinks.iterator();
          while (elinkIter.hasNext()) {
            ExternalLink externalLink = (ExternalLink) elinkIter.next();

            // The externalURI is the pointer to the technical details
            // necessary to invoke the service
            String externalURI = externalLink.getExternalURI();
            System.out.println(
              " Use the technical detail at this URI, "
              + externalURI + " to invoke the service, "  +
              + service.getName().getValue());
          }
          // Obtain usage description
          InternationalString usageDescription =
specificationLink.getUsageDescription();
          // Any parameters necessary to invoke service are in usageParameter
collection
          Collection usageParameters = specificationLink.getUsageParameters();
          // Get the specification concept from the specification link
          // This specificationConcept is equivalent to the tModel registered as
          // the technical specification
          Concept specificationConcept = (Concept)
specificationLink.getSpecificationObject();
        }
      }
    }
  }
}
```

Registry operation: publish or update information about a business service
The JAXR client also publishes Web services, another important registry operation. In addition to discovering partner organization information and services, an organization will want to register its information and services in the registry for partner use. If a JAXR client wishes to publish an organization and its services to a registry, the client uses the `LifeCycleManager` or the more focused `BusinessLifeCycleManager`. Clients familiar with UDDI should use the `BusinessLifeCycleManager`, which provides methods for saving information to the registry provider. Since this is a privileged operation, the user must set authentication information on the JAXR connection. Note that the JAXR provider, not the JAXR client, performs authentication with the registry provider. The JAXR provider negotiates this authentication with the registry provider on an as-needed basis, and the authentication is completely transparent to the JAXR client.
JAXR specification defines two interfaces – `LifeCycleManager` interface and `BusinessLifeCycleManager` interface. `LifeCycleManager` interface provides complete support for all life cycle management needs using a generic API. `BusinessLifeCycleManager` interface defines a simple business-level API for life cycle management of some important high-level interfaces in the information model. This interface provides no new functionality beyond that of `LifeCycleManager`. The goal of defining this interface is to provide an API similar to that of the publisher's API in UDDI. The intent is to provide a familiar API to UDDI developers.
Before it can submit data, the client must send its username and password to the registry in a set of credentials. The following code fragment shows how to do this:

```
// Set userid/password
String username = "myUserName";
String password = "myPassword";
// Create authentication object
PasswordAuthentication passwdAuth = new PasswordAuthentication(username,
password.toCharArray());
// Set the credential with registry provider
Set creds = new HashSet();
creds.add(passwdAuth);
connection.setCredentials(creds);
```

The client creates the organization and populates it with data before saving it. An `Organization` object is one of the more complex data items in the JAXR API. It normally includes the following:

1. A Name object
2. A `Description` object
3. A `Key` object, representing the ID by which the organization is known to the registry
4. A `PrimaryContact` object, which is a `User` object that refers to an authorized user of the registry. A `User` object normally includes a `PersonName` object and collections of `TelephoneNumber` and `EmailAddress` objects.
5. A collection of `Classification` objects
6. `Service` objects and their associated `ServiceBinding` objects

Creating an `Organization`:

```
// Create organization name and description
Organization org = blcm.createOrganization("The Coffee Break");
InternationalString s = blcm.createInternationalString("Purveyor of the finest
coffees");
org.setDescription(s);
// Create primary contact, set name
User primaryContact = blcm.createUser();
PersonName pName = blcm.createPersonName("Jane Doe");
primaryContact.setPersonName(pName);
// Set primary contact phone number
TelephoneNumber tNum = blcm.createTelephoneNumber();
tNum.setNumber("(800) 555-1212");
Collection phoneNums = new ArrayList();
phoneNums.add(tNum);
primaryContact.setTelephoneNumbers(phoneNums);
// Set primary contact email address
EmailAddress emailAddress = blcm.createEmailAddress("jane.doe@TheCoffeeBreak.com");
Collection emailAddresses = new ArrayList();
emailAddresses.add(emailAddress);
```

```
primaryContact.setEmailAddresses(emailAddresses);
// Set primary contact for organization
org.setPrimaryContact(primaryContact);
```

Steps of Adding Classification:

1. Use `BusinessQueryManager` to find the taxonomy to which the organization wants to belong to
2. Create a classification using the classification scheme and a concept (a taxonomy element) within the classification scheme
3. Add the classification to the organization

Example of Adding Classification to Organization:

```
// Set classification scheme to NAICS
ClassificationScheme cScheme = bqm.findClassificationSchemeByName(null,
"ntis-gov:naics");
// Create and add classification
Classification classification = blcm.createClassification(cScheme, "Snack and
Nonalcoholic Beverage Bars", "722213");
Collection classifications = new ArrayList();
classifications.add(classification);
org.addClassifications(classifications);
```

JAXR Provider Must Provide Taxonomies of:

1. The North American Industry Classification System (NAICS)
   http://www.census.gov/epcd/www/naics.html
2. Universal Standard Products and Services Classification (UNSPSC)
   http://www.eccma.org/unspsc/
3. ISO 3166 country codes classification system maintained by the International Organization for Standardization (ISO)
   http://www.iso.org/iso/en/prods-services/iso3166ma/index.html

Suppose the organization "Fly Away Airline Travel Agents" has a Web-based airline reservation service that its partner travel agencies must be able to use. The following code creates such a business and saves it to the registry. The business has contact information, a set of services it offers, and technical information for accessing those services:

```
public void saveOrganization() throws JAXRException {
        // Create Organization in memory
        Organization org = businessLifeCycleManager.createOrganization(
"Fly Away Airline Travel Agents");
        // Create User -- maps to Contact for UDDI
        User user = businessLifeCycleManager.createUser();
        PersonName personName = businessLifeCycleManager.createPersonName(
"Marie A Traveler");
        TelephoneNumber telephoneNumber =
businessLifeCycleManager.createTelephoneNumber();
        telephoneNumber.setNumber("781-333-3333");
        telephoneNumber.setType("office");
        Collection numbers = new ArrayList();
        numbers.add(telephoneNumber);
        EmailAddress email =
businessLifeCycleManager.createEmailAddress("marieb@airlinetravel.com", "office");
        Collection emailAddresses = new ArrayList();
        emailAddresses.add(email);
        user.setPersonName(personName);
        Collection telephoneNumbers = new ArrayList();
        telephoneNumbers.add(telephoneNumber);
        user.setTelephoneNumbers(telephoneNumbers);
        user.setEmailAddresses(emailAddresses);
        org.setPrimaryContact(user);
        // Create service with service name and description
        Service service = businessLifeCycleManager.createService(
"Fly Away Airline Reservation Service");
        service.setDescription(businessLifeCycleManager.createInternationalString(
"Flight Reservation Service"));
```

```
        // Create serviceBinding
        ServiceBinding serviceBinding =
businessLifeCycleManager.createServiceBinding();
        serviceBinding.setDescription(businessLifeCycleManager.
createInternationalString("Information for airline reservation service access"));
        //Turn validation of URI off
        serviceBinding.setValidateURI(false);
        serviceBinding.setAccessURI("http://www.airlinetravel.com:8080/services.reserva
tions.html ");

        // Create the SpecificationLink information
        SpecificationLink specificationLink =
businessLifeCycleManager.createSpecificationLink();
        // Set usage description
        specificationLink.setUsageDescription(businessLifeCycleManager.
                createInternationalString("Search for Reservations when prompted"));
        String usageParameter = "Enter travel agent id when prompted";
        Collection usageParameters = new ArrayList();
        usageParameters.add(usageParameter);
        // Set usage parameters
        specificationLink.setUsageParameters(usageParameters);
        // Set specificationConcept on the specificationLink
        Concept httpSpecificationConcept =
                (Concept)
businessLifeCycleManager.createObject(businessLifeCycleManager.CONCEPT);
        Key httpSpecificationKey =
                businessLifeCycleManager.createKey("uuid:68de9e80-ad09-469d-8a37-
088422bfbc36");
        httpSpecificationConcept.setKey(httpSpecificationKey);
        specificationLink.setSpecificationObject(httpSpecificationConcept);
        // Add the specificationLink to the serviceBinding
        serviceBinding.addSpecificationLink(specificationLink);
        // Add the serviceBinding to the service
        service.addServiceBinding(serviceBinding);
        // Add the service to the organization
        org.addService(service);
        // Add classifications to the organization
        ClassificationScheme naics =
        businessQueryManager.findClassificationSchemeByName(null, "ntis-gov:naics");
        Classification classification =
businessLifeCycleManager.createClassification(naics, "Air Transportation", "481");
        org.addClassification(classification);
        Collection orgs = new ArrayList();
        orgs.add(org);
        // Save organization and whole tree of related objects
        BulkResponse br = businessLifeCycleManager.saveOrganizations(orgs);

        if (br.getStatus() == JAXRResponse.STATUS_SUCCESS) {
                System.out.println("Successfully saved the organization  to the
registry provider.");
```

Chapter 7. J2EE Web Services

Identify the characteristics of and the services and APIs included in the J2EE platform.

Explain the benefits of using the J2EE platform for creating and deploying Web service applications.

Describe the functions and capabilities of the JAXP, DOM, SAX, JAXR, JAX-RPC, and SAAJ in the J2EE platform.

Describe the role of the WS-I Basic Profile when designing J2EE Web services.

WS-I is an open industry effort chartered to promote Web Services interoperability across platforms, applications, and programming languages. The organization brings together a diverse community of Web services leaders to respond to customer needs by providing guidance, recommended practices, and supporting resources for developing interoperable Web services.

Basic Profile is organized around base specifications. Profile adds constraints and guidance as to their interoperable usage based upon implementation experience.

Messaging: XML Representations of SOAP Mesasge

Areas Clarified:

- Fault messages
- SOAP `encodingStyle` attribute
- DTDs and PIs
- SOAP trailers
- `SOAPAction` HTTP header

SOAP `encodingStyle` Attribute

- The `soap:encodingStyle` attribute is used to indicate the use of a particular scheme in the encoding of data into XML
- However, this introduces complexity, as this function can also be served by the use of XML Namespaces. As a result, the Profile prefers the use of `literal`, non-encoded XML.
- R1005: A MESSAGE MUST NOT contain `soap:encodingStyle` attributes on any of the elements whose namespace name is "`http://schemas.xmlsoap.org/soap/envelope/`"
- R1006: A MESSAGE MUST NOT contain `soap:encodingStyle` attributes on any element that is a child of `soap:Body`
- R1007: A MESSAGE described in an `rpc-literal` binding MUST NOT contain `soap:encodingStyle` attribute on any elements are grandchildren of `soap:Body`

SOAP's use of XML: DTDs and PIs

- XML DTDs and PIs may introduce security vulnerabilities, processing overhead and ambiguity in message semantics when used in SOAP messages. As a result, these XML constructs are disallowed by section 3 of SOAP 1.1.
- R1008 A MESSAGE MUST NOT contain a Document Type Declaration (DTD)
- R1009 A MESSAGE MUST NOT contain Processing Instructions (PI)

SOAP Trailers

- The interpretation of sibling elements following the `soap:Body` element is unclear. Therefore, such elements are disallowed.
- R1011: A MESSAGE MUST NOT have any element children of `soap:Envelope` following the `soap:Body` element.

SOAP Trailers: Incorrect Usage

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' >
    <soap:Body>
        <p:Process xmlns:p='http://example.org/Operations' />
    </soap:Body>
    <m:Data xmlns:m='http://example.org/information' >
        Here is some data with the message
    </m:Data>
</soap:Envelope>
```

SOAP Trailers: Correct Usage

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/' >
    <soap:Body>
        <p:Process xmlns:p='http://example.org/Operations' >
            <m:Data xmlns:m='http://example.org/information' >
                Here is some data with the message
            </m:Data>
        </p:Process>
    </soap:Body>
</soap:Envelope>
```

SOAPAction

- Interoperability testing has demonstrated that requiring the SOAPAction HTTP header field value to be quoted increases interoperability of implementations
- Even though HTTP allows for header field values to be unquoted, some implementations require that the value be quoted.
- The SOAPAction header is purely a hint to processors. All vital information regarding the intent of a message is carried in the Envelope
- R2744: A HTTP request MESSAGE MUST contain a SOAPAction HTTP header field with a quoted value equal to the value of the soapAction attribute of soapbind:operation, if present in the corresponding WSDL description.
- R2745: A HTTP request MESSAGE MUST contain a SOAPAction HTTP header field with a quoted empty string value, if in the corresponding WSDL description, the soapAction of soapbind:operation is either not present, or present with an empty string as its value.

SOAPAction Correct Usage. Example 1.
WSDL:

```
<soapbind:operation soapAction="foo" />
```

HTTP header field:

```
SOAPAction: "foo"
```

SOAPAction Correct Usage. Example 2.
WSDL:

```
<soapbind:operation />
```

or

```
<soapbind:operation soapAction="" />
```

HTTP header field:

```
SOAPAction: ""
```

Messaging: Use of SOAP in HTTP

Areas Clarified:

- Identifying SOAP faults
- HTTP methods and extensions
- HTTP and TCP ports
- HTTP success status codes
- HTTP redirect status codes
- HTTP server error status codes

Identifying SOAP Faults

- Background: Some consumer implementations use only the HTTP status code to determine the presence of a SOAP Fault. Because there are situations where the Web infrastructure changes the HTTP status code, and for general reliability, the Profile requires that they examine the envelope.
- R1107: A RECEIVER MUST interpret SOAP messages containing only a soap:Fault element as a Fault.

HTTP Methods and Extensions

- Background: The SOAP 1.1 specification defined its HTTP binding such that two possible methods could be used, the HTTP POST method and the HTTP Extension Framework's M-POST method. The Profile requires that only the HTTP POST method be used and precludes use of the HTTP Extension Framework.
- R1132 A HTTP request MESSAGE MUST use the HTTP POST method.
- R1108 A MESSAGE MUST NOT use the HTTP Extension Framework

HTTP and TCP ports

- Background: SOAP is designed to take advantage of the HTTP infrastructure. However, there are some situations (e.g., involving proxies, firewalls and other intermediaries) where there may be harmful side effects. As a result, instances may find it advisable to use ports other than the default for HTTP (port 80).
- R1110 An INSTANCE MAY accept connections on TCP port 80 (HTTP).

HTTP Success Status Codes

- Background: HTTP uses the 2xx series of status codes to communicate success. In particular, 200 is the default for successful messages, but 202 can be used to indicate that a message has been submitted for processing. Additionally, other 2xx status codes may be appropriate, depending on the nature of the HTTP interaction.
- R1124 An INSTANCE MUST use a 2xx HTTP status code for responses that indicate a successful outcome of a request.
- R1111 An INSTANCE SHOULD use a "200 OK" HTTP status code for responses that contain a SOAP message that is not a SOAP fault.
- R1112 An INSTANCE SHOULD use either a "200 OK" or "202 Accepted" HTTP status code for a response that does not contain a SOAP message but indicates successful HTTP outcome of a request.

HTTP Redirect Status Codes

- Backgroud: There are interoperability problems with using many of the HTTP redirect status codes, generally surrounding whether to use the original method, or GET
- The Profile mandates "307 Temporary Redirect", which has the semantic of redirection with the same HTTP method, as the correct status code for redirection
- R1130 An INSTANCE MUST use HTTP status code "307 Temporary Redirect" when redirecting a request to a different endpoint.
- R1131 A CONSUMER MAY automatically redirect a request when it encounters a "307 Temporary Redirect" HTTP status code in a response.

HTTP Server Error Status Codes

- Background: HTTP uses the 5xx series of status codes to indicate failure due to a server error.
- R1126: An INSTANCE MUST use a "500 Internal Server Error" HTTP status code if the response message is a SOAP Fault.

Service Description: Document Structure

WSDL Schema Definitions

- Background: The normative schemas for WSDL appearing in Appendix 4 of the WSDL 1.1 specification have inconsistencies with the normative text of the specification. The Profile references new schema documents that have incorporated fixes for known errors.
- Although the Profile requires WSDL descriptions to be Schema valid, it does not require consumers to validate WSDL documents. It is the responsibility of a WSDL document's author to assure that it is Schema valid.
- R2028 A DESCRIPTION using the WSDL namespace (prefixed "`wsdl`" in this Profile) MUST be valid according to the XML Schema found at "`http://schemas.xmlsoap.org/wsdl/2003-02-11.xsd`".
- R2029 A DESCRIPTION using the WSDL SOAP binding namespace (prefixed "`soapbind`" in this Profile) MUST be valid according to the XML Schema found at "`http://schemas.xmlsoap.org/wsdl/soap/2003-02-11.xsd`".

Placement of WSDL `import` Element

- R2022 When they appear in a DESCRIPTION, `wsdl:import` elements MUST precede all other elements from the WSDL namespace except `wsdl:documentation`.
- R2023 When they appear in a DESCRIPTION, `wsdl:types` elements MUST precede all other elements from the WSDL namespace except `wsdl:documentation` and `wsdl:import`.

Correct Usage. Example 1.

```
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote/definitions">
        <import namespace="http://example.com/stockquote/base"
                location="http://example.com/stockquote/stockquote.wsdl"/>

        <message name="GetLastTradePriceInput">
                <part name="body" element="..."/>
        </message>
        ...
</definitions>
```

Correct Usage. Example 2.

```
<definitions name="StockQuote"
        ...
        xmlns="http://schemas.xmlsoap.org/wsdl/">
        <types>
                <schema targetNamespace="http://example.com/stockquote/schemas"
                        xmlns="http://www.w3.org/2001/XMLSchema">
                        ...
                </schema>
        </types>
        <message name="GetLastTradePriceInput">
                <part name="body" element="tns:TradePriceRequest"/>
        </message>
        ...
        <service name="StockQuoteService">
                <port name="StockQuotePort" binding="tns:StockQuoteSoap">
                        ....
                </port>
        </service>
</definitions>
```

WSDL Extensions

- Background: Requiring support for WSDL extensions that are not explicitly specified by this or another WS-I Profile can lead to interoperability problems with development tools that have not been instrumented to understand those extensions

Service Description: Types

soapenc:Array

- Background: The recommendations in WSDL 1.1 Section 2.2 for declaration of array types have been interpreted in various ways, leading to interoperability problems. Further, there are other clearer ways to declare arrays.
- R2110 In a DESCRIPTION, array declarations MUST NOT extend or restrict the soapenc:Array type.
- R2111 In a DESCRIPTION, array declarations MUST NOT use wsdl:arrayType attribute in the type declaration.
- R2112 In a DESCRIPTION, array declaration wrapper elements SHOULD NOT be named using the convention ArrayOfXXX.
- R2113 A MESSAGE containing serialized arrays MUST NOT include the soapenc:arrayType attribute.
- Given the WSDL Description:

```

<xsd:element name="MyArray1" type="tns:MyArray1Type"/>

<xsd:complexType name="MyArray1Type">
     <xsd:sequence>
             <xsd:element name="x" type="xsd:string" minOccurs="0"
maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>

```

The SOAP message would serialize as (omitting namespace declarations for clarity):

```
<MyArray1>
  <x>abcd</x>
  <x>efgh</x>
</MyArray1>
```

Service Description: Messages

Binding and Parts

- Background: There are various interpretations about how many wsdl:part elements are permitted or required for document-literal and rpc-literal bindings and how they must be defined.
- Use of wsdl:message elements with zero parts is permitted in Document styles to permit operations that can send or receive messages with empty soap:Bodys. Use of wsdl:message elements with zero parts is permitted in RPC styles to permit operations that have no (zero) parameters and/or a return value.

- For `document-literal` bindings, the Profile requires that at most one part, abstractly defined with the `element` attribute, be serialized into the `soap:Body` element.
- When a `wsdl:part` element is defined using the `type` attribute, the wire representation of that part is equivalent to an implicit (XML Schema) qualification of a `minOccurs` attribute with the value "1", a `maxOccurs` attribute with the value "1" and a nillable attribute with the value "`false`".

Declaration of `part` elements

- Background: Examples 4 and 5 in WSDL 1.1 Section 3.1 incorrectly show the use of XML Schema types (e.g. "`xsd:string`") as a valid value for the `element` attribute of a `wsdl:part` element.
- R2206 A `wsdl:message` in a DESCRIPTION containing a `wsdl:part` that uses the `element` attribute MUST refer, in that attribute, to a global element declaration.

Incorrect Usage Examples:

```
<message name="GetTradePriceInput">
        <part name="tickerSymbol" element="xsd:string"/>
        <part name="time" element="xsd:timeInstant"/>
</message>
<message name="GetTradePriceInput">
        <part name="tickerSymbol" element="xsd:string"/>
</message>
```

Correct Usage Example:

```
<message name="GetTradePriceInput">
        <part name="body" element="tns:SubscribeToQuotes"/>
</message>
```

Service Description: PortTypes

Order of `part` elements

- Background: Permitting the use of `parameterOrder` helps code generators in mapping between method signatures and messages on the wire.
- R2301 The order of the elements in the `soap:body` of a MESSAGE MUST be the same as that of the `wsdl:parts` in the `wsdl:message` that describes it.
- R2302 A DESCRIPTION MAY use the `parameterOrder` attribute of an `wsdl:operation` element to indicate the return value and method signatures as a hint to code generators.

Service Description: Bindings

Use of SOAP Binding

- Background: The Profile limits the choice of bindings to the well defined and most commonly used SOAP binding. MIME and HTTP GET/POST bindings are not permitted by the Profile.
- R2401 A `wsdl:binding` element in a DESCRIPTION MUST use WSDL SOAP Binding as defined in WSDL 1.1 Section 3.

Service Description: SOAP Binding

HTTP Transport

- Background: The profile limits the underlying transport protocol to HTTP (HTTPS is allowed too).
- R2702 A `wsdl:binding` element in a DESCRIPTION MUST specify the HTTP transport protocol with SOAP binding. Specifically, the transport attribute of its `soapbind:binding` child MUST have the value "`http://schemas.xmlsoap.org/soap/http`".

Consistency of `style` Attribute

- Background: The style, "`document`" or "`rpc`", of an interaction is specified at the `wsdl:operation` level, permitting `wsdl:bindings` whose `wsdl:operations` have different `styles`. This has led to interoperability problems.
- R2705 A `wsdl:binding` in a DESCRIPTION MUST use either be a `rpc-literal` binding or a `document-literal` binding.

Encodings and the use Attribute

- Background: The Profile prohibits the use of encodings, including the SOAP encoding.
- R2706 A `wsdl:binding` in a DESCRIPTION MUST use the value of "`literal`" for the `use` attribute in all `soapbind:body`, `soapbind:fault`, `soapbind:header` and `soapbind:headerfault` elements.

Default for use Attribute

- Background: There is an inconsistency between the WSDL 1.1 specification and the WSDL 1.1 schema regarding whether the `use` attribute is optional on `soapbind:body`, `soapbind:header`, and `soapbind:headerfault`, and if so, what omitting the attribute means.

- R2707 A `wsdl:binding` in a DESCRIPTION that contains one or more `soapbind:body`, `soapbind:fault`, `soapbind:header` or `soapbind:headerfault` elements that do not specify the `use` attribute MUST be interpreted as though the value "`literal`" had been specified in each case.

Child Element for Document-Literal Bindings

- Background: WSDL 1.1 is not completely clear what, in `document-literal` style bindings, the child element of `soap:Body` is.
- R2712 A `document-literal` binding MUST be represented on the wire as a MESSAGE with a `soap:Body` whose child element is an instance of the global element declaration referenced by the corresponding `wsdl:message` part.

One-way Operations

- Background: There are differing interpretations of how HTTP is to be used when performing one-way operations.
- One-way operations do not produce SOAP responses. Therefore, the Profile prohibits sending a SOAP envelope in response to a one-way operation. This means that transmission of one-way operations can not result in processing level responses or errors. For example, a "500 Internal Server Error" HTTP response that includes a SOAP message containing a SOAP Fault element can not be returned.
- The HTTP response to a one-way operation indicates the success or failure of the transmission of the message. Based on the semantics of the different response status codes supported by the HTTP protocol, the Profile specifies that "200" and "202" are the preferred status codes that the sender should expect, signifying that the one-way message was received. A successful transmission does not indicate that the SOAP processing layer and the application logic has had a chance to validate the message or have committed to processing it.
- Despite the fact that the HTTP 1.1 assigns different meanings to response status codes "200" and "202", in the context of the Profile they should be considered equivalent by the initiator of the request. The Profile accepts both status codes because some SOAP implementations have little control over the HTTP protocol implementation and cannot control which of these response status codes is sent.
- R2714 For one-way operations, an INSTANCE MUST NOT return a HTTP response that contains a SOAP envelope. Specifically, the HTTP response entity-body must be empty.
- R2750 A CONSUMER MUST ignore a SOAP response carried in a response from a one-way operation.
- R2727 For one-way operations, a CONSUMER MUST NOT interpret a successful HTTP response status code (i.e., 2xx) to mean the message is valid or that the receiver would process it.

Service Description: Use of XML Schema

Use of XML Schema

- Background: WSDL 1.1 uses XML Schema as one of its type systems. The Profile mandates the use of XML Schema as the type system for WSDL descriptions of Web Services.
- R2800 A DESCRIPTION MAY use any construct from XML Schema 1.0.
- R2801 A DESCRIPTION MUST use XML Schema 1.0 Recommendation as the basis of user defined datatypes and structures.

Chapter 8. Security

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.

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. To run with HTTPS, the service endpoint address must be in the form `https://`. The integrity and confidentiality of transport data, including SOAP messages and HTTP basic authentication, is confirmed when you use SSL and TLS. Web services applications can also use Federal Information Processing Standard (FIPS) approved ciphers for more secure TLS connections.

Implementing security at the transport level means, securing the network protocol, a Web Service uses for communication. 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.
HTTP basic authentication uses a user name and password to authenticate a service client to a secure endpoint. A simple way to provide authentication data for the service client is to authenticate to the protected service endpoint to the HTTP basic authentication. The basic authentication is located in the HTTP header that carries the SOAP request. When the application server receives the HTTP request, the user name and password are retrieved and verified using the authentication mechanism specific to the server. Although the basic authentication data is base64-encoded, sending data over HTTPS is recommended. The integrity and confidentiality of the data can be protected by the Secure Sockets Layer (SSL) protocol.

Security at XML level

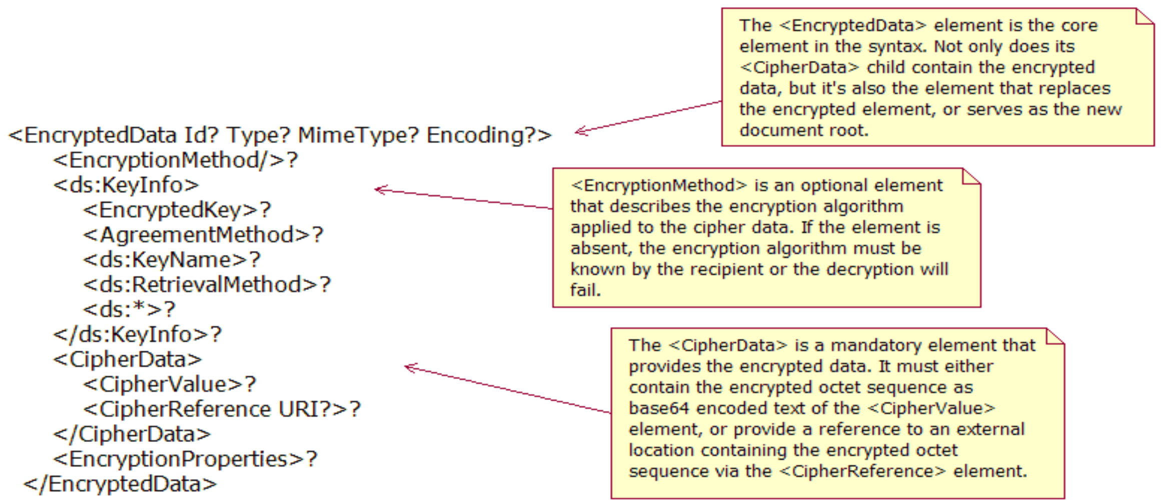
There are some standards available for securing Web Services at XML level. They are:

- **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>
```



```
<EncryptedData Id? Type? MimeType? Encoding?>
   <EncryptionMethod/>?
   <ds:KeyInfo>
     <EncryptedKey>?
     <AgreementMethod>?
     <ds:KeyName>?
     <ds:RetrievalMethod>?
     <ds:*>?
   </ds:KeyInfo>?
   <CipherData>
     <CipherValue>?
     <CipherReference URI?>?
   </CipherData>
   <EncryptionProperties>?
 </EncryptedData>
```

The <EncryptedData> element is the core element in the syntax. Not only does its <CipherData> child contain the encrypted data, but it's also the element that replaces the encrypted element, or serves as the new document root.

<EncryptionMethod> is an optional element that describes the encryption algorithm applied to the cipher data. If the element is absent, the encryption algorithm must be known by the recipient or the decryption will fail.

The <CipherData> is a mandatory element that provides the encrypted data. It must either contain the encrypted octet sequence as base64 encoded text of the <CipherValue> element, or provide a reference to an external location containing the encrypted octet sequence via the <CipherReference> element.

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=...>
        <!-- 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>
```

Note the difference between the Type attributes in the two examples. From looking at this attribute, we can tell immediately whether the plaintext is an element of just its content.
EncryptedData is shown below. In this structure "?" denotes zero or one occurrence; "+" denotes one or more occurrences; "*" denotes zero or more occurrences; and the empty element tag means the element must be empty:

```
<EncryptedData Id? Type? MimeType? Encoding?>
    <EncryptionMethod/>?
    <ds:KeyInfo>
        <EncryptedKey>?
        <AgreementMethod>?
        <ds:KeyName>?
        <ds:RetrievalMethod>?
        <ds:*>?
    </ds:KeyInfo>?
    <CipherData>
        <CipherValue>?
        <CipherReference URI?>?
    </CipherData>
    <EncryptionProperties>?
</EncryptedData>
```

EncryptedData's most important element, CipherData, either directly contains the ciphertext in CipherValue, or, if CipherReference is used, a reference to it. The other elements are optional because usually the receiving party already has the information they contain. For instance, EncryptionMethod lets you specify the algorithm and key size, but usually you and the other party will agree on those beforehand. The same goes for KeyInfo: it gives you the flexibility to give the other party the material to decrypt your message, but you'd probably want to sent it through some out-of-band mechanism. EncryptionProperties serves as another optional element used for, optional information, such as a date/time stamp.
Consider the following fictitious payment information, which includes identification information and information appropriate to a payment method (e.g., credit card, money transfer, or electronic check):

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Credit card number is sensitive information. If the application wishes to keep that information confidential, it can encrypt the CreditCard element:

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
                xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>A23B45C56</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

By encrypting the entire CreditCard element from its start to end tags, the identity of the element itself is hidden. The CipherData element contains the encrypted serialization of the CreditCard element.

Encrypting XML Element Content (Elements)

As an alternative scenario, it may be useful for intermediate agents to know that John used a credit card with a particular limit, but not the card's number, issuer, and expiration date. In this case, the content (character data or children elements) of the CreditCard element is encrypted:

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
                    Type='http://www.w3.org/2001/04/xmlenc#Content'>
      <CipherData>
        <CipherValue>A23B45C56</CipherValue>
      </CipherData>
    </EncryptedData>
  </CreditCard>
</PaymentInfo>
```

Encrypting XML Element Content (Character Data)

We can consider the scenario in which all the information except the actual credit card number can be in the clear, including the fact that the Number element exists:

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>
      <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
                      Type='http://www.w3.org/2001/04/xmlenc#Content'>
        <CipherData>
          <CipherValue>A23B45C56</CipherValue>
        </CipherData>
      </EncryptedData>
    </Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Both CreditCard and Number are in the clear, but the character data content of Number is encrypted.

Encrypting Arbitrary Data and XML Documents

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>
```

Super-Encryption: Encrypting 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. Encrypting only the content of these elements, or encrypting selected child elements is an invalid instance under the provided schema. For example, consider the following:

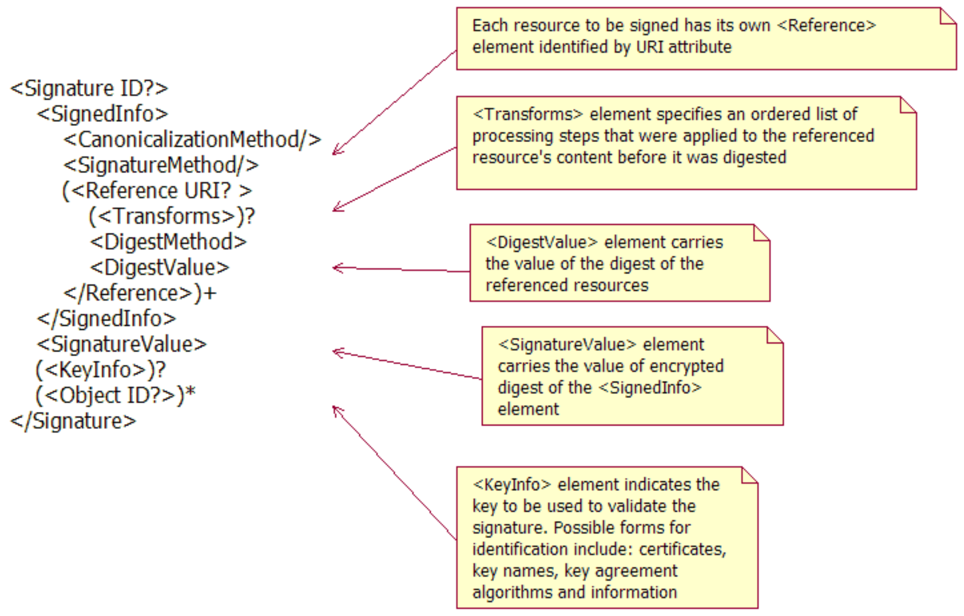
```
<pay:PaymentInfo xmlns:pay='http://example.org/paymentv2'>
  <EncryptedData Id='ED1' xmlns='http://www.w3.org/2001/04/xmlenc#'
             Type='http://www.w3.org/2001/04/xmlenc#Element'>
    <CipherData>
      <CipherValue>originalEncryptedData</CipherValue>
    </CipherData>
  </EncryptedData>
</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 W3C (World Wide Web Consortium) and the IETF (Internet Engineering Task Force) jointly coordinate this effort. 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.
  So why XML Digital Signature? XML Digital Signature provides a flexible means of signing and supports diverse sets of Internet transaction models. For example, you can sign individual items or multiple items of an XML document. 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). You can sign not only XML data, but also non-XML data. 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. 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.
  The XML signature specification is an extremely flexible tool for generating digitally signed XML documents. 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.



```
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    (<Reference URI? >
      (<Transforms>)?
      <DigestMethod>
      <DigestValue>
    </Reference>)+
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo>)?
  (<Object ID?>)*
</Signature>
```

Each resource to be signed has its own <Reference> element identified by URI attribute
<Transforms> element specifies an ordered list of processing steps that were applied to the referenced resource's content before it was digested
<DigestValue> element carries the value of the digest of the referenced resources
<SignatureValue> element carries the value of encrypted digest of the <SignedInfo> element
<KeyInfo> element indicates the key to be used to validate the signature. Possible forms for identification include: certificates, key names, key agreement algorithms and information

XML Signature Forms:

1. **Enveloped**
   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. Obviously, enveloped signatures must take care not to include their own value in the calculation of the `SignatureValue`.
   `Signature` is enveloped within the content been signed:

```
<doc Id="myID">
   <myElement>
          ...
   </myElement>
   <Signature>
          ...
          <Reference URI="#myID"/>
          ...
   </Signature>
</doc>
```

2. **Enveloping**
   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>
```

3. **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.
   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` is external to the content that is signed:

```
<Signature>
   ...
   <Reference URI="http://www.buy.com/books/purchaseWS"/>
   ...
</Signature>
```

XML Signatures are applied to arbitrary digital content (data objects) via an indirection. Data objects are digested, the resulting value is placed in an element (with other information) and that element is then digested and cryptographically signed. XML digital signatures are represented by the `Signature` element which has the following structure (where "?" denotes zero or one occurrence; "+" denotes one or more occurrences; and "*" denotes zero or more occurrences):

```
<Signature ID?>
    <SignedInfo>
        <CanonicalizationMethod/>
        <SignatureMethod/>
        (<Reference URI? >
            (<Transforms>)?
            <DigestMethod>
            <DigestValue>
        </Reference>)+
    </SignedInfo>
    <SignatureValue>
    (<KeyInfo>)?
    (<Object ID?>)*
</Signature>
```

Signatures are related to data objects via URIs. Within an XML document, signatures are related to local data objects via fragment identifiers. Such local data can be included within an enveloping signature or can enclose an enveloped signature. Detached signatures are over external network resources or local data objects that reside within the same XML document as sibling elements; in this case, the signature is neither enveloping (signature is parent) nor enveloped (signature is child). Since a `Signature` element (and its `Id` attribute value/name) may co-exist or be combined with other elements (and their IDs) within a single XML document, care should be taken in choosing names such that there are no subsequent collisions that violate the ID uniqueness validity constraint.
Simple example:

```
<Signature Id="MyFirstSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <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.w3.org/TR/2000/REC-xhtml1-20000126/">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
  <KeyInfo>
    <KeyValue>
      <DSAKeyValue>
        <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
      </DSAKeyValue>
    </KeyValue>
  </KeyInfo>
</Signature>
```

The required `SignedInfo` element is the information that is actually signed. Core validation of `SignedInfo` consists of two mandatory processes: validation of the signature over `SignedInfo` and validation of each `Reference` digest within `SignedInfo`. Note that the algorithms used in calculating the `SignatureValue` are also included in the signed information while the `SignatureValue` element is outside `SignedInfo`.

The `CanonicalizationMethod` is the algorithm that is used to canonicalize the `SignedInfo` element before it is digested as part of the signature operation. Note that this example is not in canonical form.

The `SignatureMethod` is the algorithm that is used to convert the canonicalized `SignedInfo` into the `SignatureValue`. It is a combination of a digest algorithm and a key dependent algorithm and possibly other algorithms such as padding, for example RSA-SHA1. The algorithm names 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.

How to Create an XML Signature:

4. Determine which resources are to be signed.
   This will take the form of identifying the resources through a Uniform Resource Identifier (URI).
   - `"http://www.abccompany.com/index.html"` - would reference an HTML page on the Web
   - `"http://www.abccompany.com/logo.gif"` - would reference a GIF image on the Web
   - `"http://www.abccompany.com/xml/po.xml"` - would reference an XML file on the Web
   - `"http://www.abccompany.com/xml/po.xml#sender1"` - would reference a specific element in an XML file on the Web
5. Calculate the digest of each resource.

In XML signatures, each referenced resource is specified through a Reference element and its digest (calculated on the identified resource and not the Filename>Reference element itself) is placed in a DigestValue child element like:

```
<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>
```

The DigestMethod element identifies the algorithm used to calculate the digest.

6. Collect the Reference elements
   Collect the Reference elements (with their associated digests) within a SignedInfo element like:

```
<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>
```

   The CanonicalizationMethod element indicates the algorithm was used to canonize the SignedInfo element. Different data streams with the same XML information set may have different textual representations, e.g. differing as to whitespace. To help prevent inaccurate verification results, XML information sets must first be canonized before extracting their bit representation for signature processing. The SignatureMethod element identifies the algorithm used to produce the signature value.

7. Signing
   Calculate the digest of the SignedInfo element, sign that digest and put the signature value in a SignatureValue element:

```
<SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
```

8. Add key information
   If keying information is to be included, place it in a KeyInfo element. Here the keying information contains the X.509 certificate for the sender, which would include the public key needed for signature verification:

```
<KeyInfo>
  <X509Data>
    <X509SubjectName>CN=Ed Simon,O=XMLSec
Inc.,ST=OTTAWA,C=CA</X509SubjectName>
    <X509Certificate>MIID5jCCA0+gA...lVN</X509Certificate>
  </X509Data>
</KeyInfo>
```

9. Enclose in a Signature element
   Place the SignedInfo, SignatureValue, and KeyInfo elements into a Signature element. The Signature element comprises the XML signature:

```
<?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>
```

Verifying an XML Signature.

A brief description of how to verify an XML signature:

10. Verify the signature of the SignedInfo element. To do so, recalculate the digest of the SignedInfo element (using the digest algorithm specified in the SignatureMethod element) and use the public verification key to verify that the value of the SignatureValue element is correct for the digest of the SignedInfo element.
11. If this step passes, recalculate the digests of the references contained within the SignedInfo element and compare them to the digest values expressed in each Reference element's corresponding DigestValue element.

- **XKMS (XML Key Management Specification)**
  [see next section]
- **SAML (Security Assertion Markup Language)**
  [see next section]

Federated identity and trust

There are two possible identity management architectures, one based on a centralized model and the other, on a federated 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. One advantage of the centralized model is that, because a single operator owns and controls everything, constructing and managing the identity network could be easier than with the federated model. However, the centralized model has serious downsides. The most serious one is the dangerous potential for the single operator becoming a tollgate for all transactions over the Internet. For example, the operator might charge a fee for every transaction you make. You might have to pay a few cents or dollars whenever you perform a transaction on eBay. The centralized model has another serious problem: the single operator could represent a single point of security failure or hacker attack. One more reason why the centralized model has not garnered any support, especially from the business community, is because 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. This information represents one of the most critical assets to a business, one it is not willing to give up to a third party. The federated model, driven by the Liberty Alliance Project, is designed to correct the centralized model's problems. The goal of the Liberty Alliance Project 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. The first is the role of a consumer. As a consumer, you can have multiple identity profiles, and you can ask different identity providers to maintain these profiles. For example, you might want your HMO to manage your healthcare profile and your brokerage house to maintain your brokerage profile. In fact, as a consumer, you can pick and choose which identity provider to maintain your profile based on price, credibility, service, and so on. In this model, consumers have a final say in terms of who can access what information. Consumers can be a person, a business, or a software entity. In this case, the HMO and brokerage house play the role of identity provider. 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. Identity providers are expected to compete for your business in the future in the same way HMOs, banks, and brokerage houses compete for your business today. The third role is that of 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. It should be noted that this concept of trust-based relationships between organizations and their individual or joint customers has existed in the offline business world for years; two common examples would include travel alliances and affiliate business partnerships.

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.

SAML

Security Assertions Markup Language effort, or SAML, which is being defined by the OASIS (Organization for the Advancement of Structured Information) security services technical committee. The committee aims to outline a standard XML framework for exchanging authentication and authorization information. In a nutshell, SAML is an XML-based framework for exchanging security information. As a framework, it deals with three things. First, it defines syntax and semantics of XML-encoded assertion messages. Second, it defines request and response protocols between requesting and asserting parties for exchanging security information. Third, it defines rules for using assertions with standard transport and message frameworks. For example, it defines how SAML assertion messages can transport using SOAP over HTTP.
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

XACML stands for Extensible Access Control Markup Language, and 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.
In a nutshell, XACML is a general-purpose access control policy language. This means that it provides a syntax (defined in XML) for managing access to resources.
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. Note that the PEP and PDP might both be contained within a single application, or might be distributed across several servers. In addition to providing request/response and policy languages, XACML also provides the other pieces of this relationship, namely finding a policy that applies to a given request and evaluating the request against that policy to come up with a yes or no answer.
There are many existing proprietary and application-specific languages for doing this kind of thing but XACML has several points in its favor:

- It's standard. By using a standard language, you're using something that has been reviewed by a large community of experts and users, you don't need to roll your own system each time, and you don't need to think about all the tricky issues involved in designing a new language. Plus, as XACML becomes more widely deployed, it will be easier to interoperate with other applications using the same standard language.
- It's generic. This means that rather than trying to provide access control for a particular environment or a specific kind of resource, it can be used in any environment. One policy can be written which can then be used by many different kinds of applications, and when one common language is used, policy management becomes much easier.
- It's distributed. This means that a policy can be written which in turn refers to other policies kept in arbitrary locations. The result is that rather than having to manage a single monolithic policy, different people or groups can manage sub-pieces of policies as appropriate, and XACML knows how to correctly combine the results from these different policies into one decision.
- It's powerful. While there are many ways the base language can be extended, many environments will not need to do so. The standard language already supports a wide variety of data types, functions, and rules about combining the results of different policies. In addition to this, there are already standards groups working on extensions and profiles that will hook XACML into other standards like SAML and LDAP, which will increase the number of ways that XACML can be used.

Every enterprise has a need to secure resources accessed by employees, partners, and customers. For example, browser based access to portals which aggregate resources (web pages, applications, services, etc.) are typical in today's enterprises. Clients send requests to servers for resources, but before a server can return that resource it must determine if the requester is authorized to use the resource. This is where XACML fits in. XACML provides a

policy language which allows administrators to define the access control requirements for their application resources. The language and schema support include data types, functions, and combining logic which allow complex (or simple) rules to be defined. 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. Standard Combining Algorithms are defined for Deny-Overrides, Permit-Overrides, First Applicable, and Only-One Applicable outcomes.
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.
Attributes provide the typed values that represent both a resource requester and the Policy's condition predicates. When describing the requester, attributes may include an issue date and time, issuer identification, and optionally are contained in the Subject, Environment, Resource and Action elements of the access request. The Attribute Designator element is used to retrieve attribute values from a request by specifying the name, type, and issuer of attributes. The SubjectAttributeDesignator, ResourceAttributeDesignator, ActionAttributeDesignator, and EnvironmentAttributeDesignator each retrieves attributes from the respective elements in the request. An AttributeSelector element is used to locate request attributes using XPATH queries. Since queries can return collections of attributes (all of the same primitive type) they are placed in a Bag. Functions are then used to compare attributes contained within a Bag.

XKMS (XML Key Management Specification)

XKMS stands for the XML Key Management Specification and 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. Now let's talk about why we need XKMS. To explain it, I must discuss PKI first. PKI proves important for e-commerce and Web services. However, 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.
[PKI refers to a set of security services for authentication, encryption and digital certificate management under which documents are encrypted with a private key and decrypted using a publicly available key accessible to the recipient via a network. PKI differs from private key technology, like Kerberos, in which a single key that is shared by the sender and receiver is used to encrypt and decrypt a message or document.]
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.
XKMS is a foundational specification for secure Web services, enabling Web services to register and manage cryptographic keys used for digital signatures and encryption.
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.

With XKMS, developers can integrate authentication, digital signature, and encryption services, such as certificate processing and revocation status checking, into applications in a matter of hours - without the constraints and complications associated with proprietary PKI software toolkits.

XKMS Functions

An XKMS-compliant service supports the following operations:

- 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.
- 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.
- 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.

These services can be complimented by having cryptographic capabilities on the client, but client crypto is not required. For example, if a client must generate keys, then some crypto code, resident on the client, performs the key generation. However, the client can just as easily have the XKMS service generate the keys that are subsequently managed through the service. For security, however, most client applications generate a keypair, and then register the public key with the CA.

The Benefits of XKMS

XKMS provides many benefits. The most important benefits are that XKMS is:

- Easy to use: The developer-friendly syntax used in XKMS eliminates the necessity for PKI toolkits and proprietary plug-ins. The XKMS specification allows developers to rapidly implement trust features, incorporating cryptographic support for XML digital signatures and XML encryption using standard XML toolkits.
- Quick to deploy: By simplifying application development, XKMS removes the need to delay PKI deployment because of , and instead, moves the complexity of PKI to server side components. Developers can now focus on their core competency of developing applications rather than the complexities surrounding a PKI deployment.
- Open: The common XML vocabulary used to describe authentication, authorization, and profile information in XML documents makes XKMS services completely platform, vendor, and transport-protocol-neutral. The XKMS specification has been submitted to the World Wide Web Consortium (W3C) as an open standard for distribution and registration of keys.
- Ideal for mobile devices: XKMS allow mobile devices to access full-featured PKI through ultra-minimal-footprint client device interfaces.
- Future-proof: Supports new and emerging PKI developments since the impact of future PKI developments is restricted to server-side components. By restricting the impact of future PKI developments and advancements to the server-side components, XKMS protects developers and applications from becoming incompatible with the latest developments in PKI.

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. In addition, it supports multiple security tokens; for example, tokens that contain Kerberos tickets, X.509 certificates, or SAML assertions. The following example illustrates a message with a username security token:

```
<?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#">
   <S:Header>
      <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
        <m:action>http://fabrikam123.com/getQuote</m:action>
        <m:to>http://fabrikam123.com/stocks</m:to>
        <m:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</m:id>
      </m:path>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
        <wsse:UsernameToken Id="MyID">
            <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>
```

The first two lines start the SOAP envelope. Line (003) begins the headers that are associated with this SOAP message. Lines (004) to (008) specify how to route this message (as defined in WS-Routing).

Line (009) starts the `Security` header that we define in WS-Security specification. This header contains security information for an intended receiver. This element continues until line (029).

Lines (010) to (012) specify a security token that is associated with the message. In this case, it defines username of the client using the `UsernameToken`. Note that here we assume the service knows the password - in other words, it is a shared secret.

Lines (013) to (028) specify a digital signature. This signature ensures the integrity of the signed elements (that they aren't modified). The signature uses the XML Signature specification. In this example, the signature is based on a key generated from the users' password; typically stronger signing mechanisms would be used.

Lines (014) to (021) describe the digital signature. Line (015) specifies how to canonicalize (normalize) the data that is being signed.

Lines (017) to (020) select the elements that are signed. Specifically, line (017) indicates that the `S:Body` element is signed. In this example only the message body is signed; typically additional elements of the message, such as parts of the routing header, should be included in the signature.

Line (022) specifies the signature value of the canonicalized form of the data that is being signed as defined in the XML Signature specification.

Lines (023) to (027) provide a hint as to where to find the security token associated with this signature. Specifically, lines (024) to (025) indicate that the security token can be found at (pulled from) the specified URL.

Lines (031) to (033) contain the body (payload) of the SOAP message.

Protecting the message content from being intercepted (confidentiality) or illegally modified (integrity) are primary security concerns. 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 illustrates the syntax of this header:

```
<S:Envelope>
    <S:Header>
            ...
        <Security S:actor="..." S:mustUnderstand="...">
        <!--
             actor attribute is optional, however no two instances of the
             header block may omit an actor or specify the same actor
         -->
            ...
        </Security>
            ...
    </S:Header>
    ...
</S:Envelope>
```

The following sample message illustrates the use of security tokens, signatures, and encryption. For this example, we use a fictitious "RoutingTransform" that selects the immutable routing headers along with the message body:

```
<?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/2002/04/secext"
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
  <S:Header>
     <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
        <m:action>http://fabrikam123.com/getQuote</m:action>
        <m:to>http://fabrikam123.com/stocks</m:to>
        <m:from>mailto:johnsmith@fabrikam123.com</m:from>
        <m:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</m:id>
     </m:path>
     <wsse:Security>
        <wsse:BinarySecurityToken
               ValueType="wsse:X509v3"
               Id="X509Token"
               EncodingType="wsse:Base64Binary">
        MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
        </wsse:BinarySecurityToken>
        <xenc:EncryptedKey>
            <xenc:EncryptionMethod Algorithm=
                  "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
            <ds:KeyInfo>
              <ds:KeyName>CN=Hiroshi Maruyama, C=JP</ds:KeyName>
            </ds:KeyInfo>
            <xenc:CipherData>
               <xenc:CipherValue>d2FpbmdvbGRfE0lm4byV0...
               </xenc:CipherValue>
            </xenc:CipherData>
            <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 (003)-(051) contain the SOAP message headers.

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 (010)-(050) represent the `Security` header block. This contains the security-related information for the message.

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 (041)-(043) indicate the actual signature value - specified in Line (042).
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).
The body of the message is represented by Lines (052)-(060).
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).

Liberty Project

- Create an open standard for identity, authentication and authorization.
- Objective: lower costs, accelerate commercial opportunities, and increase customer satisfaction
- Federated standard will enable every business to:
    - Maintain their own customer/employee/device data.
    - Tie data to an individual's or business's identity.
    - Share data with partners according to its business objectives, and customer's preferences.

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.

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

1. 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>
```

2. Set security properties in the 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

1. Configure SSL connector
2. 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>
```

3. 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();
```

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.

1212

Chapter 9. Developing Web Services

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.

Packaging

Port components may be packaged in a WAR file, or EJB JAR file. Port components packaged in a WAR file must use a JAX-RPC Service Endpoint for the Service Implementation Bean. Port components packaged in a EJB-JAR file must use a Stateless Session Bean for the Service Implementation Bean. The developer is responsible for packaging, either by containment or reference, the WSDL file, Service Endpoint Interface class, Service Implementation Bean class, and their dependent classes, JAX-RPC mapping file along with a Web services deployment descriptor in a J2EE module. The location of the Web services deployment descriptor in the module is module specific. WSDL files are located relative to the root of the module and are typically located in the `wsdl` directory that is co-located with the module deployment descriptor or a subdirectory of it. Mapping files are located relative to the root of the module and are typically co-located with the WSDL file.

The `wsdl` directory is a well-known location that contains WSDL files and any relative content the WSDL files may reference. WSDL files and their relative references will be published during deployment.

Stateless Session EJB Service Implementation Beans are packaged in an EJB-JAR that contains the class files and WSDL files. The packaging rules follow those defined by the Enterprise JavaBeans specification. In addition, the Web services deployment descriptor location within the EJB-JAR file is `META-INF/webservices.xml`. The `wsdl` directory is located at `META-INF/wsdl`.

JAX-RPC Service Endpoints are packaged in a WAR file that contains the class files and WSDL files. The packaging rules for the WAR file are those defined by the Servlet specification. A Web services deployment descriptor is located in a WAR at `WEB-INF/webservices.xml`. The `wsdl` directory is located at `WEB-INF/wsdl`.

Directory Structure for JAX-RPC Web Services

The structure of WAR file is shown below (`hello-web.war`):

```
/WEB-INF  <-------- "special" directory, its content is not accessible for HTTP clients
/WEB-INF/web.xml  <------------- servlet container info and service endpoint implementation
/WEB-INF/webservices.xml  <--------- describes the Web Service part of the application
/WEB-INF/wsdl/HelloService.wsdl  <--- wsdl is the Web Service contract published externally
/WEB-INF/mapping.xml  <----------- maps the wsdl to the Service Endpoint Interface [SEI]
/WEB-INF/classes/HelloServiceSEI.class  <----- compiled Service Endpoint Interface [SEI]
/WEB-INF/classes/HelloServiceImpl.class  <--- compiled Service Implementation class
```

The `webservices.xml` captures all the information that the container needs to deploy Web service applications. It provides:

- the location of WSDL - `wsdl-file`
- the mapping file `jaxrpc-mapping-file`
- the `port-component` corresponding to the ports in WSDL
- the Java Service Endpoint Interface (SEI) `service-endpoint-interface`
- the Java representation of the WSDL
- the servlet name `servlet-link` which acts as the SOAP HTTP listener for it

`webservices.xml` :

```
<webservices>
    <webservice-description>
        <webservice-description-name>HelloService</webservice-description-name>
        <wsdl-file>WEB-INF/wsdl/HelloService.wsdl</wsdl-file>
        <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
        <port-component>
            <port-component-name>HelloInterfacePort</port-component-name>
            <wsdl-port xmlns:my="http://hello">my:HelloInterfacePort</wsdl-port>
            <service-endpoint-interface>HelloServiceSEI</service-endpoint-interface>
            <service-impl-bean>
                <servlet-link>HelloInterfacePort</servlet-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

Web Container Deployment Descriptor /WEB-INF/web.xml is shown below:

```
<web-app>
    ...
    <servlet>
        <servlet-name>HelloInterfacePort</servlet-name>
        <display-name>HelloInterfacePort</display-name>
        <description>JAX-RPC endpoint - HelloInterfacePort</description>
        <servlet-class>HelloServiceImpl</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>HelloInterfacePort</servlet-name>
        <url-pattern>/HelloService</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Observe that the `servlet-link` of `webservices.xml` refers to the `servlet-name` present in `web.xml`. This allows the Web service implementation to be exposed by a servlet, and hence over HTTP (SOAP over HTTP). NOTE: the `servlet-class` in `web.xml` actually refers to the Java class (in this case `HelloServiceImpl`) implementing the Web service SEI (`service-endpoint-interface` in `webservices.xml`). It is not actually a `Servlet`.

Directory Structure for EJB 2.1 Web Services

The structure of JAR (ejb-jar) file is shown below (`hello-ejb.jar`):

```
/META-INF
/META-INF/ejb-jar.xml   <----------------------- EJB Container deployment descriptor
/META-INF/webservices.xml  <--------- describes the Web Service part of the application
/META-INF/wsdl/HelloServiceEJB.wsdl <--- wsdl is the Web Service contract published externally
/META-INF/mapping.xml <------------- maps the wsdl to the Service Endpoint Interface [SEI]
/HelloServiceSEI.class  <----------------------- compiled Service Endpoint Interface [SEI]
/HelloServiceBean.class  <--------------------- compiled EJB class (Stateless Session Bean)
```

Configuration file /META-INF/webservices.xml for an EJB Web Service:

```
<webservices>
    <webservice-description>
        <webservice-description-name>HelloServiceEJB</webservice-description-name>
        <wsdl-file>META-INF/wsdl/HelloServiceEJB.wsdl</wsdl-file>
        <jaxrpc-mapping-file>META-INF/mapping.xml</jaxrpc-mapping-file>
        <port-component>
            <port-component-name>HelloServiceInfPort</port-component-name>
            <wsdl-port xmlns:my="http://hello" >my:HelloServiceInfPort</wsdl-port>
            <service-endpoint-interface>HelloServiceSEI</service-endpoint-interface>
            <service-impl-bean>
                <ejb-link>HelloServiceEJB</ejb-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

This `webservices.xml` is more or less similar to the one for a JAX-RPC Web Service, except for `service-impl-bean` having an `ejb-link` element in it, pointing to the fact the implementation is an EJB.

EJB Container deployment descriptor (/META-INF/ejb-jar.xml):

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <display-name>HelloServiceEJB</display-name>
            <ejb-name>HelloServiceEJB</ejb-name>
            <service-endpoint>HelloServiceSEI</service-endpoint>
            <ejb-class>HelloServiceBean</ejb-class>
            <session-type>Stateless</session-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

Observe that the `ejb-link` of `webservices.xml` is used to relate to the `ejb-name` inside `ejb-jar.xml`. Moreover the `ejb-jar.xml` has a new entry `service-endpoint` as part of EJB 2.1 in J2EE 1.4 which has the fully qualified Service Endpoint Interface name.The ejb `ejb-class` should have implementation of the methods defined in Service Endpoint Interface class.

NOTE: The service implementation class need not explicitly implement the service endpoint interface (through a Java `implements` keyword).This allows for flexibility in service endpoint interface implementation because one can use an existing class and expose the business methods one wishes to expose by creating a service endpoint interface class. The service implementation class may implement other methods in addition to those defined by service endpoint interface, but only the service endpoint interface methods are exposed to the client.

Deployment Descriptors

The `webservices.xml` deployment descriptor file defines the set of Web services that are to be deployed in a Web Services for J2EE enabled container.

The developer is responsible not only for the implementation of a Web service, but also for declaring its deployment characteristics. The deployment characteristics are defined in both the module specific deployment descriptor and the `webservices.xml` deployment descriptor. Service Implementations using a stateless session bean must be defined in the `ejb-jar.xml` deployment descriptor file using the `session` element. Service Implementations using a JAX-RPC Service Endpoint must be defined in the `web.xml` deployment descriptor file using the `servlet-class` element. See the Enterprise JavaBeans and Servlet specifications for additional details on developer requirements for defining deployment descriptors. The developer is also required to provide structural information that defines the Port components within the `webservices.xml` deployment descriptor file. The developer is responsible for providing the set of WSDL documents that describe the Web services to be deployed, the Java classes that represent the Web services, and the mapping that correlates the two. The developer is responsible for providing the following information in the `webservices.xml` deployment descriptor:

- Port's name.
  A logical name for the port must be specified by the developer using the `port-component-name` element. This name bears no relationship to the WSDL port name. This name must be unique amongst all port component names in a module.
- Port's bean class.
  The developer declares the implementation of the Web service using the `service-impl-bean` element of the deployment descriptor. The bean declared in this element must refer to a class that implements the methods of the Port's Service Endpoint Interface. This element allows a choice of implementations. For a JAX-RPC Service Endpoint, the `servlet-link` element associates the `port-component` with a JAX-RPC Service Endpoint class defined in the `web.xml` by the `servlet-class` element. For a stateless session bean implementation, the `ejb-link` element associates the `port-component` with a `session` element in the `ejb-jar.xml`. The `ejb-link` element may not refer to a session element defined in another module. A servlet must only be linked to by a single `port-component`. A session EJB must only be linked to by a single `port-component`.
- Port's Service Endpoint Interface.
  The developer must specify the fully qualified class name of the Service Endpoint Interface in the `service-endpoint-interface` element. If the Service Implementation is a stateless session EJB, the developer must also specify the Service Endpoint Interface in the EJB deployment descriptor using the `service-endpoint` element.
- Port's WSDL definition.
  The `wsdl-file` element specifies a location of the WSDL description of a set of Web services. The location is relative to the root of the module and must be specified by the developer. The WSDL file may reference (e.g. import) other files contained within the module using relative references. It may also reference other files external to the module using an explicit URL. Relative imports are declared relative to the file defining the import. Imported files may import other files as well using relative locations or explicit URLs. It is recommended that the WSDL file and relative referenced files be packaged in the `wsdl` directory. Relative references must not start with a "/".
- Port's QName.
  In addition to specifying the WSDL document, the developer must also specify the WSDL port `QName` in the `wsdl-port` element for each Port defined in the deployment descriptor.
- JAX-RPC Mapping.

The developer must specify the correlation of the WSDL definition to the interfaces using the `jaxrpc-mapping-file` element. The same mapping file must be used for all interfaces associated with a `wsdl-file`.

- Handlers.
  A developer may optionally specify handlers associated with the `port-component` using the `handler` element.
- Servlet Mapping.
  A developer may optionally specify a `servlet-mapping`, in the `web.xml` deployment descriptor, for a JAX-RPC Service Endpoint. No more than one `servlet-mapping` may be specified for a servlet that is linked to by a `port-component`. The `url-pattern` of the `servlet-mapping` must be an exact match pattern (i.e. it must not contain an asterisk (“*”)).

Note that if the WSDL specifies an address statement within the port, its URI address is ignored. This address is generated and replaced during the deployment process in the deployed WSDL.

Service Reference Deployment Descriptor Information

The developer is responsible for defining a `service-ref` for each Web service a component within the module wants to reference. This includes the following information:

- Service Reference Name.
  This defines a logical name for the reference that is used in the client source code. It is recommended, but not required that the name begin with `service/`.
- Service type.
  The `service-interface` element defines the fully qualified name of the JAX-RPC Service Interface class returned by the JNDI lookup.
- Ports.
  The developer declares requirements for container managed port resolution using the `port-component-ref` element. The `port-component-ref` elements are resolved to a WSDL port by the container.
- WSDL definition.
  The `wsdl-file` element specifies a location of the WSDL description of the service. The location is relative to the root of the module. The WSDL description may be a partial WSDL, but must at least include the `portType` and `binding` elements. The WSDL description provided by the developer is considered a template that must be preserved by the assembly/deployment process. In other words, the WSDL description contains a declaration of the application’s dependency on `portTypes`, `bindings`, and `QNames`. The WSDL document must be fully specified, including the `service` and `port` elements, if the application is dependent on port `QNames` (e.g. uses the `Service.getPort(QName,Class)` method).
  The developer must specify the `wsdl-file` if any of the following `Service` methods are used.

  ```
  Call createCall() throws ServiceException;
  java.rmi.Remote getPort(java.lang.Class serviceEndpointInterface) throws
  ServiceException;
  javax.xml.namespace.QName getServiceName();
  java.util.Iterator getPorts() throws ServiceException;
  java.net.URL getWSDLDocumentLocation();
  ```

  The WSDL file may reference (e.g. import) other files contained within the module using relative references. It may also reference other files external to the module using an explicit URL. Relative imports are declared relative to the file defining the `import`. Imported files may import other files as well using relative locations or explicit URLs. Relative references must not start with a "/".
- Service Port.
  If the specified `wsdl-file` has more than one service element, the developer must specify the `service-qname`.
- JAX-RPC Mapping.
  The developer specifies the correlation of the WSDL definition to the interfaces using the `jaxrpc-mapping-file` element. The location is relative to the root of the module. The same mapping file must be used for all interfaces associated with a `wsdl-file`. The developer must specify the `jaxrpc-mapping-file` if the `wsdl-file` is specified.
- Handlers.
  A developer may optionally specify handlers associated with the `service-ref` using the `handler` element.

Below is provided an extract of the `web.xml` with the `service-ref` and items of interest noted:

```
<web-app>
    ...
    <service-ref>
        <service-ref-name>service/MyHelloServiceRef</service-ref-name>          <!-- [1] -->
        <service-interface>javax.xml.rpc.Service</service-interface>            <!-- [2] -->
        <wsdl-file>WEB-INF/wsdl/HelloService.wsdl</wsdl-file>
        <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
        <service-qname xmlns:my="http://hello">my:HelloService</service-qname>  <!-- [3] -->
        <port-component-ref>
            <service-endpoint-interface>hello.HelloInterface</service-endpoint-interface>
        </port-component-ref>
    </service-ref>
    ...
<web-app>
```

- [1] Logical name that components in your module will use to look up the Web service
- [2] The classname of the interface whose implementation will be returned by the process of JNDI lookup
- [3] The `service-qname` is the qualified name of the service, as present inside the packaged WSDL

The Servlet (or JSP) executing the look up of the Web service has code as is shown below:

```
public String consumeService (String name) {    ...
    Context ic = new InitialContext();
    Service service = (Service)ic.lookup("java:comp/env/service/MyHelloServiceRef");
    // declare the qualified name of the port, as specified in the wsdl
    QName portQName= new QName("http://hello","HelloInterfacePort");
    // get a handle on that port :
    // Service.getPort(portQName,SEI class)
    HelloInterface helloPort = (HelloInterface)service.getPort(portQName,
hello.HelloInterface.class);
    // invoke the operation : sayHello()
    resultFromService = helloPort.sayHello(name);
    ...
}
```

The client, which is a `Servlet` in this case, makes use of JNDI lookup to get an instance of the `javax.xml.rpc.Service`. It makes use of the '`Service`' object to get a handle on a '`Port`' of the Web service, these are the ports defined in the WSDL representing the service. The port obtained can be cast into a locally packaged service endpoint interface (`hello.HelloInterface`), and the available methods invoked on it. The locally packaged service endpoint interface can be either generated from the WSDL using a tool or can be already available and mapped to the WSDL using a JAX-RPC mapping file. The client can use this '`Service`' object to get a handle on the '`Call`' object as well, and make a DII Web Service call.

The important observation to be made here is that a client always accesses the Service implementation indirectly by way of a JNDI lookup - that is, through the container - and is never passed a direct reference to the Web service implementation. That way, the container can intervene and provide services (logging, security, management) for the client. Moreover a J2EE Web service client remains oblivious of how a port operates and concerns itself only with the methods a port defines.

Note: similar `service-ref` tags inside an `ejb-jar.xml` will allow an EJB or application client to look up and invoke a remote Web service.

JAX-RPC Mapping Deployment Descriptor

The JAX-RPC mapping deployment descriptor has no standard file name, though it is recommended that the file use a `.xml` suffix. There is a 1-1 correspondence between WSDL files and mapping files within a module. The JAX-RPC mapping deployment descriptor contains information that correlates the mapping between the Java interfaces and WSDL definition. A deployment tool uses this information along with the WSDL file to generate stubs and TIEs for the deployed services and service-refs.

Deployment

Deployment starts with a service enabled application or module. The deployer uses a deployment tool to start the deployment process. In general, the deployment tool validates the content as a correctly assembled deployment artifact, collects binding information from the deployer, deploys the components and Web services defined within the modules, publishes the WSDL documents representing the deployed Web services, deploys any clients using Web services, configures the server and starts the application. The deployment tool starts the deployment process by examining the deployable artifact and determining which modules are Web service enabled by looking for a webservices.xml deployment descriptor file contained within the module. Deployment of services occurs before resolution of service references. This is done to allow deployment to update the WSDL port addresses before the service references to them are processed. Validation of the artifact packaging is performed to ensure that:

- Every port in every WSDL defined in the Web services deployment descriptor has a corresponding `port-component` element.
- If the Service Implementation Bean is an EJB, the transaction attributes for the methods defined by the SEI DO NOT include `Mandatory`.
- JAX-RPC service components are only packaged within a WAR file.
- Stateless session bean Web services are only packaged within an EJB-JAR file.
- The WSDL bindings used by the WSDL ports are supported by the Web Services for J2EE runtime. Bindings that are not supported may be declared within the WSDL if no port uses them.

Deployment of each `port-component` is dependent upon the service implementation and container used. Deployment of a JAX-RPC Service Endpoint requires different handling than deployment of a session bean service. If the implementation is a JAX-RPC Service Endpoint, a servlet is generated to handle parsing the incoming SOAP request and dispatch it to an instance of the JAX-RPC service component. The generated servlet class is dependent on threading model of the JAX-RPC Service Endpoint. The `web.xml` deployment descriptor is updated to replace the JAX-RPC Service Endpoint class with the generated servlet class. If the JAX-RPC Service Endpoint was specified without a corresponding `servlet-mapping`, the deployment tool generates one. The WSDL port address for the Port component is the combination of the web app `context-root` and `url-pattern` of the `servlet-mapping`. If the implementation is a stateless session bean, the deployment tool has a variety of options available to it. In general, the deployment tool generates a servlet to handle parsing the incoming SOAP request, the servlet obtains a reference to an instance of an appropriate `EJBObject` and dispatches the request to the stateless session EJB. How the request is dispatched to the Service Implementation Bean is dependent on the deployment tool and deploy time binding information supplied by the deployer.

The deployment tool must deploy and publish all the ports of all WSDL documents described in the Web services deployment descriptor. The deployment tool updates or generates the WSDL port address for each deployed `port-component`. The updated WSDL documents are then published to a location determined by the deployer. It could be as simple as publishing to a file in the modules containing the deployed services, a URL location representing the deployed services of the server, a UDDI or ebXML registry, or a combination of these. This is required for the next step, which is resolving references to Web services.

For each service reference described in the Web services client deployment descriptors, the deployment tool ensures that the client code can access the Web service. The deployment tool examines the information provided in the client deployment descriptor (the Service interface class, the Service Endpoint Interface class, and WSDL ports the client wants to access) as well as the JAX-RPC mapping information. In general the procedure includes providing an implementation of the JAX-RPC Service interface class declared in the deployment descriptor service reference, generating stubs for all the `service-endpoint-interface` declarations (if generated Stubs are supported and the deployer decides to use them), and binding the Service class implementation into a JNDI namespace. The specifics depend on whether or not the service is declared as a client managed or container managed access.

When client managed port access is used, the deployment tool must provide generated stubs or dynamic proxy access to every port declared within the Web services client deployment descriptor. The choice of generated stub or dynamic proxy is deploy time binding information. The container must provide an implementation for a Generated Service Interface if declared within the deployment descriptor.

When container managed port access to a service is used, the container must provide generated stubs or dynamic proxy access to every port declared within the deployment descriptor. The choice of generated stub or dynamic proxy is deploy time binding information. The deployment descriptor may contain a `port-component-link` to associate the reference not only with the Service Endpoint Implementation, but with the WSDL that defines it.

Once the Web services enabled deployable artifact has been converted into a J2EE deployable artifact, the deployment process continues using normal deployment processes.

Generation of any run-time classes the container requires to support a JAX-RPC Service Endpoint or Stateless Session Bean Service Implementation is provider specific. The behavior of the run-time classes must match the deployment descriptor settings of the component. A JAX-RPC Service Endpoint must match the behavior defined by the `servlet` element in the `web.xml` deployment descriptor. A Stateless Session Bean Service Implementation must match the behavior defined by the `session` element and the `assembly-descriptor` in the `ejb-jar.xml` deployment descriptor.

The container must update and/or generate a deployed WSDL document for each declared `wsdl-file` element in the Web services deployment descriptor (`webservices.xml`). If multiple `wsdl-file` elements refer to the same location, a separate WSDL document must be generated for each. The container must not update a WSDL file located in the document root of a WAR file.

The WSDL document described by the `wsdl-file` element must contain `service` and `port` elements and every `port-component` in the deployment descriptor must have a corresponding WSDL port and vice versa. The deployment tool must update the WSDL port address element to produce a deployed WSDL document. The generated port address information is deployment time binding information. In the case of a port-component within a web module, the address is partially constrained by the context-root of the web application and partially constructed from the servlet-mapping (if specified).

The deployment tool and/or container must make the WSDL document that a service-ref is bound to available via a URL returned by the Service Interface getWSDLDocumentLocation() method. This may or may not be the same WSDL document packaged in the module. The process of publishing the bound service-ref WSDL is analogous to publishing deployed WSDL, but only the service-ref that is bound to it is required to have access to it. A Web Services for J2EE provider is required to provide a URL that maintains the referential integrity of the WSDL document the service-ref is bound to if the wsdl-file element refers to a document located in the wsdl directory or one of its subdirectories.

The deployment tool must publish every deployed WSDL document. The deployed WSDL document may be published to a file, URL, or registry. File and URL publication must be supported by the provider. File publication includes within the generated artifacts of the application. Publication to a registry, such as UDDI or ebXML, is encouraged but is not required. If publication to a location other than file or URL is supported, then location of a WSDL document containing a service from that location must also be supported. As an example, a Web services deployment descriptor declares a wsdl-file StockQuoteDescription.wsdl and a port-component which declares a port QName within the WSDL document. When deployed, the port address in StockQuoteDescription.wsdl is updated to the deployed location. This is published to a UDDI registry location. In the same application, a service-ref uses a port-component-link to refer to the deployed port-component. The provider must support locating the deployed WSDL for that port component from the registry it was published to. This support must be available to a deployed client that is not bundled with the application containing the service. Publishing to at least one location is required. Publishing to multiple locations is allowed, but not required. The choice of where (both location and how many places) to publish is deployment time binding information. A Web Services for J2EE provider is required to support publishing a deployed WSDL document if the Web services deployment descriptor (webservices.xml) wsdl-file element refers to a WSDL file contained in the wsdl directory or subdirectory. A vendor may support publication of WSDL files packaged in other locations, but these are considered non-portable. A provider may publish the static content (e.g. no JSPs or Servlets) of the entire wsdl directory and all its subdirectories if the deploy tool cannot compute the minimal set of documents to publish in order to maintain referential integrity. The recommended practice is to place WSDL files referenced by a wsdl-file element and their relative imported documents under the wsdl directory. Web Services for J2EE providers are free to organize the published WSDL documents however they see fit so long as referential integrity is maintained. For example, the wsdl directory tree may be collapsed to a flat published directory structure (updating import statements appropriately). Clients should not depend on the wsdl directory structure being maintained during publication. Access to relatively imported documents should only be attempted by traversing the published WSDL document at the location chosen by the deployer.

The container must provide an implementation of the JAX-RPC Service Interface. There is no requirement for a Service Implementation to be created during deployment. The container may substitute a Generated Service Interface Implementation for a generic Service Interface Implementation. The container must provide an implementation of the JAX-RPC Generated Service Interface if the Web services client deployment descriptor defines one. A Generated Service Interface Implementation will typically be provided during deployment. The Service Interface Implementation must provide a static stub and/or dynamic proxy for all ports declared by the service element in the WSDL description. A container provider must support at least one of static stubs or dynamic proxies, but may provide support for both. The container must make the required Service Interface Implementation available at the JNDI namespace location java:comp/env/service-ref-name where service-ref-name is the name declared within the Web services client deployment descriptor using the service-ref-name element.

If a service-ref contains a port-component-ref that contains a port-component-link, the deployer should bind the container managed Port for the SEI to the deployed port address of the port-component referred to by the port-component-link. For example, given a webservices.xml file containing:

```
<webservices>
        <webservice-description>
                <webservice-description-name>JoesServices</webservice-description-name>
                <wsdl-file>META-INF/joe.wsdl</wsdl-file>
                <jaxrpc-mapping-file>META-INF/mapping.xml</jaxrpc-mapping-file>
                <port-component>
                        <port-component-name>JoePort</port-component-name>
                        ...
                        <service-impl-bean>
                                <ejb-link>JoeEJB</ejb-link>
                        </service-impl-bean>
                </port-component>
        </webservice-description>
</webservices>
```

and a module's deployment descriptor containing:

```
<service-ref>
        <service-ref-name>service/Joe</service-ref-name>
        <service-interface>javax.xml.rpc.Service</service-interface>
        <wsdl-file>WEB-INF/joe.wsdl</wsdl-file>
        ...
        <port-component-ref>
                <service-endpoint-interface>sample.Joe</service-endpoint-interface>
                <port-component-link>JoePort</port-component-link>
        </port-component-ref>
</service-ref>
```

During deployment, the deployer must provide a binding for the port address of the JoePort port-component. This port address must be defined in the published WSDL for JoesServices. The deployer must also provide a binding for container managed port access to the sample.Joe Service Endpoint Interface. This should be the same binding used for the port address of the JoePort port-component. When providing a binding for a port-component-ref, the deployer must ensure that the port-component-ref is compatible with the Port being bound to.

EJB 2.1 Web Service deployment descriptor

The Bean Provider must use web service references to locate web service interfaces as follows.

- Assign an entry in the enterprise bean’s environment to the reference.
- The EJB specification recommends, but does not require, that all references to web services be organized in the service subcontext of the bean’s environment (i.e., in the java:comp/env/service JNDI context).
- Look up the JAX-RPC web service interrface of the referenced service in the enterprise bean's environment using JNDI.

The following example illustrates how an enterprise bean uses an web service reference to locate a web service interface and invoke a method on the service endpoint:

```
public class InvestmentBean implements SessionBean {
        public void checkPortfolio(...) {
                ...
                // Obtain the default initial JNDI context.
                Context initCtx = new InitialContext();

                // Look up the stock quote service in the environment.
                com.example.StockQuoteService sqs = 
(com.example.StockQuoteService)initCtx.lookup(
                        "java:comp/env/service/StockQuoteService");

                // Get the stub for the service endpoint
                com.example.StockQuoteProvider sqp = sqs.getStockQuoteProviderPort();

                // Get a quote
                float quotePrice = sqp.getLastTradePrice(...);
                ...
        }
}
```

In the example, the Bean Provider of the InvestmentBean enterprise bean assigned the environment entry service/StockQuoteService as the web service reference name to refer to the StockQuoteService JAX-RPC web service interface.

Although the web service reference is an entry in the enterprise bean's environment, the Bean Provider must not use a env-entry element to declare it. Instead, the Bean Provider must declare all the web service references using the service-ref elements of the deployment descriptor. This allows the ejb-jar consumer (i.e. Application Assembler or Deployer) to discover all the web service references used by the enterprise bean.

Each service-ref element describes the interface requirements that the referencing enterprise bean has for the referenced web service. The service-ref element contains an optional description element and the

mandatory `service-ref-name` and `service-interface` elements. The use of service references and the `service-ref` deployment descriptor element are described in further detail in [Web Services for J2EE, Version 1.1. http://jcp.org/en/jsr/detail?id=109, http://jcp.org/en/jsr/detail?id=921].
The `service-ref-name` element specifies the web service reference name: its value is the environment entry name used in the enterprise bean code. The `service-interface` element specifies the fully qualified name of the JAX-RPC service interface returned by the JNDI lookup.
A web service reference is scoped to the enterprise bean whose declaration contains the `service-ref` element. This means that the web service reference is not accessible to other enterprise beans at runtime, and that other enterprise beans may define `service-ref` elements with the same `service-ref-name` without causing a name conflict.
The following example illustrates the declaration of a web service reference in the deployment descriptor (`ejb-jar.xml`):

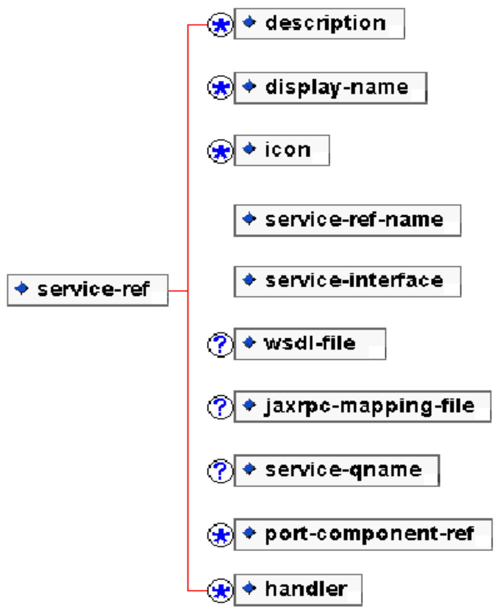
```
<session>
        ...
        <ejb-name>InvestmentBean</ejb-name>
        <ejb-class>com.wombat.empl.InvestmentBean</ejb-class>
        ...
        <service-ref>
                <description>
                        This is a reference to the stock quote
                        service used to estimate portfolio value.
                </description>
                <service-ref-name>service/StockQuoteService</service-ref-name>
                <service-interface>com.example.StockQuoteService</service-interface>
        </service-ref>
        ...
</session>
```

JAX-RPC Web Services deployment descriptor
JSR-109 [http://jcp.org/jsr/detail/109.jsp] defines the model for packaging a Web service interface with its associated WSDL description and associated classes. It defines a mechanism for JAX-RPC-enabled Web containers to link to a component that implements this Web service. A JAX-RPC Web service implementation component uses the APIs defined by the JAX-RPC specification, which defines its contract with the JAX-RPC enabled Web container. It is packaged into the WAR file. The Web service developer makes a declaration of this component using the usual `servlet` declaration. The JAX-RPC-enabled Web container must support the developer in using the Web deployment descriptor to define the following information for the endpont implementation component, using the same syntax as for HTTP Servlet components:

- a logical name which may be used to locate this endpoint description among the other Web components in the WAR
- the fully qualified Java class name of this endpoint implementation
- descriptions of the component which may be displayed in a tool
- the order in which the component is initialized relative to other Web components in the Web container
- `security-role-reference` that it may use to test whether the authenticated user is in a logical security role
- whether or not to override the identity propagated to EJBs called by this component

Any servlet initialization parameters defined by the developer for this Web component may be ignored by the container. Additionally, the JAX-RPC enabled Web component inherits the traditional Web component mechanisms for defining information:

- mapping of the component to the Web container's URL namespace using the servlet mapping technique
- authorization constraints on Web components using security constraints
- the ability to use servlet filters to provide low-level byte stream support for manipulating JAX-RPC messages using the filter mapping technique
- the timeout characteristics of any HTTP sessions that are associated with the component
- links to J2EE objects stored in the JNDI namespace



The `service-ref` declares the reference to a Web service. The `service-ref-name` declares the logical name that the components in the module use to look up the Web service. It is recommended that all service reference names start with `/service/`. The `service-interface` defines the fully qualified class name of the JAX-RPC Service interface that the client depends on. In most cases, the value will be `javax.xml.rpc.Service`. A JAX-RPC generated Service Interface class may also be specified. The `wsdl-file` element contains the URI location of a WSDL file. The location is relative to the root of the module. The `jaxrpc-mapping-file` contains the name of a file that describes the JAX-RPC mapping between the Java interaces used by the application and the WSDL description in the `wsdl-file`. The file name is a relative path within the module file. The `service-qname` element declares the specific WSDL service element that is being refered to. It is not specified if no `wsdl-file` is declared. The `port-component-ref` element declares a client dependency on the container for resolving a Service Endpoint Interface to a WSDL port. It optionally associates the Service Endpoint Interface with a particular `port-component`. This is only used by the container for a `Service.getPort(Class)` method call. The `handler` element declares the handler for a `port-component`. Handlers can access the `init-param` `name-value` pairs using the `HandlerInfo` interface. If `port-name` is not specified, the handler is assumed to be associated with all ports of the service. See JSR-109 Specification [http://www.jcp.org/en/jsr/detail?id=921] for detail. The container that is not a part of a J2EE implementation is not required to support this element. Example of `web.xml`:

```
<service-ref>
   <service-ref-name>service/MyHelloServiceRef</service-ref-name>
   <service-interface>javax.xml.rpc.Service</service-interface>
      ....
      <port-component-ref>
         ....
      </port-component-ref>
      <handler>
         <handler-name>First Handler</handler-name>
         <handler-class >example.AccountTransactionHandler</handler-class>
         <port-name>portA</port-name>
      </handler>
      <handler>
         <handler-name>Second Handler </handler-name>
         <handler-class>example.NewAccountHandler< /handler-class>
         <port-name>portB</port-name>
      </handler>
</service-ref>
```

Given a set of requirements, develop code to process XML files using the SAX, DOM, XSLT, and JAXB APIs.

1212

Given an XML schema for a document style Web service create a WSDL file that describes the service and generate a service implementation.

WSDL file:

```
<message name="AuthorNotFoundException">
     <part name="Author" type="xsd:string" />
</message>
<portType name ="BookSearch">
     <operation name="getBooksByAuthor" >
          <input message="tns:getAuthorName">
          <output message="tns:getBookList">
          <fault name="AuthorNotFoundException" message="tns:AuthorNotFoundException">
     </operation>
</portType>
```

Generated service endpoints interface (SEI):

```
public interface BookSearch extends java.rmi.Remote {
        public Books[] getBooksByAuthor(String authorName) throws
java.rmi.RemoteException,      com.acme.AuthorNotFoundException;
}
```

Generated exception class:

```
public class AuthorNotFoundException extends java.lang.Exception{
        ...
        public AuthorNotFoundException(String Author) {
        ...
        }
        public getAuthor() {...}
}
```

- `portType` = ABSTRACT INTERFACE
- `operation` = METHOD
- `message` = PARAMETERS, RETURN VALUES AND EXCEPTIONS

More detailed example (also demonstrates references in SOAP 1.1 message):

WSDL file:

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                   xmlns:tns="http://www.example.com"
             targetNamespace="http://www.example.com">
 <wsdl:types>
   <xsd:schema targetNamespace="http://www.example.com">
     <!-- "Person" type used by "compare" operation,
          note use optional attributes -->
     <xsd:complexType name="Person">
       <xsd:sequence>
         <xsd:element name="firstName" type="xsd:string" />
         <xsd:element name="lastName" type="xsd:string" />
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:ID" use="optional" />
       <xsd:attribute name="href" type="xsd:anyURI" use="optional" />
     </xsd:complexType>

   </xsd:schema>
 </wsdl:types>
 <!-- RPC style message definitions -->
 <wsdl:message name="compareInput">
   <wsdl:part name="person1" type="tns:Person" />
   <wsdl:part name="person2" type="tns:Person" />
 </wsdl:message>
 <wsdl:message name="compareOutput">
   <wsdl:part name="result" type="xsd:boolean" />
 </wsdl:message>

 <!-- Geometry portType -->
 <wsdl:portType name="Human">
   <wsdl:operation name="compare">
     <wsdl:input message="tns:compareInput" />
     <wsdl:output message="tns:compareOutput" />
   </wsdl:operation>
 </wsdl:portType>
 <!-- Binding for "Human" portType that uses SOAP encoding -->
 <wsdl:binding name="HumanBinding" type="tns:Human">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"
   <wsdl:operation name="compare">
     <soap:operation soapAction="" style="rpc" />
     <wsdl:input message="tns:compareInput">
       <soap:body namespace="http://www.example.com" use="literal" />
     </wsdl:input>
     <wsdl:output message="tns:compareOutput">
       <soap:body namespace="http://www.example.com" use="literal" />
     </wsdl:output>
   </wsdl:operation>
 </wsdl:portType>
</wsdl:definitions>
```

Will generate SEI:

```
public interface Human extends java.rmi.Remote {
        public boolean compare(Person person1, Person person2) throws
java.rmi.RemoteException;
}
```

Following Web-Service call:

```
Person one = new Person("Mikalai", "Zaikin");
Person two = new Person("Volha", "Zaikina");
boolean b = port.compare(one, two);
```

will be serialized in the following SOAP 1.1 message:

```

<soap:Envelope xmlns:soap= "http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <ns:compare xmlns:ns="http://www.example.com">
      <person1>
        <firstName>Mikalai</firstName>
        <lastName>Zaikin</lastName>
      </person1>
      <person2>
        <firstName>Volha</firstName>
        <lastName>Zaikina</lastName>
      </person2>
    </ns:compare>
  </soap:Body>
</soap:Envelope>
```

Following Web-Service call:

```
Person one = new Person("Mikalai", "Zaikin");
```

```
boolean b = port.compare(one, one);
```

will be serialized in the following SOAP 1.1 message:

```
<soap:Envelope xmlns:soap= "http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

    <ns:compare xmlns:ns="http://www.example.com">
      <person1 href="#unique_person_id" />
      <person2 href="#unique_person_id" />
    </ns:compare>

    <ns:Person id="unique_person_id" xmlns:ns="http://www.example.com">
      <firstName>Mikalai</firstName>
      <lastName>Zaikin</lastName>
    </ns:Person>

  </soap:Body>
</soap:Envelope>
```

Given a set of requirements, develop code to create an XML-based, document style, Web service using the JAX-RPC APIs.

1212

Implement a SOAP logging mechanism for testing and debugging a Web Service application using J2EE Web Service APIs.

1212

Given a set of requirements, develop code to handle system and service exceptions and faults received by a Web services client.

1212

Chapter 10. General Design and Architecture

Describe the characteristics of a service oriented architecture and how Web Services fits to this model.

Web services is a service oriented architecture which allows for creating an abstract definition of a service, providing a concrete implementation of a service, publishing and finding a service, service instance selection, and interoperable service use. In general a Web service implementation and client use may be decoupled in a variety of ways. Client and server implementations can be decoupled in programming model. Concrete implementations may be decoupled in logic and transport.

The service provider defines an abstract service description using the Web Services Description Language (WSDL). A concrete Service is then created from the abstract service description yielding a concrete service description in WSDL. The concrete service description can then be published to a registry such as Universal Description, Discovery and Integration (UDDI). A service requestor can use a registry to locate a service description and from that service description select and use a concrete implementation of the service. The abstract service description is defined in a WSDL document as a PortType. A concrete Service instance is defined by the combination of a PortType, transport & encoding binding and an address as a WSDL port. Sets of ports are aggregated into a WSDL service.

Given 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 Façade server-side patterns.

Business Delegate

Problem:

- You want to hide clients from the complexity of remote communication with business service components. Business components are exposed to clients leading to coupling and fine-grained access.

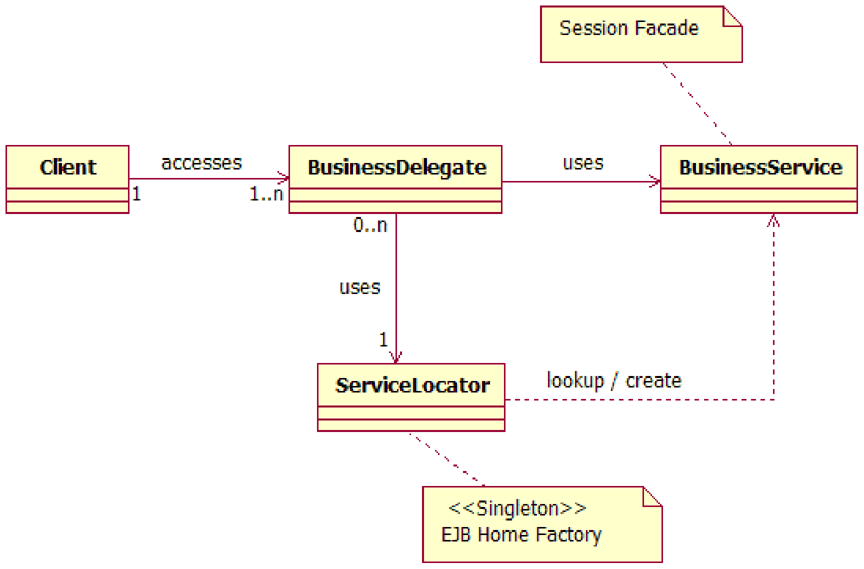
Forces:

- Clients need access to business services.
- Complexity of remote objects. You want to avoid unnecessary invocation of remote services.

- Clients and business components become tightly coupled. You want to minimize coupling between clients and the business services, thus hiding the underlying implementation details of the service, such as lookup and access.
- Business services APIs may change. You want to hide the details of service creation, reconfiguration, and invocation retries from the clients.

Solution:

- Use a Business Delegate to encapsulate access to a business service. The Business Delegate hides the implementation details of the business service, such as lookup and access mechanisms. Use a Business Delegate to reduce coupling between presentation-tier clients and business services:



Consequences:

- Reduces coupling, increases maintainability.
- Implements failure recovery: Retry or Synchronize.
- Translates network (System) Exceptions to Business service Exceptions.
- Exposes simpler, uniform Interface to the business tier.
- Performance enhancing cache.
- Hides complexities of remote services.

Service Locator

Problem:

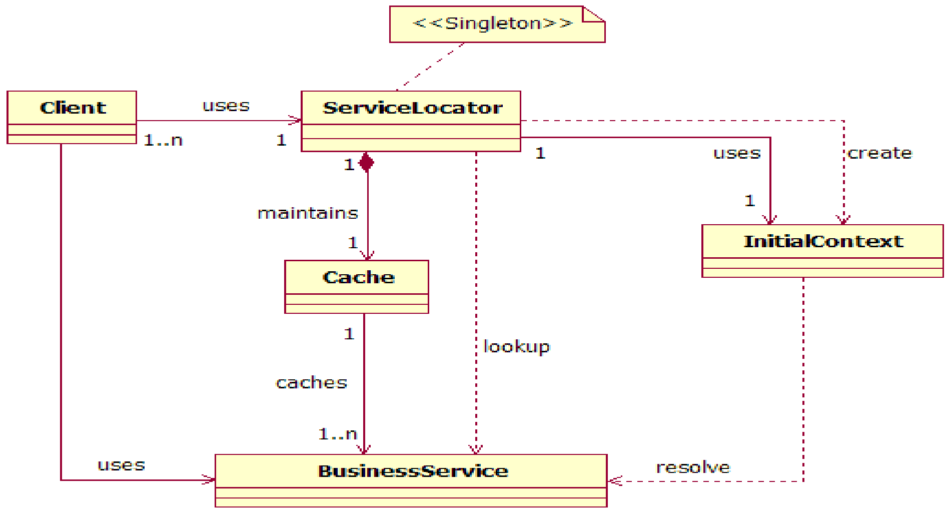
- Clients want to transparently locate business components and services in a uniform manner.

Forces:

- Vendor dependencies are exposed to the clients, we want to encapsulate vendor dependencies for registry implementations, and hide the dependency and complexity from the clients.
- Complexity and duplication of lookup and creation. Clients want to centralize and reuse the implementation of lookup mechanisms.
- Clients may need to reestablish connection to a previously accessed business object.
- Clients want to avoid performance overhead related to initial context creation and service lookups.

Solution:

- Use a Service Locator to implement and encapsulate service and component lookup. A Service Locator hides the complexities of Initial Context creation, lookup, and object re-creation:



Consequences:

- Abstracts complexity of lookups.
- Uniform lookup for all clients.
- Improves network performance.
- Improves client performance by caching.

Proxy

Problem:

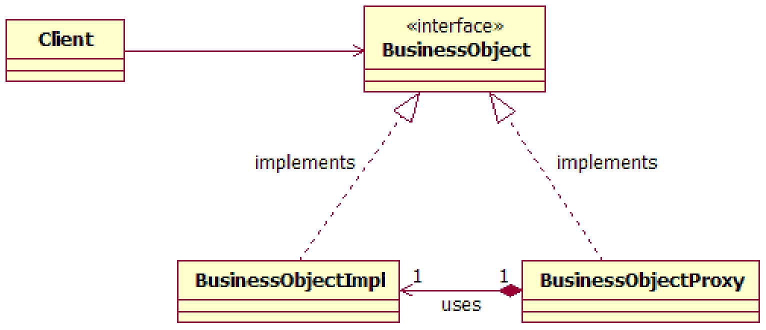
- Provide a stand-in or placeholder to another object, in order to control access to it.

Forces:

- The desired object may be inaccessible (if it exists in a different address space).
- The desired object may be expensive to instantiate and you want to delay its creation until absolutely necessary.
- The desired object may need to be protected from unauthorized access.
- You may need to perform special actions upon accessing the desired object.

Solution:

- Use a Proxy to provide a separate implementation of interface and working code for location transparency:



Consequences:

- Decouples interface and implementation by providing two objects.
- Proxy handles all incoming requests to the object, it knows how to contact the object if it is remotely located.
- Proxy passes along all authorized communication to the object (and prevents unauthorized communication).
- If the Implementation class changes, the Proxy remains the same.
- An example is the EJB's Remote Interface.

Session Façade

Problem:

- Tight coupling between clients and business tier; fine-grained access.

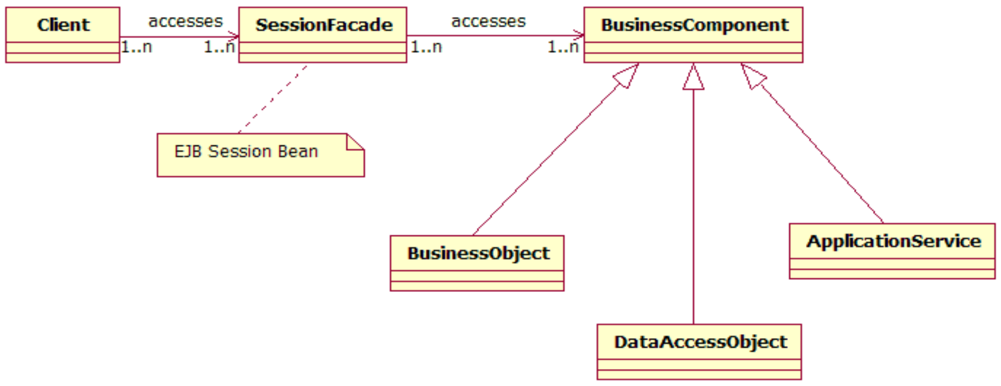
Forces:

- You want to avoid giving clients direct access to business-tier components, to prevent tight coupling with the clients.
- Numerous business objects are exposed over the network.

- No uniform access to business tier. You want to centralize and aggregate all business logic that needs to be exposed to remote clients.
- Clients are exposed to complex interactions and interdependencies of business objects. You want to hide the complex interactions and interdependencies between business components and services to improve manageability, centralize logic, increase flexibility, and improve ability to cope with changes.

Solution:

- Use a Session Façade to encapsulate business-tier components and expose a coarse-grained service to remote clients. Clients access a Session Façade instead of accessing business components directly. Use a Session Façade to encapsulate the complexity of interactions:



Consequences:

- Controller layer for business tier.
- Uniform exposure of business components.
- Reduces coupling between the tiers.
- Provides a uniform coarse-grained access.
- Centralizes business logic.
- Centralizes security management.
- Centralizes transaction control.
- Exposes fewer remote interfaces to clients.
- Improves performance, reduces fine-grained remote methods.

Adapter (Wrapper)

Problem:

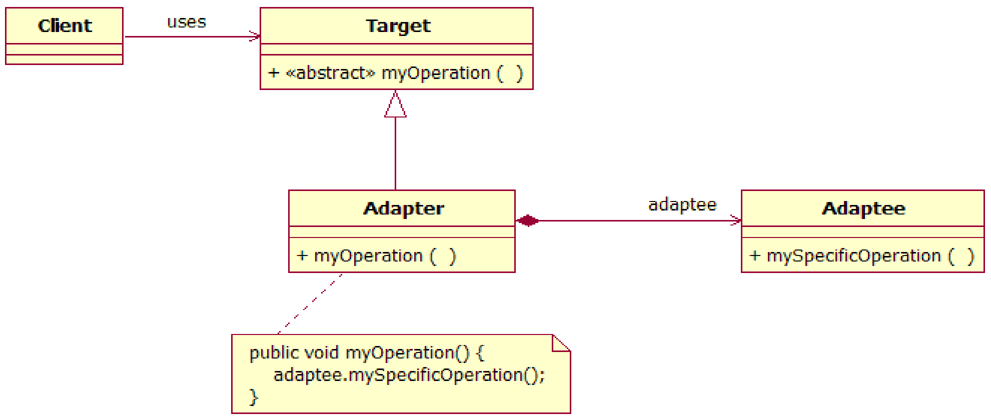
- Convert the interface of a class into another interface clients expect.

Forces:

- A potentially reusable class may not have the appropriate interface required by the particular application domain.

Solution:

- Adapter lets classes work together that could not otherwise because of incompatible interfaces. Use an Adapter pattern to adapt one interface to another:


Client
uses
Target
+ «abstract» myOperation ( )
Adapter
+ myOperation ( )
adaptee
Adaptee
+ mySpecificOperation ( )
public void myOperation() {
adaptee.mySpecificOperation();
}

Consequences:

- Adapter pattern maps the interface of one class onto another so that they can work together. These incompatible classes may come from different libraries or frameworks.
- Adapter pattern can be used to expose existing component as Web Service.

- Adapter pattern helps resolve integration issues.

Command

Problem:

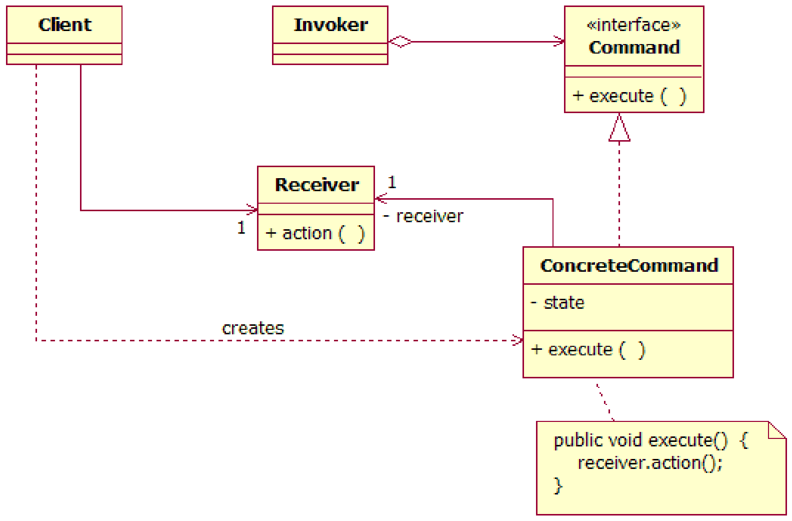
- There are a lot of similar methods, and the interface to implement that kind of object is becoming heavy.

Forces:

- Too many public methods for other objects to call. An interface that is unworkable and always changing. You feel that a method name must include prose describing the exact action, and this is preventing layering your code.

Solution:

- Encapsulate a request as a Command object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Consequences:

- Command pattern is an example of pluggable behavior that enforce client access to Web Services.
- The Command pattern is commonly used for gathering requests from client objects and packaging them into a single object for processing.
- The Command pattern allows for having well defined command interfaces that are implemented by the object that provides the processing for the client requests packaged as commands.
- Command design pattern provides a convenient way to store and execute an "Undo" function.
- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands can be manipulated and extended like any other object.
- Commands can be made into a composite command.
- Commands can be stored. Since a command encapsulates all the data for a given request, it can be created and initialized at one point and applied at another.

Web Service Broker

Problem:

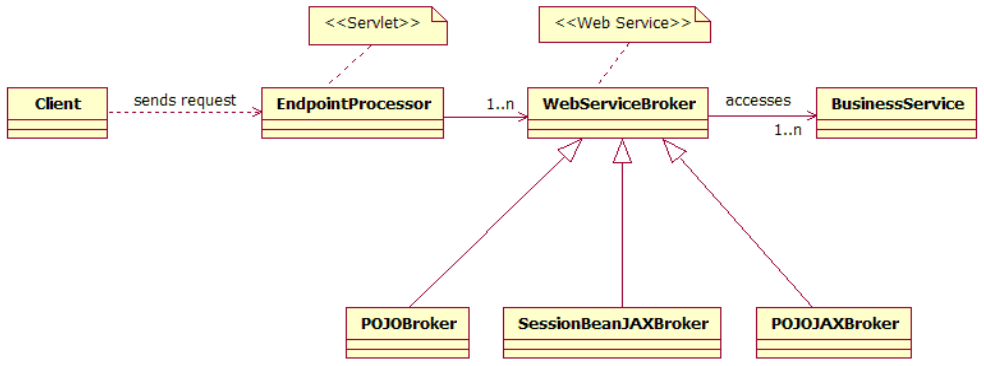
- You want to provide access to one or more services using XML and web protocols.

Forces:

- You want to reuse and expose existing services to clients.
- You want to monitor and potentially limit the usage of exposed services, based on your business requirements and system resource usage.
- Your services must be exposed using open standards to enable integration of heterogeneous applications.
- You want to bridge the gap between business requirements and existing service capabilities.

Solution:

- Use a Web Service Broker to expose and broker one or more services in your application to external clients as a Web Service using XML and standard web protocols:



Consequences:

- Introduces a layer between client and service.
- Existing remote Session Façades need be refactored to support local access.
- Network performance may be impacted due to web protocols.
- Coordinates interactions among one or more services, aggregates responses and may demarcate and compensate transactions.

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.

1212

Describe how to handle the various types of return values, faults, errors, and exceptions that can occur during a Web service interaction.

A Handler class must implement the javax.xml.rpc.handler.Handler interface:

```
package javax.xml.rpc.handler;

public interface Handler {
        boolean handleRequest(MessageContext context);
        boolean handleResponse(MessageContext context);
        boolean handleFault(MessageContext context);
        // ...
}
```

The handleRequest, handleResponse and handleFault methods for a SOAP message handler get access to the SOAPMessage from the SOAPMessageContext. The implementation of these methods can modify the SOAPMessage including the headers and body elements.

The handleRequest method performs one of the following steps after performing handler specific processing of the request SOAP message:

- Return true to indicate continued processing of the request handler chain. The HandlerChain takes the responsibility of invoking the next entity. The next entity may be the next handler in the HandlerChain or if this handler is the last handler in the chain, the next entity is the target service endpoint. The mechanism for dispatch or invocation of the target service endpoint depends on whether the request HandlerChain is on the client side or service endpoint side.
- Return false to indicate blocking of the request handler chain. In this case, further processing of the request handler chain is blocked and the target service endpoint is not dispatched. The JAX-RPC runtime system takes the responsibility of invoking the response handler chain next with the appropriate SOAPMessageContext. The Handler implementation class has the responsibility of setting the response SOAP message in the handleRequest method and perform additional processing in the handleResponse method. In the default processing model, the response handler chain starts processing from the same Handler instance (that returned false) and goes backward in the execution sequence.
- Throw the javax.xml.rpc.soap.SOAPFaultException to indicate a SOAP fault. The Handler implementation class has the responsibility of setting the SOAP fault in the SOAP message in either handleRequest and/or handleFault method. If SOAPFaultException is thrown by a server-side

request handler's `handleRequest` method, the `HandlerChain` terminates the further processing of the request handlers in this handler chain and invokes the `handleFault` method on the `HandlerChain` with the SOAP message context. Next, the `HandlerChain` invokes the `handleFault` method on handlers registered in the handler chain, beginning with the `Handler` instance that threw the exception and going backward in execution. The client-side request handler's `handleRequest` method should not throw the `SOAPFaultException`. Refer to the SOAP specification for details on the various SOAP `faultcode` values and corresponding specification.

- Throw the `JAXRPCException` or any other `RuntimeException` for any handler specific runtime error. If `JAXRPCException` is thrown by a `handleRequest` method, the `HandlerChain` terminates the further processing of this handler chain. On the server side, the `HandlerChain` generates a SOAP fault that indicates that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to a runtime error during the processing of the message. Refer to the SOAP specification for details on the various SOAP `faultcode` values. On the client side, the `JAXRPCException` or runtime exception is propagated to the client code as a `RemoteException` or its subtype.

The `handleResponse` method performs the processing of the SOAP response message. It does one of the following steps after performing its handler specific processing of the SOAP message:

- Return `true` to indicate continued processing of the response handler chain. The `HandlerChain` invokes the `handleResponse` method on the next `Handler` in the handler chain.
- Return `false` to indicate blocking of the response handler chain. In this case, no other response handlers in the handler chain are invoked. On the service endpoint side, this may be useful if response handler chooses to issue a response directly without requiring other response handlers to be invoked.
- Throw the `JAXRPCException` or any other `RuntimeException` for any handler specific runtime error. If `JAXRPCException` is thrown by the `handleResponse` method, the `HandlerChain` terminates the further processing of this handler chain. On the server side, the `HandlerChain` generates a SOAP fault that indicates that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to a runtime error during the processing of the message. On the client side, the `JAXRPCException` or runtime exception is propagated to the client code as a `RemoteException` or its subtype.

The `handleFault` method performs the SOAP fault related processing. The JAX-RPC runtime system should invoke the `handleFault` method if a SOAP fault needs to be processed by either client-side or server-side handlers. The `handleFault` method does one of the following steps after performing handler specific processing of the SOAP fault:

- Return `true` to indicate continued processing of the fault handlers in the handler chain. The `HandlerChain` invokes the `handleFault` method on the next `Handler` in the handler chain.
- Return `false` to indicate blocking of the fault processing in the handler chain. In this case, no other handlers in the handler chain are invoked. The JAX-RPC runtime system takes the further responsibility of processing the SOAP message.
- Throw `JAXRPCException` or any other `RuntimeException` for any handler specific runtime error. If `JAXRPCException` is thrown by the `handleFault` method, the `HandlerChain` terminates the further processing of this handler chain. On the server side, the `HandlerChain` generates a SOAP fault that indicates that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to a runtime error during the processing of the message. On the client side, the `JAXRPCException` or runtime exception is propagated to the client code as a `RemoteException` or its subtype.

Please note that when a `JAXRPCException` or `RuntimeException` raised on the server is converted to a SOAP fault for the purpose of being transmitted to the client, there are no guarantees that any of the information it contains will be preserved.

The following shows an example of the SOAP fault processing. In this case, the request handler `Handler_2` on the server side throws a `SOAPFaultException` in the `handleRequest` method:

1. `Handler_1.handleRequest`
2. `Handler_2.handleRequest` -> throws `SOAPFaultException`
3. `Handler_2.handleFault`
4. `Handler_1.handleFault`

Describe the role that Web services play when integrating data, application functions, or business processes in a J2EE application.

1212

Describe how to design a stateless Web service that exposes the functionality of a stateful business process.

1212

Chapter 11. Endpoint Design and Architecture

Given a scenario, design Web service applications using information models that are either procedure-style or document-style.

Choosing Between an RPC-Style (remote procedure call) and a Message-Style (document-style) Web Service. RPC-style Web services are interface driven, which means that the business methods of the underlying stateless session EJB determine how the Web service works. When clients invoke the Web service, they send parameter values to the Web service, which executes the corresponding methods and sends back the return values. The relationship is synchronous, which means that the client waits for a response from the Web service before it continues with the remainder of its application. Create an RPC-style Web service if your application has the following characteristics:

- The client invoking the Web service needs an immediate response.
- The client and Web service work in a back-and-forth, conversational way.
- The behavior of the Web service can be expressed as an interface.
- The Web service is process-oriented rather than data-oriented.

Examples of RPC-style Web services include providing the current weather conditions in a particular location; returning the current price for a given stock; or checking the credit rating of a potential trading partner prior to the completion of a business transaction. In each case the information is returned immediately, implying a synchronous relationship between the client and the Web service.

RPC is essentially a Remote Procedure Call in which the client sends a SOAP request to execute an operation on the Web Service. The SOAP request contains the name of method to be executed and the parameter it takes. The server running the Web Service converts this request to appropriate objects (java method call, EJB method call etc with parameters of defined type), executes the operation and sends the response as SOAP message to client. At the client side, this response is used to form appropriate objects and return the required information (output) to the client. RPC-style Web Services are tightly coupled because the sending parameters and return values are as described in WSDL (Web Service Description Language) file and are wrapped in the SOAP body. Following is an example SOAP Body of RPC-style Web Service, which invokes `GetStockQuote` method with input parameter `"ORCL"`:

```
<SOAP-ENV:Envelope...>
  <SOAP-ENV:Body>
    <m:GetStockQuote xmlns:m="http://hello">
      <m:Symbol>ORCL</m:Symbol>
    </m:GetStockQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

RPC-style Web Services follow call/response semantics, and hence they are synchronous, which means that the client sends the request and waits for the response till the request is processed completely.

You should create a **message-style (document-style)** Web service if your application has the following characteristics:

- The client has an asynchronous relationship with the Web service, or in other words, the client does not expect an immediate response.
- The Web service is data-oriented rather than process-oriented.

Examples of message-style Web services include processing a purchase order; accepting a request for new DSL home service; or responding to a request for quote order from a customer. In each case, the client sends an entire document, such as purchase order, to the Web service and assumes that the Web service is processing it in some way, but the client does not require an answer right away or even at all. If your Web service will work in this asynchronous, document-driven manner, then you should consider designing it as a message-style Web service. NOTE: Document-Style web servives can use both one-way (non-blocking) calls and two-way (request-response) calls, but preferrable choice will be one-way calls.

Document-Style Web Service are loosely coupled and the request/response are in the form of XML documents. The client sends the parameter to the Web Service as XML document, instead of discrete set of parameter values. The Web Service processes the document, executes the operation and constructs & sends the response to the client as an XML document. There is no direct mapping between the server objects (parameters, method calls etc) and the values in XML documents. The application has to take care of mapping the XML data values. The SOAP Body of a Document-Style carries one or more XML documents, within its body. The protocol places no constraint on how that document needs to be structured, which is totally handled at the application level. Document-Style Web Service follows asynchronous processing. Following is an example SOAP body for Document-Style Web Service:

```
<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <StockQuoteRequest symbol="IBA-USA"/>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The parameters of the methods which are to be exposed by the document style Web Service should be of type XML `element` only. The return type of the method can be either an XML `element` or `void`.

The Simple Object Access Protocol (SOAP) offers two messaging styles: RPC (Remote Procedure Call) and document style. One is for creating tightly coupled, inter-object style interfaces for Web services components; the other is for developing loosely coupled, application-to-application and system-to-system interfaces.

RPC-Style

An RPC is a way for an application running in one execution thread on a system to call a procedure belonging to another application running in a different execution thread on the same or a different system. RPC interfaces are based on a request-response model where one program calls, or requests a service of, another across a tightly coupled interface. In Web services applications, one service acts as a client, requesting a service; the other as a server, responding to that request. RPC interfaces have two parts: the call-level interface seen by the two applications, and the underlying protocol for moving data from one application to the other. NOTE, it may be not only request-response (two-way) RPC call, but also one-way RPC call (but more often it is used with two-way calls).

The call-level interface to an RPC procedure looks just like any other method call in the programming language being used. It consists of a method name and a parameter list. The parameter list is made up of the variables passed to the called procedure and those returned as part of its response.

For Web services, SOAP defines the wiring between the calling and called procedures. At the SOAP level, the RPC interface appears as a series of highly structured XML messages moving between the client and the server where the `Body` of each SOAP message contains an XML representation of the call or return stack:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
        <env:Body>
                <sm:someMethod xmlns:sm="http://www.xyz.com/sm">
                        <someParams>
                                <item>100</item>
                                <item>200</item>
                        </someParams>
                </sm:someMethod>
        </env:Body>
</env:Envelope>
```

The transformation from call-level interface to XML and back occurs through the magic of two processes: marshaling and serialization.

1. The process begins with the client calling a method implemented as a remote procedure. The client actually calls a proxy stub that acts as a surrogate for the real procedure. The proxy stub presents the same external interface to the caller as would the real procedure, but instead of implementing the procedure's functionality, implements the processes necessary for preparing and transporting data across the interface.
2. The proxy stub gathers the parameters it receives through its parameter list into a standard form, in this case, into a SOAP message, through a process called marshaling.
3. The proxy stub encodes the parameters as appropriate during the marshaling process to ensure the recipient can correctly interpret their values. Encoding may be as simple as identifying the correct structure and data type as attributes on the XML tag enclosing the parameter's value or as complex as converting the content to a standard format such as Base64. The final product of the marshaling process is a SOAP message representation of the call stack.
4. The proxy stub serializes the SOAP message across the transport layer to the server. Serialization involves converting the SOAP message into a TCP/IP buffer stream and transporting that buffer stream between the client and the server.

The server goes through the reverse process to extract the information it needs. A listener service on the server deserializes the transport stream and calls a proxy stub on the server that unmarshals the parameters, decodes and binds them to internal variables and data structures, and invokes the called procedure. The listener process may be, for example, a J2EE servlet, JSP (JavaServer Page), or Microsoft ASP (Active Server Page). The client and server reverse roles and the inverse process occurs to return the server's response to the client.

Document-Style

The difference between RPC-Style and Document-Style is primarily in the control you have over the marshaling process. With RPC-style messaging, standards govern that process. With document-style messaging, you make the decisions: you convert data from internal variables into XML; you place the XML into the Body element of the encapsulating SOAP document; you determine the schema(s), if any, for validating the document's structure; and you determine the encoding scheme, if any, for interpreting data item values. The SOAP document simply becomes a wrapper containing whatever content you decide. For example, the SOAP document shown in following example contains an XML namespace reference, `http://www.xyz.com/genealogy`, that presumably includes all the information a receiving program needs for validating the message's structure and content, and for correctly interpreting data values:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
        <env:Body>
                <xyz:family xmlns:xyz="http://www.xyz.com/genealogy">
                        <parents>
                                <father age="29">Mikalai</father>
                                <mother age="29">Volha</mother>
                        </parents>
                        <children>
                                ...
                        </children>
                </xyz:family>
        </env:Body>
</env:Envelope>
```

If you compare the steps involved in typical document-style message exchange process with those involved in processing an RPC-style message, you will notice they are essentially parallel processes:

1. The SOAP client uses an Extensible Stylesheet Language Transformation (XSLT) and the DOM parser, or some other means, to create an XML document.
2. The SOAP client places this XML document into the Body of a SOAP message.
3. The SOAP client optionally includes a namespace reference in the message that other applications can use for validating the encapsulated document's format and content. The namespace reference may be included as an attribute either on one of the SOAP elements or on the XML document's root element. If the document does not include a namespace reference, the client and server must agree on some other scheme for validating and interpreting the document's contents.
4. The SOAP client serializes the message to the SOAP server across either an HTTP or SMTP bound interface.

The SOAP server reverses the process, potentially using a different XSLT, to validate, extract, and bind the information it needs from the XML document to its own internal variables. The roles reverse and the two follow inverse processes for returning and accessing any response values. The rules guiding the marshaling process are the primary difference between this process and that for RPC-style messages. With document-style, you as the SOAP client's author create those rules.

RPC-style messaging maps to the object-oriented, component-technology space. It is an alternative to other component technologies such as DCOM and CORBA where component models are built around programmable interfaces and languages such as Java and C#. RPC-style messaging's strength in this space lies in its platform independence. It offers a standards-based, platform-independent component technology, implemented over standard Internet protocols. One of the benefits of this style's XML layer is that clients and servers can use different programming languages, or technologies, to implement their respective side of the interface, which means one side can choose one set of technologies, such as J2EE's JAX-RPC, while the other chooses a completely different set, such as .NET's C#. RPC-style messaging's standards heritage can be an important consideration in hybrid environments (one using multiple technologies such as J2EE and .NET) and can provide a transition path between different technologies.

RPC-Style messaging's weaknesses

- Strong coupling
  If you change the number, order, or data types of the parameters to the call-level interface, you must make the change on both sides of the interface.
- Synchronicity
  Most programming languages assume synchronous method calls: the calling program normally waits for the called program to execute and return any results before continuing. Web services are asynchronous by nature and, in comparison to technologies such as DCOM and CORBA, long running. You may want to take advantage of Web services' asynchronous nature to avoid the user having to wait for calls to complete by developing asynchronous RPC calls, but that adds another level of complexity to your application. Some tools hide this complexity using callbacks, or other techniques, to enable processing overlap between the

request and the response. Check to see if the tools you' re using let you choose between synchronous and asynchronous RPC calls.

- Marshaling and serialization overhead
  Marshaling and serializing XML is more expensive than marshaling and serializing a binary data stream. With XML, at least one side of the interface, and possibly both, involves some parsing in order to move data between internal variables and the XML document. There is also the cost of moving encoded text, which can be larger in size than its binary equivalent, across the interface.

The coupling and synchronicity issues are common to RPC-based component technologies. So they are really not discriminators when making comparisons between these technologies. The marshaling and serialization overhead is greater for RPC-style messaging and places this messaging style at a relative disadvantage. However, with today's high-speed processors and networks, performance is generally not an issue.

Document-style messaging is clearly an option in any situation where an XML document is one of the interface parameters. It is ideal for passing complex business documents, such as invoices, receipts, customer orders, or shipping manifests. Document-style messaging uses an XML document and a stylesheet to specify the content and structure of the information exchanged across the interface, making it an obvious choice in situations where a document's workflow involves a series of services where each service processes a subset of the information within the document. Each service can use an XSLT to validate, extract, and transform only the elements it needs from the larger XML document; with the exception of those elements, the service is insensitive to changes in other parts of the document. The XSLT insulates the service from changes in the number, order, or type of data elements being exchanged. As long as the service creating the document maintains backwards compatibility, it can add or rearrange the elements it places into a document without affecting other services. Those services can simply ignore any additional data. Document-style messaging is also agnostic on the synchronicity of the interface; it works equally well for both synchronous and asynchronous interfaces.

Document-style messaging's weaknesses

- No standard service identification mechanism
  With document-style messaging, the client and server must agree on a service identification mechanism: a way for a document's recipient to determine which service(s) need to process that document. SOAP header entries offer one option; you can include information in the document's header that helps identify the service(s) needed. WS-Routing makes just such a proposal. Another option is to name elements in the Body of the message for the services that need to process the payload the elements contain. You might ask how that differs from schema-based RPC-style messaging. You would be right in assuming there is little or no difference except possibly in terms of the number of "calls" that can be made per message. A third option is to perform either structure or content analysis as part of a service selection process in order to identify the services needed to process the document.
- Marshaling and serialization overhead
  Document-style messaging suffers from the same drawbacks as RPC-style messaging in this area. However, the problem may be more severe with document-style messaging. Document-style messaging incurs overhead in three areas: in using DOM, or another technique, to build XML documents; in using DOM, or SAX, to parse those documents in order to extract data values; and in mapping between extracted data values and internal program variables. Tools generating equivalent RPC-style interfaces optimize these transformations. You may have trouble achieving the same level of efficiency in your applications using standard tools.

There are two compelling reasons to use document-style messaging. One is to gain the independence it provides. Its strength lies in decoupling interfaces between services to the point that they can change completely independently of one another. The other is that document-style messaging puts the full power of XML for structuring and encoding information at your disposal. The latter is one reason many consider document-style superior to RPC-style messaging.

RPC-style messaging's strength is as a bridging component technology. It is a good option for creating new components and for creating interfaces between Web services and existing components - you simply wrap existing components with RPC-style Web services interfaces. RPC-style messaging is also an excellent component standard in situations where you are using multiple technologies, such as J2EE and .NET, and want to develop sharable components.

Document-style messaging's strengths are in situations where an XML document is part of the data being passed across the interface, where you want to leverage the full power of XML and XSL, and in instances where you want to minimize coupling between services forming an interface, such as in application-to-application and system-to-system interfaces.

WSDL Example for RPC-Style:

```
...
<message name="myMethodRequest">
    <part name="x" type="xsd:int"/>
</message>
```

```
<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
    </operation>
</portType>
...
```

NOTE: `part` element has attribute `type`.

RPC-Literal SOAP message for this request:

```
<soap:Envelope>
    <soap:Body>
        <myMethod>
            <x>5</x>
        </myMethod>
    </soap:Body>
</soap:Envelope>
```

WSDL Example for Document-Style:

```
<types>
    <schema>
        <element name="xElement" type="xsd:int"/>
    </schema>
</types>

<message name="myMethodRequest">
    <part name="x" element="xElement"/>
</message>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
    </operation>
</portType>
```

NOTE: `part` element has attribute `element` with value of globally declared element.

Document-Literal SOAP message:

```
<soap:Envelope>
    <soap:Body>
        <xElement>5</xElement>
    </soap:Body>
</soap:Envelope>
```

Describe the function of the service interaction and processing layers in a Web service.

See blueprints.

Describe the tasks performed by each phase of an XML-based, document oriented, Web service application, including the consumption, business processing, and production phases.

1212

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.

1. Refactor Server's Web Service to one-way (asynchronous) from two-way (synchronous).
2. Add some identification information in SOAP message (can be implemented using header with unique `ID` value).
3. Deploy new Web Service on the Client and allow Server call back client when Server processed incoming message and ready to return results to caller (Client).

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.

1212

Appendixes

Bibliography

[BP-1.0] *Basic Profile Version 1.0 - Final Specification (http://www.ws-i.org/Profiles/BasicProfile-1.0.html)*.
[SOAP-1.1] *Simple Object Access Protocol (SOAP) 1.1 (http://www.w3.org/TR/2000/NOTE-SOAP-20000508)*.
[SOAP-1.1-Attachments] *SOAP Messages with Attachments (http://www.w3.org/TR/SOAP-attachments)*.
[WSDL-1.1] *Web Services Description Language (WSDL) 1.1 (http://www.w3.org/TR/wsdl)*.
[UDDI-2.0-DS] *UDDI Version 2.03 Data Structure Reference (http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm)*.
[UDDI-2.0-API] *UDDI Version 2.04 API Specification (http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm)*.
[XML-DSIG] *XML Digital Signature (http://www.w3.org/Signature)*.
[XML-SIG] *XML-Signature Syntax and Processing (http://www.w3.org/TR/2002/REC-xmldsig-core-20020212)*.
[XML-ENCR] *XML Encryption Syntax and Processing (http://www.w3.org/TR/2002/REC-xmlenc-core-20021210)*.
[SAML] *SAML (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security)*.
[XACML] *XACML (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml)*.
[WS-SEC] *Web Services Security (WS-Security) (http://www-106.ibm.com/developerworks/webservices/library/ws-secure/)*.
[J2EE-1.4] *J2EE 1.4 Specification - Final Release (http://java.sun.com/j2ee/1.4/download.html)*.
[JSR-921] *JSR 921: Implementing Enterprise Web Services 1.1 (http://jcp.org/en/jsr/detail?id=921)*.

Appendix A. First Appendix

SAAJ API

SAAJ Interfaces

```
package javax.xml.soap;
public interface Node extends org.w3c.dom.Node {
    public abstract String getValue();
    public abstract void setParentElement(SOAPElement parent)
        throws SOAPException;
    public abstract SOAPElement getParentElement();
    public abstract void detachNode();
    public abstract void recycleNode();
    public abstract void setValue(String value);
}
package javax.xml.soap;
public interface Text extends Node, org.w3c.dom.Text {
    public abstract boolean isComment();
}
package javax.xml.soap;
import java.util.Iterator;
public interface SOAPElement extends Node, org.w3c.dom.Element {
    public abstract SOAPElement addChildElement(Name name) throws SOAPException;
    public abstract SOAPElement addChildElement(String localName)
        throws SOAPException;
    public abstract SOAPElement addChildElement(String localName, String prefix)
        throws SOAPException;
    public abstract SOAPElement addChildElement(
        String localName, String prefix, String uri) throws SOAPException;
    public abstract SOAPElement addChildElement(SOAPElement element)
        throws SOAPException;
    public abstract SOAPElement addTextNode(String text) throws SOAPException;
    public abstract SOAPElement addAttribute(Name name, String value)
        throws SOAPException;
    public abstract SOAPElement addNamespaceDeclaration(
        String prefix, String uri) throws SOAPException;
    public abstract String getAttributeValue(Name name);
    public abstract Iterator getAllAttributes();
    public abstract String getNamespaceURI(String prefix);
    public abstract Iterator getNamespacePrefixes();
    public abstract Name getElementName();
    public abstract boolean removeAttribute(Name name);
    public abstract boolean removeNamespaceDeclaration(String prefix);
    public abstract Iterator getChildElements();
    public abstract Iterator getChildElements(Name name);
    public abstract void setEncodingStyle(String encodingStyle)
        throws SOAPException;
    public abstract String getEncodingStyle();
    public abstract void removeContents();
    public abstract Iterator getVisibleNamespacePrefixes();

package javax.xml.soap;
import org.w3c.dom.Document;
import java.util.Locale;
public interface SOAPBody extends SOAPElement {
    public abstract SOAPFault addFault() throws SOAPException;
    public abstract boolean hasFault();
    public abstract SOAPFault getFault();
    public abstract SOAPBodyElement addBodyElement(Name name)
        throws SOAPException;
    public abstract SOAPFault addFault(Name faultCode,
                                       String faultString,
                                       Locale locale) throws SOAPException;
    public abstract SOAPFault addFault(Name faultCode, String faultString) throws
SOAPException;
    public abstract SOAPBodyElement addDocument(Document document) throws
SOAPException;
}
package javax.xml.soap;
public interface SOAPBodyElement extends SOAPElement {}
package javax.xml.soap;
import java.util.Locale;
public interface SOAPFault extends SOAPBodyElement {
    public abstract void setFaultCode(String faultCode) throws SOAPException;
    public abstract String getFaultCode();
    public abstract void setFaultActor(String faultActor) throws SOAPException;
    public abstract String getFaultActor();
    public abstract void setFaultString(String faultString) throws SOAPException;
    public abstract String getFaultString();
    public abstract Detail getDetail();
    public abstract Detail addDetail() throws SOAPException;
    public abstract void setFaultCode(Name name) throws SOAPException;
    public abstract Name getFaultCodeAsName();
    public abstract void setFaultString(String faultString, Locale locale) throws
SOAPException;
    public abstract Locale getFaultStringLocale();
}
package javax.xml.soap;
public interface SOAPEnvelope extends SOAPElement {
    public abstract Name createName(String localName, String prefix, String uri)
        throws SOAPException;

    public abstract Name createName(String localName) throws SOAPException;
    public abstract SOAPHeader getHeader() throws SOAPException;
    public abstract SOAPBody getBody() throws SOAPException;
    public abstract SOAPHeader addHeader() throws SOAPException;
    public abstract SOAPBody addBody() throws SOAPException;
}
package javax.xml.soap;
import java.util.Iterator;
public interface SOAPHeader extends SOAPElement {
    public abstract SOAPHeaderElement addHeaderElement(Name name)
        throws SOAPException;
    // does not detach headers
    public abstract Iterator examineHeaderElements(String actor);
    // detaches headers
    public abstract Iterator extractHeaderElements(String actor);
    public abstract Iterator examineMustUnderstandHeaderElements(String actor);
    public abstract Iterator examineAllHeaderElements();
    public abstract Iterator extractAllHeaderElements();
}
package javax.xml.soap;
public interface SOAPHeaderElement extends SOAPElement {
    public abstract void setActor(String actorURI);
    public abstract String getActor();
    public abstract void setMustUnderstand(boolean mustUnderstand);
    public abstract boolean getMustUnderstand();
}
```

SAAJ API Classes.

```
package javax.xml.soap;
import java.io.IOException;
import java.io.InputStream;
public abstract class MessageFactory {
    protected MessageFactory() {}
    public static MessageFactory newInstance() throws SOAPException {        ...
    }
    public abstract SOAPMessage createMessage() throws SOAPException;
    public abstract SOAPMessage createMessage(
        MimeHeaders mimeheaders, InputStream inputstream)
            throws IOException, SOAPException;
    ...
}
package javax.xml.soap;
import javax.activation.DataHandler;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Iterator;
public abstract class SOAPMessage {
    public SOAPMessage() {}
    public abstract String getContentDescription();
    public abstract void setContentDescription(String description);
    public abstract SOAPPart getSOAPPart();
    public abstract void removeAllAttachments();
    public abstract int countAttachments();
    public abstract Iterator getAttachments();
    public abstract Iterator getAttachments(MimeHeaders headers);
    public abstract void addAttachmentPart(AttachmentPart attachmentpart);
    public abstract AttachmentPart createAttachmentPart();
    public AttachmentPart createAttachmentPart(DataHandler datahandler) { ... }
    public abstract MimeHeaders getMimeHeaders();
    public AttachmentPart createAttachmentPart(Object content, String contentType) {
        ...
    }
    public abstract void saveChanges() throws SOAPException;
    public abstract boolean saveRequired();
    public abstract void writeTo(OutputStream out) throws SOAPException, IOException;
    public SOAPBody getSOAPBody() throws SOAPException {
        ...
    }
    public SOAPHeader getSOAPHeader() throws SOAPException {        ...    }
    public void setProperty(String property, Object value) throws SOAPException  {
        ...
    }
    public Object getProperty(String property) throws SOAPException {     ...   }
}
package javax.xml.soap;
import javax.xml.transform.Source;
import java.util.Iterator;
public abstract class SOAPPart implements org.w3c.dom.Document {
    public SOAPPart() {}
    public abstract SOAPEnvelope getEnvelope() throws SOAPException;
    public String getContentId() {        ...    }
    public String getContentLocation() {        ...        }
    public void setContentId(String contentId) {        ...    }
    public void setContentLocation(String contentLocation) {        ...    }
    public abstract void removeMimeHeader(String header);
    public abstract void removeAllMimeHeaders();
    public abstract String[] getMimeHeader(String name);
    public abstract void setMimeHeader(String name, String value);
    public abstract void addMimeHeader(String name, String value);
    public abstract Iterator getAllMimeHeaders();
    public abstract Iterator getMatchingMimeHeaders(String names[]);
    public abstract Iterator getNonMatchingMimeHeaders(String names[]);
    public abstract void setContent(Source source) throws SOAPException;
    public abstract Source getContent() throws SOAPException;
}
package javax.xml.soap;
import java.util.Locale;
public interface SOAPFault extends SOAPBodyElement {
    public abstract void setFaultCode(String faultCode) throws SOAPException;
    public abstract String getFaultCode();
    public abstract void setFaultActor(String faultActor) throws SOAPException;
    public abstract String getFaultActor();
    public abstract void setFaultString(String faultString)
    public abstract String getFaultString();
    public abstract Detail getDetail();
    public abstract Detail addDetail() throws SOAPException;
    public abstract void setFaultCode(Name name) throws SOAPException;
    public abstract Name getFaultCodeAsName();
    public abstract void setFaultString(String faultString, Locale locale) throws
SOAPException;
    public abstract Locale getFaultStringLocale();
```

Second Section

sdsds

Third Section

sdsds